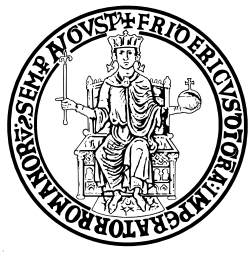
Università degli Studi di Napoli Federico II

FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI
Corso di Laurea in Informatica



Una procedura di Model Checking in un dominio lineare astratto per la verifica di programmi C con array

Tesi di Laurea di:
Dario Della Monica
Matricola: 961/22

Relatore:
Prof. Massimo Benerecetti

Correlatore:
Prof. Giovanni Criscuolo

Anno Accademico 2006/2007

Ringraziamenti

Finalmente è finita!!! Dopo 19 “lunghi brevi” anni la scuola è finita e mi ritrovo nel momento più bello, in cui ricordare e rendere grazie a tutti coloro che in qualche modo hanno contribuito o hanno reso più bello il percorso che mi ha condotto fin qui. E’ bello rendere grazie a chi di dovere perchè è importante far capire che nessuno fa nulla da solo, ma un uomo è potentissimo se parte di un ingranaggio di persone che remano tutte nella stessa direzione. Pertanto queste parole sono rivolte a tutti coloro che, anche se non esplicitamente menzionati, hanno remato con me, per un breve o lungo periodo.

Queste non vogliono essere delle banali parole di circostanza ma dei sentiti ringraziamenti a persone a cui in un modo o nell’altro voglio bene e meritano il mio rispetto e la mia stima. Si sa che spesso è difficile dire delle cose dal vivo, magari perchè a volte è difficile ritagliarsi una situazione, un momento adeguato a fare determinate esternazioni, pertanto esse restano solo nelle intenzioni, senza mai concretizzarsi.

Pertanto approfitto di questo momento di riflessione per dire “cose” e rendere grazie.

Innanzitutto appare indispensabile menzionare e ringraziare gli artefici principali delle mie gioie e soddisfazioni, i miei genitori, che con tanti sacrifici lavorano giorno dopo giorno alla costruzione della mia vita felice. Sacrifici che vengono moltiplicati nei momenti di bisogno o nel momento di dover accontentare qualche mio “capriccio”, detto tra virgolette perchè alcuni “capricci” non sono veri e propri capricci bensì tappe indispensabili alla formazione sociale e culturale di un individuo. Per quanto mi riguarda, ho sempre ricevuto il meglio sia da mia madre che mi ha sempre dimostrato tutte le attenzioni, la comprensione, l’affetto e l’altruismo che solo una mamma può trasmettere, che da mio padre, che mi ha insegnato a vivere e “sopravvivere”, a rischiare, a non avere paura e a credere in me stesso almeno tanto quanto lui credeva in me.

Ritornando ai momenti del bisogno, devo ringraziare una persona che si dimostra sempre preziosa ed impeccabile quando serve. La mia badante

saltuaria e a tempo pieno, mia sorella Tamara, sempre pronta ad assecondare e soddisfare le mie richieste ed esigenze, anche le più stravaganti, e sempre pronta a cogliere e diffondere un momento di allegria.

Devo ringraziare poi un'altra grottesca figura che ultimamente ha preso a ronzarmi attorno, richiedendo attenzioni, energie e tempo ma ripagandole in maniera sublime, contribuendo a migliorare con la sua perenne allegria (tranne quando fa la parte di quella incazzata e rompe le ... sinergie) la qualità della vita di tutte le persone che hanno la fortuna entrarvi in contatto. Si tratta di Immacolata a cui devo molto e a cui va tutta la mia gratitudine e non solo. Ah, dimenticavo... il soggetto presenta anche un'insospettabile intraprendenza. Non cambiare, se possibile!!!

Porgo i miei ringraziamenti anche al professor Benerecetti (il Bene, per i nemici), che ho imparato ad apprezzare per la sua umanità, prima che per la sua competenza e preparazione. Spesso "incompreso" (siamo sicuri???) e messo sotto accusa per la sua eccessiva "curiosità" e "magnanimità" dalle mamme dei laureandi che preferirebbero meno domande per i loro figli in sede di seduta di laurea. Scherzi a parte, porgo i miei più sinceri ringraziamenti per la pazienza ed il tempo (davvero tanto) dedicatomi, sottolineando che per me si è trattato comunque di tempo trascorso piacevolmente.

Ora viene la parte più difficile... ringraziare una moltitudine di gente che merita, sperando di non dimenticare nessuno, anche se conto sul fatto che queste persone non sono affatto permalose e di certo non si offenderanno se verranno omesse, ben sapendo che se non vengono menzionate sarà solo perchè si sa che il periodo pre-laurea è carico di impegni e scadenze e pertanto la memoria vien di gran lunga limitata (figuriamoci la mia, già abbastanza corta).

Innanzitutto voglio ringraziare casa Abate, in particolare Helpettino, Sabalbano e 3sidente, la cui ospitalità sicura ed incondizionata mi ha reso la vita estremamente più facile. E' stato un piacere di pazzi scroccare da voi un letto a due piazze, telefono, internet, "leggere" insalate alla Elvio con TUTTO dentro. E' stato bello partecipare a discussioni politiche, con annessi progetti per un domani migliore, portate avanti fino alle 4 del mattino in condizioni di precaria lucidità. Ma la cosa più bella è che grazie al goffo, e sottolineo ingiusto, soprannome gentilmente affibiatomi dal fiorellino, non ho mai dovuto cucinare nè fare neanche il più piccolo lavoro in casa :-).

Un grazie, anche se MENO SENTITO, va anche al Sukkiotto e a 'Ntonio per la loro ospitalità. Si tratta tuttavia di un grazie meno sentito, a costo di sembrare scortese, perchè i due soggetti hanno avuto il coraggio di farmi dormire in pianerottolo, al freddo e al gelo di una notte d'estate napoletana e soprattutto mi hanno fatto pagare fitto e bollette per ben 2 (DUE) mesi e tutto ciò solo perchè 'Ntonio è più avaro di zio Paperone. P.S.: Vediamo che

fai mo che hai conosciuto a mia sorella e se le dici ciò che promettevi, ehehe. P.P.S.: ragazzi scherzo logicamente. Grazie davvero.

Come non menzionare l'imponente Annapaola (per gli amici "Grasso" o "AnnaProvola", si resta sempre in campo alimentare e non poteva essere altrimenti per una brillante dottoressa laureata con una tesi sul McDonald. Aspettiamo tutti di leggere quella sul Burger King). Ricordo i giorni felici in cui si scomodava a venire fin nel mio studiolo ad insultarmi a domicilio e ricordo con affetto quando non potendoli mangiare perchè perennemente a dieta con scarsissimi (se non inesistenti) risultati, mi portava i gelati mentre io ero impegnato a scrivere 'sta tesi. A tal proposito Annapaola ti voglio dire una cosa: "Ricorda, non sei grassa... sei solo robusta di costituzione"... "Ah, dimenticavo... ti trovo dimagrita!!!".

Un ricordo va anche a biologi e biologhe, in particolare ad Anna e Valentina, che venivano gentilmente ad importunarmi tra una lezione e l'altra con il solo scopo... mmm... ora che ci penso venivano senza motivo, CHE VENIVATE A FARE!!!!! Ricorderò con affetto modi e soprattutto luogo del nostro primo incontro. Ma un grazie grande quanto una casa va anche a GAYtano il pisciaiuolo, per la sua ospitalità nella sua bellissima casa fatta di letti, anzi materassi, appoggiati sul pavimento, ma davvero comodi. Ricorda sempre: LA CAPOLISTA SE NE VA.

Se Ligabue cantava "Lambrusco e pop-corn", devo ringraziare Eugenietta, inventrice nonchè generosa dispensatrce del famosissimo connubio Levinska e pop-corn, oltre che di frasi ad effetto per la mia tesi... Ricorda che sei di VITALE IMPORTANZA per me e non vedo l'ora in cui potremo dichiarare al mondo il nostro amore... ahia Imma stavo solo scherzando (no Eugenia era tutto vero).

Mi sembra di aver finito. Spero di non aver dimenticato nessuno, anche se in realtà ci sono molte altre persone che andrebbero menzionate come Rosaria, per la sua attività di verifica della tesi oppure Roberta, che mi ha ospitato a casina sua, come Alessandro Siani, altro fornitore di giacigli temporanei, che mi ha anche rivelato l'esistenza del TAVOLO DA PING-PONG DELL'ULTIMO PIANO, teatro di tante battaglie tra me e il Punziano che mi hanno visto sempre perdente o ancora Nicola e Fulvio con cui ho condiviso il laboratorio e che talvolta mi hanno fornito piccoli e preziosi suggerimenti e ancora tante tante altre persone con cui ho condiviso momenti ed emozioni, belle, brutte ma mai insignificanti. A tali persone che per un motivo o per un altro non compaiono qui, un SENTITO GRAZIE, perchè grazie a voi sono cresciuto, ho imparato qualcosa dalla vostra personalità che valeva la pena apprezzare e imitare, perchè ognuno ha qualcosa da insegnare a qualsiasi altra persona su questa terra, la bravura sta nel riuscire a cogliere gli aspetti positivi di ognuno e riuscire a farli propri. Pertanto spero che anche

io sia riuscito a trasmettere qualcosa ad ognuno di voi così che il cammino effettuato insieme non sia stato inutile.

Ciao e grazie a tutti.

Indice

Introduzione **7**

1 Verifica di software **11**
1.1 Introduzione alla verifica formale 11
1.2 Model checking simbolico 13
1.3 Tecniche di astrazione per il software 14
1.4 Domini di astrazione 16
1.5 Un esempio di esecuzione di Eureka 17

2 Astrazione **24**
2.1 Programmi lineari con array generalizzati 24
2.1.1 Espressioni lineari generalizzate con array ed espressioni lineari booleane generalizzate con array 24
2.1.2 Sintassi dei programmi lineari con array generalizzati . 26
2.1.3 Rappresentazione tramite CFG 28
2.1.4 Scope delle variabili 29
2.1.5 Semantica di programmi lineari con array 30
2.2 Introduzione all'astrazione 38
2.2.1 Astrazione sintattica 38
2.2.2 Astrazione semantica 38
2.2.3 Un ulteriore beneficio derivante dal processo di astrazione 39
2.3 Meccanismo di astrazione preesistente in Eureka 41
2.3.1 Analisi dei limiti del precedente meccanismo di astrazione utilizzato in Eureka 42
2.4 Un nuovo meccanismo di astrazione 46
2.5 Semantica lineare astratta per programmi lineari con array . . 49
2.5.1 Astrazione $\hat{w}$ della funzione di valutazione concreta w . 50
2.5.2 Estensione della funzione di valutazione astratta $\hat{w}$ alla funzione $\bar{\hat{w}}$ 50
2.5.3 Scope delle variabili astratte 52
2.5.4 Definizione di stato astratto 53

2.5.5 Relazione di transizione astratta e percorsi astratti . . 53
2.5.6 Funzioni di astrazione α e concretizzazione γ 57
2.6 Correttezza dell'astrazione proposta 57
2.6.1 Teorema di correttezza dell'astrazione 62

3 Model checking **70**
3.1 Interprocedural Data Flow Analysis per programmi lineari . . 70
3.1.1 Path Edge e Summary Edge 70
3.2 Model Checking simbolico per programmi lineari 73
3.2.1 Codifica in aritmetica lineare degli statement del programma (α e β) 74
3.2.2 Rappresentazione simbolica tramite ADLC 75
3.2.3 Manipolazione degli ADLC 76
3.2.4 Controparte simbolica dell'algorimtmo di Interprocedural Data Flow Analysis 78
3.3 Interprocedural Data Flow Analysis per astrazioni di programmi lineari con array 81
3.3.1 Path Edge astratti 82
3.3.2 Procedura simbolica di model checking per astrazioni di programmi lineari con array 91

4 Simulazione e raffinamento del modello **115**
4.1 Verifica di fattibilità della traccia 115
4.2 Raffinamento del modello 117
4.3 Meccanismo di raffinamento in Eureka 119

5 Implementazione **124**
5.1 Architettura di Eureka 124
5.2 Modulo per l'astrazione 126
5.3 Modulo di verifica 134
5.3.1 PPL: The Parma Polyhedra Library 135
5.3.2 Costruzione dell'ADLC corrispondente alla relazione di transizione e verifica del modello 139
5.4 Simulazione e raffinamento 140
5.4.1 CVC Lite 141

Conclusioni **145**

Bibliografia **147**

Introduzione

Il presente lavoro di tesi trova la sua collocazione nell'ambito della verifica automatica di software, in particolare esso contribuisce allo sviluppo di Eureka, un tool di verifica automatica di sistemi.

Molte attività umane traggono benefici in termini di velocità, precisione ed affidabilità dall'automazione, totale o parziale, dei processi in essi coinvolti. In particolare, l'attenzione è posta sia ad attività personali, lavorative o di svago, sia e soprattutto ad attività di pubblico interesse. Basti pensare al crescente utilizzo di personal computer per le più disparate attività, all'automazione di processi produttivi industriali, del sistema bancario, fino ad arrivare all'importanza dei calcolatori nella gestione di operazioni spaziali o all'utilizzo di strumenti robotici ad alta precisione per il loro impiego in campo medico.

Se in alcune di tali attività errori di sistemi informatici sono tollerati, nella maggior parte dei casi appare evidente che potrebbero avere conseguenze molto spiacevoli e comportare ingenti danni a livello economico, ma anche in termini di salute o vite umane. Tali sistemi, il cui corretto funzionamento è di vitale importanza, sono definiti **safety critical**. Fanno parte di questa categoria i software per la gestione di satelliti, i sistemi per il controllo del traffico aereo e ferroviario, i database bancari, i sistemi sanitari e le applicazioni utilizzate nell'ambito militare.

Attualmente il mezzo di verifica software più diffuso ed utilizzato consiste nel processo di simulazione e test, che prevede l'esecuzione del software a fronte di determinati input e la verifica che il sistema fornisca le risposte corrette. Sebbene tale tecnica renda possibile l'individuazione di numerosi bug, è quasi sempre impossibile simulare tutti i comportamenti di un sistema anche molto semplice, pertanto spesso non è in grado di fornire una risposta esaustiva circa l'assenza di errori. È quindi fondamentale, per i sistemi safety critical, sostituire o affiancare tale meccanismo di verifica con tecniche formali capaci di dare una risposta esaustiva circa la presenza o meno di errori.

Esistono, pertanto, numerosi mezzi di verifica formale, tra cui il **theorem proving** e il **model checking**, i quali, attraverso la descrizione formale del

sistema in esame e della proprietà che si desidera verificare, permettono, in maniera automatica o semiautomatica, di dimostrare che tutti i comportamenti del sistema verifichino la proprietà desiderata, superando i limiti delle tecniche di testing e simulazione.

Nel caso del theorem proving, sia il sistema che le proprietà che si desiderano verificare devono essere prima espressi in una logica appropriata, poi viene costruita una dimostrazione del fatto che tali proprietà siano derivabili dalla descrizione del sistema. Tale approccio è molto potente e flessibile, tanto è vero che può essere applicato a sistemi molto complessi ed anche a stati infiniti, ma difetta nella costruzione della dimostrazione, che non è facile e spesso viene richiesta l'interazione da parte dell'utente, che suggerisce quali passi di prova eseguire, affinché il sistema riesca a portarla a termine.

Il model checking, invece, prevede sia la descrizione del sistema attraverso un Labeled Transaction System (LTS - sistema a transizioni etichettato) che quella della proprietà da verificare attraverso una logica temporale (ad esempio Linear Temporal Logic, LTL, o Computational Tree Logic, CTL). La procedura di model checking termina con risposta affermativa qualora il modello verifichi la proprietà, altrimenti esibisce un controesempio di percorso di esecuzione ammissibile nel sistema, per cui una delle proprietà specificate non è valida.

Un grande limite del model checking sta nel fatto di dover rappresentare esplicitamente tutti gli stati del sistema e ciò implica un carico di memoria che limita la dimensione del sistema del quale si desidera eseguire verifica. Per ovviare tale problema sono state introdotte tecniche per la rappresentazione simbolica degli stati, grazie alle quali è stato possibile aumentare il numero degli stessi, estendendo quindi l'insieme di sistemi per i quali è possibile eseguire verifica. Si parla quindi di model checking simbolico, attraverso il quale si sono ottenuti ottimi risultati nella verifica di circuiti elettronici di piccole e medie dimensioni.

Il model checking simbolico è comunque applicabile solo a sistemi con un determinato dominio di applicazione, ad esempio programmi booleani oppure programmi lineari. Attualmente non è possibile, ad esempio, eseguire model checking di programmi con array. Pertanto per la verifica di tale classe di programmi è necessario introdurre una fase preliminare che trasformi i programmi con array in programmi con un dominio più semplice (programmi booleani o programmi lineari). Tale operazione è portata a termine dal processo di astrazione. L'astrazione è un processo che permette di mappare un sistema in un altro molto più semplice attraverso la riduzione del numero dei suoi stati o attraverso la mappatura del linguaggio del programma originale in un linguaggio che sia più facile da manipolare. Attraverso la costruzione di modelli astratti è stato possibile sfruttare tecniche di model checking anche

per la verifica di software, dando quindi origine al software model checking.

Affinché la verifica possa terminare con successo è indispensabile trovare il giusto livello di astrazione. A tal fine si è introdotto il paradigma CEGAR (Counter Example Guided Abstraction Refinement) che permette di giungere al giusto livello di astrazione attraverso una o più iterazioni del ciclo Astrazione-Verifica-Raffinamento. L'approccio è stato seguito con successo nella realizzazione del tool di verifica SLAM di Microsoft che prevede l'utilizzo del dominio dei programmi booleani come dominio astratto. Nonostante il tool fornisca ottimi risultati nell'ambito della verifica di driver, si è mostrato poco efficiente nella verifica di altre tipologie di software, in particolar modo per programmi in cui la manipolazione dei dati è predominante rispetto al flusso di controllo.

Si è deciso di progettare e sviluppare un tool di verifica che superi i limiti di SLAM ma che ne erediti i pregi. Nasce così il progetto Eureka, volto alla verifica di programmi lineari con array, ovvero programmi nei quali le variabili sono di tipo numerico e ogni espressione è un'espressione lineare. Prevede l'utilizzo dei programmi lineari come dominio astratto e l'applicazione del paradigma CEGAR per giungere al giusto livello di astrazione.

Il dominio astratto scelto ha il vantaggio di essere più vicino ai programmi concreti rispetto al dominio dei programmi booleani. Tale caratteristica dovrebbe ridurre il numero di cicli astrazione-raffinamento e quindi migliorare le prestazioni complessive. Inoltre i programmi lineari sono più adatti alla rappresentazione di programmi in cui la manipolazione dei dati è predominante.

Nell'ambito del progetto Eureka il mio contributo è mirato ad una futura estensione del tool alla verifica di programmi C con puntatori. Tenendo presente che la memoria di un programma può essere vista come uno sconfinato array, i cui elementi (le varie locazioni di memoria) sono accessibili tramite i puntatori, un primo e fondamentale passo verso l'introduzione degli stessi nel processo di verifica di Eureka è quello di riuscire a gestire, in fase di astrazione e raffinamento, i generici elementi di un array. D'altronde il linguaggio C tratta in maniera perfettamente analoga elementi di un array e locazioni di memoria puntate da puntatori. Nella precedente versione di eureka, un array veniva modellato in fase di astrazione attraverso variabili legate in maniera indissolubile ad uno specifico e prefissato elemento dell'array. Pertanto il mio impegno è stato quello di ridefinire ed estendere il meccanismo di astrazione e raffinamento, per poter eseguire tali fasi su generici elementi di un array.

La presente tesi è articolata nei seguenti capitoli:

- Nel primo capitolo verrà introdotto il problema della verifica automatica, evidenziandone le problematiche e le strategie utilizzate per renderla

più efficiente e fattibile su un dominio sempre più esteso di programmi. Verrà dunque introdotto il paradigma CEGAR e verranno menzionati alcuni tool di verifica sviluppati secondo tale paradigma. Infine verrà introdotto e illustrato in maniera molto intuitiva il funzionamento di Eureka.

- Il secondo capitolo illustra in maniera dattagliata la fase di astrazione. Dapprima verrà evidenziata la necessità di tale fase nell'ambito della verifica formale tramite tecniche di model checking, successivamente verrà descritto il meccanismo di astrazione proposto nel presente lavoro di tesi, mostrando i benefici che esso apporta al tool. Infine verrà esibita una dimostrazione di correttezza della strategia proposta.

- Il terzo capitolo descrive il modulo di Model Checking di Eureka, che si occupa di verificare la raggiungibilità di nodi rappresentanti stati di errore del programma, all'interno del modello astratto. Anche per il modulo di verifica verranno mostrate le modifiche apportate durante il presente lavoro e presentata una dimostrazione di correttezza e completezza.

- Nel quarto capitolo vengono illustrati i moduli di verifica e raffinamento utilizzati in Eureka.

- Infine, il quinto capitolo riguarda i dettagli implementativi del tool. Verranno mostrati ed illustrati i principali algoritmi che implementano la fasi del processo di verifica, ponendo particolare attenzione ai moduli esterni utilizzati all'interno di Eureka.

Capitolo 1

Verifica di software

Verificare la correttezza di un software risulta spesso fondamentale affinchè questo possa essere utilizzato con successo. Test e simulazione spesso non bastano a questo scopo. Per questo motivo c'è bisogno di tecniche alternative, ad esempio il model checking, che possano fornire informazioni certe circa alcune caratteristiche del software in esame. In questo capitolo si introduce il tema della verifica automatica e le sue problematiche. Si descrive la sua evoluzione nel tempo e le soluzioni che sono state trovate per renderla sempre più efficiente e per accrescere il numero di sistemi sui quali è possibile effettuare verifica formale. Si introduce quindi l'utilizzo di modelli astratti per la verifica tramite tecniche di model checking e come questi possano essere costruiti automaticamente attraverso l'utilizzo di strategie di verifica come il paradigma CEGAR. L'attenzione verrà poi focalizzata sul tool Eureka, analizzandone obiettivi applicativi, scelte e strategie funzionali (mettendole a confronto con le scelte intraprese da altri tool di verifica) e fornendone un esempio di esecuzione per illustrare l'intuizione del funzionamento dei vari moduli che lo compongono, al fine di preparare il lettore alla comprensione delle innovazioni apportate con il presente lavoro di tesi.

1.1 Introduzione alla verifica formale

Con il passare del tempo l'uomo è sempre più dipendente da sistemi complessi (ad esempio sistemi informatici, componenti elettronici e sistemi per le telecomunicazioni) e ciò rende sempre più importante che tali sistemi siano corretti. Semplicemente notando quali e quanti sistemi vengono utilizzati quotidianamente è possibile rendersi conto quanto la vita umana sia condizionata da elementi il cui corretto funzionamento è spesso considerato ovvio. Fra questi esistono sistemi la cui correttezza è un requisito fondamentale.

Tali sistemi vengono solitamente definiti safety critical, in quanto un loro errore potrebbe comportare non solo ingenti danni economici, ma anche seri rischi per la vita stessa. Nella maggior parte dei casi la verifica di correttezza, o meglio dell'assenza di bugs, è affidata a varie fasi di test e simulazione. Tali tecniche si basano sull'utilizzo del sistema stesso (o di un suo modello) allo scopo di verificare che, in funzione di un dato input, il comportamento del sistema sia quello desiderato. Purtroppo tale approccio è soggetto all'impossibilità di verificare ogni input possibile del sistema in tempi accettabili. Tale limite comporta che se le fasi di test o simulazione permettono di verificare la presenza di un errore, esso è certo, ma se test e simulazione non trovano errori non è detto che non ce ne siano.

La verifica formale supera tale limitazione perchè fornisce risposte esatte, a patto che sia stata utilizzata correttamente. Essa non opera sul sistema concreto, bensì su un modello la cui correttezza è fondamentale per la riuscita della verifica. Un meccanismo di verifica formale controlla che per tutti i possibili comportamenti del sistema valga la proprietà che si desidera verificare. L'analisi è esaustiva sul modello formale, ed è possibile grazie al ragionamento simbolico. In che modo il sistema e la proprietà devono essere espressi dipende dal tipo di meccanismo che si usa per effettuare la verifica. Due meccanismi utilizzati per la verifica sono Theroem proving e Model checking.

Il primo prevede che il sistema sia descritto attraverso un insieme Γ_M di assiomi e regole di inferenza, mentre la proprietà ϕ attraverso un teorema. Il sistema di theroem proving cerca una dimostrazione per il teorema a partire dagli assiomi sfruttando le regole di inferenza.

Il secondo prevede che il sistema sia descritto attraverso un sistema a transizioni etichettato (LTS) per indicare come il sistema evolve nel tempo e che la proprietà da verificare sia espressa in un linguaggio accettato dal model checker. Un sistema a transizioni etichettato è una quadrupla (S, S_0, Act, R) dove:

- S è un insieme finito di stati;
- $S_0 \subseteq S$ è l'insieme degli stati iniziali;
- Act è un insieme di etichette;
- $R \subseteq S \times Act \times S$ è una relazione di transizione totale, ovvero per ogni $s \in S$ esistono $s' \in S$ e $a \in Act$ tali che $(s, a, s') \in R$.

Solitamente il linguaggio utilizzato per esprimere le proprietà è espresso in formule di logiche temporali, che permettono di descrivere proprietà legate al tempo senza doverlo rappresentare esplicitamente [CGP99]. Una volta

costruito il modello M e fornita la proprietà ϕ che si desidera verificare, il model checker esegue una ricerca esaustiva sullo spazio degli stati del modello al fine di stabilire se la proprietà è soddisfatta in ogni suo stato. Questo equivale a dire che il comportamento del modello è tale per cui, data una sua valutazione, la proprietà ϕ è sempre valida, ovvero $M \models \phi$. Tale ricerca risulta particolarmene onerosa data la dimensione del modello. Infatti i model checker sono soggetti al problema dell'esplosione degli stati [CGJ+00b], ovvero aumentando il numero delle componenti del sistema cresce anche il numero degli stati del modello. Tale crescita, in alcuni casi, è esponenziale. Questo problema limita l'applicazione delle tecniche di model checking a sistemi relativamente piccoli.

1.2 Model checking simbolico

Per ridurre la quantità di memoria necessaria a immagazinare tutti gli stati presi in considerazione si è pensato di passare da una rappresentazione esplicita degli stati ad una simbolica.

Grazie a McMillan [BCM+90] è stata sfruttata nel model checking una nuova rappresentazione (simbolica) definita da Bryant basata sui diagrammi binari di decisione (BDD) [Bry86]. In tale rappresentazione ogni stato è codificato mediante un assegnamento di valori booleani all'insieme di variabili di stato associate al sistema. Introducendo su queste strutture un ordinamento sulle variabili ed applicando le regole di riduzione si ottengono i *Reduced Ordered Binary Decision Diagram* ROBDD[1]. E' possibile, inoltre, esprimere anche la relazione di transizione attraverso formule booleane in funzione di due insiemi di variabili, uno che codifica lo stato di partenza della transizione e uno che codifica lo stato di arrivo della transizione. Tale formula è poi rappresentata in un diagramma binario di decisione (ROBDD). Grazie a questo tipo di rappresentazione si è riusciti ad effettuare verifica su sistemi composti da più di 10^{20} stati [CGP99] e si sono ottenuti ottimi risultati nella verifica di circuiti elettronici isolati di medie dimensioni e sistemi il cui numero di stati non è eccessivamente alto.

La rappresentazione degli stati tramite ROBDD non è l'unica possibile. Approcci differenti prevedono l'utilizzo di sistemi di vincoli sulle variabili. Un sistema di vincoli ci permette di definire, per ogni variabile, uno o più sottoinsiemi del dominio entro il quale essa può assumere valore. Tale approccio ci permette di esprimere in modo molto succinto anche insiemi infiniti

[1] Gli ROBDD sono a tutti gli effetti una rappresentazione canonica di formule booleane, del tutto equivalente ad una tabella di verità o una formula proposizionale

e, per questo motivo, è particolarmente conveniente quando le variabili del modello sono definite su un dominio numerico infinito.

1.3 Tecniche di astrazione per il software

Un altro problema da affrontare quando si tenta di eseguire verifica automatica di software mediante model checking deriva dall'impossibilità di eseguire tale tecnica su programmi con array. E' necessario, quindi, mappare un programma da tale dominio ad un altro più semplice, per il quale sia possibile effettuare verifica tramite una procedura di model checking. Tale mapping viene effettuato in fase di astrazione, durante la quale il programma concreto, definito all'interno del dominio dei programmi lineari con array, viene trasformato in un programma astratto, appartenente, nel caso di Eureka, al dominio dei programmi lineari.

Tale corrispondenza deve, però, garantire una certa relazione tra programma astratto e concreto in merito alle proprietà da verificare, in particolare sarebbe auspicabile che una proprietà fosse valida nel programma astratto se e solo se lo è nel programma concreto, in modo da poter estendere anche al programma concreto l'esito della verifica di una determinata proprietà sul programma astratto.

Nei programmi lineari con array, ed in particolare in Eureka, l'astrazione utilizzata è conservativa ed in quanto tale permette di affermare solo un verso dell'implicazione e cioè che, se una proprietà risulta valida nel programma astratto, allora lo è anche in quello concreto; non vale però il contrario, infatti una proprietà che risulta violata da una traccia di esecuzione del programma astratto, potrebbe comunque essere valida nel programma concreto, dove tale traccia di esecuzione non è fattibile. Tali tracce di esecuzione, ammissibili nel modello astratto ma non nel programma concreto, sono definite **spurie**[2].

Trovare il giusto livello di dettaglio per la costruzione del modello astratto è alla base della riuscita della verifica. Sono stati introdotti vari meccanismi per la costruzione automatica del modello che cercano di trovare la giusta astrazione attraverso raffinamenti sempre più dettagliati di un modello di partenza. Clarke descrive una procedura di Astrazione-Raffinamento guidato da controesempi (CEGAR, Counter Example Guided Abstraction Refinement) grazie al quale è possibile giungere al giusto livello di astrazione [CGJ+00a]. Considerato un modello K e una proprietà p da verificare, tale procedura prevede un ciclo di astrazione-verifica-raffinamento, illustrato in figura 1.1, che può essere schematizzato nelle seguenti fasi:

[2]Per maggiori chiarimenti si rimanda al capitolo 2

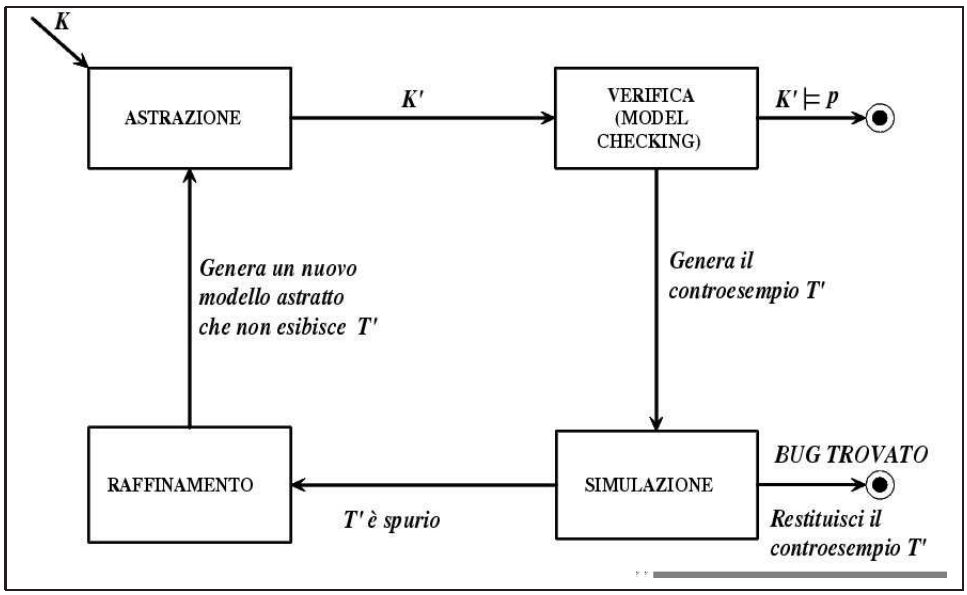


Figura 1.1: Ciclo CEGAR

1. Astrazione: il modello concreto K viene mappato nel suo modello astratto K'

2. Verifica: viene effettuata la verifica, tramite model checking, della proprietà p sul modello astratto K'. Se la proprietà risulta valida nel modello astratto ($K' \models p$), allora lo è anche nel modello concreto K, essendo l'astrazione conservativa, pertanto il ciclo termina affermando che la proprietà è verificata dal modello concreto ($K \models p$). Se, invece, dalla verifica risulta che $K' \not\models p$, allora il model checker fornisce un controesempio T' (traccia del modello astratto) che falsifica la proprietà

3. Simulazione: la traccia T' viene analizzata, per verificarne la fattibilità in K, tramite un theorem proving. In altre parole si verifica che T' appartenga anche all'insieme di tracce ammissibili del programma concreto. Se T' è una traccia fattibile del programma concreto, allora il ciclo termina fornendo la traccia di esecuzione T' che falsifica p; se invece T' è una traccia spuria, allora va eliminata dal modello astratto

4. Raffinamento: si occupa proprio di individuare gli elementi che erano stati ignorati nella precedente astrazione e che, introdotti nel nuovo modello astratto, garantiscono che la traccia spuria T' non faccia più parte di esso

5. Ritorna al passo 1

Il ciclo descritto da Clarke è l'approccio scelto per la realizzazione di molti tool di verifica basati su model checking, fra i quali si cita il progetto SLAM di Microsoft utilizzato per realizzare SDV (Static Driver Verifier), tool per la verifica dei driver per Windows.

Tuttavia, il paradigma CEGAR non è l'unico approccio possibile. Henzinger, Jhala, Majumdar e Sutre suggeriscono una variante a tale ciclo. L'idea introdotta prevede il *raffinamento a richiesta*, ovvero si raffinano solo quelle regioni di codice in cui è necessaria un'astrazione più vicina al modello concreto. Grazie al controesempio si determina quale regione di codice deve essere raffinata e, a partire da questa, si crea un nuovo livello di astrazione, relativo alla regione in questione, tale da rendere il controesempio non più fattibile. A questo punto la verifica può ripartire proprio dalla regione di codice appena raffinata [HJMS02]. Tale approccio prende il nome di *Lazy Abstraction* e ha il vantaggio di avere regioni di codice astratte con granularità differenti e di evitare di eseguire più volte la verifica su regioni di codice il cui livello di astrazione non cambia. Tale tecnica però risulta particolarmente onerosa per la verifica di sistemi di grandi dimensione. Il tool di verifica realizzato a Berkeley che sfrutta tale tecnica prende il nome di BLAST (Berkeley Lazy Abstraction Software verification Tool) [HJMS03].

I moduli di model checking di SLAM e BLAST calcolano la raggiungibilità di alcuni stati adeguatamente etichettati. Infatti, il problema di stabilire se un programma sequenziale P gode di una qualche proprietà ϕ può essere ridotto ad un problema di violazione di un'affermazione (*assertion failure*) in una versione propriamente modificata del programma di partenza. Poichè ci si aspetta che ogni proprietà ϕ valga in ogni stato del modello, ovvero che ϕ sia un'invariante del programma, la sua negata non deve essere mai possibile. Aggiungendo quindi in P, per ogni proprietà ϕ, del codice extra che può essere raggiunto solo se $\neg\phi$ è verificata, si riduce il problema della verifica di ϕ al calcolo della raggiungibilità del codice che esprime $\neg\phi$ [BR00a].

1.4 Domini di astrazione

La fase di astrazione consiste dunque nella traduzione di un programma da un dominio concreto ad un dominio astratto più semplice e su cui è possibile eseguire la procedura di verifica.

Nell'ambito del software model checking scelta molto diffusa è utilizzare il dominio dei programmi booleani come dominio di astrazione, infatti questo dominio si presta particolarmente alla verifica formale di software per due

motivi: primo, il numero di stati di un programma booleano è finito, quindi il problema della raggiungibilità è decidibile; secondo, i programmi booleani hanno gli stessi costrutti di controllo di un linguaggio di programmazione procedurale.

Un esempio di applicazione del paradigma CEGAR per la verifica di software su un dominio astratto costituito dai programmi booleani è il progetto SLAM di Microsoft [BMMR01]. Infatti esso prevede che i modelli astratti siano rappresentati con programmi booleani e la rappresentazione simbolica di stati e transizioni avviene tramite ROBDD.

E' evidente, comunque, che il costo della verifica è direttamente proporzionale al numero di iterazioni del ciclo e, proprio per ridurre tale numero, si è deciso di utilizzare all'interno di Eureka un dominio astratto, quello dei programmi lineari, il più vicino possibile a quello concreto dei programmi lineari con array, differentemente da quanto fatto da SLAM della Microsoft, in modo da avere astrazioni con un minor numero di tracce spurie e conseguentemente un minor numero di iterazioni Astrazione-Raffinamento.

1.5 Un esempio di esecuzione di Eureka

Il progetto Eureka [ABM06] nasce con l'obiettivo di realizzare un tool per la verifica formale di software. Specificatamente, il tool è in grado di verificare il frammento di linguaggio C corrispondente al dominio dei programmi lineari con array (in realtà la classe di programmi verificati da Eureka è rappresentata da un'estensione di tale dominio, il dominio dei programmi lineari con array generalizzati, introdotto in sezione 2.1).

Per descrivere in maniera un po' più dettagliata il comportamento delle varie fasi del ciclo, viene presentato di seguito un esempio di esecuzione di Eureka sul programma di tabella 1.1

La prima iterazione comincia con la costruzione del modello $\widehat{P_0}$, mostrato in tabella 1.2, il cui livello di astrazione è massimo. Costruire un modello astratto a partire da uno concreto significa, in pratica, semplificare il modello concreto ignorando parte dell'informazione in esso contenuta. In Eureka l'informazione che viene astratta è quella riguardante il valore di alcune variabili ed array del programma.

Infatti, alcune variabili vengono eliminate dal modello astratto e sostituite dal simbolo `u` (indefinito) in tutte le espressioni in cui compaiono e gli assegnamenti ad esse vengono sostituiti dall'istruzione vuota (`skip`).

Ma più interessante è vedere cosa accade con gli array del programma, visto che scopo primario dell'astrazione è costruire un modello che non preveda la loro presenza, problematica per il model checker.

P

```
    void main() {
      int i, a[3];
[1]   a[1] = 1;
[2]   i = 0;
[3]   while ( (a[i] != 1) && (i < 3) ) {
[4]     a[i] = 2*i;
[5]     i = i+1;
      }
[6]   assert (i <= 1);
    }
```

Tabella 1.1: Un semplice programma P su cui viene mostrato un esempio di esecuzione di Eureka

In fase di astrazione, gli array non vengono più considerati nella loro interezza, ma vengono tradotti nel modello astratto con una variabile scalare per ogni elemento. Così, un generico array a di dimensione 5, verrà tradotto, nel modello astratto, dall'insieme di variabili scalari $\{a_0, a_1, a_2, a_3, a_4\}$. Ancora una volta, non tutte le variabili corrispondenti ad elementi dell'array faranno parte del modello astratto, ma solo un loro sottoinsieme, individuato dall'insieme $R(a) \subseteq \{0, 1, \ldots, dim(a) - 1\}$ di indici dell'array a tale che se $i \in R(a)$ allora a_i sarà una variabile del modello astratto per ogni $i \in \{0, 1, \ldots, dim(a) - 1\}$.

Quindi, ogni accesso a`[`i`]` ad array presente in un'espressione del programma concreto, viene sostituito, in quello astratto, dall'istruzione condizionale

$$(i{==}k_1)\,?\,a_{k_1} : (i{==}k_2)\,?\,a_{k_2} : \ldots : (i{==}k_n)\,?\,a_{k_n} : \mathtt{u}$$

con $R(a) = \{k_1, \ldots, k_n\}$. Intuitivamente, se il valore dell'espressione i che indicizza l'accesso ad array è uguale ad uno degli indici numerali presenti in $R(a)$, e cioè se vale $i = k_j$ per qualche $k_j \in R(a)$, allora l'espressione a`[`i`]` viene sostituita dalla variabile astratta a_{k_j}, corrispondente proprio all'elemento k_j-esimo dell'array a; se invece l'espressione i non è uguale a nessuno dei numerali in $R(a)$ ($i \neq k_j$ per ogni $k_j \in R(a)$), l'espressione a`[`i`]` è sostituita dal simbolo `u`.

Invece, ogni assegnamento ad array a`[`i`]=`e`;` viene sostituito dall'assegnamento parallelo

$$a_{k_1}{=}(i{==}k_1\,?\,e : a_{k_1}), a_{k_2}{=}(i{==}k_2\,?\,e : a_{k_2}), \ldots, a_{k_n}{=}\ (i{==}k_n\,?\,e : a_{k_n})\,;$$

con $R(a) = \{k_1, \ldots, k_n\}$. Intuitivamente, viene modificata solo la variabile

che cattura il comportamento della locazione dell'array identificata dall'espressione i. Il valore delle variabili astratte che modellano altre locazioni dell'array non viene modificato.

Quindi, un modello astratto viene generato a partire da un sottoinsieme V' dell'insieme V di variabili scalari del programma e da una classe di insiemi di indici $I = \{R(a) \mid a \in A\}$, dove A rappresenta l'insieme degli array del programma.

Tuttavia, per semplicità di trattazione, nel corso del presente elaborato, i modelli astratti saranno generati sempre rispetto all'intero insieme di variabili scalari; l'unica informazione astratta dal modello astratto è quella riguardante gli array del programma, si dirà quindi che un modello è stato generato rispetto ad una classe di insiemi di indici I, sottointendendo che l'astrazione avviene anche rispetto all'intero insieme V di variabili di programma.

Ritornando all'esempio di esecuzione, l'astrazione della prima iterazione avviene, come già detto, con il massimo livello di astrazione, cioè tutta l'informazione del programma astratto (si ricorda che ci si riferisce sempre all'informazione relativa a variabili array) è ignorata. Il programma astratto viene costruito rispetto alla classe di insiemi vuoti di indici $I = \{R(a) \mid a \in A_P\}$, dove $R(a) = \emptyset$ per ogni $a \in A_P$

$\widehat{P_0}$

```
    void main() {
      int i;
[1]   ;
[2]   i = 0;
[3]   while ( (u != 1) && (i < 3) ) {
[4]     ;
[5]     i = i+1;
      }
[6]   assert (i <= 1);
    }
```

Tabella 1.2: Astrazione iniziale $\widehat{P_0}$ di P

Su tale modello viene eseguita la procedura di model checking, illustrata nel capitolo 3, che rivela la presenza di una traccia τ che porta al fallimento dell'istruzione `assert` (*assert failure*).

La traccia in questione viene sottoposta al test di fattibilità nel programma concreto P, attraverso la fase di simulazione, che consiste nella gene-

razione di un insieme Γ di formule, che rappresentano la traduzione degli statement della traccia, e la successiva verifica della sua soddisfacibilità. Infatti, poichè tale insieme di formule viene costruito in modo tale da rispettare la semantica degli statement, accade che per ogni possibile esecuzione, nel programma concreto, della traccia in esame, esiste un modello (valutazione delle variabili che soddisfa l'insieme di formule Γ) corrispondente e quindi l'eventuale insoddisfacibilità di Γ ($\Gamma \vdash \bot$) implica che la traccia non è fattibile nel programma concreto: è una traccia spuria.

Durante la fase di simulazione, la traccia τ, che consiste in una successione di interi, ognuno dei quali individua un'istruzione del programma, viene innanzitutto convertita in una successione di statement. In questa fase, ogni intero viene tradotto con l'istruzione corrispondente all'interno del programma, eccetto i casi di istruzioni condizionali del tipo `if(`e`)` o `assert(`e`);` oppure i casi di istruzioni iterative del tipo `while(`e`)`. In tali casi si usa, per la traduzione, uno statement di tipo `assume(`e'`);`, la cui guardia e' corrisponde alla guardia e dello statement condizionale o iterativo della traccia oppure alla sua negazione `!`e, a seconda che l'esecuzione prosegua seguendo il ramo vero o il ramo falso dello statement stesso.

A titolo di esempio, si consideri la tabella 1.3, in cui compaiono, in corrispondenza di ogni passo di esecuzione della traccia (colonna a sinistra), la riga di codice relativa all'istruzione eseguita (seconda colonna) e lo statement originale, ottenuto come appena descritto (terza colonna). Per quanto riguarda i primi due passi di esecuzione della traccia, si utilizzano i rispettivi statement del programma concreto. Il terzo passo di esecuzione coinvolge un'istruzione iterativa (`while`) e viene, quindi, tradotto con uno statement di tipo `assume`. Poichè l'esecuzione continua seguendo il ramo $true$, allora la guardia dell'`assume` è uguale a quella del `while`. Quando, però, si esce dal ciclo, ovvero l'esecuzione prosegue lungo il ramo $false$, come accade nel dodicesimo passo di esecuzione, l'istruzione `while` viene sostituita dall'istruzione `assume` con la guardia negata.

Le ragioni dell'utilizzo dello statement `assume` al posto di un condizionale o iterativo vanno ricercate nella necessità di eliminare le ramificazioni, presenti nel programma proprio a causa di tali costrutti, ed ottenere quindi un insieme di statement lineari (non ramificati). Infatti non ha piu senso mantenere uno statement di tipo `if`, considerato che la "scelta" del ramo da seguire è già avvenuta. Con uno statement di tipo `assume(`e`);` si intende appunto asserire la validità dell'espressione e in quel determinato punto del programma.

Una volta ottenuto l'insieme di statement corrispondenti alla traccia, il processo di traduzione di quest'ultima in un insieme di formule passa per la previa operazione di rinomina delle variabili che occorrono negli statement

della traccia, in modo tale che nessuna di esse venga assegnata due volte durante l'esecuzione. Tale operazione consiste nell'aggiungere un pedice ad ogni variabile (inizialmente 0) e nell'incrementarlo ogni volta che la variabile compare come membro sinistro di uno statement di assegnamento nella traccia. Intuitivamente, con tale meccanismo si cattura l'evoluzione temporale del valore di una variabile lungo la traccia di esecuzione. Infatti, in fase di simulazione, la stessa variabile con pedice diverso rappresenta la stessa variabile in due istanti (stati) differenti.

	τ	Statement originali	Statement rinominati	Formule
1	[1]	`a[1]=1;`	a_1`[1]=1;`	$a_1 = store(a_0, 1, 1)$
2	[2]	`i=0;`	i_1`=0;`	$i_1 = 0$
3	[3]	`assume(a[i]!=1&&i<3);`	`assume(`a_1`[`i_1`]!=1&&`i_1`<3);`	$select(a_1, i_1) \neq 1 \wedge i_1 < 3$
4	[4]	`a[i]=2*i;`	a_2`[`i_1`]=2*`i_1`;`	$a_2 = store(a_1, i_1, 2 * i_1)$
5	[5]	`i=i+1;`	i_2`=`i_1`+1;`	$i_2 = i_1 + 1$
6	[3]	`assume(a[i]!=1&&i<3);`	`assume(`a_2`[`i_2`]!=1&&(`i_2`<3);`	$select(a_2, i_2) \neq 1 \wedge i_2 < 3$
7	[4]	`a[i]=2*i;`	a_3`[`i_2`]=2*`i_2`;`	$a_3 = store(a_2, i_2, 2 * i_2)$
8	[5]	`i=i+1;`	i_3`=`i_2`+1;`	$i_3 = i_2 + 1$
9	[3]	`assume(a[i]!=1&&i<3);`	`assume(`a_3`[`i_3`]!=1&&(`i_3`<3);`	$select(a_3, i_3) \neq 1 \wedge i_3 < 3$
10	[4]	`a[i]=2*i;`	a_4`[`i_3`]=2*`i_3`;`	$a_4 = store(a_3, i_3, 2 * i_3)$
11	[5]	`i=i+1;`	i_4`=`i_3`+1;`	$i_4 = i_3 + 1$
12	[3]	`assume(!(a[i]!=1&&i<3));`	`assume(!(`a_4`[`i_4`]!=1&&`i_4`<3));`	$\neg(select(a_4, i_4) \neq 1 \wedge i_4 < 3)$
13	[6]	`assume(i>1);`	`assume(`i_4`>1);`	$i_4 > 1$

Tabella 1.3: Processo di traduzione della traccia τ in un insieme Γ di formule

Dopo la fase di rinomina, ogni statement rinominato viene effettivamente tradotto in una formula logica che ne codifica il comportamento, come mostrato in tabella 1.3, in cui compare l'intero processo di traduzione della traccia in formule.

Ad esempio, si consideri il terzo passo di esecuzione, cui corrisponde lo statement rinominato `assume(`a_1`[`i_1`]!=1 &&` i_1`<3);`. Si tratta di un `assume` la cui guardia è costituita dalla congiunzione di due formule booleane, una delle quali coinvolge un accesso ad array. L'accesso ad array è tradotto con la formula $select(a_1, i_1)$ appartenente alla teoria degli array. Le espressioni booleane vengono tradotte tramite la banale traduzione degli operatori del programma con i rispettivi operatori booleani o relazionali; in questo caso `&&` viene sostituito con l'operatore booleano $\wedge$, l'operatore `!=` con $\neq$ e l'operatore `<` resta invariato. Infine, poichè la traduzione di uno statement di tipo `assume(`e`);`, corrisponde alla traduzione dell'espressione e che

esso asserisce, la formula risultante dalla traduzione dell'intero statement è $(select(a_1, i_1) \neq 1) \wedge (i_1 < 3)$.

Da notare, infine, come gli assegnamenti ad array vengano tradotti in maniera diversa rispetto agli accessi ad array all'interno di espressioni. In tali casi si utilizza un'altra formula della teoria degli insiemi, la formula $store(a, i, e)$, il cui significato è quello di uguagliare l'elemento dell'array a indicizzato dall'espressione i con il valore dell'espressione e; la funzione restituisce l'array a. Pertanto, facendo sempre riferimento alla tabella 1.3, in corrispondenza del primo passo di esecuzione della traccia, lo statement a_1`[1]=1;` viene tradotto con la formula $a_1 = store(a_0, 1, 1)$.

L'insieme Γ di formule così generato viene dato in input ad un theorem prover che ne verifica la soddisfacibilità e quindi stabilisce la fattibilità o meno della traccia nel programma concreto.

Nell'esempio in questione, Γ è insoddisfacibile, infatti le formule riportate di seguito (precedute dai rispettivi passi di esecuzione della traccia) sono inconsistenti:

- $1 \rightarrow a_1 = store(a_0, 1, 1)$
- $2 \rightarrow i_1 = 0$
- $4 \rightarrow a_2 = store(a_1, i_1, 2 * i_1)$
- $5 \rightarrow i_2 = i_1 + 1$
- $6 \rightarrow select(a_2, i_2) \neq 1 \wedge i_2 < 3$

Pertanto si può concludere che la traccia è spuria e bisogna raffinare il modello.

A tale scopo, si utilizza la prova Π di insoddisfacibilità di Γ per stabilire le "cause" dell'insoddisfacibilità, individuare, cioè, quegli elementi di array tali che l'astrazione di P rispetto ad essi non annovera la traccia τ nell'insieme delle sue tracce ammissibili.

Tra le suddette formule (1, 2, 4, 5 e 6), da cui dipende la prova di insoddisfacibilità, compare un solo accesso ad array (formula di tipo *select*), che individua proprio l'indice i_2, e l'array a, su cui raffinare. Come verrà descritto più dettagliatamente nel capitolo 4, si cerca di ridurre l'indice ad un numerale, utilizzando le formule dell'insieme Γ; in questo caso è immediato vedere che i_2 è uguale al numerale 1, utilizzando le uguaglianze dei passi di esecuzione 2 e 5. Il processo di raffinamento restituisce pertanto l'indice 1 associato all'array a.

Comincia, a questo punto, una nuova iterazione del ciclo CEGAR, con la costruzione del nuovo modello astratto $\widehat{P_1}$, mostrato in tabella 1.4, generato astraendo rispetto all'insieme di indici restituito dal refiner $\{R(a)\}$, con $R(a) = \{1\}$ e con a unica variabile array del programma.

$\widehat{P_1}$

```
    void main() {
      int i,a1;
[1]   a1 = (1==1)?1:a1;
[2]   i = 0;
[3]   while ( (((i==1)?a1:u) != 1) && (i < 3) ) {
[4]     a1 = (i==1)?2*i:a1;
[5]     i = i+1;
      }
[6]   assert(i <= 1);
    }
```

Tabella 1.4: Raffinamento $\widehat{P_1}$ di $\widehat{P_0}$

La fase di verifica sul modello astratto $\widehat{P_1}$ rivela che nessuna traccia di esecuzione porta al raggiungimento dello stato di errore, pertanto, poichè l'astrazione è conservativa, lo stato di errore non è raggiungibile neanche nel programma concreto P.

Capitolo 2

Astrazione

Poichè non è esiste attualmente una procedura in grado di eseguire model checking su di un programma con array, per verificare tale classe di programmi sono state sviluppate delle strategie che prevedono una trasformazione sintattica o semantica del programma in modo da eliminare ogni riferimento a tali strutture dati. Questa trasformazione è eseguita in fase di astrazione e nel presente capitolo verrà introdotto in generale il concetto di astrazione, prima di presentare la strategia di astrazione preesistente in Eureka. Verrà illustrato infine il nuovo meccanismo di astrazione proposto nel presente lavoro di tesi, con la relativa dimostrazione di correttezza.

2.1 Programmi lineari con array generalizzati

La fase di astrazione risulta dunque indispensabile quando si voglia eseguire verifica formale di programmi con array. Verrà presentato nel corso di tale sezione il dominio dei programmi lineari con array e un'estensione di tale dominio, rappresentata dai programmi lineari con array generalizzati, che è la classe di programmi che il tool Eureka è in grado di verificare. In particolare verranno presentate sintassi e semantica di tali classi di programmi.

2.1.1 Espressioni lineari generalizzate con array ed espressioni lineari booleane generalizzate con array

Prima di introdurre la sintassi di un programma lineare con array generalizzato si introducono le definizioni di espressioni lineari generalizzate con array

ed espressioni lineari booleane generalizzate con array.

Sia $\mathcal{D}$ un dominio numerico (come ad esempio l'insieme dei naturali o degli interi), sia W un insieme di variabili di programma e siano $V \subseteq W$ un insieme di variabili scalari definite su $\mathcal{D}$ e $A \subseteq W$ un insieme di variabili array, tali che V e A formano una partizione dell'insieme W, in simboli $W = V \cup A$ e $V \cap A = \emptyset$, e sia `u` un nuovo simbolo non appartenente a W, con cui si indica un valore indefinito, cioè un qualsiasi elemento del dominio $\mathcal{D}$ scelto in maniera non deterministica, l'insieme E_W di espressioni lineari generalizzate con array sull'insieme di variabili W è induttivamente definito come segue:

1. `u` $\in E_W$
2. $c \in E_W$ per ogni $c \in \mathcal{D}$
3. $v \in E_W$ per ogni $v \in V$
4. c`*`$e \in E_W$ per ogni $c \in \mathcal{D}$ e per ogni $e \in E_W$
5. e_1`+`$e_2 \in E_W$ per ogni $e_1, e_2 \in E_W$
6. b`?`e_1`:`$e_2 \in E_W$ per ogni $e_1, e_2 \in E_W$ e per ogni espressione lineare booleana generalizzata con array b
7. a`[`e`]` $\in E_W$ per ogni $a \in A$ e per ogni $e \in E_W$

L'insieme di espressioni lineari generalizzate è definito induttivamente dalle regole 1-6 della definizione precedente, mentre gli insiemi di espressioni lineari con array ed espressioni lineari sono definiti dalle regole $2, 3, 4, 5, 7$ e dalle regole $2, 3, 4, 5$, rispettivamente. Si definisce infine l'insieme $E_{\Theta,W} \subseteq E_W$, definito induttivamente dalle regole 2-7. Da notare come all'interno delle espressioni in $E_{\Theta,W}$ non compaia il simbolo di indefinito `u`.

Un'espressione b è detta lineare booleana generalizzata con array sull'insieme di variabili W se è della forma e_1 *op* e_2 con *op* $\in \{$`<,<=,>,>=,==,!=`$\}$ e $e_1, e_2 \in E_W$

Indichiamo con B_W l'insieme di tutte le espressioni lineari booleane generalizzate con array sull'insieme di variabili W.

Siano $b_1, b_2 \in B_W$, si assume nel corso del seguente elaborato che ogni occorrenza di `!`b_1, b_1`&&`b_2 e b_1`||`b_2 in un programma sia sostituita dalle espressioni equivalenti `(`b_1`?0:1)`, `(`b_1`?`b_2`:0)` e `(`b_1`?1:`b_2`)`, rispettivamente. Inoltre ogni espressione lineare booleana generalizzata con array b che compare fuori della guardia di un espressione o di uno statement condizionale è sostituita dall'espressione equivalente `(`b`?1:0)`.

Le definizioni di espressioni lineari booleane generalizzate, espressioni booleane con array e esprssioni booleane sono analoghe alla precedente, con

la differenza che gli operandi e_1 ed e_2 sono espressioni lineari generalizzate, espressioni lineari con array ed espressioni lineari rispettivamente.

2.1.2 Sintassi dei programmi lineari con array generalizzati

La struttura di un programma P lineare con array generalizzato [ABM05a] [ABM05b] consiste in dichiarazioni di variabili globali e definizioni di procedure. Una procedura è a sua volta composta da dichiarazioni di variabili locali e una successione di statement, ciascuno dei quali può essere:

- assegnamento a variabile (semplice o parallelo);
- assegnamento ad array (semplice);
- statement condizionale (`if(`e`)`, `assert(`e`);` e `assume(`e`);`);
- statement iterativo (`while(`e`)`);
- statement nullo (`skip`);
- chiamata a procedura;
- ritorno da procedura (`return;`).

Senza allontanarsi troppo dai comuni programmi imperativi, si assume che per ogni programma P esista una procedura chiamata `main`, da cui parte l'esecuzione del programma.

Le variabili sono definite su un dominio numerico $\mathcal{D}$, le espressioni appartengono all'insieme E_W di espressioni lineari con array generalizzate e le guardie degli statement condizionali o iterativi sono delle espressioni lineari booleane con array generalizzate.

Un programma P si dice lineare con array se le espressioni che lo compongono sono espressioni lineari con array o espressioni lineari booleane con array.

Nel corso del presente elaborato l'aggettivo "generalizzate" verrà omesso, pertanto le espressioni lineari (booleane) con array generalizzate verranno indicate più semplicemente come espressioni lineari (booleani) con array. Anche quando si parla di programmi lineari con array, ci si riferisce ai programmi lineari con array generalizzati.

Viene fornita, in tabella 2.1, la sintassi dettagliata di un programma lineare con array, secondo la notazione BNF.

prog	::=	*decl* proc**	Dichiarazione variabili globali,
		`int main(){`*decl* sseq*`}`	main e altre procedure
decl	::=	*type id* [`,` *id*]*`;`	Dichiarazione di variabili
type	::=	`int` | `float`	Tipo delle variabili
id	::=	[a-zA-Z_][a-zA-Z0-9_]*	Identificatore
proc	::=	*type id*`(`*par*`){`*decl* sseq*`}`	Definizione di procedura
par	::=	ϵ | *type id* [`,` *type id*]*	Lista dei parametri di definizione di procedura
sseq	::=	*lstmt**	Sequenza di istruzioni
lstmt	::=	*stmt*	Istruzione senza etichetta
		| *id* `:` *stmt*	Istruzione con etichetta
stmt	::=	`;`	Skip
		| *id*`=`*expr* [`,` *id*`=`*expr*]*`;`	Assegnamento a variabile
		| *id*`[`*expr*`]=`*expr*`;`	Assegnamento ad array
		| *cond*	Statement Condizionale
		| `while(`*pred*`){`*sseq*`}`	Statement iterativo
		| *id*`(`*parCh*`);`	Chiamata a procedura
		| `return;`	Ritorno da procedura
cond	::=	`if(`*pred*`){`*sseq*`}`	If senza else
		| `if(`*pred*`){`*sseq*`}else{`*sseq*`}`	If con else
		| `assert(`*pred*`);`	Asserzione
		| `assume(`*pred*`);`	Assunzione
parCh	::=	ϵ | *id* [`,` *id*]*	Lista dei parametri di chiamata a procedura
expr	::=	`u`	Simbolo di indefinito
		| *mon*	Monomio
		| *const*`*`*expr*	Costante per espressione
		| *expr*`+`*expr*	Espressione lineare
		| *pred*`?`*expr*`:`*expr*	Espressione condizionale
		| *id*`[`*expr*`]`	Accesso ad array
mon	::=	*umon* | `-`*umon*	Monomio con e senza segno
umon	::=	*const*`*`*id*	Monomio senza segno
		| *const*	Costante numerica
		| *id*	Variabile
const	::=	[0-9][0-9]*	Costante numerica
pred	::=	*expr op expr*	Predicato semplice
		| `!`*pred*	Negazione
		| *pred*`&&`*pred*	Congiunzione
		| *pred*`||`*pred*	Disgiunzione
op	::=	`<` | `<=` | `>` | `>=` | `==` | `!=`	Operatore Relazionale

Tabella 2.1: Sintassi dei programmi lineari con array

Dalla tabella emerge anche che la sintassi descritta è abbastanza simile a quella di un qualsiasi linguaggio imperativo, eccetto che per la possibilità di utilizzare il simbolo di indefinito `u`. La necessità di questa estensione del linguaggio va ricercata nell'utilità di introdure il non determinismo per modellare interazioni esterne. Infatti, per poter eseguire verifica di programmi che richiedono interazione da parte dell'utente, bisogna modellare le immissioni esterne con il simbolo di indefinito in modo tale da trattare non deterministicamente tutti i possibili casi.

2.1.3 Rappresentazione tramite CFG

Un programma lineare con array è, in altre parole, una stringa di caratteri formata secondo regole sintattiche. Pertanto, appare evidente che, per essere manipolato in maniera efficiente da un sistema automatico che ne verifichi la correttezza, la stringa di codice che lo rappresenta debba essere convertita in un'opportuna struttura dati, più adatta alla manipolazione.

Si utilizza a tale scopo una particolare struttura dati, chiamata CFG (Control Flow Graph) [ASU86], che, associata ad un programma P, ne codifica gli statement e il flusso di controllo.

Il CFG di un programma P è un grafo diretto $G_P = (N_P, Succ_P)$, dove

- $N_P = \{0, 1, \ldots, n, n+1, n+2, \ldots, n+p\}$ rappresenta l'insieme dei nodi in cui:
  - n rappresenta il numero di statement di P
  - p rappresenta il numero di procedure di P
  - il nodo i corrisponde all'i-esimo statement per ogni $1 \leq i \leq n$
  - il nodo $j+n$ corrisponde all'exit node della j-esima procedura per ogni $1 \leq j \leq p$
  - il nodo 0 modella il fallimento degli statement di tipo `assert(`e`);`
- $Succ_P : N_P \rightarrow 2^{N_P}$ è la funzione di successore che ad ogni nodo i associa l'insieme dei suoi nodi successori $Succ_P(i)$.

Ogni istruzione del programma è, quindi, codificata in un nodo di tale grafo, in cui compaiono anche un nodo extra per ogni procedura (exit node), cui si giunge al termine dell'esecuzione della procedura cui il nodo si riferisce.

Per ogni nodo i tale che $1 \leq i \leq n$, si indica con stm_i lo statement che corrisponde all'i-esima istruzione di P.

Se stm_i è `if(`e`)`, `while(`e`)`, `assert(`e`);` o `assume(`e`);`, allora $Succ_P(i) = \{Tsucc_P(i), Fsucc_P(i)\}$, dove $Tsucc_P(i)$ ($Fsucc_P(i)$) denota il successore di

i quando e viene valutata vera (falsa, rispettivamente). In particolare, se lo statement stm_i è `assert(`e`);`, allora $Fsucc_P(i) = 0$, ovvero il successore nel caso in cui la guardia e è falsa è il nodo 0 di errore. Invece, gli statement di tipo `assume(`e`);` sono tali per cui, se e viene valutata vera, allora l'esecuzione del programma continua, altrimenti il programma termina senza errori, pertanto, se stm_i è `assume(`e`);`, allora $Fucc_P(i)$ rappresenta l'exit node del main.

Se pr è una procedura di P, allora $First_P(pr)$ è il vertice corrispondente al primo statement della procedura pr e $RetPt_P(i)$ è il vertice corrispondente all'istruzione `skip` immediatamente successivo alla chiamata della procedura[1].

Se stm_i corrisponde ad una chiamata alla procedura pr, allora $Succ_P(i) = \{First_P(pr)\}$; sia invece stm_j uno statement che occorre all'interno di pr, allora $ProcOf_P(j) = pr$.

Inoltre, sia stm_i un `return;`, allora $Succ_P(i) = \{Exit_P(pr)\}$, dove $Exit_P(pr)$ indica l'exit node associato alla procedura pr e $Succ_P(Exit_P(pr)) = \{j \in N_P \mid stm_k = pr\texttt{(e);}$ e $j = RetPt_P(k)\}$.

Infine, se $Succ_P(i) = \{i_1\}$, cioè $Succ_P(i)$ ha cardinalità 1, allora $sSucc_P(i) = i_1$, ossia $sSucc_P(i)$ individua proprio l'unico nodo successore di i nel CFG di P.

2.1.4 Scope delle variabili

Come abbiamo visto, all'interno di un programma è possibile definire più procedure ed all'interno di ognuna di esse è possibile dichiarare un insieme di variabili locali alla procedura, visibili ed utilizzabili solo da parte di istruzioni della procedura stessa. Esiste, inoltre, la possibilità di dichiare variabili globali, la cui visibilità è estesa a tutti gli statement del programma.

Tali informazioni, contenute nella stringa del programma, non vengono codificate all'interno del control flow diagram. Per tenere traccia degli scope delle variabili del programma si utilizzano dei vettori di stringhe.

In particolare, sia $i \in N_P$, si definiscono i seguenti insiemi:

- $Globals_P$: insieme delle variabili globali di P
- $Formals_P(i)$: insieme dei parametri formali della procedura contenente il nodo i

[1]Per semplificare la costruzione del CFG si è deciso di inserire uno statement *skip* nell'istruzione immediatamente successiva ad ogni chiamata a procedura

- $sLocals_P(i)$: insieme delle variabili strettamente locali alla procedura contenente il nodo i (ovvero le variabili dichiarate all'interno della procedura, senza considerare i suoi parametri formali)
- $Locals_P(i)$: insieme delle variabili locali alla procedura contenente il nodo i (cioè le variabili strettamente locali e i paramentri formali della procedura)
- $InScope_P(i)$: insieme delle variabili nello scope della procedura contenente il nodo i (ovvero le variabili locali alla procedura e le variabili globali)

Valgono, pertanto, le seguenti relazioni tra i suddetti insiemi:

- $Formals_P(i) \subseteq Locals_P(i)$
- $sLocals_P(i) \subseteq Locals_P(i)$
- $Locals_P(i) = sLocals_P(i) \cup Formals_P(i)$
- $sLocals_P(i) \cap Formals_P(i) = \emptyset$
- $Locals_P(i) \cup Globals_P = InScope_P(i)$

2.1.5 Semantica di programmi lineari con array

Verrà introdotta ora la semantica dei programmi lineari con array, verrà cioè attribuito un significato alle varie stringhe che rappresentano le istruzioni (statement) che compongono il programma. In altre parole, la semantica di un programma descrive gli effetti che ogni statement ha sullo stato attuale e cattura dunque le transizioni tra stati.

Funzione di valutazione

Catturare l'evoluzione dello stato di un programma durante i suoi passi di esecuzione significa catturare l'evoluzione del valore delle sue variabili.

La relazione tra una variabile di programma e il valore che essa assume in un determinato istante dell'esecuzione è catturata da una funzione detta di valutazione, che rappresenta una corrispondenza tra le variabili del programma è il dominio $\mathcal{D}$ su cui sono definite.

Formalmente, sia V_P l'insieme di variabili scalari del programma P e sia A_P l'insieme delle variabili array di P; si indichi, inoltre, con Var_P l'unione degli insiemi V_P e A_P, si definisce una valutazione $w : W \rightarrow \mathcal{D}$, con $W \subseteq$

Var_P, come una funzione che associa ad ogni variabile scalare dell'insieme $V_P \cap W$ un valore all'interno del dominio $\mathcal{D}$ e ad ogni variabile array in $A_P \cap W$ una corrispondenza finita da $\{0, \ldots, dim(a) - 1\}$ in $\mathcal{D}$

La funzione di valutazione w appena definita viene estesa alla funzione $\overline{w} : E_W \cup B_W \rightarrow 2^{\mathcal{D}}$ di valutazione di espressioni lineari (booleane) con array.

Da notare come la funzione $\overline{w}$ non associa ad ogni espressione un solo valore del dominio $\mathcal{D}$ ma un insieme di valori. Ciò accade perchè le espressioni lineari (booleane) con array utilizzate possono assumere un comportamento non deterministico a causa della possibile presenza del simbolo `u` di indefinito, che può assumere, in maniera non deterministica appunto, un qualsiasi valore del dominio $\mathcal{D}$. In realtà il non determinismo è introdotto anche da eventuali accessi ad array fuori dell'intervallo consentito, ma tali accessi possono essere evitati con opportuni controlli ed in ogni caso l'aspetto significativo che si vuole catturare è il non determinismo introdotto dal simbolo di indefinito, utile, come già detto, ai fini della verifica, per modellare le interazioni esterne.

Viene data di seguito la definizione della funzione $\overline{w}$ di valutazione di espressioni lineari (booleane) con array; sia $e \in E_W \cup B_W$ un'espressione lineare con array o un'espressione lineare booleana con array, la sua valutazione $\overline{w}(e)$ dipende dal tipo di espressione:

$$\overline{w}(e) = \begin{cases} \{e\} & \text{se } e = c \in \mathcal{D} \\ \{w(e)\} & \text{se } e = v \in V_P \cap W \\ \{w(a)(d_1) \mid d_1 \in \overline{w}(e_1)\} & \text{se } e = a\texttt{[}e_1\texttt{]},\ a \in A_P \cap W,\ e_1 \in E_W,\ \overline{w}(e_1) \subseteq \{0, \ldots, dim(a) - 1\} \\ \{c \cdot d \mid d \in \overline{w}(e_1)\} & \text{se } e = c\texttt{*}e_1,\ c \in \mathcal{D},\ e_1 \in E_W \\ \{d_1\ op\ d_2 \mid d_1 \in \overline{w}(e_1) \text{ e } d_2 \in \overline{w}(e_2)\} & \text{se } e = e_1\ op\ e_2,\ e_1, e_2 \in E_W \cup B_W,\ op \in \{\texttt{+},\texttt{<},\texttt{<=},\texttt{>},\texttt{>=},\texttt{==},\texttt{!=}\} \\ \overline{w}(e_1) & \text{se } e = \texttt{(}b\texttt{?}e_1\texttt{:}e_2\texttt{)},\ b \in B_W,\ e_1, e_2 \in E_W,\ 0 \notin \overline{w}(b) \\ \overline{w}(e_2) & \text{se } e = \texttt{(}b\texttt{?}e_1\texttt{:}e_2\texttt{)},\ b \in B_W,\ e_1, e_2 \in E_W,\ \overline{w}(b) = \{0\} \\ \overline{w}(e_1) \cup \overline{w}(e_2) & \text{se } e = \texttt{(}b\texttt{?}e_1\texttt{:}e_2\texttt{)},\ b \in B_W,\ e_1, e_2 \in E_W,\ \{0, d\} \subseteq \overline{w}(b) \text{ per qualche } d \neq 0 \\ \mathcal{D} & \text{altrimenti} \end{cases}$$

Da notare che l'ultimo caso della definizione riguarda espressioni costituite dal solo simbolo di indefinito `u` o da un accesso ad array $a\texttt{[}e_1\texttt{]}$ fuori dall'intervallo consentito (ovvero l'intervallo di indici $\{0, 1, \ldots, dim(a) - 1\}$), che vengono valutate con un qualsiasi elemento del dominio $\mathcal{D}$ scelto in maniera non deterministica. Solo in queste due evenienze si introduce direttamente nondeterminismo. L'idea intuitiva è infatti quella di associare ad ogni espressione l'insieme di tutti i valori che essa può assumere in maniera non deterministica. Un'espressione può assumere più di un valore proprio per via del fatto che una o più sottoespressioni in essa contenute siano contemplate nell'ultimo caso della definizione (e cioè `u` oppure accesso ad array fuori intervallo). Infatti, poichè la valutazione avviene ricorsivamente attraverso la valutazione delle sue sottoespressioni, un'espressione e contenente il simbolo `u` (oppure un accesso $a\texttt{[}e_1\texttt{]}$ fuori intervallo) viene associata a tutti i possibili valori assunti da e sostituendo di volta in volta il simbolo `u` (l'accesso $a\texttt{[}e_1\texttt{]}$, rispettivamente) con tutti i valori del dominio $\mathcal{D}$ (eventualmente infinito), le variabili e gli accessi "legali" (cioè all'interno dell'intervallo consentito)

ad array con i rispettivi valori da essi assunti in base alla valutazione w e applicando la semantica classica degli operatori aritmetici e relazionali.

Definizione di stato di un programma

Come si è detto, l'evoluzione di uno stato del programma ad ogni passo di esecuzione può essere catturato dall'evoluzione del valore delle sue variabili, ma in realtà ciò non è del tutto corretto, o meglio è incompleto, infatti per individuare in maniera univoca lo stato di un programma e poter stabilire, senza ambiguità, quale sarà il suo stato al prossimo passo di esecuzione, c'è bisogno, oltre che della valutazione attuale delle variabili, anche dell'istruzione di programma che verrà eseguita, quindi del nodo del CFG in cui ci si trova.

Pertanto si definisce uno stato s del programma P come una coppia $\langle$nodo, funzione di valutazione$\rangle$, in simboli $s = \langle i, w\rangle$ dove $i \in N_P$ e $w : Var_P \cap InScope_P(i) \rightarrow \mathcal{D}$ è stata definita precedentemente. Uno stato è detto iniziale se e soltanto se $i = First_P(main)$.

S_P indica l'insieme di possibili stati in cui può trovarsi il programma P.

Relazione di transizione e nozione di percorso

Le transizioni tra stati di un programma P sono catturate dalla relazione di transizione $\langle i_1, w_1\rangle \xrightarrow{\sigma}_P \langle i_2, w_2\rangle$, che costituisce quindi un'interpretazione semantica degli statement di P. Si noti che ogni transizione tra stati è associata ad un'etichetta σ che può essere la stringa vuota ϵ oppure un'espressione della forma $\textsc{call}(i, w)$ o $\textsc{ret}(i, w)$ con $i \in N_P$ e $w : Locals_P(i) \rightarrow \mathcal{D}$. Le stringhe della forma $\textsc{call}(i, w)$ e $\textsc{ret}(i, w)$ sono associate a transizioni in partenza da nodi corrispondenti a chiamate a procedura o ad exit node di una procedura di P; a tutte le altre transizioni è associata l'etichetta costituita dalla stringa vuota ϵ. La necessità di introdurre etichette associate alle chiamate e ritorni da procedura nasce dall'esigenza di conservare il giusto nodo da cui riprendere l'esecuzione al termine della procedura chiamata e tenere traccia della valutazione delle variabili locali nel punto di chiamata, che deve coincidere con quella delle locali nel punto di ritorno dalla procedura stessa. Nel resto dell'elaborato, con i caratteri in grassetto verranno denotati vettori di variabili, espressioni o elementi. Le funzioni di valutazioni di variabili o espressioni si intendono estese in maniera ovvia a vettori di variabili o espressioni. L'assegnamento parallelo verrà denotato con $\mathbf{x} = \mathbf{e}$;. Inoltre, siano $\mathbf{c} = \langle c_1, c_2, \ldots, c_n\rangle$ e $\mathbf{d} = \langle d_1, d_2, \ldots, d_n\rangle$ n-uple di elementi dell'insieme X e $\mathcal{D}$ rispettivamente, per ogni funzione $f : X \rightarrow \mathcal{D}$, si denota con $f[\mathbf{d}/\mathbf{c}]$

la funzione f' tale che $f'(c_k) = d_k$ per ogni $k = 1, 2, \ldots, n$ e $f'(c) = f(c)$ per ogni $c \neq c_k$ con $k = 1, 2, \ldots, n$.

Infine, si definisce la relazione di transizione $\langle i_1, w_1 \rangle \xrightarrow{\sigma}_P \langle i_2, w_2 \rangle$ come segue:

- se stm_{i_1} è `skip` o `return;`, allora

  $$\langle i_1, w_1 \rangle \xrightarrow{\epsilon}_P \langle sSucc_P(i_1), w_1 \rangle$$

  Gli statement `skip` e `return;` non hanno effetto sulla valutazione delle variabili, così l'unico stato successivo possibile è rappresentato dalla coppia formata dal nodo successore (che nel caso di tali statement è unico) e dalla stessa valutazione dello stato attuale.

- se stm_{i_1} è un assegnamento parallelo (**y=e;**), allora

  $$\langle i_1, w_1 \rangle \xrightarrow{\epsilon}_P \langle sSucc_P(i_1), w_1[\mathbf{d}/\mathbf{y}] \rangle$$

  per qualche $\mathbf{d} \in \overline{w_1}(\mathbf{e})$. Se i corrisponde ad un assegnamento parallelo, allora gli stati successori sono tutti quelli formati dall'unico nodo successore del nodo i e da una delle funzioni di valutazione w_2 che corrispondono alla funzione w_1 dello stato di partenza su tutte le variabili eccetto che per quelle in $\mathbf{y}$. Il valore di w_2 in corrispondenza di ogni variabile in $\mathbf{y}$ è una delle valutazioni possibili per la rispettiva espressione in $\mathbf{e}$.

  Da notare che se $\mathbf{y}$ e $\mathbf{e}$ sono vettori di arietà 1, lo statement descrive un assegnamento semplice, che viene trattato quindi in maniera del tutto analoga.

- se stm_{i_1} è un assegnamento ad array (a`[`e_1`]=`e_2`;`), allora

  $$\langle i_1, w_1 \rangle \xrightarrow{\epsilon}_P \langle sSucc_P(i_1), w_1[(w_1(a)[d_2/d_1])/a] \rangle$$

  per qualche $d_1 \in \overline{w_1}(e_1)$ e $d_2 \in \overline{w_1}(e_2)$. Anche in un assegnamento ad un array a, la funzione di valutazione di un possibile stato di arrivo conincide con quella di partenza su tutte le variabili, eccetto che per l'elemento dell'array a indicizzato da un possibile valore assunto dall'espressione e_1, il cui nuovo valore nello stato di arrivo è uno dei possibili valori per e_2.

- se stm_{i_1} è uno statement condizionale o iterativo (`if(`e`)`, `while(`e`)`, `assert(`e`);` o `assume(`e`);`), allora

  $$\langle i_1, w_1 \rangle \xrightarrow{\epsilon}_P \langle i_2, w_1 \rangle$$

  dove

- $i_2 = Fsucc_P(i_1)$ se $\{0\} = \overline{w_1}(e)$
- $i_2 = Tsucc_P(i_1)$ se $0 \notin \overline{w_1}(e)$
- $i_2 \in Succ_P(i_1)$ se esiste $d \neq 0$ tale che $\{0, d\} \subseteq \overline{w_1}(e)$

Uno statement condizionale non altera il valore delle variabili del programma, pertanto la funzione di valutazione di un possibile stato di arrivo coincide con quella dello stato di partenza. Invece, i possibili nodi di arrivo possono essere il nodo $Tsucc_P(i)$ se l'unica valutazione possibile della guardia è vero ($0 \notin \overline{w_1}(e)$), il nodo $Fsucc_P(i)$ se l'unica valutazione possile per la guardia è falso ($\overline{w_1}(e) = \{0\}$). Notare che in questi due casi (la guardia ammette il solo valore vero o il solo valore falso) esiste un unico possibile stato successore. Infine, nel caso in cui la guardia possa assumere non deterministicamente valore vero o falso ($\{0, d\} \subseteq \overline{w_1}(e)$ per qualche $d \neq 0$), allora il nodo successore è scelto anch'esso in maniera non deterministica tra quelli dell'insieme $Succ_P(i) = \{Tsucc_P(i), Fsucc_P(i)\}$ ed esistono due possibili stati di arrivo della transizione.

- se stm_{i_1} è una chiamata a procedura (pr`(e);`), allora

$$\langle i_1, w_1 \rangle \xrightarrow{\text{CALL}(RetPt_P(i_1), w)}_P \langle First_P(pr), w_2 \rangle$$

dove $w : Locals_P(i_1) \to \mathcal{D}$ è tale che $w(\mathbf{x}) = w_1(\mathbf{x})$, $w_2(\mathbf{y}) \in \overline{w_1}(\mathbf{e})$ e $w_2(\mathbf{g}) = w_1(\mathbf{g})$, con $\mathbf{x} = Locals_P(i_1)$, $\mathbf{y} = Formals_P(First_P(pr))$ e $\mathbf{g} = Globals_P$. In una chiamata a procedura (pr`(e);`), bisogna tener traccia dello stato attuale del programma prima di poter cambiare procedura e quindi scope delle variabili. Pertanto, la relazione di transizione conserva (nell'etichetta ad essa associata) il punto di ritorno della procedura ($RetPt_P(i_1)$) e, nella funzione di valutazione w, la valutazione delle variabili locali nel nodo corrispondente alla chiamata a procedura (infatti w e w_1 coincidono sulla valutazione delle variabili locali, come affermato dalla condizione $w(\mathbf{x}) = w_1(\mathbf{x})$). Tali elementi dovranno coincidere con quelli dell'etichetta della transizione relativa all'uscita dalla procedura pr. Un possibile stato di arrivo della transizione è identificato dal primo nodo della procedura invocata ($First_P(pr)$) e da una funzione di valutazione w_2 tale che w_2 coincide con la valutazione w_1 dello stato di partenza sulle variabili globali ($w_2(\mathbf{g}) = w_1(\mathbf{g})$) e assegna ai suoi parametri formali una possibile valutazione delle espressioni che costituiscono i parametri effettivi della chiamata a procedura ($w_2(\mathbf{y}) \in \overline{w_1}(\mathbf{e})$).

- se $i_1 = Exit_P(pr)$, allora

$$\langle i_1, w_1 \rangle \xrightarrow{\text{RET}(i_2,w)}_P \langle i_2, w_2 \rangle$$

  dove $i_2 \in Succ_P(i_1)$, $w_2(\mathbf{g}) = w_1(\mathbf{g})$ e $w_2(\mathbf{x}) = w(\mathbf{x})$ e con $\mathbf{g} = Globals_P$ e $\mathbf{x} = Locals_P(i_2)$. Se il nodo i è l'exit node di una procedura pr, allora bisogna propagare anche al prossimo stato gli effetti della procedura sulle variabili globali ($w_2(\mathbf{g}) = w_1(\mathbf{g})$) e ritornare allo scope della procedura che aveva invocato pr. Quindi una funzione di valutazione w_2 per un possibile stato di arrivo è definita, oltre che sulle variabili globali, sulle variabili locali del nodo di arrivo e, su di esse, deve coincidere con la funzione w ($w_2(\mathbf{x}) = w(\mathbf{x})$), nella quale era stata memorizzata la valutazione delle variabili locali al momento della chiamata a procedura. Anche il nodo dello stato di arrivo coincide con quello che era stato memorizzato nell'etichetta della relazione di transizione relativa alla chiamata alla procedura pr.

Data una formalizzazione di come evolve lo stato di un programma nel corso dell'esecuzione di una traccia, si fa notare che, a causa dell'introduzione del non determinismo con il simbolo `u`, non si è parlato di funzione ma di relazione di transizione, perchè da uno stato è possibile raggiungere, in maniera non deterministica, più di uno stato.

La funzione $post_P : 2^{S_P} \to 2^{S_P}$ mette in relazione un insieme di stati di partenza con un insieme di stati che è possibile raggiungere al prossimo passo di esecuzione del programma P ed è definita di seguito. Sia $S \subseteq S_P$:

$$post_P(S) = \{s' \mid s \xrightarrow{\sigma}_P s' \text{ e } s \in S\}$$

Un *path*, percorso, è una sequenza $\langle i_0, w_0 \rangle \xrightarrow{\sigma_1}_P \langle i_1, w_1 \rangle \xrightarrow{\sigma_2}_P \ldots \xrightarrow{\sigma_n}_P \langle i_n, w_n \rangle$ tale che $\langle i_k, w_k \rangle \xrightarrow{\sigma_{k+1}}_P \langle i_{k+1}, w_{k+1} \rangle$ con $k = 0, \ldots, n-1$[2].

Da notare che non tutti i percorsi rappresentano possibili tracce di esecuzione, infatti in transizioni del tipo $\langle i_1, w_1 \rangle \xrightarrow{\text{RET}(i_2,w)}_P \langle i_2, w_2 \rangle$, il nodo e la valutazione dello stato di arrivo sono vincolati a quelli indicati nell'etichetta $\text{RET}(i_2, w)$ della transizione (infatti, per come è stata definita la relazione, il nodo di arrivo deve essere lo stesso di quello indicato nell'etichetta e la valutazione di arrivo w_2 deve corrispondere, sulle variabili locali, alla valutazione w presente nell'etichetta della transizione), ma non esiste alcun vincolo sui nodi e sulle valutazioni da indicare nell'etichetta stessa, che dovrebbero

[2]Di qui in avanti l'espressione $\langle i_0, w_0 \rangle \xrightarrow{\sigma_1}_P \langle i_1, w_1 \rangle \xrightarrow{\sigma_2}_P \ldots \xrightarrow{\sigma_n}_P \langle i_n, w_n \rangle$ verrà denotata con l'espressione $\langle i_0, w_0 \rangle \xrightarrow{\Sigma_0^n}_P \langle i_n, w_n \rangle$, dove $\Sigma_0^n = \sigma_1 \sigma_2 \ldots \sigma_n$

essere vincolati a quelli della chiamata alla procedura cui si riferisce l'exit node identificato dal nodo i_1.

Pertanto, si introducono le nozioni di *same-level valid path* e *valid path*, rispettivamente percorso valido dello stesso livello e percorso valido [RHS95]. Un percorso è detto valido dello stesso livello se e soltanto se la sequenza di etichette associata al percorso è una stringa nel linguaggio delle parentesi bilanciate definita dal non-terminale *matched* relativo alla seguente grammatica context-free:

$$matched ::= \epsilon \mid \langle \ \textsc{call}(i, w) \ \rangle \ matched \ \langle \ \textsc{ret}(i, w) \ \rangle \ matched$$

In pratica, un percorso valido dello stesso livello da $\langle i_0, w_0 \rangle$ a $\langle i_n, w_n \rangle$ descrive una trasmissione di effetto da $\langle i_0, w_0 \rangle$ a $\langle i_n, w_n \rangle$, dove i_0 e i_n sono nodi appartenenti alla stessa procedura. Un tale percorso è caratterizzato da una sequenza di passi di esecuzione durante i quali lo stack delle chiamate a procedura può crescere temporaneamente, per poi tornare al suo stato iniziale, senza mai diventare più piccolo di quest'ultimo. Questo equivale a dire che, nella sequenza di etichette associate al percorso, per ogni terminale relativo ad una chiamata a procedura esiste un terminale relativo al ritorno della procedura stessa.

Un percorso è detto valido se e soltanto se la sequenza di etichette ad esso associate è una stringa definita dal non-terminale *valid* relativo alla seguente grammatica context-free:

$$valid ::= matched \mid valid \ \langle \ \textsc{call}(i, w) \ \rangle \ matched$$

Di fatto un percorso da $\langle i_0, w_0 \rangle$ a $\langle i_n, w_n \rangle$ è detto valido se descrive la trasmissione di effetto da $\langle i_0, w_0 \rangle$ a $\langle i_n, w_n \rangle$ attraverso una sequenza di passi di esecuzione che può terminare con qualche record di attivazione sullo stack delle chiamate a procedura, anche stavolta senza, però, mai decrescere rispetto allo stato iniziale dello stack. Questo equivale ad avere nella sequenza di etichette associate al percorso un numero di terminali $\textsc{call}(i, w)$ maggiore del numero di terminali della forma $\textsc{ret}(i, w)$.

Un percorso è detto inizializzato se e soltanto se il suo stato iniziale s_0 è nella forma $\langle First_P(main), w_0 \rangle$ per qualche w_0.

Si definisce inoltre la nozione di semantica di P, in simboli $[\![P]\!]$, come l'insieme di tutti i percorsi validi inizializzati del programma P secondo la sua relazione di transizione $\xrightarrow{\sigma}_P$.

Uno stato $\langle i, w \rangle$ è detto raggiungibile in P se e soltanto se esiste un percorso $\pi \in [\![P]\!]$ valido inizializzato che raggiunge lo stato $\langle i, w \rangle$. Un vertice $i \in N_P$ è raggiungibile se e soltanto se esiste una valutazione w tale che $\langle i, w \rangle$ è raggiungibile.

2.2 Introduzione all'astrazione

Una qualsiasi procedura di model checking prende in input un programma, rappresentato mediante un'opportuna struttura dati, e una semantica associata alle stringhe (semplici entità sintattiche) che rappresentano i suoi statement. In base alla semantica associata, il model checking computa la raggiungibilità o meno di uno statement del programma passato in input. Come già accennato in precedenza, non esiste attualmente una procedura di model checking che computi la raggiungibilità di uno statement di programma rispetto ad una semantica definita su un insieme di variabili che comprende anche delle variabili di tipo array. Bisogna quindi passare in ingresso al model checker una semantica definita su un insieme di sole variabili scalari.

Esistono due strade per passare da una semantica definita anche su variabili array ad una definita solo su variabili scalari: attraverso un processo di astrazione sintattica oppure mediante un procedimento di astrazione semantica.

2.2.1 Astrazione sintattica

Per astrazione sintattica s'intende il processo di trasformazione di un programma P definito sul dominio dei programmi lineari con array in un programma astratto $\widehat{P}$ definito sul dominio dei programmi lineari attraverso la sostituzione, mediante un processo puramente sintattico, di ogni statement stm_i di P con quello astratto $\widehat{stm_i}$ equivalente, secondo la semantica di P, a quello originale e tale da non contenere eventuali accessi ad array presenti in stm_i [CAM04].

Il programma $\widehat{P}$ ottenuto in seguito al processo di astrazione, unitamente alla medesima semantica (ovvero la medesima relazione di transizione) associata al programma concreto originale, può a questo punto essere dato in ingresso al model checker che, applicando la relazione semantica ai nuovi statement del programma astratto e al nuovo insieme di variabili di $\widehat{P}$ (all'interno del quale non compaiono variabili di tipo array) verificherà la raggiungibilità o meno di uno statement di errore all'interno di quest'ultimo.

L'intero processo è illustrato in figura 2.1.

2.2.2 Astrazione semantica

Un'altra possibile strategia, schematizzata in figura 2.2 di astrazione consiste nel definire una nuova relazione semantica attraverso una relazione di

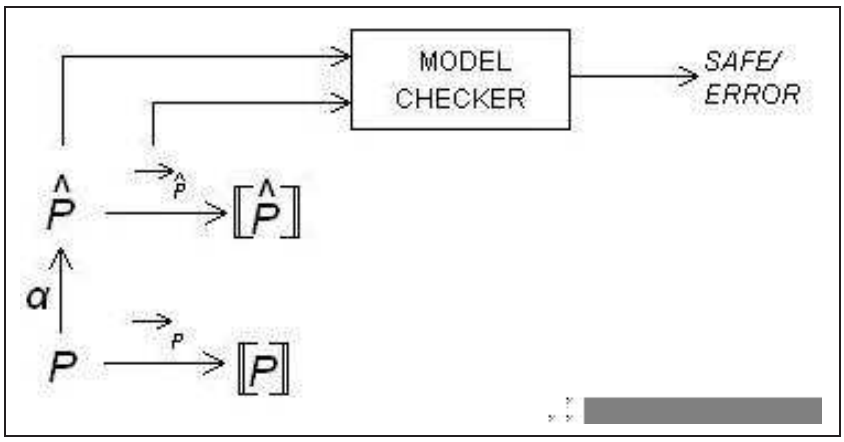


Figura 2.1: Schema alla base del processo di astrazione sintattica: la funzione α effettua la trasformazione sintattica da P a $\widehat{P}$. La semantica, ossia la relazione di transizione, è la stessa, ma viene applicata su un diverso insieme di stringhe sintattiche (statement) e su un ristretto dominio di variabili, che non contiene array. Con $[\![P]\!]$ e $[\![\widehat{P}]\!]$ si denotano rispettivamente l'insieme di percorsi di P e l'insieme dei percorsi astratti di $\widehat{P}$

transizione astratta e dare in ingresso al model checker il programma concreto originale e la nuova semantica astratto computata. Il processo di model checking verificherà se tra tutti i possibili percorsi di esecuzione calcolati in base alla nuova relazione di transizione astratta ne esiste almento uno che conduca ad uno statement di errore.

E' evidente che, affinchè il model checker possa eseguire la verifica, è necessario che la semantica astratta venga definita su un dominio di sole variabili scalari. Inoltre, la relazione di transizione astratta deve essere definita in maniera tale che l'evoluzione degli stati astratti in un percorso astratto avvenga coerentemente a quanto accadeva con la relazione di transizione concreta tra stati, ovvera l'evoluzione dello stato delle variabili (e degli array) del programma concreto e il flusso di controllo dello stesso devono essere catturati e preservati dalla relazione di transizione astratta, ma su questa tematica si tornerà in maniera più precisa nella prossima sezione.

2.2.3 Un ulteriore beneficio derivante dal processo di astrazione

Oltre alla necessità di eseguire la fase di astrazione preliminarmente a quella di model checking dettata dal bisogno di ridurre il dominio di variabili elimi-

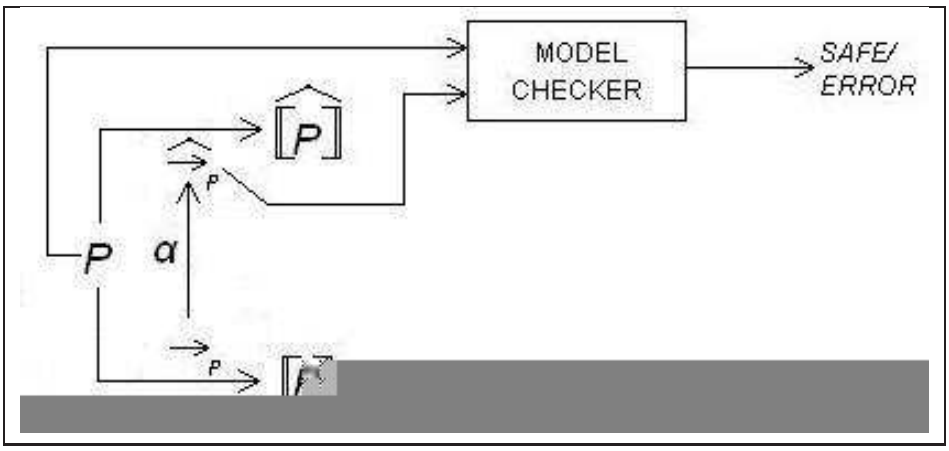


Figura 2.2: Schema alla base del processo di astrazione semantica: la funzione α definisce la nuova relazione di transizione astratta, che viene data in input al model checker unitamente al programma concreto. Con $[\![P]\!]$ e $\widehat{[\![P]\!]}$ si denotano rispettivamente l'insieme di percorsi di P e l'insieme dei suoi percorsi astratti rispetto alla semantica astratta

nando gli array, l'astrazione è molto utile nel campo della verifica automatica tramite model checking perchè consente di creare modelli semplificati a partire da un programma concreto. Infatti, poichè la complessità dell'algoritmo di model checking è esponenziale, come verrà fatto notare nel capitolo 3, rispetto al numero di variabili, sarebbe auspicabile poter ridurre il più possibile il numero di variabili coinvolte in tale processo. In fase di astrazione è possibile ottenere quest'ulteriore beneficio, infatti il risultato del processo di astrazione, sia essa sintattica che semantica, dipende dal nuovo insieme di variabili da tenere in considerazione in fase di verifica. E' chiaro che quanto più grande è il numero di variabili che si astrae via dalla verifica tanto più grande è l'informazione che si perde in fase di astrazione e tanto più il modello astratto si discosterà dall'originale e proprio perciò, come evidenziato nel capitolo 1, l'introduzione di una fase di astrazione in un tool di verifica non può prescindere dall'introduzione di un modulo che si occupi di raffinare un modello astratto troppo distante da quello originale.

In sintesi, si conclude che per un meccanismo di astrazione è auspicabile la presenza di due proprietà fondamentali [Cla03]:

- **feasibility**: il modello astratto deve essere significativamente più piccolo e semplice del programma concreto cui si riferisce in modo da ridurre la complessità computazionale del model checking, anche a costo di eseguire un numero maggiore di iterazioni CEGAR

- **preservation**: il modello astratto deve preservare **tutti** i percorsi di esecuzione del programma concreto. Come conseguenza di ciò accade che se una proprietà ϕ risulta vera in tutti i percorsi del modello astratto, allora sicuramente essa è valida anche nel programma concreto. Tale proprietà, però, non garantisce che l'astrazione contenga **solo** percorsi del programma concreto. Anzi, generalmente l'astrazione contiene percorsi di esecuzione non presenti nel programma concreto, e come conseguenza di ciò potrà verificarsi che una proprietà sia falsa nel modello astratto pur essendo valida nel programma originario, in quanto la traccia di esecuzione che la falsifica non corrisponde ad un percorso del programma concreto. Tali tracce di esecuzione vengono definite spurie

2.3 Meccanismo di astrazione preesistente in Eureka

Per capire come il presente lavoro si colloca nell'ambito del progetto Eureka, appare opportuno illustrare il meccanismo di astrazione preesistente per poter cogliere le differenze con quello proposto in questa tesi e i benefici che quest'ultimo ha apportato al tool.

Innanzitutto va detto che il meccanismo preesistente in Eureka, a differenza di quello proposto nel presente lavoro, segue una strategia di astrazione sintattica, ovvero consiste nel prendere in considerazione, dato un programma P, solo un sottoinsieme delle sue variabili e, rispetto ad esso, costruisce la versione astratta $\widehat{P}$ del programma concreto attraverso una manipolazione sintattica delle stringhe rappresentanti i suoi statement.

In altre parole, se si indica con V_P l'insime di variabili scalari del programma P, con A_P l'insieme delle sue variabili array e con Var_P l'unione degli insiemi V_P e A_P, contenente tutte le variabili del programma, è possibile definire l'astrazione di P rispetto a $V \subseteq V_P$ e rispetto alla classe di insiemi di indici $I = \{R(a) \mid a \in A_P\}$[3], con $R(a) \subseteq \{0, 1, \ldots dim(a) - 1\}$ per ogni $a \in A_P$, come il programma ottenuto sostituendo:

- ogni riferimento ad una variabile di $V_P \setminus V$ all'inerno di un'espressione con il simbolo di indefinito (`u`);

[3]Anche se si è detto che l'astrazione può avvenire rispetto a $V \subseteq V_P$ e rispetto ad I, nel corso della tesi si supporrà, a soli fini esemplificativi, che $V = V_P$ e cioè che si introducono nell'astrazione tutte le variabili scalari e si parlerà semplicemente di astrazione rispetto ad I. Resta il fatto che tutti i concetti e teoremi enunciati valgono anche nel caso generale di astrazione rispetto a $V \subseteq V_P$ e rispetto ad I

- ogni assegnamento ad una variabile di $V_P \setminus V$ con lo statement vuoto (`skip`);
- ogni occorrenza di array a`[`e`]` all'interno di un espressione con l'espressione:

  $$\texttt{(}e\texttt{==}k_1\texttt{)?}a_{k_1}\texttt{:(}e\texttt{==}k_2\texttt{)?}a_{k_2}\texttt{:}\ldots\texttt{:(}e\texttt{==}k_n\texttt{)?}a_{k_n}\texttt{:u}$$

  con $R(a) = \{k_1, k_2, \ldots, k_n\}$;
- ogni assegnamento ad array a`[`e_1`]=`e_2`;` con l'assegnamento parallelo

  $$a_{k_1}\texttt{=(}e_1\texttt{==}k_1\texttt{?}e_2\texttt{:}a_{k_1}\texttt{)},\ldots,a_{k_n}\texttt{=(}e_1\texttt{==}k_n\texttt{?}e_2\texttt{:}a_{k_n}\texttt{)}$$

  con $R(a) = \{k_1, k_2, \ldots, k_n\}$;

L'idea alla base di tale meccanismo è quella di eliminare tutte le variabili array e modellare ogni elemento di array presente in P nel modo più banale ed intuitivo, cioè attraverso l'introduzione di una variabile scalare che ne catturi l'evoluzione. In questo modo si ha che ogni variabile array $a \in A_P$ viene modellata attraverso l'inserimento, nel dominio di variabili astratte, di un numero di variabili pari alla dimensione di a $(a_0, a_1, \ldots, a_{dim(a)-1})$. Coma già evidenziato nel corso di tale capitolo, per questioni di efficienza, è preferibile astrarre con il dominio di variabili più piccolo possibile, pertanto per ogni array $a \in A_P$, non si modellano tutti i suoi elementi ma solo un sottoinsieme di essi individuato da $R(a)$. Il dominio di variabili astratte è dunque costituito dall'insieme $V_{\widehat{P}} = V_P \cup \{a_k \mid a \in A_P \text{ e } k \in R(a)\}$, dove per ogni $a \in A_P$, $R(a)$ è un insieme di interi, ognuno dei quali individua, in maniera fissata, un preciso elemento dell'array a. Per ogni intero in $R(a)$, denotato con k, esiste quindi nel dominio di variabili astratte, una variabile scalare a_k, che cattura l'evoluzione del k-esimo elemento dell'array a e la cui valutazione è legata in maniera indissolubile ad un preciso elemento di a, il k-esimo appunto, indipendentemente dalla possibile esecuzione del programma e dalla funzione di valutazione utilizzata.

2.3.1 Analisi dei limiti del precedente meccanismo di astrazione utilizzato in Eureka

Il meccanismo di astrazione immerso all'interno del ciclo CEGAR di cui si è parlato ampiamente nel capitolo 1, viene ripetuto ad ogni iterazione in maniera incrementale, cioè includendo insiemi di indici sempre più estesi.

Tale meccanismo funziona molto bene quando il refiner riesce ad individuare un indice numerico corrispondente all'elemento di un array che rende

```
    void main() {
      int a[5];

[1]   a[3] = 3;
[2]   assert(a[3] == 3);
    }
```

Tabella 2.2: Programma per cui il meccanismo di astrazione preesistente in Eureka era sufficientemente efficiente

non fattibile, nel programma concreto, una traccia esistente, invece, in quello astratto, come accade nel programma P di tabella 2.2.

Se si analizza nel dettaglio il comportamento di Eureka sul predetto esempio, si osserva che, durante la prima iterazione del ciclo, il programma P viene astratto rispetto al minor livello di dettaglio possibile, infatti l'astrazione avviene rispetto all'insieme $\{R(a)\}$, con $R(a) = \emptyset$ e con a unico array del programma. In pratica vengono sostituite tutte le occorrenze di array nelle espressioni con `u` e tutti gli assegnamenti ad array vengono rimpiazzati da uno `skip` in modo da generare il programma astratto $\widehat{P_0}$, mostrato in tabella 2.3.

Da notare che, anche con l'astrazione più brutale, che astrae dal programma il maggior numero di elementi possibile (tutti gli elementi di tutti gli array), l'astrazione conserva sempre lo stesso flusso di controllo del programma concreto.

La procedura di model checking su $\widehat{P_0}$ rileva la presenza di una traccia che porta all'errore attraverso il fallimento dell'assert (si parla di fallimento di assert quando la guardia e di uno statement di tipo `assert(`e`);` viene valutata falsa e il programma termina con un errore).

```
    void main() {
      ;

[1]   ;
[2]   assert(u == 3);
    }
```

Tabella 2.3: $\widehat{P_0}$: astrazione iniziale del programma concreto P

In fase di simulazione, con l'utilizzo di un theorem prover, emerge che la traccia è spuria, in quanto non è fattibile nel programma concreto. Il

refiner, utilizzando la prova generata dal theorem prover individua nell'elemento 3 dell'array a la causa della non fattibilità della traccia nel programma concreto.

Così comincia una nuova iterazione del ciclo, attraverso la costruzione di un nuovo modello astratto, stavolta rispetto alla classe di indici $\{R(a)\}$, con $R(a) = \{3\}$.

Dalla tabella 2.4 si può notare come, nel nuovo modello astratto $\widehat{P_1}$, la traccia spuria, che aveva portato al fallimento dell'assert nella precedente iterazione, non è più fattibile.

```
      void main() {
        int a3 ;

[1]     a3 = 3;
[2]     assert(a3 == 3);
      }
```

Tabella 2.4: $\widehat{P_1}$: astrazione alla seconda iterazione CEGAR del programma concreto P

Il model checker rivela l'assenza di errori nel nuovo programma astratto, pertanto il ciclo si arresta concludendo che anche il programma concreto non contiene errori.

Il problema di tale meccanismo è che non si riescono ad individuare proprietà generali riguardanti generici elementi (o gruppi di elementi) di array, ma solo proprietà riguardanti un fissato elemento (o gruppi di elementi) di array.

Il concetto sarà più chiaro con l'aiuto di un esempio. Si consideri il programma di tabella 2.5.

Al primo ciclo CEGAR, Eureka trova una traccia spuria che porta al fallimento dell'assert. L'elemento che rende spuria tale traccia è il generico elemento x-esimo dell'array a, con x non assegnato, ma appartenente all'insieme di indici dell'array ($\{0, 1, \ldots, dim(a) - 1\}$). Il processo di raffinamento restituisce un indice non numerico, si tratta infatti dell'espressione x, associata all'array a. Il meccanismo di astrazione precedentemente utilizzato non riesce, dunque, ad aggiungere un indice all'insieme $R(a)$ per un nuovo tentativo di astrazione. Per non ripetere il processo di astrazione sugli stessi insiemi di indici, ottenendo così lo stesso modello astratto del ciclo precedente, l'astrazione viene effettuata rispetto all'intero insieme di indici di a. Solo a questo punto il model checker conclude che il programma è privo di errore, ponendo fine all'esecuzione del ciclo.

```
    void main() {
      int a[5];
      int x;

[1]   if ( (x >= 0) && (x < 5) ) {
[2]     a[x] = 3;
[3]     assert(a[x] == 3);
      }
    }
```

Tabella 2.5: Programma per cui il meccanismo di astrazione preesistente in Eureka diventa pericolosamente inefficiente

Come si può notare, Eureka riesce ad eliminare la traccia spuria solamente astraendo rispetto a tutti gli elementi dell'array, quindi in questi casi la dimensione dell'astrazione[4] dipende dalla dimensione dell'array e ciò potrebbe essere evitato se si riuscisse ad esprimere il fatto che qualsiasi valore assuma x, l'espressione all'interno dello statement di tipo `assert` non verrà mai valutata falsa. Sarebbe estremamente potente poter astrarre rispetto al generico x-esimo elemento dell'array a e garantire che, per qualsiasi valutazione di x ammessa dal programma, la traccia che porta al fallimento dell'assert non appartiene più neanche all'insieme di tracce astratte.

Un altro esempio in cui la precedente versione del meccanismo di astrazione appare alquanto inadeguata è presentato nella tabella 2.6

Anche in questo caso, se si raggiunge lo statement `assert`, qualsiasi valore assuma x (che comunque resta compreso nell'insieme di indici di a), vale sempre la relazione di uguaglianza tra l'x-esimo elemento dell'array a e l'$(x+1)$-esimo elemento del vettore b. Ancora una volta il meccanismo di astrazione preesistente deve astrarre rispetto a tutti gli elementi di a e di b, mentre sarebbe sufficiente astrarre rispetto al generico x-esimo elemento di a e al generico $(x+1)$-esimo elemento di b, riuscendo così ad eliminare la traccia spuria dall'astrazione con l'aggiunta di sole due variabili astratte, indipendentemente dalle dimensioni dei vettori a e b.

Riuscire a rendere la dimensione dell'astrazione indipendente da quella degli array in essa coivolti comporterebbe quindi notevoli benefici per l'efficienza del tool. Infatti un'astrazione la cui dimensione dipende da quella degli array ha una forte influenza negativa sulle prestazioni generali, cosiderato che risultati empirici mostrano un sensibile degrado di prestazioni già

[4]Per dimensione dell'astrazione si intende il numero di variabili nel dominio astratto

```
    void main() {
      int a[5];
      int b[6];
      int x;

[1]   i=0;
[2]   while (i<5) {
[3]     a[i] = b[i+1];
      }

[4]   if ( (x >= 0) && (x < 5) ) {
[5]     assert(a[x] == b[x+1]);
      }
    }
```

Tabella 2.6: Ulteriore esempio per cui il precedente meccanismo di astrazione appare inadeguato

per array di dimensione superiore ai 20 elementi.

2.4 Un nuovo meccanismo di astrazione

L'idea alla base del presente lavoro di tesi è quella di utilizzare delle variabili per indicizzare elementi di array in maniera dinamica. Infatti, ad ognuna di tali variabili viene assegnata un'espressione e, in base alle possibili valutazioni di quest'ultima in diverse tracce di esecuzione, tali variabili puntano, ad ogni esecuzione del programma, ad un diverso elemento dell'array.

Si indica con Θ l'insieme di tali nuove variabili, ognuna delle quali inizializzata, ed in seguito mai più modificata, con un'espressione $e \in E_{\Theta,W}$. Ogni variabile $\theta \in \Theta$ rappresenta, quindi, un indice di array e, per ognuna di esse, viene introdotta nel programma astratto un'altra nuova variabile a_θ, con $a \in A_P$, che rappresenta l'elemento dell'array a indicizzato da θ.

Un'esecuzione di un programma può essere vista come una successione di valutazioni delle sue variabili, che fotografano l'evoluzione del suo stato ad ogni passo di esecuzione.

Pur non considerando il non determinismo dovuto all'introduzione, nella sintassi, del simbolo `u` (indefinito), ad un programma non è associato un unico percorso di esecuzione, bensì un insieme di tracce di esecuzione, ognuna a partire da una possibile valutazione iniziale delle sue variabili. Da notare

che nel caso in cui le variabili siano definite su un dominio infinito $\mathcal{D}$, allora esistono infiniti possibili assegnamenti alle variabili del programma e quindi infinite valutazioni iniziali e di conseguenza infinite tracce di esecuzione.

Se consideriamo una variabile $\theta \in \Theta$, aggiunta nel programma astratto ed a cui è stata assegnata un'espressione $e_\theta \in E_{\theta,Var_P}$, la valutazione di tale variabile è uguale alla valutazione dell'espressione e_θ al momento dell'assegnamento, che, come già detto, può variare a seconda della valutazione iniziale delle variabili di programma ed è proprio in questo senso che una variabile θ indicizza dinamicamente un array.

Infatti la stessa variabile θ, e quindi la relativa variabile a_θ per qualche $a \in A_P$, possono riferirsi a diverse locazioni dell'array in diverse tracce di esecuzione del programma in dipendenza della valutazione iniziale.

Quindi, se si astrae rispetto ad una variabile θ associata ad un array $a \in A_P$ è come se si astraesse contemporaneamente per tutti i valori corrispondenti alle possibili valutazioni dell'espressione e_θ ad essa associata e sarà possibile eliminare un insieme più ampio di tracce spurie in un'unica iterazione del ciclo CEGAR.

Al solo scopo di fornire un'idea intuitiva di come vengono sfruttate le variabili $\theta \in \Theta$ per modellare dinamicamente elementi di array, viene presentato in tabella 2.7 il programma astratto, con riferimento al programma concreto di tabella 2.6, costruito sfruttando la strategia appena illustrata. Si evidenzia però che nel nuovo meccanismo di astrazione non avviene la costruzione sintattica del programma astratto perchè quando si astrae rispetto ad un insieme di indici rappresentato da variabili anzichè costanti non è possibile attuare una tale trasformazione (o almeno non è così banale; i problemi nascono quando due o più variabili $\theta \in \Theta$ assumono lo stesso valore, puntando quindi allo stesso elemento di un array), così come accadeva con il precedente meccanismo di astrazione, dal programma concreto P a quello astratto $\widehat{P}$. L'astrazione proposta nel presente elaborato è esclusivamente a livello semantico (per meglio comprendere la differenza tra astrazione sintattica e semantica si rimanda alla sezione 2.2) e consiste quindi nella definizione di una nuova relazione di transizione, e quindi di una semantica astratta su un diverso dominio di variabili astratte (che non contiene array) che verrà utilizzata dal model checker per verificare la raggiungibilità di nodi di errore nel programma concreto rispetto alla sua semantica astratta.

Come già visto nella sezione precedente, durante la prima iterazione il model checker trova una traccia che porta all'errore, il simulatore rivela che tale traccia è una traccia spuria ed il processo di raffinamento individua le espressioni x, associata al vettore a, e $x + 1$ associata a b. Secondo il nuovo meccanismo di astrazione proposto, vengono introdotte nel programma astratto la variabile θ_1, a cui viene assegnata la variabile x, e la variabile

P	$\widehat{P}$
`void main() {`	`void main() {`
`  int a[5];`	`  int` a_{θ_1}`;`
`  int b[6];`	`  int` b_{θ_2}`;`
`  int x;`	`  int x;`
`[1]  i=0;`	`  i=0;` $\langle\theta_1$`=x;`θ_2`=x+1;>`
`[2]  while (i<5) {`	`  while (i<5) {`
`[3]  a[i] = b[i+1];`	`  a`$_{\theta_1}$`=(`θ_1`==i)?(`θ_2`=i+1?b`$_{\theta_2}$`:u):a`$_{\theta_1}$`;`
`  }`	`  }`
`[4]  if ( (x>=0) && (x<5) ) {`	`  if ( (x>=0) && (x<5) ) {`
`[5]  assert(a[x] ==`	`  assert((`θ_1`==x?a`$_{\theta_1}$`:u)`
`  == b[x+1]);`	`  ==(`θ_2`==x+1?b`$_{\theta_2}$`:u));`
`  }`	`  }`
`}`	`}`

Tabella 2.7: Astrazione $\widehat{P}$ del programma concreto P costruita sfruttando la nuova strategia proposta

θ_2, a cui si assegna l'espressione x+1. Gli statement di assegnamento delle variabili θ_1 e θ_2 sono raggruppate in parentesi angolari perchè non si tratta di veri e propri statement del programma. Infatti, tali assegnamenti non verranno inseriti davvero nel CFG del programma, ma, come vedremo nel capitolo 3, si farà in modo che abbiano effetto in fase di model checking. Vengono introdotte, inoltre, le variabili a_{θ_1} e b_{θ_2} che rappresentano dinamicamente, in base alle valutazioni iniziali di x e x+1, un elemento dell'array a e b rispettivamente.

Quindi si esegue astrazione non più rispetto ad un insieme di indici numerici, bensi ad un insieme di espressioni che indicizzano un array[5].

Si definisce, quindi, il nuovo dominio di variabili astratte relativo all'astrazione di P rispetto ad $I = \{\theta(a) \mid a \in A_P\}$ come $\widehat{V_P} = V_P \cup \{a_\theta | a \in A_P \wedge \theta \in \theta(a)\} \cup \Theta$, con $\Theta = \bigcup_{a \in A_P} \theta(a)$ e dove $\theta(a)$ è l'insieme contenente tutte le nuove variabili $\theta \in \Theta$ associate ad a rispetto cui astrarre. Intuitivamente l'insieme $\theta(a)$ è l'analogo dell'insieme $R(a)$ utilizzato nel precedente mec-

[5]In realtà gli insiemi di indici numerici $R(a)$ ci sono ancora, oltre ai nuovi insiemi di indici-espressioni, ma nel corso di tale lavoro di tesi verranno tralasciati per illustrare in maniera più chiara e semplice le innovazioni introdotte

canismo di astrazione, infatti contiene gli indici (sottoforma di espressioni) corrispondenti agli elementi dell'array a che si vuole includere nell'astrazione.

Nella successiva fase di model checking si analizzano tutte le possibili tracce di esecuzione appartenenti all'astrazione di P. Stavolta non è possibile giungere al fallimento dell'assert perchè, per ogni valore iniziale di x, e quindi di $x+1$, se si giunge allo statement assert le variabili astratte a_{θ_1} e a_{θ_2} sono uguali.

In altre parole si riesce, in questo modo, a modellare nel programma astratto un insieme di elementi di array del programma concreto con l'introduzione di una sola coppia di variabili, θ e a_θ. Intuitivamente, così facendo si riesce a modellare la proprietà che, sempre in riferimento al programma di tabella 2.6, per ogni valore di x, se $0 \leq x < 5$ allora a`[`x`]=`b`[`x`+1]`, e di conseguenza l'assert, non può fallire.

L'estensione del meccanismo di astrazione preesistente a quello descritto in questa sezione, oltre a migliorare l'efficienza di Eureka sotto gli aspetti di cui si è parlato nella sezione precedente, presenta il vantaggio di trattare gli elementi di un array in maniera più generica e non fissando le variabili a locazioni specifiche dell'array, preparando così il terreno per una futura estensione del tool alla verifica di programmi con puntatori come accennato nell'introduzione.

2.5 Semantica lineare astratta per programmi lineari con array

Come già detto, l'astrazione proposta è a livello semantico, il che significa che si vuole definire una semantica lineare astratta, ossia una nuova relazione di transizione su un dominio di variabili astratte che non contiene variabili array. La verifica di raggiungibilità avverrà sulle stringhe di statement del programma concreto, cui però viene data un'interpretazione diversa, che non coinvolge variabili array.

In questa sezione verrà definita dunque la relazione di transizione astratta calcolata in fase di astrazione. Prima però occorre definire le nozioni di valutazione astratta e di stato astratto. Infine, verrà fornita la dimostrazione di correttezza del meccanismo di astrazione proposto.

2.5.1 Astrazione $\hat{w}$ della funzione di valutazione concreta w

Sia w una funzione di valutazione sull'insieme di variabili di un programma concreto P, occorre definire un'analoga funzione di valutazione $\widehat{w}$ sul dominio di variabili astratte.

Quindi, si consideri l'astrazione di P rispetto alla classe di insiemi di indici $I = \{\theta(a) \mid a \in A_P\}$, si ha che per ogni funzione di valutazione concreta w sull'insieme di variabili $W \subseteq Var_P$ del programma P, esiste un insieme $\widehat{\Omega}(w)$ di valutazioni astratte tali che una generica valutazione $\widehat{w} \in \widehat{\Omega}(w)$ rappresenta una possibile astrazione di quella concreta w rispetto ad I.

Sia $\widehat{A_W} = \{a_\theta \mid a \in A_P \cap W \text{ e } \theta \in \theta(a)\}$, $\widehat{w}$ è definita come segue:

$\widehat{w} : (V_P \cap W) \cup \Theta \cup \widehat{A_W} \to \mathcal{D}$ tale che:

1. $\widehat{w}(v) = w(v)$ per ogni $v \in V_P \cap W$
2. $\widehat{w}(a_\theta) = w(a)(\widehat{w}(\theta))$ per ogni $a_\theta \in \widehat{A_W}$

Come si può notare, non è stato definito il comportamento che assume la funzione in corrispondenza delle variabili $\theta \in \Theta$, che possono assumere, dunque, un qualsiasi valore del dominio $\mathcal{D}$, fino al momento della loro inizializzazione con la relativa espressione e_θ. Comunque il vincolo sulle valutazioni delle variabili $\theta \in \Theta$ non viene catturato dalla relazione di transizione, che si limita a lasciare inalterata la valutazione di tali variabili nel corso della traccia di esecuzione, bensì verranno poi ristretti i percorsi lungo i quali la valutazione delle $\theta \in \Theta$ è quella desiderata attraverso le nozioni di valutazioni ammissibili e percorsi ammissibili. Proprio per questo motivo, per qualsiasi valutazione concreta w e per ogni classe di insiemi di indici I, esiste un insieme di valutazioni (anzichè un'unica valutazione) astratte, ognuna delle quali differisce dall'altra per il valore di almeno una variabile $\theta \in \Theta$.

2.5.2 Estensione della funzione di valutazione astratta $\hat{w}$ alla funzione $\bar{\hat{w}}$

Analogamente a quanto fatto per le valutazioni concrete, occorre estendere la funzione di valutazione di variabili astratte $\widehat{w}$ alla funzione astratta $\overline{\widehat{w}}$ di valutazione di espressioni.

Pertanto, si denoti con $\theta_1(a) = \{\widehat{w}(\theta) \mid \theta \in \theta(a)\}$ per ogni $a \in A_P$, per ogni valutazione astratta $\widehat{w} : (V_P \cap W) \cup \Theta \cup \widehat{A_W} \to \mathcal{D}$ si definisce la funzione

$\overline{\widehat{w}} : E_W \cup B_W \cup \Theta \cup \widehat{A_P} \to \mathcal{D}$ in base al tipo di espressione:

$$\overline{\widehat{w}}(e) = \begin{cases} \{e\} & \text{se } e = c \in \mathcal{D} \\ \{\widehat{w}(e)\} & \text{se } e = v \in (V_P \cap W) \cup \Theta \cup \widehat{A_P} \\ \{\widehat{w}(a_\theta) \mid \theta \in \theta(a) \text{ e } \widehat{w}(\theta) \in \overline{\widehat{w}}(e_1)\} & \text{se } e = a\texttt{[}e_1\texttt{]},\ \overline{\widehat{w}}(e_1) \subseteq \theta_1(a),\ a \in A_P \cap W,\ e_1 \in E_W \\ \{c \cdot d \mid d \in \overline{\widehat{w}}(e_1)\} & \text{se } e = c\texttt{*}e_1,\ c \in \mathcal{D},\ e_1 \in E_W \\ \{d_1\ op\ d_2 \mid d_1 \in \overline{\widehat{w}}(e_1) \text{ e } d_2 \in \overline{\widehat{w}}(e_2)\} & \text{se } e = e_1\ op\ e_2,\ e_1, e_2 \in E_W,\ op \in \{\texttt{+},\texttt{<},\texttt{<=},\texttt{>},\texttt{>=},\texttt{==},\texttt{!=}\} \\ \overline{\widehat{w}}(e_1) & \text{se } e = \texttt{(}b\texttt{?}e_1\texttt{:}e_2\texttt{)},\ b \in B_W,\ e_1, e_2 \in E_W,\ 0 \notin \overline{\widehat{w}}(b) \\ \overline{\widehat{w}}(e_2) & \text{se } e = \texttt{(}b\texttt{?}e_1\texttt{:}e_2\texttt{)},\ b \in B_W,\ e_1, e_2 \in E_W,\ \overline{\widehat{w}}(b) = \{0\} \\ \overline{\widehat{w}}(e_1) \cup \overline{\widehat{w}}(e_2) & \text{se } e = \texttt{(}b\texttt{?}e_1\texttt{:}e_2\texttt{)},\ b \in B_W,\ e_1, e_2 \in E_W,\ \{0, d\} \subseteq \overline{\widehat{w}}(b) \text{ per qualche } d \neq 0 \\ \mathcal{D} & \text{altrimenti} \end{cases}$$

La valutazione degli accessi ad array $a\texttt{[}e_1\texttt{]}$ avviene in maniera sostanzialmente diversa rispetto alla valutazione concreta $\overline{w}$. Vengono modellati nell'astrazione solo gli elementi dell'array indicizzati dalle variabili $\theta \in \theta(a)$, le cui valutazioni sono contenute nell'insime $\theta_1(a)$. Pertanto, se le possibili valutazioni per e_1 coincidono con valutazioni di qualche $\theta \in \theta(a)$ (condizione $\overline{\widehat{w}}(e_1) \subseteq \theta_1(a)$), allora un possibile valore per l'espressione sarà il valore della rispettiva a_θ, altrimenti se e_1 indicizza un elemento di a non modellato all'interno dell'astrazione ($\overline{\widehat{w}}(e_1) \not\subseteq \theta_1(a)$, trattato nell'ultimo caso della definizione), allora l'espressione ammette un qualsiasi valore scelto all'interno del dominio $\mathcal{D}$ in maniera non deterministica ($\overline{\widehat{w}}(e) = \mathcal{D}$), analogamente a come veniva valutato l'accesso fuori intervallo dalla funzione concreta $\overline{w}$.

Ad esempio, sia a un array di dimensione 10, sia $\theta(a) = \{\theta_1, \theta_2\}$, con $\theta_1 = x$, $\theta_2 = x + 1$ e $\widehat{w}(x) = 3$ e sia $\overline{\widehat{w}}(e_1) = \{3, 7\}$, allora l'espressione $\overline{\widehat{w}}(a\texttt{[}e_1\texttt{]}) = \mathcal{D}$ perchè e_1 può assumere un valore che indicizza un elemento, il settimo, che non è inserito nel dominio astratto, cui appartengono solo il

terzo e il quarto elemento di a. Se invece $\theta(a) = \{\theta_1, \theta_2, \theta_3\}$, con $\theta_3 = x + 4$, allora $\widehat{w}(a\,[e_1]) = \{\widehat{w}(a_{\theta_1}), \widehat{w}(a_{\theta_3})\}$

Come si può vedere, è molto forte l'analogia con l'estensione $\overline{w}$ della funzione di valutazione concreta w, ma si è dovuta definire una nuova nozione di estensione $\widehat{\cdot}$ per le valutazioni astratte innanzitutto perchè, come già illustrato, gli accessi ad array vengono trattati in maniera differente. Inoltre, anche il dominio di applicazione viene esteso in maniera diversa. Infatti, mentre nel caso concreto il dominio W rappresentante un insieme di variabili, veniva esteso al dominio $E_W \cup B_W$ di espressioni e espressioni booleane costruite utilizzando proprio l'insieme di variabili W, nel caso astratto il dominio della funzione di valutazione $\widehat{w}$ è l'insieme di variabili del programma astratto, mentre il dominio dell'estensione è l'insieme di espressioni (booleane e non) generate a partire dalle variabili concrete (insieme W) unito all'insieme Θ.

2.5.3 Scope delle variabili astratte

Visto che nell'astrazione sono state introdotte nuove variabili, vanno definiti gli insiemi, già definiti in sezione 2.1.4 per le variabili concrete, che tengono traccia degli scope delle variabili astratte.

Sia $i \in N_P$, si definiscono seguenti insiemi:

- $Globals_{\widehat{P}} = (Globals_P \cap V_P) \cup \Theta \cup \widehat{AG_P}$: insieme delle variabili globali dell'astrazione di P, con $\widehat{AG_P} = \{a_\theta \mid a \in A_P \cap Globals_P \text{ e } \theta \in \theta(a)\}$
- $Formals_{\widehat{P}}(i) = Formals_P(i)$: insieme dei parametri formali astratti della procedura contenente il nodo i
- $sLocals_{\widehat{P}}(i) = (V_P \cap sLocals_P(i)) \cup \widehat{A_P}(i)$: insieme delle variabili astratte strettamente locali alla procedura contenente il nodo i, con $\widehat{A_P}(i) = \{a_\theta \mid a \in A_P \cap sLocals_P(i) \text{ e } \theta \in \theta(a)\}$
- $Locals_{\widehat{P}}(i) = (V_P \cap Locals_P(i)) \cup \widehat{A_P}(i)$: insieme delle variabili astratte locali alla procedura contenente il nodo i
- $InScope_{\widehat{P}}(i) = Globals_{\widehat{P}} \cup Locals_{\widehat{P}}(i)$: insieme delle variabili astratte nello scope della procedura contenente il nodo i

Anche tra gli insiemi di variabili astratte valgono le stesse relazioni esistenti tra gli insiemi di variabili concrete menzionate in sezione 2.1.4.

2.5.4 Definizione di stato astratto

Prima di pensare alla definizione della relazione di transizione astratta tra stati, occorre definire la nozione di stato astratto sfruttando la funzione di valutazione astratta sul nuovo dominio di variabili appena definita. Sia $s = \langle i, w \rangle$ uno stato del programma concreto P, un possibile stato astratto $\widehat{s}$ dell'astrazione di P rispetto alla classe di insiemi di indici $\{\theta(a) \mid a \in A_P\}$ è rappresentato da una coppia ⟨nodo, funzione di valutazione astratta⟩, in simboli $\widehat{s} = \langle i, \widehat{w} \rangle$ dove $i \in N_P$ è lo stesso nodo dello stato concreto s e $\widehat{w} \in \widehat{\Omega}(w)$ rappresenta una possibile valutazione astratta $\widehat{w}$ di quella concreta w. Si fa notare il fatto che, proprio in funzione del fatto che l'astrazione di un programma concreto P preserva il flusso di controllo e quindi il CFG di P, il passaggio da uno stato concreto alla sua versione astratta non coinvolge in alcun modo il nodo ma solo la valutazione delle variabili.

Appare evidente, quindi, che per ogni stato concreto non esiste un unico, ma un insieme di stati astratti ad esso associati. Sia $s = \langle i, w \rangle$ uno stato concreto di P, l'insieme dei possibili stati astratti $\widehat{s}$ appartenenti all'astrazione di P rispetto alla classe di insiemi di indici $\{\theta(a) \mid a \in A_P\}$ si denota con $h(s) = \{\widehat{s} = \langle i, \widehat{w} \rangle \mid \widehat{w} \in \widehat{\Omega}(w)\}$

Si indica con $\widehat{S_P}$ l'insieme di stati astratti del programma P.

2.5.5 Relazione di transizione astratta e percorsi astratti

Per catturare formalmente come uno stato astratto $\widehat{s} = \langle i, \widehat{w} \rangle$ evolve in quello successivo, a seconda dello statement stm_i del programma concreto P associato al nodo i, si definisce una nuova semantica degli statement attraverso la definizione della relazione di transizione astratta $\widehat{\xrightarrow{\sigma}_P} \subseteq \widehat{S_P} \times \widehat{S_P}$, che cattura l'evoluzione del modello astratto ed è definita come segue a seconda del tipo di statement del programma P:

- stm_i : `skip` o `return;` allora

$$\langle i, \widehat{w_1} \rangle \widehat{\xrightarrow{\epsilon}_P} \langle sSucc_P(i), \widehat{w_1} \rangle$$

  Si ricorda che il modello astratto conserva lo stesso flusso e lo stesso CFG del concreto, pertanto si possono utlizzare le funzioni $Succ_P$, $sSucc_P$, $Fsucc_P$ e $Tsucc_P$ definite per il programma concreto.

- stm_i : **y=e;** allora

$$\langle i, \widehat{w_1} \rangle \widehat{\xrightarrow{\epsilon}_P} \langle sSucc_P(i), \widehat{w_1}[\mathbf{d}/\mathbf{y}] \rangle$$

per qualche $\mathbf{d} \in \overline{\widehat{w_1}}(\mathbf{e})$.

- $stm_i : a\texttt{[}e_1\texttt{]=}e_2\texttt{;}$ allora

$$\langle i, \widehat{w_1}\rangle \widehat{\xrightarrow{\epsilon}_P} \langle sSucc_P(i), \widehat{w_2}\rangle$$

con $\widehat{w_2} = \widehat{w_1}[d_1/a_{\theta_1^a}, \ldots, d_m/a_{\theta_m^a}]$ dove

$$\begin{array}{ll}
\theta(a) = \{\theta_1^a, \theta_2^a, \ldots, \theta_m^a\} & \\
d_i \in \overline{\widehat{w_1}}(e_2) & \text{se } \overline{\widehat{w_1}}(e_1\texttt{==}\theta_i^a) = \{1\} \\
d_i = \widehat{w_1}(a_{\theta_i^a}) & \text{se } \overline{\widehat{w_1}}(e_1\texttt{==}\theta_i^a) = \{0\} \\
d_i \in \overline{\widehat{w_1}}(e_2) \cup \{\widehat{w_1}(a_{\theta_i^a})\} & \text{altrimenti} \\
\text{con } 1 \leq i \leq m &
\end{array}$$

In un assegnamento ad un array a, bisogna aggiornare, con il valore dell'espressione e_2, le variabili astratte che modellano l'elemento dell'array che viene modificato dallo statement (quello indicizzato dall'espressione e_1) e lasciare invariate tutte le altre variabili astratte. Pertanto, si consideri una generica variabile $\theta_i^a \in \theta(a)$, se l'espressione e_1, che rappresenta l'indice dell'elemento dell'array a che viene assegnato nello statement, ammette una valutazione che coincide con quella di θ_i^a ($\overline{\widehat{w_1}}(e_1\texttt{==}\theta_i^a) = \{1\}$), allora una possibile valutazione per uno stato di arrivo della transizione assegna alla variabile $a_{\theta_i^a}$ uno dei possibili valori dell'espressione e_2 ($d_i \in \overline{\widehat{w_1}}(e_2)$). Se invece la valutazione dell'espressione e_1 non coincide con la valutazione della variabile θ_i^a ($\overline{\widehat{w_1}}(e_1\texttt{==}\theta_i^a) = \{0\}$), allora le possibili valutazioni degli stati di arrivo conincidono con quella dello stato di partenza sulla variabile $a_{\theta_i^a}$ ($d_i = \widehat{w_1}(a_{\theta_i^a})$). Infine, se la valutazione di e_1 ammette sia valori uguali alla valutazione di θ_i^a sia valori differenti, le valutazioni degli stati di arrivo possono sia lasciare inalterato il valore di $a_{\theta_i^a}$, che aggiornarlo con un valore possibile per e_2 ($d_i \in \overline{\widehat{w_1}}(e_2) \cup \{\widehat{w_1}(a_{\theta_i^a})\}$).

- stm_i : `if(`e`)`, `while(`e`)`, `assert(`e`);` o `assume(`e`);` allora

$$\langle i, \widehat{w_1}\rangle \widehat{\xrightarrow{\epsilon}_P} \langle i_1, \widehat{w_1}\rangle$$

dove

  - $i_1 = Fsucc_P(i)$ se $\{0\} = \overline{\widehat{w_1}}(e)$
  - $i_1 = Tsucc_P(i)$ se $0 \notin \overline{\widehat{w_1}}(e)$

– $i_1 \in Succ_P(i)$ se esiste $d \neq 0$ tale che $\{0, d\} \subseteq \widehat{\overline{w_1}}(e)$.

- $stm_i : pr\,(\mathbf{e})\,;$ allora

$$\langle i, \widehat{w_1}\rangle \xrightarrow{\widehat{\text{CALL}(RetPt_P(i), \widehat{w'})}}_P \langle First_P(pr), \widehat{w_2}\rangle$$

dove $\widehat{w'} : Locals_{\widehat{P}}(i) \to \mathcal{D}$ è tale che $\widehat{w'}(\mathbf{x}) = \widehat{w_1}(\mathbf{x})$, $\widehat{w_2}(\mathbf{y}) \in \widehat{\overline{w_1}}(\mathbf{e})$ e $\widehat{w_2}(\mathbf{g}) = \widehat{w_1}(\mathbf{g})$, con $\mathbf{x} = Locals_{\widehat{P}}(i)$, $\mathbf{y} = Formals_{\widehat{P}}(First_P(pr))$ e $\mathbf{g} = Globals_{\widehat{P}}$.

- $i = Exit_P(pr)$ allora

$$\langle i, \widehat{w_1}\rangle \xrightarrow{\widehat{\text{RET}(i_1, \widehat{w'})}}_P \langle i_1, \widehat{w_2}\rangle$$

dove $i_1 \in Succ_P(i)$, $\widehat{w_2}(\mathbf{g}) = \widehat{w_1}(\mathbf{g})$ e $\widehat{w_2}(\mathbf{x}) = \widehat{w'}(\mathbf{x})$ con $\mathbf{g} = Globals_{\widehat{P}}$ e $\mathbf{x} = Locals_{\widehat{P}}(i_1)$.

Si fa notare che la relazione di transizione astratta prevede che la valutazione delle variabili $\theta \in \Theta$ non venga mai modificata. Infatti, sebbene tali variabili vengano assegnate con un'opportuna espressione restituita dal processo di raffinamento in un opportuno punto della traccia di esecuzione del programma, computato anch'esso in fase di raffinamento, si assume per semplicità e chiarezza che tali variabili vengano inizializzate con l'opportuno valore nella prima istruzione del main. Si sottolinea comunque che tutti i concetti e teoremi enunciati nel corso di tale lavoro possono essere estesi al caso generale di assegnamento di una variabile $\theta \in \Theta$ in qualsiasi punto della traccia di esecuzione con un notevole appesantimento del formalismo. Pertanto, sebbene la relazione di transizione astratta catturi il vincolo per cui una variabile $\theta \in \Theta$ non modifica il proprio valore una volta che è stata opportunamente inizializzata nel primo statement del main, si sottolinea che il vincolo riguardante la valutazione iniziale delle stesse (il cui valore deve corrispondere a quello dell'espressione con cui vengono inizializzate) non è catturato da tale semantica, che ammette qualsiasi valutazione iniziale per tali variabili. L'inisieme di percorsi che rispettano tale vincolo sarà modellato con l'ausilio dalle nozioni di valutazioni ammissibili e percorsi ammissibili, definite nel corso del capitolo.

Si definisce di seguito la versione astratta $\widehat{post_P} : 2^{\widehat{S_P}} \to 2^{\widehat{S_P}}$ della funzione $post_P$ definita per il caso concreto. Tale funzione mette in relazione un insieme di stati astratti di partenza con l'insieme di tutti gli stati astratti che è possibile raggiungere attraverso l'applicazione della relazione di transizione astratta appena introdotta ed è definita come segue. Sia $\widehat{S} \subseteq \widehat{S_P}$:

$$\widehat{post_P}(\widehat{S}) = \{\widehat{s'} \mid \widehat{s} \xrightarrow{\widehat{\sigma}}_P \widehat{s'} \text{ e } \widehat{s} \in \widehat{S}\}$$

Analogamente al caso concreto si introduce la nozione di percorso astratto, che è definito essere una sequenza $\langle i_0, \widehat{w_0}\rangle \xrightarrow{\widehat{\sigma_1}}_P \langle i_1, \widehat{w_1}\rangle \xrightarrow{\widehat{\sigma_2}}_P \ldots \xrightarrow{\widehat{\sigma_n}}_P \langle i_n, \widehat{w_n}\rangle$ tale che $\langle i_k, \widehat{w_k}\rangle \xrightarrow{\widehat{\sigma_{k+1}}}_P \langle i_{k+1}, \widehat{w_{k+1}}\rangle$ con $k = 0, \ldots, n-1$[6].

Ancora una volta, si fa notare che non tutti i percorsi rappresentano possibili tracce di esecuzione, per il problema delle chiamate e ritorni da procedura. Infatti, analogamente al caso concreto, l'uscita da una procedura, alla cui chiamata corrisponde nella relazione di transizione un'etichetta del tipo $\textsc{call}(i, \widehat{w})$, deve corrispondere ad una transizione etichettata da $\textsc{ret}(i, \widehat{w})$, con il nodo e la funzione di valutazione che coincidono esattamente.

Pertanto, le nozioni di percorso valido dello stesso livello e percorso valido, definite in sezione 2.1.5 possono applicarsi anche ai percorsi astratti.

Un percorso astratto è detto inizializzato se e soltanto se il suo stato iniziale $\widehat{s_0}$ è nella forma $\langle First_P(main), \widehat{w_0}\rangle$ per qualche $\widehat{w_0}$.

A questo punto sorge, per i percorsi astratti, un ulteriore problema, generato dalla presenza delle variabili $\theta \in \Theta$, che, per definizione della funzione di valutazione astratta $\widehat{w}$ possono assumere un qualsiasi valore, sebbene l'utilizzo desiderato per tali variabili è quello di poterle inizializzare, in un determinato punto del programma (che si è supposto essere il primo nodo del main), con un'opportuna espressione e mai più modificarle.

Dunque, all'inizio del programma ogni variabile $\theta \in \Theta$ viene inizializzata assegnandola ad un'opportuna espressione $e_\theta \in E_{\Theta,W}$ (definito in sezione 2.1.1). Quindi bisogna selezionare, tra tutti i percorsi astratti validi e inizializzati, quelli che partono da una valutazione iniziale che attribuisca a ciascuna variabile $\theta \in \Theta$ un valore che coincide con quello relativo all'espressione usata per la sua inizializzazione.

Formalmente, sia $\Theta_{ass} : \Theta \to E_{\Theta,Var_P}$ una funzione iniettiva che cattura la corrispondenza tra una variabile $\theta \in \Theta$ e l'espressione con la quale viene inizializzata, si definiscono le nozioni di valutazione astratta ammissibile e percorso astratto inizializzato ammissibile.

L'insieme di valutazioni astratte ammissibili è definito essere l'insieme $\widehat{\Omega}_{\Theta_{ass}} = \{\widehat{w} \mid \{\widehat{w}(\theta)\} = \overline{\widehat{w}}(e_\theta), \theta \in \Theta \text{ e } e_\theta = \Theta_{ass}(\theta)\}$. Uno stato astratto $\widehat{s} = \langle i, \widehat{w}\rangle$ è detto ammissibile se $\widehat{w} \in \widehat{\Omega}_{\Theta_{ass}}$.

Un percorso astratto inizializzato è detto ammissibile se e soltanto se il suo stato iniziale $\widehat{s_0} = \langle First_P(main), \widehat{w_0}\rangle$ è uno stato ammissibile.

Da notare che l'espressione e_θ ammette sicuramente un'unica valutazione possibile e ciò è garantito dalla sua appartenenza all'insieme $E_{\Theta,W}$ e dalla definizione di quest'ultimo.

---

[6]Di qui in avanti l'espressione $\langle i_0, \widehat{w_0}\rangle \xrightarrow{\widehat{\sigma_1}}_P \langle i_1, \widehat{w_1}\rangle \xrightarrow{\widehat{\sigma_2}}_P \ldots \xrightarrow{\widehat{\sigma_n}}_P \langle i_n, \widehat{w_n}\rangle$ sarà semplificata con l'espressione $\langle i_0, \widehat{w_0}\rangle \xrightarrow{\widehat{\Sigma_0^n}}_P \langle i_n, \widehat{w_n}\rangle$, dove $\Sigma_0^n = \sigma_1\sigma_2\ldots\sigma_n$

Si definisce inoltre la nozione di semantica astratta di P, in simboli $\widehat{[\![P]\!]}$, come l'insieme di tutti i percorsi astratti validi inizializzati e ammissibili del programma P secondo la relazione di transizione astratta $\widehat{\xrightarrow{\sigma}_P}$.

Uno stato astratto $\langle i, \widehat{w}\rangle$ è detto raggiungibile in un'astrazione di P se e soltanto se esiste un percorso $\widehat{\pi} \in \widehat{[\![P]\!]}$ astratto valido inizializzato e ammissibile che raggiunge lo stato astratto $\langle i, \widehat{w}\rangle$. Un vertice $i \in N_P$ è raggiungibile in un'astrazione di P se e soltanto se esiste una valutazione $\widehat{w}$ tale che $\langle i, \widehat{w}\rangle$ è uno stato astratto raggiungibile nell'astrazione di P.

2.5.6 Funzioni di astrazione α e concretizzazione γ

Finalmente è possibile introdurre le funzioni di astrazione e di concretizzazione di insiemi di stati, sfruttando la funzione di astrazione di stati h, definita in precedenza.

Sia $S \subseteq S_P$ e $\widehat{S} \subseteq \widehat{S_P}$, si definiscono le funzioni di astrazione α e concretizzazione γ come segue:

$$\begin{aligned}
&\alpha : 2^{S_P} \to 2^{\widehat{S_P}} \text{ tale che} \\
&\quad \alpha(S) = \bigcup_{s \in S} h(s) \\
&\gamma : 2^{\widehat{S_P}} \to 2^{S_P} \text{ tale che} \\
&\quad \gamma(\widehat{S}) = \bigcup_{\widehat{s} \in \widehat{S}} \{s \in S_P \mid \widehat{s} \in h(s)\}
\end{aligned}$$

La coppia di funzioni $\langle \alpha, \gamma\rangle$ forma una Galois Connection $\langle 2^{S_P}, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^{\widehat{S_P}}, \subseteq\rangle$, di cui si ricorda la proprietà fondamentale:

$$\forall X \subseteq S_P, \forall Y \subseteq \widehat{S_P},\ \alpha(X) \subseteq Y \iff X \subseteq \gamma(Y)$$

2.6 Correttezza dell'astrazione proposta

Ora che si è data una definizione formale di come si ottiene la semantica astratta a partire dalla relazione di transizione concreta $\xrightarrow{\sigma}_P$ e da una classe di insiemi di indici I, verrà esibita la dimostrazione di correttezza di tale meccanismo, cioà verrà dimostrato che l'astrazione è conservativa.

Dire che un'astrazione è conservativa significa dire che l'insieme delle tracce di esecuzione del programma concreto rispetto alla relazione di transizione concreta è incluso nell'insieme di tracce di esecuzione eseguibili secondo la

semantica astratta, il che equivale a dire che l'astrazione dell'insieme S di stati concreti raggiungibili, secondo la relazione di transizione concreta, a partire da un arbitrario stato concreto s deve essere incluso nell'insieme di stati astratti raggiungibili da una delle possibili astrazioni di s, cioè da un generico $\widehat{s} \in h(s)$, in accordo con la relazione di transizione astratta.

In pratica, come schematizzato in figura 2.3, si vuol dimostrare che $\widehat{\xrightarrow{\sigma}_P}$ è un "upper approximation" di $\xrightarrow{\sigma}_P$, in simboli

$$post_P(S) \subseteq \gamma \circ \widehat{post_P} \circ \alpha(S)$$

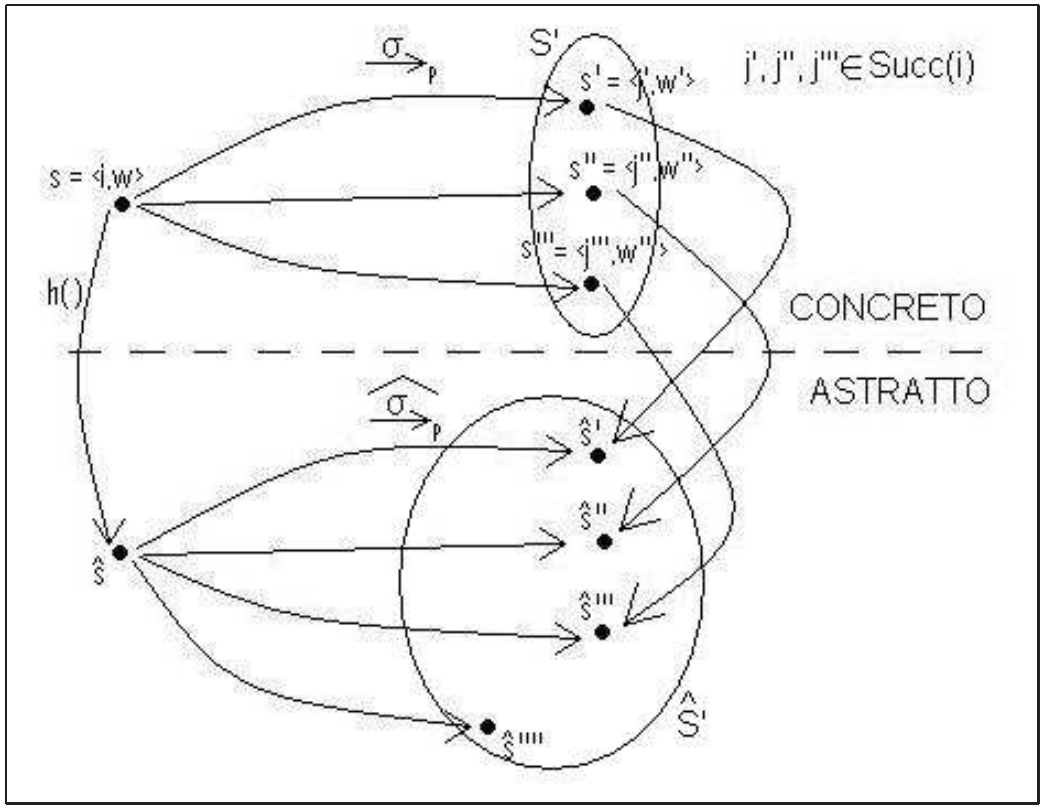


Figura 2.3: Schematizzazione della dimostrazione di soundness

Nel corso della dimostrazione di correttezza, verrà utilizzato più volte il seguente lemma:

Lemma 1 *$\overline{w}(e) \subseteq \overline{\widehat{w}}(e)$ per ogni valutazione w su $W \subseteq V_P \cup A_P$ e per ogni $\widehat{w} \in \widehat{\Omega}(w)$*

Dimostrazione La prova procede per induzione sulla struttura di e.

Caso base $e \in (V_P \cap W) \cup \mathcal{D} \cup \{\mathtt{u}\}$

- $e \in V_P \cap W$. $\widehat{w}(e) = w(e) \in \mathcal{D}$
  $\overline{w}(e) = \{w(e)\}$
  $\overline{\widehat{w}}(e) = \{\widehat{w}(e)\}$ quindi $\overline{\widehat{w}}(e) = \overline{w}(e)$
- $e \in \mathcal{D}$. $\overline{w}(e) = \{e\} = \overline{\widehat{w}}(e)$
- $e = \mathtt{u}$. $\overline{\widehat{w}}(e) = \mathcal{D}$ quindi $\overline{w}(e) \subseteq \mathcal{D} = \overline{\widehat{w}}(e)$

Passo induttivo Ipotesi induttiva: la tesi è valida per tutte le sotto-espressioni e' occorrenti in e. La dimostrazione procede per casi secondo la struttura di e:

1. $e = a[i]$. Si considerano 2 casi

   a) $\overline{\widehat{w}}(i) \subseteq \theta_1(a)$ Per ipotesi induttiva

   $$\overline{w}(i) \subseteq \overline{\widehat{w}}(i) \tag{2.1}$$

   $\overline{w}(e) = \{w(a)(d) \mid d \in \overline{w}(i)\}$ (definizione di $\bar{\cdot}$)
   Si prenda un generico elemento di $\overline{w}(e)$.
   Si consideri ad esempio l'elemento $w(a)(k)$ con un fissato $k \in \overline{w}(i)$
   Dall'ipotesi del caso a) e dalla definizione di $\bar{\widehat{\cdot}}$

   $$\overline{\widehat{w}}(e) = \{\widehat{w}(a_\theta) \mid \theta \in \theta(a) \wedge \widehat{w}(\theta) \in \overline{\widehat{w}}(i)\} \tag{2.2}$$

   Per l'ipotesi induttiva si ha che

   $$k \in \overline{\widehat{w}}(i) \tag{2.3}$$

   e per ipotesi del caso a)

   $$k \in \theta_1(a) = \{\widehat{w}(\theta) \mid \theta \in \theta(a)\} \tag{2.4}$$

   quindi

   $$\exists \overline{\theta} \in \theta(a) \mid k = \widehat{w}(\overline{\theta}) \tag{2.5}$$

   Quindi per sostituzione in 2.3

   $$\widehat{w}(\overline{\theta}) \in \overline{\widehat{w}}(i) \tag{2.6}$$

Da 2.2, 2.5 e 2.6 si ha che

$$\widehat{w}(a_{\overline{\theta}}) \in \overline{\widehat{w}}(e) \tag{2.7}$$

Ma della definizione di $\widehat{\cdot}$ si ha che $\widehat{w}(a_{\overline{\theta}}) = w(a)(\widehat{w}(\overline{\theta}))$
e dalla 2.5

$$w(a)(\widehat{w}(\overline{\theta})) = w(a)(k) \tag{2.8}$$

Quindi si conclude che $w(a)(k) = \widehat{w}(a_{\overline{\theta}}) \in \overline{\widehat{w}}(e)$

b) $\overline{\widehat{w}}(i) \not\subseteq \theta_1(a)$
Quindi ci deve essere un elemento $k \in \overline{\widehat{w}}(i) \wedge k \notin \theta_1(a)$
Quindi $k \in \overline{\widehat{w}}(i) \setminus \theta_1(a) \neq \emptyset$
Quindi (per definizione di $\overline{\widehat{\cdot}}$) $\overline{\widehat{w}}(e) = \mathcal{D}$
Infine $\overline{w}(e) \subseteq \mathcal{D} \Longrightarrow \overline{w}(e) \subseteq \overline{\widehat{w}}(e)$

2. $e = e_1\ op\ e_2$ con $op \in \{\texttt{+}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{==}, \texttt{!=}\}$.
$\overline{\widehat{w}}(e) = \{d_1\ op\ d_2 \mid d_1 \in \overline{\widehat{w}}(e_1) \text{ e } d_2 \in \overline{\widehat{w}}(e_2)\}$
$\overline{w}(e) = \{d_1\ op\ d_2 \mid d_1 \in \overline{w}(e_1) \text{ e } d_2 \in \overline{w}(e_2)\}$
Poichè $\overline{w}(e_1) \subseteq \overline{\widehat{w}}(e_1) \wedge \overline{w}(e_2) \subseteq \overline{\widehat{w}}(e_2)$ (per ipotesi induttiva) allora $\overline{w}(e) \subseteq \overline{\widehat{w}}(e)$

3. $e = c\texttt{*}e_1$.
$\overline{\widehat{w}}(e) = \{c \cdot d \mid d \in \overline{\widehat{w}}(e_1)\}$
$\overline{w}(e) = \{c \cdot d \mid d \in \overline{w}(e_1)\}$
Poichè $\overline{w}(e_1) \subseteq \overline{\widehat{w}}(e_1)$ (per ipotesi induttiva) allora $\overline{w}(e) \subseteq \overline{\widehat{w}}(e)$

4. $e = b\texttt{?}e_1\texttt{:}e_2$. Si distinguono 3 casi:

a) $\overline{w}(b) = \{0\}$
Dunque $\overline{w}(e) = \overline{w}(e_2)$ (definizione di $\overline{\cdot}$).
Per ipotesi induttiva valgono $\overline{w}(e_2) \subseteq \overline{\widehat{w}}(e_2)$ e $\overline{w}(b) \subseteq \overline{\widehat{w}}(b)$.
Quindi vale che $\overline{\widehat{w}}(b) = \{0\}$ oppure che $\{0, d\} \subseteq \overline{\widehat{w}}(b)$ per qualche $d \neq 0$ e di conseguenza $\overline{\widehat{w}}(e) = \overline{\widehat{w}}(e_2)$ oppure $\overline{\widehat{w}}(e) = \overline{\widehat{w}}(e_1) \cup \overline{\widehat{w}}(e_2)$, rispettivamente. In ogni caso, poichè $\overline{w}(e) = \overline{w}(e_2) \subseteq \overline{\widehat{w}}(e_2)$, vale che $\overline{w}(e) \subseteq \overline{\widehat{w}}(e)$.

b) $0 \notin \overline{w}(b)$, ovvero $\overline{w}(b) \subseteq \mathcal{D} \setminus \{0\}$
Dunque $\overline{w}(e) = \overline{w}(e_1)$ (definizione di $\overline{\cdot}$).
Per ipotesi induttiva valgono $\overline{w}(e_1) \subseteq \overline{\widehat{w}}(e_1)$ e $\overline{w}(b) \subseteq \overline{\widehat{w}}(b)$.

Quindi vale che $\overline{\widehat{w}}(b) \subseteq \mathcal{D} \setminus \{0\}$ oppure che $\{0, d\} \subseteq \overline{\widehat{w}}(b)$ per qualche $d \neq 0$ e di conseguenza $\overline{\widehat{w}}(e) = \overline{\widehat{w}}(e_1)$ oppure $\overline{\widehat{w}}(e) = \overline{\widehat{w}}(e_1) \cup \overline{\widehat{w}}(e_2)$, rispettivamente. In ogni caso, poichè $\overline{w}(e) = \overline{w}(e_1) \subseteq \overline{\widehat{w}}(e_1)$, vale che $\overline{w}(e) \subseteq \overline{\widehat{w}}(e)$.

c) $\{0, d\} \subseteq \overline{w}(b)$

Dunque $\overline{w}(e) = \overline{w}(e_1) \cup \overline{w}(e_2)$ (definizione di $\bar{\cdot}$).

Per ipotesi induttiva valgono $\overline{w}(e_1) \subseteq \overline{\widehat{w}}(e_1)$, $\overline{w}(e_2) \subseteq \overline{\widehat{w}}(e_2)$ e $\overline{w}(b) \subseteq \overline{\widehat{w}}(b)$.

Quindi vale anche $\{0, d\} \subseteq \overline{\widehat{w}}(b)$ per qualche $d \neq 0$ e di conseguenza $\overline{\widehat{w}}(e) = \overline{\widehat{w}}(e_1) \cup \overline{\widehat{w}}(e_2)$. Pertanto, dall'ipotesi induttiva segue $\overline{w}(e) \subseteq \overline{\widehat{w}}(e)$.

□

Dalle definizioni delle funzioni α, γ, $post_P$ e $\widehat{post_P}$ si ricavano i seguenti corollari:

Corollario 1 *Siano $S_1, S_2 \in 2^{S_P}$ tali che $S_1 \subseteq S_2$ e siano $\widehat{S}_1, \widehat{S}_2 \in 2^{\widehat{S_P}}$ tali che $\widehat{S}_1 \subseteq \widehat{S}_2$, valgono le seguenti relazioni:*

1. $\alpha(S_1) \subseteq \alpha(S_2)$
2. $post_P(S_1) \subseteq post_P(S_2)$
3. $\gamma(\widehat{S}_1) \subseteq \gamma(\widehat{S}_2)$
4. $\widehat{post_P}(\widehat{S}_1) \subseteq \widehat{post_P}(\widehat{S}_2)$

Corollario 2 *Siano $S_1, S_2 \in 2^{S_P}$ e siano $\widehat{S}_1, \widehat{S}_2 \in 2^{\widehat{S_P}}$, valgono le seguenti relazioni:*

1. $\alpha(S_1) \cup \alpha(S_2) = \alpha(S_1 \cup S_2)$
2. $post_P(S_1) \cup post_P(S_2) = post_P(S_1 \cup S_2)$
3. $\gamma(\widehat{S}_1) \cup \gamma(\widehat{S}_2) = \gamma(\widehat{S}_1 \cup \widehat{S}_2)$
4. $\widehat{post_P}(\widehat{S}_1) \cup \widehat{post_P}(\widehat{S}_2) = \widehat{post_P}(\widehat{S}_1 \cup \widehat{S}_2)$

2.6.1 Teorema di correttezza dell'astrazione

E' finalmente possibile, utilizzando definizioni e concetti introdotti nel corso di tale capitolo, esibire la dimostrazione che il meccanismo di astrazione proposto nel presente lavoro di tesi è corretto, e cioè che l'astrazione da esso prodotta è conservativa. Con riferimento alla fig. 2.3, si vuole dimostrare che $S' \subseteq \gamma(\widehat{S}')$

Dal seguente teorema è possibile ricavare la correttezza dell'astrazione proposta:

Teorema 1 *Per ogni stato $s \in S_P$ e per ogni stato astratto $\widehat{s} \in \alpha(\{s\})$, vale che $post_P(\{s\}) \subseteq \gamma \circ \widehat{post_P}(\{\widehat{s}\})$*[7]

Dimostrazione Sia $s = \langle i, w_1 \rangle \in S_P$ un arbitrario stato del programma concreto P e sia $\widehat{s} = \langle i, \widehat{w_1} \rangle$ un arbitrario stato astratto tale che $\widehat{s} \in h(s)$ per qualche $\widehat{w_1} \in \widehat{\Omega}(w_1)$. Si noti inoltre che per ogni $s \in S_P$ in base alla definizione di α, vale che $\alpha(s) = \bigcup_{\overline{s} \in \{s\}} h(\overline{s}) = h(s)$. Vale dunque che $\widehat{s} \in h(s) = \alpha(s)$ e, per la definizione di γ si ha che $s \in \gamma(\widehat{s})$. La prova procede per casi sul tipo di statement stm_i corrispondente al nodo i:

- stm_i : `skip` o `return;`. Dalla definizione di relazione di transizione concreta si ha che

  $$\langle i, w_1 \rangle \xrightarrow{\epsilon}_P \langle sSucc_P(i), w_1 \rangle$$

  quindi, dalla definizione di $post_P$ si ha che

  $$post_P(s) = \{s' = \langle sSucc_P(i), w_1 \rangle\}$$

  Dalla relazione di transizione astratta si ha invece

  $$\langle i, \widehat{w_1} \rangle \widehat{\xrightarrow{\epsilon}_P} \langle sSucc_P(i), \widehat{w_1} \rangle$$

  e di conseguenza

  $$\widehat{post_P}(\widehat{s}) = \{\widehat{s'} = \langle sSucc_P(i), \widehat{w_1} \rangle\}$$

  Poichè $\widehat{w_1} \in \widehat{\Omega}(w_1)$, allora $\widehat{s'} \in h(s')$ e di conseguenza $s' \in \gamma(\widehat{s'})$. Da ciò segue che $s' \in \gamma \circ \widehat{post_P}(\widehat{s})$ e cioè $\{s'\} = post_P(s) \subseteq \gamma \circ \widehat{post_P}(\widehat{s})$.

[7] Nel corso della dimostrazione, per facilitarne la lettura e la comprensione, gli insiemi costituiti da un solo elemento utilizzati come argomento di funzioni saranno denotati semplicemente dall'elemento che li compone. Quindi le espressioni $post_P(\{s\})$, $\gamma(\{s\})$, ecc. saranno denotate semplicemente con $post_P(s)$ e $\gamma(s)$, rispettivamente

- stm_i : **x=e;** dove **x** è una tupla di variabili scalari e **e** una tupla di espressioni lineari con array. In base alla relazione di transizione concreta si ha che

  $$\langle i, w_1\rangle \xrightarrow{\epsilon}_P \langle sSucc_P(i), w_2\rangle \text{ dove } w_2 \in \Omega' = \{w_1[\mathbf{d}/\mathbf{x}] \mid \mathbf{d} \in \overline{w_1}(\mathbf{e})\}$$

  quindi, per la definizione della funzione $post_P$, si afferma che

  $$post_P(s) = \{\langle sSucc_P(i), w_2\rangle \mid w_2 \in \Omega'\}$$

  Si prenda un arbitrario elemento di $post_P(s)$ e lo si denoti con $s' = \langle sSucc_P(i), w'\rangle$, con $w' \in \Omega'$, cioè $w' = w_1[\mathbf{d}/\mathbf{x}]$, ovvero w' differisce da w_1 solo per le valutazioni $w'(\mathbf{x}) = \mathbf{d}$ per qualche $\mathbf{d} \in \overline{w_1}(\mathbf{e})$.

  Dalla relazione di transizione astratta si ha che

  $$\langle i, \widehat{w_1}\rangle \widehat{\xrightarrow{\epsilon}_P} \langle sSucc_P(i), \widehat{w_2}\rangle \text{ dove } \widehat{w_2} \in \widehat{\Omega}' = \{\widehat{w_1}[\mathbf{d}/\mathbf{x}] \mid \mathbf{d} \in \overline{\widehat{w_1}}(\mathbf{e})\}$$

  e dalla definizione della funzione $\widehat{post_P}$, si afferma che

  $$\widehat{post_P}(\widehat{s}) = \{\langle sSucc_P(i), \widehat{w_2}\rangle \mid \widehat{w_2} \in \widehat{\Omega}'\}$$

  Si consideri di nuovo $s' = \langle sSucc_P(i), w'\rangle$ con $w' \in \Omega'$.

  Vale che $\alpha(s') = h(s')$.

  Notare, inoltre, che per ogni $w \in \Omega'$ esiste una valutazione $\widehat{w} \in \widehat{\Omega}(w)$ tale che $\widehat{w}(\theta) = \widehat{w_1}(\theta)$ per ogni $\theta \in \Theta$.

  Si prenda proprio una tale $\widehat{w'} \in \widehat{\Omega}(w')$ tale che $\widehat{w'}(\theta) = \widehat{w_1}(\theta)$ per ogni $\theta \in \Theta$.

  Poichè $\widehat{w'} \in \widehat{\Omega}(w')$, allora $\widehat{s'} = \langle sSucc_P(i), \widehat{w'}\rangle \in h(s') = \alpha(s')$.

  Quindi

  $$s' \in \gamma(\widehat{s'}) \tag{2.9}$$

  Poichè w' differisce da w_1 solo per $w'(\mathbf{x}) = \mathbf{d}$ per qualche $\mathbf{d} \in \overline{w_1}(e)$ e poichè abbiamo scelto $\widehat{w'}$ in modo tale che $\widehat{w'}(\theta) = \widehat{w_1}(\theta)$ per ogni $\theta \in \Theta$, allora anche $\widehat{w'}$ differisce da $\widehat{w_1}$ solo per $\widehat{w'}(\mathbf{x}) = \mathbf{d}$ per qualche $\mathbf{d} \in \overline{w_1}(\mathbf{e}) \subseteq \overline{\widehat{w_1}}(\mathbf{e})$

  Quindi $\widehat{w'} = \widehat{w_1}[\mathbf{d}/\mathbf{x}]$ con $\mathbf{d} \in \overline{w_1}(\mathbf{e}) \subseteq \overline{\widehat{w_1}}(\mathbf{e})$, quindi $\widehat{w'} \in \widehat{\Omega}'$ e dunque $\widehat{s'} = \langle sSucc_P(i), \widehat{w'}\rangle \in \widehat{post_P}(\widehat{s})$.

  Vale allora, per l'equazione precedente

  $$\{\widehat{s'}\} \subseteq \widehat{post_P}(\widehat{s})$$

e quindi dal corollario 1 e dall'equazione precedente vale

$$\gamma(\widehat{s'}) \subseteq \gamma \circ \widehat{post_P}(\widehat{s})$$

Quindi dall'equazione precedente e dalla 2.9 vale che

$$s' \in \gamma(\widehat{s'}) \subseteq \gamma \circ \widehat{post_P}(\widehat{s})$$

e quindi

$$s' \in \gamma \circ \widehat{post_P}(\widehat{s}) \tag{2.10}$$

Poichè s' rappresenta un arbitrario elemento di $post_P(s)$, allora la 2.10 vale per ogni elemento di $post_P(s)$, pertanto si conclude che

$$post_P(s) \subseteq \gamma \circ \widehat{post_P}(\widehat{s})$$

- $stm_i : a\texttt{[}j\texttt{]=}e\texttt{;}$. Dalla relazione di transizione concreta si ha che

  $$\langle i, w_1\rangle \xrightarrow{\epsilon}_P \langle sSucc_P(i), w_2\rangle$$

  dove $w_2 \in \Omega' = \{w'' = w_1[(w_1(a)[d/k])/a] \mid k \in \overline{w_1}(j) \text{ e } d \in \overline{w_1}(e)\}$

  Si scelga, quindi, un'arbitraria valutazione $w'' \in \Omega'$, cui corrisponde lo stato arbitrario $s'' = \langle sSucc_P(i), w''\rangle \in post_P(s)$

  Quindi si ha $w'' = w_1[w_1(a)[d/k]/a]$ con un fissato $k \in \overline{w_1}(j)$ ed un fissato $d \in \overline{w_1}(e)$

  Si considerano i due casi:

  1. $k \in \theta_1(a)$, cioè esiste $\overline{\theta} \in \theta(a) \mid k = \widehat{w_1}(\overline{\theta})$. Poichè per il lemma 1 $\overline{w_1}(j) \subseteq \overline{\widehat{w_1}}(j)$, allora sia ha anche che $k \in \overline{\widehat{w_1}}(j)$.
     Quindi $\{1\} \subseteq \overline{\widehat{w_1}}(j\texttt{==}\overline{\theta})$.
     Si consideri la relazione di transizione astratta:

     $$\langle i, \widehat{w_1}\rangle \widehat{\xrightarrow{\epsilon}_P} \langle sSucc_P(i), \widehat{w_2}\rangle$$

     per qualche $\widehat{w_2} \in \Omega_1 = \{\widehat{w_2} = \widehat{w_1}[d_1/a_{\theta_1^a}, \ldots, d_m/a_{\theta_m^a}]$ dove

     $$\begin{array}{ll}
     \theta(a) = \{\theta_1^a, \theta_2^a, \ldots, \theta_m^a\} & \\
     d_i \in \overline{\widehat{w_1}}(e) & \text{se } \overline{\widehat{w_1}}(j\texttt{==}\theta_i^a) = \{1\} \\
     d_i = \widehat{w_1}(a_{\theta_i^a}) & \text{se } \overline{\widehat{w_1}}(j\texttt{==}\theta_i^a) = \{0\} \\
     d_i \in \overline{\widehat{w_1}}(e) \cup \{\widehat{w_1}(a_{\theta_i^a})\} & \text{altrimenti} \\
     \text{con } 1 \leq i \leq m\} &
     \end{array}$$

Si consideri, quindi, l'insieme di valutazioni Ω_2 così definito:

$$\Omega_2 = \{\widehat{w_1}[d/a_{\theta_1^+}, \ldots, d/a_{\theta_r^+}] \mid d \in \overline{\widehat{w_1}}(e)\}$$

con $\theta^+ = \{\theta_1^+, \ldots, \theta_r^+\} = \{\theta \in \theta(a) \mid \widehat{w_1}(\theta) = k\}$

Intuitivamente appartengono a Ω_2 tutte le valutazioni di Ω_1 che assegnano alle variabili a_θ tale $\widehat{w}(\theta) = k$ un valore $d \in \overline{\widehat{w_1}}(e)$. E' evidente che $\Omega_2 \subseteq \Omega_1$.

Inoltre, per come è definito, θ^+ è sicuramente non vuoto, infatti $\overline{\theta} \in \theta^+$ in quanto $\widehat{w_1}(\overline{\theta}) = k$.

Si fa notare che w'' differisce da w_1 solo per $w''(a)(k) = d$ con $d \in \overline{w_1}(e)$ fissato.

Si scelga un $\widehat{w''} \in \widehat{\Omega}(w'')$ tale che $\widehat{w''}(\theta) = \widehat{w_1}(\theta)$ per ogni $\theta \in \Theta$.

Dunque, poichè $\widehat{w''}(\theta) = \widehat{w_1}(\theta)$ per ogni $\theta \in \Theta$, anche $\widehat{w''}$ differisce da $\widehat{w_1}$ solo per $\widehat{w''}(a_\theta) = d$ per ogni a_θ tale che $\theta \in \theta(a)$ e $\widehat{w''}(\theta) = k$ e con $d \in \overline{w_1}(e) \subseteq \overline{\widehat{w_1}}(e)$ (per il lemma 1).

Ma allora $\widehat{w''}$ e $\widehat{w_1}$ differiscono per tutte le variabili a_θ tale che $\theta \in \theta^+$ poichè $\widehat{w''}(\theta) = \widehat{w_1}(\theta)$ per ogni $\theta \in \Theta$, quindi $\widehat{w''} \in \Omega_2 \subseteq \Omega_1$. Quindi $\widehat{w''} \in \Omega_1$ e quindi $\widehat{s''} = \langle sSucc_P(i), \widehat{w''}\rangle \in \widehat{post_P}(\widehat{s})$, da cui si ricava che

$$\{\widehat{s''}\} \subseteq \widehat{post_P}(\widehat{s}) \tag{2.11}$$

Inoltre, poichè $\widehat{s''} = \langle sSucc_P(i), \widehat{w''}\rangle$ e poichè $\widehat{w''} \in \widehat{\Omega}(w'')$, allora $\widehat{s''} \in h(\langle sSucc_P(i), w''\rangle) = \alpha(\langle sSucc_P(i), w''\rangle)$, con $\langle sSucc_P(i), w''\rangle = s'' \in post_P(s)$.

Quindi $s'' \in \gamma(\widehat{s''})$ dalla definizione della funzione γ.

Inoltre, dalla 2.11 e dal corollario 1 si ricava $\gamma(\widehat{s''}) \subseteq \gamma \circ \widehat{post_P}(\widehat{s})$ e dalle due relazioni precedenti si ottiene $s'' \in \gamma(\widehat{s''}) \subseteq \gamma \circ \widehat{post_P}(\widehat{s})$.

Poichè s'' è un elemento arbitraro di $post_P(s)$, allora la relazione precedente vale per tutti gli elementi $s' \in post_P(s)$, pertanto si arriva alla conclusione:

$$post_P(s) \subseteq \gamma \circ \widehat{post_P}(\widehat{s})$$

2. $k \notin \theta_1(a)$, allora per ogni $k' \in \theta_1(a), w''(a)(k') = w_1(a)(k')$. Poichè $\overline{w_1}(j) \subseteq \overline{\widehat{w_1}}(j)$ abbiamo anche che $k \in \overline{\widehat{w_1}}(j)$. Quindi per ogni $\theta \in \theta(a), 0 \in \overline{\widehat{w_1}}(j{=}{=}\theta)$.

   Pertanto $\widehat{w_1} \in \Omega_1$ e $\widehat{s'} = \langle sSucc_P(i), \widehat{w_1}\rangle \in \widehat{post_P}(\widehat{s})$.

Sia $\widehat{w''} \in \widehat{\Omega}(w'')$ tale che $\widehat{w''}(\theta) = \widehat{w_1}(\theta)$ per ogni $\theta \in \Theta$, $\widehat{s''} = \langle sSucc_P(i), \widehat{w''} \rangle \in h(s'') = \alpha(s'')$, con $s'' = \langle sSucc_P(i), w'' \rangle \in post_P(s)$. Pertanto per la definizione di γ, si ha che

$$s'' \in \gamma(\widehat{s''}) \tag{2.12}$$

Poichè w_1 e w'' differiscono al più soltanto per il valore di un elemento di array non appartenente a $\theta_1(a)$, allora vale che $\widehat{w''} = \widehat{w_1} \in \Omega_1$, dunque anche $\widehat{s''} = \widehat{s'} \in \widehat{post_P}(\widehat{s})$.

Allora si ha che $\{\widehat{s''}\} \subseteq \widehat{post_P}(\widehat{s})$ e dall'applicazione del corollario 1 si ottiene $\gamma(\widehat{s''}) \subseteq \gamma \circ \widehat{post_P}(\widehat{s})$ e dall'ultima equazione e dalla 2.12 si ha che $s'' \in \gamma \circ \widehat{post_P}(\widehat{s})$.

Ancora una volta, poichè s'' è un elemento arbitraro di $post_P(s)$, allora la relazione vale per tutti gli elementi $s' \in post_P(s)$, pertanto si arriva alla conclusione:

$$post_P(s) \subseteq \gamma \circ \widehat{post_P}(\widehat{s})$$

- stm_i : `if(`e`)`, `while(`e`)`, `assert(`e`);` o `assume(`e`);` dove e è un'espressione lineare booleana con array. In accordo con la definizione della relazione di transizione distinguiamo 3 casi:

  1. $\{0, d\} \subseteq \overline{w_1}(e)$, con $d \neq 0$, allora $\langle i, w_1 \rangle \xrightarrow{\epsilon}_P \langle i', w_1 \rangle$, dove $i' \in Succ_P(i)$. Poichè $\overline{w_1}(e) \subseteq \overline{\widehat{w_1}}(e)$, per il lemma 1, allora vale anche $\{0, d\} \subseteq \overline{\widehat{w_1}}(e)$, con $d \neq 0$. Quindi $\langle i, \widehat{w_1} \rangle \widehat{\xrightarrow{\epsilon}_P} \langle i', \widehat{w_1} \rangle$, dove $i' \in Succ_P(i)$. Pertanto per ogni $s' \in post_P(s)$ esiste un $\widehat{s'} \in h(s')$ tale che $\widehat{s'} \in \widehat{post_P}(\widehat{s})$ e quindi si ha che $\{\widehat{s'}\} \subseteq \widehat{post_P}(\widehat{s})$ da cui, per il corollario 1 si ha che

  $$\gamma(\widehat{s'}) \subseteq \gamma \circ \widehat{post_P}(\widehat{s}) \tag{2.13}$$

  Poichè $\widehat{s'} \in h(s') = \alpha(s')$, allora $s' \in \gamma(\widehat{s'})$ e da ciò e dalla 2.13 segue $s' \in \gamma \circ \widehat{post_P}(\widehat{s})$. Poichè tali relazioni valgono per ogni $s' \in post_P(s)$, allora si arriva alla conclusione:

  $$post_P(s) \subseteq \gamma \circ \widehat{post_P}(\widehat{s})$$

  2. $\overline{w_1}(e) = \{0\}$. Allora $\langle i, w_1 \rangle \xrightarrow{\epsilon}_P \langle Fsucc_P(i), w_1 \rangle$. Poichè $\overline{w_1}(e) \subseteq \overline{\widehat{w_1}}(e)$ (lemma 1) allora ci sono due possibili alternative: $\overline{\widehat{w_1}}(e) = \{0\}$ oppure $\{0, d\} \subseteq \overline{\widehat{w_1}}(e)$, con $d \neq 0$. Quindi $\langle i, \widehat{w_1} \rangle \widehat{\xrightarrow{\epsilon}_P} \langle i', \widehat{w_1} \rangle$, dove, a seconda dei casi, $i' = Fsucc_P(i)$ oppure $i' \in Succ_P(i)$. In entrambi i casi la tesi segue immediatamente.

3. $\overline{w_1}(e) \subseteq (\mathcal{D} \setminus \{0\})$. Allora $\langle i, w_1\rangle \xrightarrow{\epsilon}_P \langle Tsucc_P(i), w_1\rangle$. La dimostrazione è analoga a quella del caso precedente.

- stm_i : $pr(\mathbf{e})$;, allora $\langle i, w_1\rangle \xrightarrow{\text{CALL}(RetPt_P(i),w')}_P \langle i_2 = First_P(pr), w_2\rangle$ è una transizione di stati da $\langle i, w_1\rangle$ con $w'(\mathbf{x}) = w_1(\mathbf{x})$, $w_2(\mathbf{y}) = \mathbf{d}$ per qualche $\mathbf{d} \in \overline{w_1}(\mathbf{e})$ e $w_2(\mathbf{g}) = w_1(\mathbf{g})$, con $\mathbf{x} = Locals_P(i)$, $\mathbf{y} = Formals_P(i_2)$ e $\mathbf{g} = Globals_P$.

  Si prenda un arbitrario stato $s' = \langle i_2, w_2\rangle \in post_P(s)$ e si consideri lo stato astratto $\widehat{s'} = \langle i_2, \widehat{w_2}\rangle \in h(s') = \alpha(s')$, con $\widehat{w_2} \in \widehat{\Omega}(w_2)$ e tale che $\widehat{w_2}(\theta) = \widehat{w_1}(\theta)$ per ogni $\theta \in \Theta$. Poichè $\widehat{s'} \in h(s')$, allora, dalla definizione della funzione γ vale che

$$s' \in \gamma(\widehat{s'}) \tag{2.14}$$

  E' immediato verificare che $\widehat{w_2}(\mathbf{g}) = \widehat{w_1}(\mathbf{g})$ e $\widehat{w'}(\mathbf{x}) = \widehat{w_1}(\mathbf{x})$, con $\mathbf{g} = Globals_{\widehat{P}}$ e $\mathbf{x} = Locals_{\widehat{P}}(i)$ e con $\widehat{w'} \in \widehat{\Omega}(w')$. Inoltre, poichè $\overline{w_1}(\mathbf{e}) \subseteq \overline{\widehat{w_1}}(\mathbf{e})$, allora vale anche $\mathbf{d} \in \overline{\widehat{w_1}}(\mathbf{e})$. Quindi $\widehat{w_2}(\mathbf{y}) = w_2(\mathbf{y}) = \mathbf{d}$ per qualche $\mathbf{d} \in \overline{\widehat{w_1}}(\mathbf{e})$, con $\mathbf{y} = Formals_{\widehat{P}}(i_2)$.

  Pertanto $\langle i, \widehat{w_1}\rangle \xrightarrow{\widehat{\text{CALL}(RetPt_P(i),w')}}_P \langle i_2, \widehat{w_2}\rangle$ è una transizione astratta di stati. Pertanto $\widehat{s'} \in \widehat{post_P}(\widehat{s})$ e cioè $\{\widehat{s'}\} \subseteq \widehat{post_P}(\widehat{s})$. Dalla precedente e dal corollario 1 si ha che

$$\gamma(\widehat{s'}) \subseteq \gamma \circ \widehat{post_P}(\widehat{s}) \tag{2.15}$$

  Da 2.14 e da 2.15 si conclude che

$$s' \in \gamma \circ \widehat{post_P}(\widehat{s})$$

  Poichè s' è scelto arbitrariamente dall'insieme $post_P(s)$, allora vale che

$$post_P(s) \subseteq \gamma \circ \widehat{post_P}(\widehat{s})$$

- $i = Exit_P(pr)$ allora $\langle i, w_1\rangle \xrightarrow{\text{RET}(i_1,w')}_P \langle i_1, w_2\rangle$ è una transizione di stati di P. Dalla definizione della relazione di transizione abbiamo che $w_2(\mathbf{g}) = w_1(\mathbf{g})$ e $w_2(\mathbf{x}) = w'(\mathbf{x})$, con $\mathbf{g} = Globals_P$ e $\mathbf{x} = Locals_P(i_1)$. Si consideri uno stato arbitrario $s' = \langle i_1, w_2\rangle \in post_P(s)$.

  Si consideri, inoltre, l'astrazione $\widehat{s'} = \langle i_1, \widehat{w_2}\rangle \in h(s') = \alpha(s')$, con $\widehat{w_2} \in \widehat{\Omega}(w_2)$ e tale che $\widehat{w_2}(\theta) = \widehat{w_1}(\theta)$ per ogni $\theta \in \Theta$. Analogamente al caso

precedente è immediato verificare che $\widehat{w_2}(\mathbf{g}) = \widehat{w_1}(\mathbf{g})$ e $\widehat{w_2}(\mathbf{x}) = \widehat{w'}(\mathbf{x})$ con $\mathbf{g} = Globals_{\widehat{P}}$ e $\mathbf{x} = Locals_{\widehat{P}}(i_1)$ e con $\widehat{w'} \in \widehat{\Omega}(w')$.

Quindi $\langle i, \widehat{w_1}\rangle \xrightarrow{\widehat{\mathrm{RET}(i_1, w')}}_P \langle i_1, \widehat{w_2}\rangle$ è una transizione astratta di stati. La tesi segue in maniera analoga al caso precedente.

□

Dal teorema 1 e dal corollario 2 è possibile ricavare il seguente corollario

Corollario 3 *Per ogni $S \in 2^{S_P}$, vale che*

$$post_P(S) \subseteq \gamma \circ \widehat{post_P} \circ \alpha(S)$$

Infine, il seguente teorema garantisce che l'astrazione è conservativa

Teorema 2 *Sia $\pi = s_0 \xrightarrow{\sigma_1}_P s_1 \xrightarrow{\sigma_2}_P s_2 \xrightarrow{\sigma_3}_P \ldots \xrightarrow{\sigma_n}_P s_n$ un percorso valido inizializzato di P, ovvero $\pi \in [\![P]\!]$, allora esiste un percorso astratto valido inizializzato ammissibile $\widehat{\pi} \in \widehat{[\![P]\!]}$, tale che $\widehat{\pi} = \widehat{s_0} \xrightarrow{\widehat{\sigma_1}}_P \widehat{s_1} \xrightarrow{\widehat{\sigma_2}}_P \widehat{s_2} \xrightarrow{\widehat{\sigma_3}}_P \ldots \xrightarrow{\widehat{\sigma_n}}_P \widehat{s_n}$ e tale che $\widehat{s_i} \in h(s_i)$ per ogni $i \in \{0, 1, 2, \ldots, n\}$*

Dimostrazione La dimostrazione procede per induzione sulla lunghezza del percorso π.

Caso base Se $\pi = s_0 \xrightarrow{\sigma_1}_P s_1$ è un percorso valido inizializzato di P di lunghezza 1, il teorema segue immediatamente dal teorema 1. Infatti dal teorema 1, si ha che per una qualsiasi astrazione $\widehat{s_0} \in h(s_0)$ e per ogni $s_1 \in post_P(s_0)$, esiste uno stato $\widehat{s_1} \in \widehat{post_P}(\widehat{s_0})$ tale che $\widehat{s_1} \in h(s_1)$ e pertanto il percorso $\widehat{\pi} = \widehat{s_0} \xrightarrow{\widehat{\sigma_1}}_P \widehat{s_1}$ è un percorso astratto valido inizializzato.

Se si sceglie, tra le astrazioni di s_0, lo stato astratto $\widehat{s_0} = \langle First_P(main), \widehat{w_0}\rangle$ tale che $\widehat{w_0}$ è una valutazione ammissibile, allora il percorso $\widehat{\pi}$ è un percorso astratto valido inizializzato e ammissibile.

Passo induttivo Ipotesi induttiva: la tesi è valida per tutti i percorsi π' di lunghezza $n-1$ e la si vuole dimostrare per l'arbitrario $\pi = s_0 \xrightarrow{\sigma_1}_P s_1 \xrightarrow{\sigma_2}_P s_2 \xrightarrow{\sigma_3}_P \ldots \xrightarrow{\sigma_n}_P s_n$ di lunghezza n.

Si consideri il prefisso del percorso π, denotato con $\pi' = s_0 \xrightarrow{\sigma_1}_P s_1 \xrightarrow{\sigma_2}_P s_2 \xrightarrow{\sigma_3}_P \ldots \xrightarrow{\sigma_{n-1}}_P s_{n-1}$. Si tratta di un percorso valido inizializzato di P di lunghezza $n-1$. Per ipotesi induttiva esiste $\widehat{\pi'}$ percorso astratto valido inizializzato e ammissibile tale che $\widehat{\pi'} = \widehat{s_0} \xrightarrow{\widehat{\sigma_1}}_P \widehat{s_1} \xrightarrow{\widehat{\sigma_2}}_P \widehat{s_2} \xrightarrow{\widehat{\sigma_3}}_P \ldots \xrightarrow{\widehat{\sigma_{n-1}}}_P \widehat{s_{n-1}}$, con $\widehat{s_i} \in h(s_i)$ per ogni $i \in \{0, 1, 2, \ldots, n-1\}$.

Ma, poichè $s_n \in post_P(s_{n-1})$ e poichè $\widehat{s_{n-1}} \in h(s_{n-1})$, segue dal teorema 1 che

$$s_n \in \gamma \circ \widehat{post_P}(\widehat{s_{n-1}})$$

e cioè, per la definizione della funzione γ, che esiste uno stato astratto $\widehat{s_n} \in \widehat{post_P}(\widehat{s_{n-1}})$ tale che $\widehat{s_n} \in h(s_n)$.

Pertanto, esiste un percorso astratto valido inizializzato e ammissibile $\widehat{\pi} = \widehat{s_0} \xrightarrow{\widehat{\sigma_1}}_P \widehat{s_1} \xrightarrow{\widehat{\sigma_2}}_P \widehat{s_2} \xrightarrow{\widehat{\sigma_3}}_P \ldots \xrightarrow{\widehat{\sigma_{n-1}}}_P \widehat{s_{n-1}} \xrightarrow{\widehat{\sigma_n}}_P \widehat{s_n}$, con $\widehat{s_i} \in h(s_i)$ per ogni $i \in \{0, 1, 2, \ldots, n\}$.
□

Dal teorema precedente è possibile affermare il seguente corollario:

Corollario 4 (Correttezza dell'astrazione) *Si consideri un programma concreto P e una sua astrazione rispetto la classe di insiemi di indici I e sia un nodo $i \in N_P$ raggiungibile in P secondo la sua semantica concreta $[\![P]\!]$, allora i è raggiungibile in P secondo la sua semantica astratta $\widehat{[\![P]\!]}$*

Dimostrazione Sia $i \in N_P$ un nodo raggiungibile in P rispetto alla sua semantica concreta $[\![P]\!]$, allora esiste un percorso $\pi \in [\![P]\!]$ tale che $\pi = s_0 \xrightarrow{\sigma_1}_P s_1 \xrightarrow{\sigma_2}_P \ldots \xrightarrow{\sigma_n}_P s_n$, dove $s_k = \langle i_k, w_k \rangle$, con $k = 0, 1, \ldots, n$, $i_0 = First_P(main)$ e $i_n = i$. Per il teorema 2 esiste un percorso $\widehat{\pi} \in \widehat{[\![P]\!]}$ tale che $\widehat{\pi} = \widehat{s_0} \xrightarrow{\widehat{\sigma_1}}_P \widehat{s_1} \xrightarrow{\widehat{\sigma_2}}_P \ldots \xrightarrow{\widehat{\sigma_n}}_P \widehat{s_n}$, dove $\widehat{s_k} = \langle i_k, \widehat{w_k} \rangle \in h(s_k)$, con $k = 0, 1, \ldots, n$, $i_0 = First_P(main)$ e $i_n = i$, pertanto i è raggiungibile in P anche rispetto alla sua semantica astratta $\widehat{[\![P]\!]}$.
□

Capitolo 3

Model checking

Obbiettivo di tale capitolo è illustrare la procedura di model checking di Eureka e descrivere le modifiche ad essa introdotte nell'ambito del presente lavoro di tesi.

In particolare, verrà presentato l'algoritmo di Data Flow Analysis, utilizzato per il calcolo della ragiungibilità degli statement, ed una sua controparte simbolica, adatta all'automazione.

A tale scopo verranno introdotti i concetti di Path Edge e Summary Edge, utilizzati dall'algoritmo di Data Flow Analysis, e la nozione di ADLC, utilizzato per la rappresentazione e manipolazione simbolica di quest'ultimi.

3.1 Interprocedural Data Flow Analysis per programmi lineari

Si definisce in questa sezione una procedura per il calcolo della raggiungibilità degli statement all'interno di programmi lineari (Interprocedural Data Flow Analysis) [Cam06].

Tale procedura estende quella descritta in [BR00b], che è essa stessa costruita a partire dall'algoritmo definito da Reps, Horwitz e Sagiv in [RHS95].

3.1.1 Path Edge e Summary Edge

Sia $i \in N_P$ un nodo del CFG e sia $e = First_P(ProcOf_P(i))$. Un *path edge* $\pi_i = \langle w_e, w_i \rangle$ *incidente in* i è una coppia di valutazioni tale che esiste un percorso valido $\langle First_P(main), w_0 \rangle \xrightarrow{\sum_0^e}_P \langle e, w_e \rangle$ e un percorso valido dello stesso livello $\langle e, w_e \rangle \xrightarrow{\sum_e^i}_P \langle i, w_i \rangle$ per qualche valutazione w_0. In altre parole, un pa-

th edge rappresenta un suffisso di un percorso valido da $\langle First_P(main), w_0\rangle$ a $\langle i, w_i\rangle$.

I *summary edge* sono delle strutture dati che memorizzano il risultato dell'analisi di procedure, evitando lavoro ridondande nell'elaborazione di una sequenza di esecuzione. L'effetto sullo stato del programma di una chiamata a procedura con una determinata lista di argomenti viene memorizzato, così che la prossima volta che la stessa procedura viene invocata con la stessa lista di parametri effettivi, non c'è bisogno di analizzarla di nuovo, perchè si sa già quali effetti avrà sullo stato del programma e si è, quindi, in grado di stabilire quale sarà il prossimo stato.

Nel corso della trattazione della procedura di model checking non verrà considerato l'utilizzo dei summary edge, in quanto scopo di tale capitolo è dare una descrizione intuitiva di come viene verificata la raggiungibilità degli statement, quindi tale ottimizzazione può essere omessa.

Sia $i \in N_P$, si denota con $\mathbf{\Pi}_i(P)$ l'insieme di path edge incidenti in i associati al programma P. Il vertice i è raggiungibile se e solo se $\mathbf{\Pi}_i(P) \neq \emptyset$.

Sia $\mathcal{Q}(P)$ la classe di insiemi indicizzata da N_P tale che $\mathcal{Q}_i(P) = \{\langle w_1, w_2\rangle \mid w_1, w_2 : InScope_P(i) \rightarrow \mathcal{D}\}$ per ogni $i \in N_P$.

Siano Π, Π^1 due classi di insiemi indicizzate da N_P tali che $\Pi_i \subseteq \mathcal{Q}_i(P)$ e $\Pi^1_i \subseteq \mathcal{Q}_i(P)$ per ogni $i \in N_P$. Si definisce la relazione binaria $\Pi \Rightarrow \Pi^1$ come segue. Sia $i \in N_P$, $\Pi^1_j = \Pi_j$ per ogni $j \notin Succ_P(i)$ e:

- se i corrisponde ad uno `skip` o `return;`, allora

  $$\Pi^1_{sSucc_P(i)} = \Pi_{sSucc_P(i)} \cup \Pi_i$$

  Tutti i path edge del nodo i vengono propagati al suo successore, senza alcuna modifica.

- se i corrisponde ad uno statement di assegnamento del tipo **y=e;**, allora

  $$\Pi^1_{sSucc_P(i)} = \Pi_{sSucc_P(i)} \cup \{\langle w_e, w_i[\mathbf{d}/\mathbf{y}]\rangle \mid \langle w_e, w_i\rangle \in \Pi_i, \mathbf{d} \in \overline{w_i}(\mathbf{e})\}$$

  I nuovi path edge vengono calcolati a partire dai path edge associati al nodo i, in maniera tale che la valutazione delle variabili assegnate nel nodo ($\mathbf{y}$) coincida con una delle possibili valutazioni delle espressioni in $\mathbf{e}$ nel nodo i.

- se i corrisponde ad uno statement del tipo `if(`e`)`, `while(`e`)`, `assume(`e`);` oppure `assert(`e`);`, allora

  $$\Pi^1_{Tsucc_P(i)} = \Pi_{Tsucc_P(i)} \cup \{\langle w_e, w_i\rangle \in \Pi_i \mid d \in \overline{w_i}(e) \text{ per qualche } d \neq 0\}$$
  $$\Pi^1_{Fsucc_P(i)} = \Pi_{Fsucc_P(i)} \cup \{\langle w_e, w_i\rangle \in \Pi_i \mid 0 \in \overline{w_i}(e)\}$$

A partire dai path edge di i, vengono propagati nel successore del ramo vero $Tsucc_P(i)$ (successore del ramo falso $Fsucc_P(i)$, rispettivamente) i path edge per cui la guardia e ammette la valutazione vero (falso, rispettivamente) nel nodo i.

- se i corrisponde ad una chiamata a procedura pr`(a);`, allora

$$\Pi^1_{sSucc_P(i)} = \Pi_{sSucc_P(i)} \cup \{\langle w_j, w_j\rangle \mid \langle w_e, w_i\rangle \in \Pi_i,\ w_j(\mathbf{g}) = w_i(\mathbf{g}),\ w_j(\mathbf{y}) \in \overline{w_i}(\mathbf{a}),\ \mathbf{g} = Globals_P,\ \mathbf{y} = Formals_P(First_P(pr))\}$$

dove la condizione $w_j(\mathbf{g}) = w_i(\mathbf{g})$ propaga i valori delle variabili globali nel vertice i corrispondente alla chiamata a procedura al vertice iniziale della procedura pr, mentre $w_j(\mathbf{y}) \in \overline{w_i}(\mathbf{a})$ modella il passaggio dei parametri assegnando i valori dei parametri effettivi $\mathbf{a}$ nel nodo della chiamata ai parametri formali $\mathbf{y}$ di pr.

- se $i = Exit_P(pr)$ è l'exit node corrispondente alla procedura pr, allora

$$\Pi^1_j = \Pi_j \cup \{\langle w_e, w_j\rangle \mid \langle w_e, w_k\rangle \in \Pi_k,\ \langle w_h, w_i\rangle \in \Pi_i,\ w_h(\mathbf{y}) \in \overline{w_k}(\mathbf{a}),\ w_h(\mathbf{g}) = w_k(\mathbf{g}),\ w_j(\mathbf{z}) = w_k(\mathbf{z}),\ w_j(\mathbf{g}) = w_i(\mathbf{g}),\ j = RetPt_P(k),\ \mathbf{y} = Formals_P(First_P(pr)),\ \mathbf{z} = Locals_P(k) \text{ e } \mathbf{g} = Globals_P\}$$

se $j \in Succ_P(i)$. Da notare che, se $j \in Succ_P(i)$, allora j è il primo statement seguente una chiamata a procedura $stm_k = pr(\mathbf{a})$; (condizione $j = RetPt_P(k)$), mentre e e h sono i vertici associati ai primi statement della procedura contenente l'istruzione stm_k (la chiamata a procedura) e di quella contenente il nodo i (l'exit node della procedura chiamata), rispettivamente. Π^1_j è ottenuto da Π_j aggiungendo tutte le coppie $\langle w_e, w_j\rangle$ ottenute dai path edge $\langle w_e, w_k\rangle \in \Pi_k$ e $\langle w_h, w_i\rangle \in \Pi_i$ tali che i valori associati in k (rispetto alla valutazione w_k) ai parametri effettivi $\mathbf{a}$ ed alle variabili globali $\mathbf{g}$ sono uguali ai valori associati in h (rispetto alla valutazione w_h) ai parametri formali $\mathbf{y}$ della procedura pr ed alle variabili globali, rispettivamente (condizioni $w_h(\mathbf{y}) \in \overline{w_k}(\mathbf{a})$ e $w_h(\mathbf{g}) = w_k(\mathbf{g})$). Nel path edge calcolato, w_j è definita essere uguale a w_k sulle variabili locali $\mathbf{z}$ dell'istruzione corrispondente al nodo k relativo alla chiamata a procedura (condizione $w_j(\mathbf{z}) = w_k(\mathbf{z})$) e uguale a w_i sulle variabili globali (condizione $w_j(\mathbf{g}) = w_i(\mathbf{g})$), ereditando, quindi, gli effetti della procedura sulle variabili globali.

I seguenti teoremi (dimostrati in [ABM07]) garantiscono la correttezza e completezza dell'algoritmo.

Teorema 3 (Correttezza) *Se* $\Pi \Rightarrow \Pi^1$*, allora per ogni* $i \in N_P$ $\Pi_i \subseteq \mathbf{\Pi}_i(P)$ *implica che per ogni* $i \in N_P$ $\Pi^1_i \subseteq \mathbf{\Pi}_i(P)$*.*

Sia Π^0 tale che

$$\Pi^0_{First_P(main)} = \{\langle w, w\rangle \mid w(x) \in \mathcal{D} \text{ per ogni } x \in InScope_P(First_P(main))\}$$

e tale che $\Pi^0_j = \emptyset$ per ogni $j \in N_P \setminus \{First_P(main)\}$ e si denoti con $\Rightarrow^*$ la chiusura riflessiva e transitiva di $\Rightarrow$, vale il seguente teorema.

Teorema 4 (Completezza) *Sia* Π^0 *definito come sopra. Se* $\langle w_h, w_j\rangle \in \mathbf{\Pi}_j(P)$*, allora esiste* Π^1 *tale che* $\Pi^0 \Rightarrow^* \Pi^1$ *e* $\langle w_h, w_j\rangle \in \Pi^1_j$*.*

Corollario 5 *Sia* Π^0 *definito come sopra, allora per ogni* $i \in N_P$*,* i *è raggiungibile se e solo se* $\Pi_i \neq \emptyset$ *per qualche* Π *tale che* $\Pi^0 \Rightarrow^* \Pi$

3.2 Model Checking simbolico per programmi lineari

La relazione binaria definita nel paragrafo precedente deve però essere espressa in una forma adatta alla sua automazione, affinchè sia possibile costruire un modulo software concreto che la computi. Tale operazione verrà effettuata in tre passi: dapprima verranno definite delle formule dell'aritmetica lineare che codificano la semantica degli assegnamenti paralleli ($\alpha(\mathbf{y}, \mathbf{e})$) e delle espressioni lineari ($\beta^+(e)$ e $\beta^-(e)$); dopodicchè verranno introdotte le strutture dati (chiamate Abstract Disjunctive Linear Constraints, in breve ADLC) che verranno utilizzate per la rappresentazione simbolica di insiemi di path edge; infine, verrà data una caratterizzazione astratta della procedura di model checking come una relazione, definita induttivamente, sulla classe di ADLC indicizzata sull'insieme di nodi N_P.

Per ogni nodo i del control flow graph del programma P, la procedura costruisce in maniera incrementale una rappresentazione simbolica dell'insieme di path edge incidenti in i. La raggiungibilità di uno statement stm_i viene stabilita attraverso la verifica che l'insieme di path edge incidenti in i si non vuoto.

3.2.1 Codifica in aritmetica lineare degli statement del programma (α e β)

Il primo passo verso la codifica della semantica dei programmi lineari nell'aritmetica lineare consiste nell'eliminare le espressioni condizionali che occorrono nel programma. Siano e e ne espressioni lineari, B un insieme di espressioni lineare booleane atomiche. Si definisce la relazione $e \rightarrow (B, ne)$ come la più piccola relazione tale che $e \rightarrow (\emptyset, e)$ per ogni $e \in V \cup \mathcal{D} \cup \{u\}$ e chiusa rispetto alle seguenti regole di inferenza:

$$\frac{e_1 \rightarrow (B_1, ne_1) \qquad e_2 \rightarrow (B_2, ne_2)}{(e_1 \; op \; e_2) \rightarrow (B_1 \cup B_2, (ne_1 \; op \; ne_2))} \text{ se } op \in \{*, +, <, <=, >, >=, ==, !=\}$$

$$\frac{b \rightarrow (B, nb) \qquad e_1 \rightarrow (B_1, ne_1)}{(b?e_1 : e_2) \rightarrow (B \cup B_1 \cup \{nb^+\}, ne_1)} \qquad \frac{b \rightarrow (B, nb) \qquad e_2 \rightarrow (B_2, ne_2)}{(b?e_1 : e_2) \rightarrow (B \cup B_2 \cup \{nb^-\}, ne_2)}$$

dove $b^+(b-)$ è b `!=` 0 (b `==` 0, rispettivamente) se b è un'espressione lineare e b (`!`b, rispettivamente) se b è un'espressione booleana. Può essere mostrato che se $e \rightarrow (B, ne)$, allora nè l'espressione lineare ne nè l'insieme di espressioni lineari booleane B contengono espressioni condizionali.

Il secondo passo verso la codifica in aritmetica lineare consiste nel codificare i simboli di indefinito. Sia U un insieme infinito di variabili tale che $U \cap V = \emptyset$. Se e è un'espressione lineare, si indica con e^* l'espressione ottenuta da e sostituendo ogni occorrenza di di `!`b, e_1 `!=` e_2 e e_1 `==` e_2, con $\neg b$, $\neg(e_1 = e_2)$ e $e_1 = e_2$, rispettivamente, e ogni occorrenza di `u` in e con una differente variabile in U. Se w è una formula e $u_1, u_2, \ldots, u_k$ sono le variabili di U che occorrono in w, con $\exists U.w$ si denota la formula $\exists u_1.\exists u_2. \ldots \exists u_k.w$. Da notare che se e è un'espressione lineare senza condizionali, allora e^* è un'espressione del linguaggio dell'aritmetica lineare.

E' ora possibile codificare la semantica delle espressioni lineari in aritmetica lineare. Le espressioni di un programma lineare possono occorrerre come guardia in statement condizionali, come membro destro di un assegnamento o come parametri di una chiamata a procedura. La semantica di un'espressione lineare che rappresenta la guardia di un condizionale dipende da come procede l'esecuzione, lungo il ramo vero (*true branch*, eseguito nel casi in cui la guardia viene valutata vera) o il ramo falso (*false brach*, eseguito in caso di guardia falsa). Quindi, bisogna codificare sia la versione positiva che negativa della semantica della guardia. Formalmente, sia e un'espressione lineare, si definiscono le seguenti funzioni:

$$\beta^+(e) = \bigvee \left\{ \exists U.(ne^+ \wedge \bigwedge B)^* \mid e \rightarrow (B, ne) \right\}$$

$$\beta^-(e) = \bigvee \left\{ \exists U.(ne^- \wedge \bigwedge B)^* \mid e \to (B, ne) \right\}$$

Siano ora y e e una variabile e un'espressione lineare tale che y non occorre in e. Per codificare la semantica di un'espressione che occorre come membro destro in uno statement di assegnamento, si definisce:

$$\alpha(y, e) = \bigvee \left\{ \exists U.(y = ne \wedge \bigwedge B)^* \mid e \to (B, ne) \right\}$$

Ad esempio, sia $e =$`(3*x+(y<0?u:y))` allora $e \to$ (`{y<0}`,`3*x+u`) e $e \to$ (`{!(y<0)}`,`3*x+y`). Quindi $\alpha(z, e) = (\exists u_1.(y < 0 \wedge z = 3 * x + u_1) \vee (\neg(y < 0) \wedge z = 3 * x + y))$ che può essere semplificata con la formula logicamente equivalente $(y < 0 \vee (\neg(y < 0) \wedge z = 3 * x + y))$. Invece, $\beta^+(e) = (\exists u_1.(y < 0 \wedge \neg(3 * x + u_1 = 0)) \vee (\neg(y < 0) \wedge \neg(3 * x + y = 0)))$, che può essere semplificata con $(y < 0 \vee (\neg(y < 0) \wedge \neg(3 * x + y = 0)))$.

Il seguente lemma afferma formalmente la relazione tra la codifica simbolica con le formule $\alpha()$, $\beta^+()$ e $\beta^-()$ e la semantica delle espressioni lineari (per una dimostrazione del lemma si rimanda al lavoro in [ABM07]).

Lemma 2 *Sia y una variabile ed e un'espressione lineare (booleana), tale che y non occorre in e, allora per ogni valutazione w:*

- $\models_w \beta^+(e)$ *se e solo se* $d \in \overline{w}(e)$, *per qualche* $d \neq 0$;
- $\models_w \beta^-(e)$ *se e solo se* $0 \in \overline{w}(e)$;
- *se e è un'espressione lineare, allora* $\overline{w}(e) = \{d \in \mathcal{D} \mid \models_{w[d/y]} \alpha(y, e)\}$

Per consentire la codifica di assegnamenti paralleli, $\alpha()$ può essere generalizzata a k-uple di variabili ed espressioni come segue. Siano $\mathbf{y}$ e $\mathbf{e}$ k-uple di variabili ed espressioni lineari con $k > 0$ tali che nessuna variabile in $\mathbf{y}$ occorre in $\mathbf{e}$. Si definisce:

$$\alpha(\mathbf{y}, \mathbf{e}) = \bigvee \left\{ \bigwedge_{i=1}^{k} \left(\exists U.(y_i = ne_i \wedge \bigwedge B_i)^* \right) \mid e_i \to (B_i, ne_i) \,\mathrm{con}\, i = 1, \ldots, k \right\}$$

3.2.2 Rappresentazione simbolica tramite ADLC

Sia e un'espressione, si denota con e' l'espressione ottenuta da e attraverso la sostituzione di ogni variabile v in e con la corrispondente versione primata v'. Tale notazione è estesa a insiemi (tuple) di espressioni nell'ovvia maniera, cioè $E' = \{e' \mid e \in E\}$ ($\mathbf{e}' = \langle e'_1, e'_2, \ldots, e'_n \rangle$ se $\mathbf{e} = \langle e_1, e_2, \ldots, e_n \rangle$, rispettivamente).

Sia e un'espressione lineare definita su $\mathcal{D}$ della forma $c_0 + c_1x_1 + \ldots + c_nx_n$, con $c_0, c_1, \ldots, c_n$ costanti e $x_1, x_2, \ldots, x_n$ variabili, definite entrambe sul dominio $\mathcal{D}$. Un vincolo lineare (*linear constraint*) è una relazione binaria in una delle forme $e < 0$, $e \leq 0$, $e > 0$, $e \geq 0$, $e = 0$, $e \neq 0$. Una formula lineare (*linear formula*) è una combinazione booleana di vincoli lineari. Una formula in forma normale disgiuntiva (Disjunctive Linear Constraint, DLC) è una disgiunzione di congiunti, ovvero una formula della forma $D = \bigvee_i \exists U. \bigwedge_j c_{ij}$, con c_{ij} vincolo lineare. Una formula in forma normale disgiuntiva astratta (Abstract Disjunctive Linear Constraint, ADLC) di arietà n è un'espressione nella forma $\lambda\mathbf{x}.\lambda\mathbf{x}'.D$[1], dove D è un DLC e $\mathbf{x}$, $\mathbf{x}'$ sono n-uple contenenti le variabili libere che occorrono in D.

Intuitivamente questo tipo di notazione permette di catturare la semantica di un programma, ovvero permette di esprimere come l'evoluzione di un programma ha effetto sulle sue variabili. Con $\mathbf{x}$ si indica una valutazione delle variabili all'inizio della procedura, mentre con $\mathbf{x}'$ si indica una valutazione delle variabili nello stato attuale. Formalmente, la semantica di un ADLC $\delta = \lambda\mathbf{x}\mathbf{x}'.D$ è data da un insieme di coppie $\langle w_1, w_2\rangle$ di valutazioni, dove w_1 è una valutazione per le variabili non primate $\mathbf{x}$ e w_2 è tale che w_2' [2] è una valutazione per le variabili primate $\mathbf{x}'$, tali che la loro unione $w_1 \cup w_2'$ è una valutazione che soddisfa il DLC. In formule, sia $\delta = \lambda\mathbf{x}\mathbf{x}'.D$ un ADLC:

$$[\![\delta]\!] = \{\langle w_1, w_2\rangle \mid \models_{w_1 \cup w_2'} D,\ w_1, w_2 : V \to \mathcal{D}\}$$

Pertanto un ADLC δ può essere visto come la rappresentazione simbolica dell'insieme di path edge contenuti nell'insieme $[\![\delta]\!]$ rappresentante la sua semantica.

3.2.3 Manipolazione degli ADLC

Una volta definiti gli ADLC, bisogna introdurre quali operazioni possono essere eseguite su di essi per manipolarli. Si introducono, quindi, in questa sezione, le seguenti operazioni:

- *Applicazione.* Sia $\lambda\mathbf{x}\mathbf{x}'.D$ un ADLC di arietà n e $\mathbf{s}$ e $\mathbf{t}$ due n-uple di espressioni lineari. L'applicazione di $\delta = \lambda\mathbf{x}\mathbf{x}'.D$ alla coppia $(\mathbf{s}, \mathbf{t})$, che in simboli si esprime con $\delta(\mathbf{s}, \mathbf{t})$, è il DLC ottenuto dalla simultanea sostituzione, all'interno di D, dell'i-esimo elemento di $\mathbf{x}$ ($\mathbf{x}'$) con l'i-esimo elemento di $\mathbf{s}$ ($\mathbf{t}$, rispettivamente), per ogni $i = 1, \ldots, n$.

[1] Per semplicità introduciamo la notazione $\lambda\mathbf{x}\mathbf{x}'.D$, del tutto equivalente a $\lambda\mathbf{x}.\lambda\mathbf{x}'.D$

[2] Qui e in seguito, data una valutazione $w : V \to \mathcal{D}$, si denota con w' la valutazione $w' : V' \to \mathcal{D}$ tale che $w'(x') = w(x)$ per ogni $x' \in V'$. Analogamente con w'' si denota la valutazione $w'' : V'' \to \mathcal{D}$ tale che $w''(x'') = w(x)$ per ogni $x'' \in V''$

Ad esempio, sia $\delta = \lambda \mathbf{x}\mathbf{x}'.\{a = 10 \wedge b' = 5\}$, dove $\mathbf{x} = \langle a, b\rangle$ e $\mathbf{x}' = \langle a', b'\rangle$. Siano $\mathbf{s} = \langle c, d\rangle$ e $\mathbf{t} = \langle c', d'\rangle$. Si ottiene $\delta(\mathbf{s}, \mathbf{t}) = \{c = 10 \wedge d' = 5\}$.

- *Disgiunzione* ($\sqcup$). Siano D_1 e D_2 due DLC, allora $D_1 \sqcup D_2$ è il DLC logicamente equivalente a $D_1 \vee D_2$. La disgiunzione è estesa agli ADLC come segue. Siano δ_1 e δ_2 due ADLC di uguale arietà, allora $\delta_1 \sqcup \delta_2 = \lambda \mathbf{x}\mathbf{x}'.(\delta_1(\mathbf{x}, \mathbf{x}') \sqcup \delta_2(\mathbf{x}, \mathbf{x}'))$.

  Ad esempio, sia $\delta_1 = \lambda \mathbf{x}\mathbf{x}'.\{a' = 1 \wedge b < 5\}$ e $\delta_2 = \lambda \mathbf{x}\mathbf{x}'.\{a' = a \wedge b > 5\}$, allora $\delta_1 \sqcup \delta_2 = \lambda \mathbf{x}\mathbf{x}'.\{(a' = 1 \wedge b < 5) \vee (a' = a \wedge b > 5)\}$.

- *Congiunzione* ($\sqcap$). Siano D_1 e D_2 due DLC, allora $D_1 \sqcap D_2$ è il DLC logicamente equivalente a $D_1 \wedge D_2$. La congiunzione è estesa agli ADLC come segue. Siano δ_1 e δ_2 due ADLC aventi la stessa arietà, allora $\delta_1 \sqcap \delta_2 = \lambda \mathbf{x}\mathbf{x}'.(\delta_1(\mathbf{x}, \mathbf{x}') \sqcap \delta_2(\mathbf{x}, \mathbf{x}'))$.

  Ad esempio, sia $\delta_1 = \lambda \mathbf{x}\mathbf{x}'.\{a' < 5 \wedge b < 4\}$ e $\delta_2 = \lambda \mathbf{x}\mathbf{x}'.\{a' < 2 \wedge b < 6\}$, allora $\delta_1 \sqcap \delta_2 = \lambda \mathbf{x}\mathbf{x}'.\{a' < 2 \wedge b < 4\}$.

- *Negazione* ($\sim$). Sia D un DLC, $\sim D$ è il DLC ottenuto prendendo il negato $\neg D$ di D è portandolo in forma normale disgiuntiva. La negazione può essere estesa agli ADLC come segue: $\sim \delta = \lambda \mathbf{x}\mathbf{x}'.(\sim \delta(\mathbf{x}, \mathbf{x}'))$.

  Ad esempio, sia $\delta = \lambda \mathbf{x}\mathbf{x}'.\{a' = 1 \wedge b < 5\}$, vale allora $\sim \delta = \lambda \mathbf{x}\mathbf{x}'.\{a' \neq 1 \vee b \geq 5\}$.

- *Quantifier Elimination* ($\exists$). Sia D un DLC, allora $\exists \mathbf{x}.D$ è un DLC equivalente a D ottenuto eliminando da D le variabili in $\mathbf{x}$.

  Ad esempio, sia $\delta = \lambda \mathbf{x}\mathbf{x}'.\{b = a' + 1 \wedge a = a' \wedge a' > 0\}$ ed indicando con $\mathbf{x}'$ tutte le variabili primate del DLC, vale che $\exists \mathbf{x}'.\{b = a' + 1 \wedge a = a' \wedge a' > 0\} = \{b = a + 1 \wedge a > 0\}$.

- *Entailment* ($\sqsubseteq$). Siano δ_1 e δ_2 due ADLC aventi la stessa arietà, $\delta_1 \sqsubseteq \delta_2$ se e soltanto se tutte le coppie di valutazioni che soddisfano δ_1 soddisfano anche δ_2, cioè $[\![\delta_1]\!] \subseteq [\![\delta_2]\!]$.

  Ad esempio, sia $\delta_1 = \lambda \mathbf{x}\mathbf{x}'.\{a' = 1 \wedge b < 5\}$ e $\delta_2 = \lambda \mathbf{x}\mathbf{x}'.\{a' = 1 \wedge b = 4\}$, allora vale che $\delta_2 \sqsubseteq \delta_1$.

3.2.4 Controparte simbolica dell'algorimtmo di Interprocedural Data Flow Analysis

Una volta definito che path edge, summary edge e relazione di transizione vengono rappresentati tramite ADLC ed una volta stabilite quali operazioni manipolano tali strutture è possibile, finalmente, fornire la procedura di model checking per il calcolo della raggiungibilità degli statement di un programma lineare.

Con $\mathcal{A}(P)$ si denota la classe di insiemi di ADLC indicizzata sui nodi del grafo N_P tale che $\mathcal{A}_i(P)$ è l'insieme di ADLC di arietà $\mid InScope_P(i) \mid$ per ogni $i \in N_P$. Siano $\Delta, \Delta^1 \in \mathcal{A}(P)$[3], si definisce la relazione binaria $\Delta \rightarrow \Delta^1$ come segue. Sia $i \in N_P$, $\Delta^1_j = \Delta_j$ per ogni $j \notin Succ_P(i)$ e

- se i corrisponde ad uno statement del tipo `skip` o ad un `return;`, allora

$$\Delta^1_{sSucc_P(i)} = \Delta_{sSucc_P(i)} \sqcup \Delta_i$$

- se i corrisponde ad uno statement di assegnamento del tipo **y=e;**, allora

$$\Delta^1_{sSucc_P(i)} = \Delta_{sSucc_P(i)} \sqcup \lambda \mathbf{x}\mathbf{x}''.\exists \mathbf{x}'.(\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \alpha(\mathbf{y}'', \mathbf{e}') \sqcap \mathbf{z}'' = \mathbf{z}')$$

  dove $\mathbf{x} = InScope_P(i)$ e $\mathbf{z} = InScope_P(i) \setminus \mathbf{y}$.

  L'idea intuitiva è la seguente: a partire dai path edge rappresentati dal DLC $\Delta_i(\mathbf{x}, \mathbf{x}')$, i nuovi path edge sono calcolati impostando prima i valori delle variabili assegnate nello statement ($\mathbf{y}''$) ai valori delle espressioni con cui vengono assegnate nel vertice i ($\mathbf{e}'$) e lasciando i valori delle altre variabili in scope invariati ($\mathbf{z}'' = \mathbf{z}'$); infine attraverso l'applicazione del quantificatore esistenziale viene eliminata la dipendenza dalle variabili associate al vertice intermedio i.

- se i corrisponde ad uno statement del tipo `if(`b`)`, `while(`b`)`, `assert(`b`);` oppure `assume(`b`);`, allora

$$\Delta^1_{Tsucc_P(i)} = \Delta_{Tsucc_P(i)} \sqcup \lambda \mathbf{x}\mathbf{x}'.(\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \beta^+(b'))$$

$$\Delta^1_{Fsucc_P(i)} = \Delta_{Fsucc_P(i)} \sqcup \lambda \mathbf{x}\mathbf{x}'.(\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \beta^-(b'))$$

[3]Con un abuso di notazione si semplifica l'espressione $\Delta_i \in \mathcal{A}_i(P)$ per ogni $i \in N_P$ con l'espressione $\Delta \in \mathcal{A}(P)$

con $\mathbf{x} = InScope_P(i)$.

Il vincolo $\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \beta^+(b')$ nella prima regola semplicemente seleziona quei path edge del vertice i che soddisfano anche la codifica della condizione associata al ramo vero dello statement condizionale ($\beta^+(b')$) e li propagano al successore vero del vertice i ($Tsucc_P(i)$). L'intuizione della seconda regola per il ramo falso è analoga.

- se i corrisponde ad una chiamata a procedura del tipo $pr(\mathbf{e})$;, allora

$$\Delta^1_{sSucc_P(i)} = \Delta_{sSucc_P(i)} \sqcup \lambda \mathbf{w}\mathbf{w}''.(\exists \mathbf{x}\mathbf{x}'.(\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \alpha(\mathbf{f}'', \mathbf{e}') \sqcap \sqcap \mathbf{g}'' = \mathbf{g}') \sqcap \mathbf{w} = \mathbf{w}'')$$

con $\mathbf{x} = InScope_P(i)$, $\mathbf{g} = Globals_P$, $\mathbf{w} = InScope_P(sSucc_P(i))$ e $\mathbf{f} = Formals_P(First_P(pr))$.

I path edge del primo vertice della procedura pr sono calcolati selezionando i path edge del nodo in cui avviene la chiamata a pr ($\Delta(\mathbf{x}, \mathbf{x}')$) e assegnando ai parametri formali di pr i valori dei parametri effettivi della chiamata ($\alpha(\mathbf{f}'', \mathbf{e}')$) e propagando i valori delle variabili globali ($\mathbf{g}'' = \mathbf{g}'$). Le quantificazioni esistenziali delle versioni primate e non delle variabili nello scope del vertice i eliminano le dipendenze delle valutazioni, mentre l'ulteriore vincolo $\mathbf{w} = \mathbf{w}''$ forza le due valutazioni del nuovo path edge a coincidere sulle variabili locali della procedura.

- se i corrisponde ad un vertice $Exit_P(pr)$, allora

$$\Delta^1_j = \Delta_j \sqcup \lambda \mathbf{w}\mathbf{w}''.\exists \mathbf{w}'''.(\exists \mathbf{x}'\mathbf{z}''.(\Delta_k(\mathbf{w}, \mathbf{w}''') \sqcap \Delta_i(\mathbf{x}', \mathbf{x}'') \sqcap \alpha(\mathbf{f}', \mathbf{e}''') \sqcap \sqcap \mathbf{g}' = \mathbf{g}''') \sqcap \mathbf{l}'' = \mathbf{l}''')$$

con $j \in Succ_P(i)$, k tale che $stm_k = pr(\mathbf{e})$; e $RetPt_P(k) = j$. Inoltre vale che $\mathbf{w} = InScope_P(k)$, $\mathbf{f} = Formals_P(First_P(pr))$, $\mathbf{l} = Locals_P(k)$, $\mathbf{x} = InScope_P(i)$, $\mathbf{z} = Locals_P(i)$ e $\mathbf{g} = Globals_P$.

Intuitivamente il vincolo $(\Delta_k(\mathbf{w}, \mathbf{w}''') \sqcap \Delta_i(\mathbf{x}', \mathbf{x}'') \sqcap \alpha(\mathbf{f}', \mathbf{e}''') \sqcap \mathbf{g}' = \mathbf{g}''')$ seleziona tutte le coppie di path edge (coppie in cui il primo elemento è un path edge incidente nel vertice k in cui avviene la chiamata alla procedura pr e che soddisfa il DLC $\Delta_k(\mathbf{w}, \mathbf{w}''')$ e il secondo un path edge incidente nell'exit node i che soddisfa $\Delta_i(\mathbf{x}', \mathbf{x}'')$) tali che i parametri formali di pr ($\mathbf{f}'$) nel primo vertice di pr hanno lo stesso valore di quello che assumono i parametri effettivi ($\mathbf{e}'''$) nel nodo k e tali che anche i valori delle variabili globali coincidono nel primo nodo di pr e nel nodo k (condizione $\mathbf{g}' = \mathbf{g}'''$). In altre parole, il vincolo seleziona

tutte le coppie di path edge tali che il nodo corrispondente alla chiamata e il primo nodo di pr coincidono rispetto ai valori dei parametri formali e attuali ed ai valori delle variabili globali. La quantificazione esistenziale su $\mathbf{x}'$ (variabili nello scope di i, valutate nel primo vertice della procedura pr) e $\mathbf{z}''$ (locali di pr, valutate nell'exit node della procedura) elimina le dipendenze sui vertici intermedi. L'ulteriore vincolo $\mathbf{l}'' = \mathbf{l}'''$ forza i valori delle variabili locali del vertice j, vertice di ritorno della procedura pr, a coincidere con i valori che tali variabili avevano nel punto di chiamata della stessa (nodo k). Infine, attraverso la quantificazione esistenziale delle variabili nello scope di k ($\mathbf{w}'''$), si ottengono i path edge desiderati.

A titolo di esempio si consideri lo statement stm_i corrispondente all'assegnamento parallelo $y_1, y_2 = ((y_1 > 0)?y_1 + 1 : u), y_2 + 1$ e sia $InScope_P(i) = \{x, y_1, y_2\}$. Si ha che $\mathbf{e} = \langle((y_1 > 0)?y_1 + 1 : u), y_2 + 1\rangle$, $\mathbf{y} = \langle y_1, y_2\rangle$ e $\mathbf{z} = \langle x\rangle$. Applicando α si ottiene $\alpha(\mathbf{y}'', \mathbf{e}') = ((y_1'' = y_1' + 1 \wedge y_2'' = y_2' + 1 \wedge y_1' > 0) \vee \exists u_1.(y_1'' = u_1 \wedge y_2'' = y_2' + 1 \wedge \neg(y_1' > 0)))$. Quindi il nuovo insieme di path edge che va aggiunto a $\Delta_{sSucc_P(i)}$ è rappresentato dal seguente ADLC

$$\begin{array}{l}\lambda x, y_1, y_2, x'', y_1'', y_2''.\exists x', y_1', y_2'.(\Delta_i(\langle x, y_1, y_2\rangle, \langle x', y_1', y_2'\rangle) \sqcap \\ \sqcap((y_1'' = y_1' + 1 \wedge y_2'' = y_2' + 1 \wedge y_1' > 0 \wedge x'' = x') \vee \\ \vee(y_2'' = y_2' + 1 \wedge \neg(y_1' > 0) \wedge x'' = x')))\end{array}$$

Come ulteriore esempio, sia stm_i lo statement $if(y > 0 \&\& y == 2 * u)$, con $InScope_P(i) = \{x, y\}$. Si ha quindi, dopo alcune semplificazioni, $\beta^+(b') = \exists u_1.(y' > 0 \wedge y' = 2 * u_1)$ e $\beta^-(b') = \exists u_1.(y' > 0 \wedge \neg(y' = 2 * u_1)) \vee \neg(y' > 0)$. Di conseguenza si ha che il nuovo insieme di path edge aggiunto a $\Delta_{Tsucc_P(i)}$ (a $\Delta_{Fsucc_P(i)}$, rispettivamente) è rappresentato dal seguente ADLC:

$$\lambda x, y, x', y'.\exists u_1.(\Delta_i(\langle x, y\rangle, \langle x', y'\rangle) \sqcap (y' > 0 \wedge y' = 2 * u_1 \wedge x' = x))$$

$$\begin{array}{c}(\lambda x, y, x', y'.\exists u_1.(\Delta_i(\langle x, y\rangle, \langle x', y'\rangle) \sqcap ((y' > 0 \wedge \neg(y' = 2 * u_1) \wedge x' = x) \vee \\ \vee(\neg(y' > 0) \wedge x' = x))), \text{ rispettivamente})\end{array}$$

Di fatto la relazione $\Delta_i \rightarrow \Delta_i^1$ appena descritta è la controparte "meccanizzabile" della relazione $\Pi \Rightarrow \Pi^1$ descritta nella sezione 3.1.1. Sfruttando tale relazione è possibile realizzare una procedura per il calcolo della raggiungibilità degli statement di un programma sottoposto a verifica. Tale procedura prevede che lo stato iniziale del sistema sia catturato dall'insieme di ADLC

$\Delta^0 \in \mathcal{A}$ definito come segue: $\Delta^0 \in \mathcal{A}(P)$ è tale che $\Delta^0_j = \lambda \mathbf{x}\mathbf{x}'.(\mathbf{x} = \mathbf{x}')$, con $\mathbf{x} \in InScope_P(j)$, se $j = First_P(main)$ e $\Delta^0_i = \lambda \mathbf{x}\mathbf{x}'.\bot$ con $\mathbf{x} \in InScope_P(i)$, per ogni $i \in N_P \setminus \{j\}$. Intuitivamente, con l'insieme di ADLC Δ^0 si vuole catturare il fatto che inizialmente l'unico nodo di cui si è certi della raggiungibilità è il primo nodo del main ed i path edge ad esso associati sono tutte le coppie di valutazioni che coincidono sui valori di tutte le variabili nello scope. Per tutti gli altri nodi la raggiungibilità non è ancora stata verificata e ciò è catturato dal fatto che non esistono coppie di valutazioni appartenenti alla semantica di Δ^0_j per ogni $j \in N_P \setminus \{First_P(main)\}$, infatti non esistono coppie di valutazioni che soddisfano il DLC insoddisfacibile, rappresentato dalla formula contraddittoria $\bot$.

Sia $\Delta \in \mathcal{A}(P)$, si definisce $[\![\Delta]\!]$ come la classe di insiemi indicizzata sull'insieme N_P dei nodi del CFG tale che per ogni $i \in N_P$ $[\![\Delta]\!]_i \in [\![\Delta]\!]$ e $[\![\Delta]\!]_i = [\![\Delta_i]\!]$. La correttezza e completezza dell'algoritmo deriva dal seguente teorema (per la dimostrazione si fa riferimento a [ABM07]).

Teorema 5 (Correttezza e Completezza) *Sia P un programma lineare e siano $\Delta, \Delta^1 \in \mathcal{A}(P)$. Vale che $\Delta \rightarrow \Delta^1$ se e soltanto se $[\![\Delta]\!] \Rightarrow [\![\Delta^1]\!]$*

Si denoti con $\rightarrow^*$ la chiusura riflessiva e transitiva della relazione $\rightarrow$ e sia $\Delta^0 \in \mathcal{A}(P)$ definito in modo tale che $\Delta^0_{First_P(main)} = \lambda \mathbf{x}\mathbf{x}'.(\mathbf{x} = \mathbf{x}')$ e $\Delta^0_j = \lambda \mathbf{x}\mathbf{x}'.\bot$ per ogni $j \in N_P \setminus \{First_P(main)\}$, vale dunque il seguente corollario, che è la controparte simbolica del corollario 5.

Corollario 6 *Sia P un programma lineare e sia $\Delta^0 \in \mathcal{A}(P)$ definito come sopra, allora per ogni $i \in N_P$, i è raggiungibile se e solo se $\Delta_i \not\sqsubseteq \lambda \mathbf{x}\mathbf{x}'.\bot$, per qualche Δ tale che $\Delta^0 \rightarrow^* \Delta$.*

3.3 Interprocedural Data Flow Analysis per astrazioni di programmi lineari con array

Le relazioni definite nelle precedenti sezioni fanno riferimento alla semantica di programmi lineari e pertanto erano utilizzabili con il meccanismo di astrazione precedente, che mappava il programma concreto P, definito nel dominio dei programmi lineari con array, in uno astratto $\widehat{P}$, definito nel dominio dei programmi lineari (senza array), attraverso la sostituzione di ogni statement stm_i di P con la sua versione astratta $\widehat{stm_i}$.

Con l'astrazione proposta in tale elaborato non esiste una tale corrispondenza sintattica. Esiste solo un'astrazione del modello a livello semantico,

pertanto non è possibile utilizzare le relazioni $\Rightarrow$ e $\rightarrow$ sull'astrazione sintattica di P (che non esiste), bensì è necesssario definire delle nuove relazioni "astratte" $\widehat{\Rightarrow}$ e $\widehat{\rightarrow}$ definite sugli statement concreti di P ma che agiscono tenendo presente la semantica astratta degli stessi.

Pertanto, in questa sezione l'algoritmo di interprocedural data flow analysis per il calcolo della raggiungibilità degli statement in programmi lineari, descritto in sezione 3.1, verrà adattato al calcolo della raggiungibilità all'interno di astrazioni di programmi lineari con array, generate secondo la nuova strategia di astrazione proposta in tale lavoro di tesi e descritta nel corso del capitolo 2.

3.3.1 Path Edge astratti

A tale fine, occorre ridefinire in maniera opportuna molti dei concetti, introdotti nelle sezioni precedenti di tale capitolo per i programmi lineari, in modo tale da estenderne la validità ad astrazioni di programmi lineari con array. In tale sezione verrà data pertanto la definizione di path edge astratto.

Sia P un programma lineare con array e sia $i \in N_P$ un nodo del CFG e sia $e = First_P(ProcOf_P(i))$. Un path edge astratto $\widehat{\pi_i} = \langle \widehat{w_e}, \widehat{w_i} \rangle$ incidente in i è una coppia di valutazioni tale che esiste un percorso astratto valido ammissibile $\langle First_P(main), \widehat{w_0} \rangle \xrightarrow{\widehat{\Sigma_0^e}}_P \langle e, \widehat{w_e} \rangle$ e un percorso astratto valido dello stesso livello $\langle e, \widehat{w_e} \rangle \xrightarrow{\widehat{\Sigma_e^i}}_P \langle i, \widehat{w_i} \rangle$ per qualche valutazione ammissibile $\widehat{w_0}$.

Sia $i \in N_P$, si denota con $\widehat{\mathbf{\Pi}}_i(P)$ l'insieme di path edge astratti incidenti in i associati all'astrazione di P. Il vertice i è raggiungibile nell'astrazione di P se e solo se $\widehat{\mathbf{\Pi}}_i(P) \neq \emptyset$.

Sia $\widehat{\mathcal{Q}}(P)$ la classe di insiemi indicizzata da N_P tale che $\widehat{\mathcal{Q}}_i(P) = \{\langle \widehat{w_1}, \widehat{w_2} \rangle \mid \widehat{w_1}, \widehat{w_2} : InScope_{\widehat{P}}(i) \rightarrow \mathcal{D}\}$ per ogni $i \in N_P$.

Siano $\widehat{\Pi}, \widehat{\Pi}^1$ due classi di insiemi indicizzate da N_P tali che $\widehat{\Pi}_i \subseteq \widehat{\mathcal{Q}}_i(P)$ e $\widehat{\Pi}_i^1 \subseteq \widehat{\mathcal{Q}}_i(P)$ per ogni $i \in N_P$.

Si definisce la relazione binaria $\widehat{\Pi} \widehat{\Rightarrow} \widehat{\Pi}^1$ come segue. Sia $i \in N_P$, $\widehat{\Pi}_j^1 = \widehat{\Pi}_j$ per ogni $j \notin Succ_P(i)$ e:

- se stm_i è uno `skip` o `return;`, allora

$$\widehat{\Pi}^1_{sSucc_P(i)} = \widehat{\Pi}_{sSucc_P(i)} \cup \widehat{\Pi}_i$$

- se stm_i è uno statement di assegnamento del tipo y=e`;`, allora

$$\widehat{\Pi}^1_{sSucc_P(i)} = \widehat{\Pi}_{sSucc_P(i)} \cup \{\langle \widehat{w_e}, \widehat{w_i}[d/y] \rangle \mid \langle \widehat{w_e}, \widehat{w_i} \rangle \in \widehat{\Pi}_i, d \in \overline{\widehat{w_i}}(e)\}$$

- se stm_i è uno statement di assegnamento del tipo a`[`e_1`]=`e_2`;`, allora

$$\widehat{\Pi}^1_{sSucc_P(i)} = \widehat{\Pi}_{sSucc_P(i)} \cup \{\langle \widehat{w_e}, \widehat{w_j} \rangle \mid \langle \widehat{w_e}, \widehat{w_i} \rangle \in \widehat{\Pi}_i \text{ e } \widehat{w_j} \in \Omega_i\}$$

$$\begin{array}{ll} \text{con } \Omega_i = \{\widehat{w_i}[d_1/a_{\theta^a_1}, \ldots, d_m/a_{\theta^a_m}] \mid \theta(a) = \{\theta^a_1, \theta^a_2, \ldots, \theta^a_m\} \text{ e} & \\ \quad d_l \in \overline{\widehat{w_i}}(e_2) & \text{se } \overline{\widehat{w_i}}(e_1\texttt{==}\theta^a_l) = \{1\} \\ \quad d_l = \widehat{w_i}(a_{\theta^a_l}) & \text{se } \overline{\widehat{w_i}}(e_1\texttt{==}\theta^a_l) = \{0\} \\ \quad d_l \in \overline{\widehat{w_i}}(e_2) \cup \{\widehat{w_i}(a_{\theta^a_l})\} & \text{altrimenti} \\ \quad \text{con } 1 \leq l \leq m\} & \end{array}$$

In un assegnamento ad un array a, i path edge propagati sono calcolati a partire da quelli del nodo i, aggiornando però i valori delle variabili a_θ per ogni $\theta \in \theta(a)$. Ad ogni variabile a_θ per qualche $\theta \in \theta(a)$ viene assegnato uno dei valori possibili per l'espressione e_2 rispetto alla valutazione $\overline{\widehat{w_i}}$, nel caso in cui a_θ modella proprio l'unico elemento di a indicizzato dall'unico valore possibile per e_1 (condizione $\overline{\widehat{w_i}}(e_1\texttt{==}\theta^a_l) = \{1\}$); se invece a_θ modella un elemento dell'array non indicizzato dai possibili valori dell'espressione e_1, allora il suo valore non viene modificato ($\overline{\widehat{w_i}}(e_1\texttt{==}\theta^a_l) = \{0\}$); infine, se e_1 ammette sia un valore che coincide con quello di θ che valori differenti (ultimo caso della definizione precedente), ovvero a_θ modella uno degli elementi dell'array indicizzati dai valori di e_1 (ci sono però anche possibili valori di e_1 che indicizzano altri elementi), allora la valutazione di a_θ può, in maniera non deterministica, essere aggiornata con uno dei valori possibili per e_2 oppure restare inalterata.

- se stm_i è uno statement del tipo `if(`e`)`, `while(`e`)`, `assume(`e`);` oppure `assert(`e`);`, allora

$$\begin{array}{l} \widehat{\Pi}^1_{Tsucc_P(i)} = \widehat{\Pi}_{Tsucc_P(i)} \cup \{\langle \widehat{w_e}, \widehat{w_i} \rangle \in \widehat{\Pi}_i \mid d \in \overline{\widehat{w_i}}(e) \text{ per qualche } d \neq 0\} \\ \widehat{\Pi}^1_{Fsucc_P(i)} = \widehat{\Pi}_{Fsucc_P(i)} \cup \{\langle \widehat{w_e}, \widehat{w_i} \rangle \in \widehat{\Pi}_i \mid 0 \in \overline{\widehat{w_i}}(e)\} \end{array}$$

- se stm_i è una chiamata a procedura pr`(a);`, allora

$$\begin{array}{l} \widehat{\Pi}^1_{sSucc_P(i)} = \widehat{\Pi}_{sSucc_P(i)} \cup \{\langle \widehat{w_j}, \widehat{w_j} \rangle \mid \langle \widehat{w_e}, \widehat{w_i} \rangle \in \widehat{\Pi}_i,\ \widehat{w_j}(\mathbf{y}) \in \overline{\widehat{w_i}}(\mathbf{a}), \\ \qquad\qquad \widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g}),\ \mathbf{g} = Globals_{\widehat{P}} \text{ e} \\ \qquad\qquad \mathbf{y} = Formals_{\widehat{P}}(First_P(pr))\} \end{array}$$

- se $i = Exit_P(pr)$ è l'exit node corrispondente alla procedura pr, allora

$$\begin{aligned}\widehat{\Pi}^1_j = \widehat{\Pi}_j \cup \{\langle\widehat{w_e}, \widehat{w_j}\rangle \mid \langle\widehat{w_e}, \widehat{w_k}\rangle \in \widehat{\Pi}_k,\ \langle\widehat{w_h}, \widehat{w_i}\rangle \in \widehat{\Pi}_i,\ \widehat{w_h}(\mathbf{y}) \in \overline{\widehat{w_k}}(\mathbf{a}),\\ \widehat{w_h}(\mathbf{g}) = \widehat{w_k}(\mathbf{g}),\ \widehat{w_j}(\mathbf{z}) = \widehat{w_k}(\mathbf{z}),\ \widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g}),\\ j = RetPt_P(k),\ \mathbf{y} = Formals_{\widehat{P}}(First_P(pr)),\\ \mathbf{z} = Locals_{\widehat{P}}(k) \text{ e } \mathbf{g} = Globals_{\widehat{P}}\}\end{aligned}$$

  se $j \in Succ_P(i)$.

I seguenti teoremi garantiscono la correttezza e completezza dell'algoritmo basato sulla relazione $\widehat{\Rightarrow}$ appena definita.

Teorema 6 (Correttezza) *Se $\widehat{\Pi}\widehat{\Rightarrow}\widehat{\Pi}^1$, allora per ogni $i \in N_P$ $\widehat{\Pi}_i \subseteq \widehat{\mathbf{\Pi}}_i(P)$ implica che per ogni $i \in N_P$ $\widehat{\Pi}^1_i \subseteq \widehat{\mathbf{\Pi}}_i(P)$.*

Dimostrazione Per ipotesi vale che $\widehat{\Pi} \subseteq \widehat{\mathbf{\Pi}}(P)$[4]. Inoltre, poichè vale $\widehat{\Pi}\widehat{\Rightarrow}\widehat{\Pi}^1$, allora $\widehat{\Pi}^1$ si ottiene da $\widehat{\Pi}$ attraverso l'applicazione di uno dei casi della definizione di $\widehat{\Rightarrow}$. Quindi, sia i un generico nodo in N_P, se $j \notin Succ_P(i)$, allora $\widehat{\Pi}^1_j = \widehat{\Pi}_j$ e poichè $\widehat{\Pi} \subseteq \widehat{\mathbf{\Pi}}(P)$, allora $\widehat{\Pi}_j \subseteq \widehat{\mathbf{\Pi}}_j(P)$ e quindi anche $\widehat{\Pi}^1_j \subseteq \widehat{\mathbf{\Pi}}_j(P)$. Invece, se $j \in Succ_P(i)$, allora $\widehat{\Pi}^1_j = \widehat{\Pi}_j \cup \widehat{\Pi}^*$ per qualche $\widehat{\Pi}^*$ e si vuole dimostrare che $\widehat{\Pi}^1_j \subseteq \widehat{\mathbf{\Pi}}_j(P)$. Questo equivale a dimostrare che vale sia $\widehat{\Pi}_j \subseteq \widehat{\mathbf{\Pi}}_j(P)$ che $\widehat{\Pi}^* \subseteq \widehat{\mathbf{\Pi}}_j(P)$. La prima relazione segue immediatamente dall'ipotesi $\widehat{\Pi} \subseteq \widehat{\mathbf{\Pi}}(P)$, mentre per la seconda bisogna mostrare che ogni coppia di valutazioni in $\widehat{\Pi}^*$ è un path edge astratto incidente in j e si procede per casi sul tipo di statement stm_i, assumendo che valga $First_P(main) = 1$:

- Se stm_i è un `return;` o uno `skip`, allora $j = sSucc_P(i)$ e $\widehat{\Pi}^* = \widehat{\Pi}_i$. Sia $\langle\widehat{w_e}, \widehat{w_i}\rangle \in \widehat{\Pi}_i$. Per ipotesi $\widehat{\Pi}_i \subseteq \widehat{\mathbf{\Pi}}_i(P)$. Questo significa che $\langle\widehat{w_e}, \widehat{w_i}\rangle$ è un path edge astratto incidente in i e quindi esiste un percorso astratto valido ammissibile $\langle 1, \widehat{w_0}\rangle \widehat{\xrightarrow{\Sigma^e_0}}_P \langle e, \widehat{w_e}\rangle \widehat{\xrightarrow{\Sigma^i_e}}_P \langle i, \widehat{w_i}\rangle$. Poichè secondo la definizione della relazione di transizione astratta $\langle j, \widehat{w_i}\rangle$ è un possibile successore di $\langle i, \widehat{w_i}\rangle$, ovvero vale $\langle i, \widehat{w_i}\rangle \widehat{\xrightarrow{\epsilon}}_P \langle j, \widehat{w_i}\rangle$, allora tale percorso può essere esteso al percorso

  $$\langle 1, \widehat{w_0}\rangle \widehat{\xrightarrow{\Sigma^e_0}}_P \langle e, \widehat{w_e}\rangle \widehat{\xrightarrow{\Sigma^i_e}}_P \langle i, \widehat{w_i}\rangle \widehat{\xrightarrow{\epsilon}}_P \langle j, \widehat{w_i}\rangle$$

  che è anch'esso un percorso astratto valido ammissibile. Quindi $\langle\widehat{w_e}, \widehat{w_i}\rangle$ è un path edge astratto incidente in j.

[4] $\widehat{\Pi} \subseteq \widehat{\mathbf{\Pi}}(P)$ è un'abbreviazione per $\widehat{\Pi}_i \subseteq \widehat{\mathbf{\Pi}}_i(P)$ per ogni $i \in N_P$

- Se stm_i è un assegnamento ad una variabile scalare y=e;, allora $j = sSucc_P(i)$ e

$$\widehat{\Pi}^* = \{\langle \widehat{w_e}, \widehat{w_i}[d/y]\rangle \mid \langle \widehat{w_e}, \widehat{w_i}\rangle \in \widehat{\Pi}_i, d \in \overline{\widehat{w_i}}(e)\}$$

Sia $\langle \widehat{w_1}, \widehat{w_2}\rangle \in \widehat{\Pi}^*$. Per la definizione di $\widehat{\Pi}^*$ si ha che $\widehat{w_1} = \widehat{w_e}$ e $\widehat{w_2} = \widehat{w_i}[d/y]$ per qualche $\langle \widehat{w_e}, \widehat{w_i}\rangle \in \widehat{\Pi}_i$ e $d \in \overline{\widehat{w_i}}(e)$. Per ipotesi $\widehat{\Pi}_i \subseteq \widehat{\mathbf{\Pi}}_i(P)$ da cui si ha che $\langle \widehat{w_e}, \widehat{w_i}\rangle$ è un path edge astratto incidente in i e quindi esiste un percorso astratto valido ammissibile $\langle 1, \widehat{w_0}\rangle \widehat{\xrightarrow{\Sigma_0^e}}_P \langle e, \widehat{w_e}\rangle \widehat{\xrightarrow{\Sigma_e^i}}_P \langle i, \widehat{w_i}\rangle$. Poichè secondo la definizione della relazione di transizione astratta $\langle j, \widehat{w_i}[d/y]\rangle$ è un possibile successore di $\langle i, \widehat{w_i}\rangle$, ovvero vale (per la condizione $d \in \overline{\widehat{w_i}}(e)$) $\langle i, \widehat{w_i}\rangle \widehat{\xrightarrow{\epsilon}}_P \langle j, \widehat{w_i}[d/y]\rangle$, allora tale percorso può essere esteso a

$$\langle 1, \widehat{w_0}\rangle \widehat{\xrightarrow{\Sigma_0^e}}_P \langle e, \widehat{w_e}\rangle \widehat{\xrightarrow{\Sigma_e^i}}_P \langle i, \widehat{w_i}\rangle \widehat{\xrightarrow{\epsilon}}_P \langle j, \widehat{w_i}[d/y]\rangle$$

che è anch'esso un percorso astratto valido ammissibile. Quindi $\langle \widehat{w_1}, \widehat{w_2}\rangle$ è un path edge astratto incidente in j.

- Se stm_i è un assegnamento ad array a[e_1]=e_2;, allora $j = sSucc_P(i)$ e

$$\widehat{\Pi}^* = \{\langle \widehat{w_e}, \widehat{w_j}\rangle \mid \langle \widehat{w_e}, \widehat{w_i}\rangle \in \widehat{\Pi}_i \text{ e } \widehat{w_j} \in \Omega_i\}$$

con $\Omega_i = \{\widehat{w_i}[d_1/a_{\theta_1^a}, \ldots, d_m/a_{\theta_m^a}] \mid \theta(a) = \{\theta_1^a, \theta_2^a, \ldots, \theta_m^a\}$ e

$d_l \in \overline{\widehat{w_i}}(e_2)$	se $\overline{\widehat{w_i}}(e_1 == \theta_l^a) = \{1\}$
$d_l = \widehat{w_i}(a_{\theta_l^a})$	se $\overline{\widehat{w_i}}(e_1 == \theta_l^a) = \{0\}$
$d_l \in \overline{\widehat{w_i}}(e_2) \cup \{\widehat{w_i}(a_{\theta_l^a})\}$	altrimenti
con $1 \leq l \leq m\}$	

Sia $\langle \widehat{w_1}, \widehat{w_2}\rangle \in \widehat{\Pi}^*$. Per la definizione di $\widehat{\Pi}^*$ si ha che $\widehat{w_1} = \widehat{w_e}$ e $\widehat{w_2} \in \Omega_i$ per qualche $\langle \widehat{w_e}, \widehat{w_i}\rangle \in \widehat{\Pi}_i$. Per ipotesi $\widehat{\Pi}_i \subseteq \widehat{\mathbf{\Pi}}_i(P)$ da cui si ha che $\langle \widehat{w_e}, \widehat{w_i}\rangle$ è un path edge astratto incidente in i e quindi esiste un percorso astratto valido ammissibile $\langle 1, \widehat{w_0}\rangle \widehat{\xrightarrow{\Sigma_0^e}}_P \langle e, \widehat{w_e}\rangle \widehat{\xrightarrow{\Sigma_e^i}}_P \langle i, \widehat{w_i}\rangle$. Poichè secondo la definizione della relazione di transizione astratta $\langle j, \widehat{w_2}\rangle$ è un possibile successore di $\langle i, \widehat{w_i}\rangle$, ovvero vale (per la condizione $\widehat{w_2} \in \Omega_i$) $\langle i, \widehat{w_i}\rangle \widehat{\xrightarrow{\epsilon}}_P \langle j, \widehat{w_2}\rangle$, allora tale percorso può essere esteso a

$$\langle 1, \widehat{w_0}\rangle \widehat{\xrightarrow{\Sigma_0^e}}_P \langle e, \widehat{w_e}\rangle \widehat{\xrightarrow{\Sigma_e^i}}_P \langle i, \widehat{w_i}\rangle \widehat{\xrightarrow{\epsilon}}_P \langle j, \widehat{w_2}\rangle$$

che è anch'esso un percorso astratto valido ammissibile. Quindi $\langle \widehat{w_1}, \widehat{w_2}\rangle$ è un path edge astratto incidente in j.

- Se stm_i è uno statement condizionale o iterativo (`if(`e`)`, `while(`e`)`, `assert(`e`);` oppure `assume(`e`);`), allora $j \in \{Tsucc_P(i), Fsucc_P(i)\}$. Se $j = Tsucc_P(i)$, allora

$$\widehat{\Pi}^* = \{\langle \widehat{w_e}, \widehat{w_i}\rangle \in \widehat{\Pi}_i \mid d \in \overline{\widehat{w_i}}(e) \text{ per qualche } d \neq 0\}$$

  Sia $\langle \widehat{w_1}, \widehat{w_2}\rangle \in \widehat{\Pi}^*$. Per la definizione di $\widehat{\Pi}^*$ si ha che $\widehat{w_1} = \widehat{w_e}$ e $\widehat{w_2} = \widehat{w_i}$ per qualche $\langle \widehat{w_e}, \widehat{w_i}\rangle \in \widehat{\Pi}_i$ tale che $d \in \overline{\widehat{w_i}}(e)$ per qualche $d \neq 0$. Per ipotesi $\widehat{\Pi}_i \subseteq \widehat{\mathbf{\Pi}}_i(P)$ da cui si ha che $\langle \widehat{w_e}, \widehat{w_i}\rangle$ è un path edge astratto incidente in i e quindi esiste un percorso astratto valido ammissibile $\langle 1, \widehat{w_0}\rangle \widehat{\xrightarrow{\Sigma_0^e}}_P \langle e, \widehat{w_e}\rangle \widehat{\xrightarrow{\Sigma_e^i}}_P \langle i, \widehat{w_i}\rangle$. Poichè secondo la definizione della relazione di transizione astratta $\langle j, \widehat{w_i}\rangle$ è un possibile successore di $\langle i, \widehat{w_i}\rangle$, ovvero vale $\langle i, \widehat{w_i}\rangle \widehat{\xrightarrow{\epsilon}}_P \langle j, \widehat{w_i}\rangle$, allora tale percorso può essere esteso a

$$\langle 1, \widehat{w_0}\rangle \widehat{\xrightarrow{\Sigma_0^e}}_P \langle e, \widehat{w_e}\rangle \widehat{\xrightarrow{\Sigma_e^i}}_P \langle i, \widehat{w_i}\rangle \widehat{\xrightarrow{\epsilon}}_P \langle j, \widehat{w_i}\rangle$$

  che è anch'esso un percorso astratto valido ammissibile. Quindi $\langle \widehat{w_1}, \widehat{w_2}\rangle$ è un path edge astratto incidente in j. La dimostrazione per $j = Fsucc_P(i)$ è analoga e quindi omessa.

- Se stm_i è una chiamata a procedura pr`(a);`, allora $j = sSucc_P(i)$ e

$$\widehat{\Pi}^* = \{\langle \widehat{w_j}, \widehat{w_j}\rangle \mid \langle \widehat{w_e}, \widehat{w_i}\rangle \in \widehat{\Pi}_i,\ \widehat{w_j}(\mathbf{y}) \in \overline{\widehat{w_i}}(\mathbf{a}),\ \widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g}),\ \mathbf{g} = Globals_{\widehat{P}} \text{ e } \mathbf{y} = Formals_{\widehat{P}}(First_P(pr))\}$$

  Sia $\langle \widehat{w_1}, \widehat{w_2}\rangle \in \widehat{\Pi}^*$. Per la definizione di $\widehat{\Pi}^*$ si ha che $\widehat{w_1} = \widehat{w_j}$ e $\widehat{w_2} = \widehat{w_j}$, dove $\widehat{w_j}$ è una valutazione tale che $\widehat{w_j}(\mathbf{y}) \in \overline{\widehat{w_i}}(\mathbf{a})$ e $\widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g})$ per qualche $\langle \widehat{w_e}, \widehat{w_i}\rangle \in \widehat{\Pi}_i$ con $\mathbf{g} = Globals_{\widehat{P}}$ e $\mathbf{y} = Formals_{\widehat{P}}(First_P(pr))$. Per ipotesi $\widehat{\Pi}_i \subseteq \widehat{\mathbf{\Pi}}_i(P)$ da cui si ha che $\langle \widehat{w_e}, \widehat{w_i}\rangle$ è un path edge astratto incidente in i e quindi esiste un percorso astratto valido ammissibile $\langle 1, \widehat{w_0}\rangle \widehat{\xrightarrow{\Sigma_0^e}}_P \langle e, \widehat{w_e}\rangle \widehat{\xrightarrow{\Sigma_e^i}}_P \langle i, \widehat{w_i}\rangle$. Poichè secondo la definizione della relazione di transizione astratta $\langle j, \widehat{w_j}\rangle$ è un possibile successore di $\langle i, \widehat{w_i}\rangle$, ovvero vale (per le condizioni $\widehat{w_j}(\mathbf{y}) \in \overline{\widehat{w_i}}(\mathbf{a})$ e $\widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g})$) $\langle i, \widehat{w_i}\rangle \widehat{\xrightarrow{\text{CALL}(RetPt_P(i), \widehat{w})}}_P \langle j, \widehat{w_j}\rangle$ dove $\widehat{w} : Locals_{\widehat{P}}(i) \rightarrow \mathcal{D}$ è tale che $\widehat{w}(\mathbf{x}) = \widehat{w_i}(\mathbf{x})$, con $\mathbf{x} = Locals_{\widehat{P}}(i)$, allora tale percorso può essere esteso a

$$\langle 1, \widehat{w_0}\rangle \widehat{\xrightarrow{\Sigma_0^e}}_P \langle e, \widehat{w_e}\rangle \widehat{\xrightarrow{\Sigma_e^i}}_P \langle i, \widehat{w_i}\rangle \widehat{\xrightarrow{\text{CALL}(RetPt_P(i), \widehat{w})}}_P \langle j, \widehat{w_j}\rangle$$

  che è anch'esso un percorso astratto valido ammissibile. Quindi $\langle \widehat{w_1}, \widehat{w_2}\rangle$ è un path edge astratto incidente in j.

- Se $i = Exit_P(pr)$, allora

$$\begin{aligned}\widehat{\Pi}^* = \{\langle\widehat{w_e},\widehat{w_j}\rangle \mid \langle\widehat{w_e},\widehat{w_k}\rangle \in \widehat{\Pi}_k,\ \langle\widehat{w_h},\widehat{w_i}\rangle \in \widehat{\Pi}_i,\ j = RetPt_P(k),\\ \widehat{w_h}(\mathbf{g}) = \widehat{w_k}(\mathbf{g}), \widehat{w_j}(\mathbf{z}) = \widehat{w_k}(\mathbf{z}), \widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g}), \widehat{w_h}(\mathbf{y}) \in \overline{\widehat{w_k}}(\mathbf{a}),\\ \mathbf{y} = Formals_{\widehat{P}}(First_P(pr)),\ \mathbf{z} = Locals_{\widehat{P}}(k)\,\mathrm{e}\,\mathbf{g} = Globals_{\widehat{P}}\}\end{aligned}$$

se $j \in Succ_P(i)$. Sia $\langle\widehat{w_1},\widehat{w_2}\rangle \in \widehat{\Pi}^*$. Per la definizione di $\widehat{\Pi}^*$ si ha che $\widehat{w_1} = \widehat{w_e}$ e $\widehat{w_2} = \widehat{w_j}$, dove $\widehat{w_j}$ è una valutazione tale che $\widehat{w_j}(\mathbf{z}) = \widehat{w_k}(\mathbf{z})$ e $\widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g})$ per qualche $\langle\widehat{w_e},\widehat{w_k}\rangle \in \widehat{\Pi}_k$ e $\langle\widehat{w_h},\widehat{w_i}\rangle \in \widehat{\Pi}_i$ tali che $\widehat{w_h}(\mathbf{y}) \in \overline{\widehat{w_k}}(\mathbf{a})$ e $\widehat{w_h}(\mathbf{g}) = \widehat{w_k}(\mathbf{g})$ con $j = RetPt_P(k)$, $\mathbf{y} = Formals_{\widehat{P}}(First_P(pr))$, $\mathbf{z} = Locals_{\widehat{P}}(k)$ e $\mathbf{g} = Globals_{\widehat{P}}$. Per ipotesi $\widehat{\Pi}_i \subseteq \widehat{\mathbf{\Pi}}_i(P)$ e $\widehat{\Pi}_k \subseteq \widehat{\mathbf{\Pi}}_k(P)$ da cui si ha che $\langle\widehat{w_e},\widehat{w_k}\rangle$ e $\langle\widehat{w_h},\widehat{w_i}\rangle$ sono path edge astratti incidenti in k e i rispettivamente e quindi esistono due percorsi astratti validi e ammissibili $\langle 1,\widehat{w_0}\rangle \xrightarrow{\widehat{\Sigma_0^e}}_P \langle e,\widehat{w_e}\rangle \xrightarrow{\widehat{\Sigma_e^k}}_P \langle k,\widehat{w_k}\rangle$ e $\langle 1,\widehat{w_0}\rangle \xrightarrow{\widehat{\Sigma_0^h}}_P \langle h,\widehat{w_h}\rangle \xrightarrow{\widehat{\Sigma_h^i}}_P \langle i,\widehat{w_i}\rangle$. Poichè k rappresenta il nodo della chiamata a procedura $pr(\mathbf{a})$; (condizione $j = RetPt_P(k)$), allora in base alla relazione di transizione astratta vale (per le condizioni $\widehat{w_h}(\mathbf{g}) = \widehat{w_k}(\mathbf{g})$ e $\widehat{w_h}(\mathbf{y}) \in \overline{\widehat{w_k}}(\mathbf{a})$) $\langle k,\widehat{w_k}\rangle \xrightarrow{\widehat{\mathrm{CALL}(RetPt_P(k),\widehat{w})}}_P \langle h,\widehat{w_h}\rangle$ dove $\widehat{w} : Locals_{\widehat{P}}(k) \to \mathcal{D}$ è tale che $\widehat{w}(\mathbf{z}) = \widehat{w_k}(\mathbf{z})$, con $\mathbf{z} = Locals_{\widehat{P}}(k)$. Esiste allora un percorso astratto valido ammissibile

$$\langle 1,\widehat{w_0}\rangle \xrightarrow{\widehat{\Sigma_0^e}}_P \langle e,\widehat{w_e}\rangle \xrightarrow{\widehat{\Sigma_e^k}}_P \langle k,\widehat{w_k}\rangle \xrightarrow{\widehat{\mathrm{CALL}(RetPt_P(k),\widehat{w})}}_P \langle h,\widehat{w_h}\rangle \xrightarrow{\widehat{\Sigma_h^i}}_P \langle i,\widehat{w_i}\rangle$$

Ancora una volta, in base alla relazione di transizione astratta $\langle j,\widehat{w_j}\rangle$ è un possibile successore di $\langle i,\widehat{w_i}\rangle$, ovvero vale (per le condizioni $\widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g})$, $\widehat{w}(\mathbf{z}) = \widehat{w_k}(\mathbf{z})$ e $\widehat{w_j}(\mathbf{z}) = \widehat{w_k}(\mathbf{z})$) $\langle i,\widehat{w_i}\rangle \xrightarrow{\widehat{\mathrm{RET}(j,\widehat{w})}}_P \langle j,\widehat{w_j}\rangle$, pertanto il percorso precedente può essere esteso a

$$\begin{aligned}&\langle 1,\widehat{w_0}\rangle \xrightarrow{\widehat{\Sigma_0^e}}_P \langle e,\widehat{w_e}\rangle \xrightarrow{\widehat{\Sigma_e^k}}_P \langle k,\widehat{w_k}\rangle \xrightarrow{\widehat{\mathrm{CALL}(RetPt_P(k),\widehat{w})}}_P \langle h,\widehat{w_h}\rangle \xrightarrow{\widehat{\Sigma_h^i}}_P\\ &\langle i,\widehat{w_i}\rangle \xrightarrow{\widehat{\mathrm{RET}(j,\widehat{w})}}_P \langle j,\widehat{w_j}\rangle\end{aligned}$$

che è anch'esso un percorso astratto valido ammissibile. Quindi $\langle\widehat{w_1},\widehat{w_2}\rangle$ è un path edge astratto incidente in j.

□

Ora, assumendo ancora che valga $First_P(main) = 1$, sia $\widehat{\Pi}^0$ tale che

$$\begin{aligned}\widehat{\Pi}_1^0 = \{\langle\widehat{w},\widehat{w}\rangle \mid \widehat{w}(x) \in \mathcal{D} \text{ per ogni } x \in InScope_{\widehat{P}}(1)\\ \text{e } \widehat{w}(\theta) = \overline{\widehat{w}}(e_\theta) \text{ per ogni } \theta \in \Theta \text{ con } e_\theta = \Theta_{ass}(\theta)\}\end{aligned}$$

e tale che $\widehat{\Pi}^0_j = \emptyset$ per ogni $j \in N_P \setminus \{1\}$ e si denoti con $\widehat{\Rightarrow}^*$ la chiusura riflessiva e transitiva di $\widehat{\Rightarrow}$, vale il seguente teorema.

Teorema 7 (Completezza) *Sia $\widehat{\Pi}^0$ definito come sopra. Se $\langle \widehat{w_h}, \widehat{w_j} \rangle \in \widehat{\mathbf{\Pi}}_j(P)$, allora esiste $\widehat{\Pi}^1$ tale che $\widehat{\Pi}^0 \widehat{\Rightarrow}^* \widehat{\Pi}^1$ e $\langle \widehat{w_h}, \widehat{w_j} \rangle \in \widehat{\Pi}^1_j$.*

Dimostrazione Sia $\langle \widehat{w_h}, \widehat{w_j} \rangle \in \widehat{\mathbf{\Pi}}_j(P)$, allora esiste un percorso astratto valido ammissibile $\widehat{\tau} = \langle 1, \widehat{w_0} \rangle \widehat{\xrightarrow{\Sigma^h_0}_P} \langle h, \widehat{w_h} \rangle \widehat{\xrightarrow{\Sigma^j_h}_P} \langle j, \widehat{w_j} \rangle$ per qualche valutazione $\widehat{w_0}$. La prova è per induzione sulla lunghezza di $\widehat{\tau}$.

Caso base Se $\widehat{\tau}$ ha lunghezza 0, cioè $\widehat{\tau} = \langle 1, \widehat{w_0} \rangle$, allora si considera $\widehat{\Pi}^1 = \widehat{\Pi}^0$. Vale sia $\widehat{\Pi}^0 \widehat{\Rightarrow}^* \widehat{\Pi}^1$ che $\langle \widehat{w_h}, \widehat{w_j} \rangle = \langle \widehat{w_0}, \widehat{w_0} \rangle \in \widehat{\Pi}^0_1 = \widehat{\Pi}^1_1$.

Passo induttivo Ipotesi induttiva: la tesi è valida per tutti i percorsi astratti validi e ammissibili $\widehat{\tau'}$ di lunghezza $n-1$ e la si vuole dimostrare per l'arbitrario percorso astratto valido ammissibile $\widehat{\tau} = \langle 1, \widehat{w_0} \rangle \widehat{\xrightarrow{\Sigma^h_0}_P} \langle h, \widehat{w_h} \rangle \widehat{\xrightarrow{\Sigma^j_h}_P} \langle j, \widehat{w_j} \rangle$ di lunghezza n. Sia $\widehat{\tau'} = \langle 1, \widehat{w_0} \rangle \widehat{\xrightarrow{\Sigma^e_0}_P} \langle e, \widehat{w_e} \rangle \widehat{\xrightarrow{\Sigma^i_e}_P} \langle i, \widehat{w_i} \rangle$ il prefisso di $\widehat{\tau}$ di lunghezza $n-1$. Poichè anche $\widehat{\tau'}$ è un percorso astratto valido ammissibile, allora $\langle \widehat{w_e}, \widehat{w_i} \rangle \in \widehat{\mathbf{\Pi}}_i(P)$ e quindi per ipotesi induttiva esiste $\widehat{\Pi}'$ tale che $\widehat{\Pi}^0 \widehat{\Rightarrow}^* \widehat{\Pi}'$ e $\langle \widehat{w_e}, \widehat{w_i} \rangle \in \widehat{\Pi}'_i$. La prova prosegue in base al tipo di statement stm_i.

- Se stm_i è un `return;` oppure uno `skip`, allora

  $$\widehat{\tau} = \langle 1, \widehat{w_0} \rangle \widehat{\xrightarrow{\Sigma^h_0}_P} \langle h, \widehat{w_h} \rangle \widehat{\xrightarrow{\Sigma^i_h}_P} \langle i, \widehat{w_i} \rangle \widehat{\xrightarrow{\epsilon}_P} \langle j, \widehat{w_j} \rangle$$

  con $\widehat{w_j} = \widehat{w_i}$ e $\langle h, \widehat{w_h} \rangle = \langle e, \widehat{w_e} \rangle$. Dalla definizione di $\widehat{\Rightarrow}$, esiste $\widehat{\Pi}^1$ tale che $\widehat{\Pi}' \widehat{\Rightarrow} \widehat{\Pi}^1$ (e quindi $\widehat{\Pi}^0 \widehat{\Rightarrow}^* \widehat{\Pi}^1$) con $\widehat{\Pi}^1_j = \widehat{\Pi}'_j \cup \widehat{\Pi}'_i$. Da ciò e da $\langle \widehat{w_e}, \widehat{w_i} \rangle \in \widehat{\Pi}'_i$ segue immediatamente che $\langle \widehat{w_h}, \widehat{w_j} \rangle \in \widehat{\Pi}^1_j$.

- Se stm_i è un assegnamento a variabile scalare y=e;, allora

  $$\widehat{\tau} = \langle 1, \widehat{w_0} \rangle \widehat{\xrightarrow{\Sigma^h_0}_P} \langle h, \widehat{w_h} \rangle \widehat{\xrightarrow{\Sigma^i_h}_P} \langle i, \widehat{w_i} \rangle \widehat{\xrightarrow{\epsilon}_P} \langle j, \widehat{w_j} \rangle$$

  con $\widehat{w_j} = \widehat{w_i}[d/y]$, con $d \in \overline{\widehat{w_i}}(e)$ e $\langle h, \widehat{w_h} \rangle = \langle e, \widehat{w_e} \rangle$. Dalla definizione di $\widehat{\Rightarrow}$, esiste $\widehat{\Pi}^1$ tale che $\widehat{\Pi}' \widehat{\Rightarrow} \widehat{\Pi}^1$ (e quindi $\widehat{\Pi}^0 \widehat{\Rightarrow}^* \widehat{\Pi}^1$) con

  $$\widehat{\Pi}^1_j = \widehat{\Pi}'_j \cup \{ \langle \widehat{w_e}, \widehat{w_i}[d/y] \rangle \mid \langle \widehat{w_e}, \widehat{w_i} \rangle \in \widehat{\Pi}'_i, d \in \overline{\widehat{w_i}}(e) \}$$

  Da ciò e da $\langle \widehat{w_e}, \widehat{w_i} \rangle \in \widehat{\Pi}'_i$ segue immediatamente che $\langle \widehat{w_h}, \widehat{w_j} \rangle \in \widehat{\Pi}^1_j$.

- Se stm_i è un assegnamento ad array a`[`e_1`]=`e_2`;`, allora

$$\widehat{\tau} = \langle 1, \widehat{w_0}\rangle \widehat{\xrightarrow{\Sigma_0^h}}_P \langle h, \widehat{w_h}\rangle \widehat{\xrightarrow{\Sigma_h^i}}_P \langle i, \widehat{w_i}\rangle \widehat{\xrightarrow{\epsilon}}_P \langle j, \widehat{w_j}\rangle$$

con $\widehat{w_j} = \widehat{w_i}[d_1/a_{\theta_1^a}, \ldots, d_m/a_{\theta_m^a}]$ dove
$\theta(a) = \{\theta_1^a, \theta_2^a, \ldots, \theta_m^a\}$
$d_l \in \overline{\widehat{w_i}}(e_2)$ se $\overline{\widehat{w_i}}(e_1 == \theta_l^a) = \{1\}$
$d_l = \widehat{w_i}(a_{\theta_l^a})$ se $\overline{\widehat{w_i}}(e_1 == \theta_l^a) = \{0\}$
$d_l \in \overline{\widehat{w_i}}(e_2) \cup \{\widehat{w_i}(a_{\theta_l^a})\}$ altrimenti
con $1 \leq l \leq m$

e con $\langle h, \widehat{w_h}\rangle = \langle e, \widehat{w_e}\rangle$. Dalla definizione di $\widehat{\Rightarrow}$, esiste $\widehat{\Pi}^1$ tale che $\widehat{\Pi}' \widehat{\Rightarrow} \widehat{\Pi}^1$ (e quindi $\widehat{\Pi}^0 \widehat{\Rightarrow}^* \widehat{\Pi}^1$) con

$$\widehat{\Pi}_j^1 = \widehat{\Pi}_j' \cup \{\langle \widehat{w_e}, \widehat{w_j}\rangle \mid \langle \widehat{w_e}, \widehat{w_i}\rangle \in \widehat{\Pi}_i' \text{ e } \widehat{w_j} \in \Omega_i\}$$

con $\Omega_i = \{\widehat{w} \mid \widehat{w} = \widehat{w_i}[d_1/a_{\theta_1^a}, \ldots, d_m/a_{\theta_m^a}]$ dove
$\theta(a) = \{\theta_1^a, \theta_2^a, \ldots, \theta_m^a\}$
$d_l \in \overline{\widehat{w_i}}(e_2)$ se $\overline{\widehat{w_i}}(e_1 == \theta_l^a) = \{1\}$
$d_l = \widehat{w_i}(a_{\theta_l^a})$ se $\overline{\widehat{w_i}}(e_1 == \theta_l^a) = \{0\}$
$d_l \in \overline{\widehat{w_i}}(e_2) \cup \{\widehat{w_i}(a_{\theta_l^a})\}$ altrimenti
con $1 \leq l \leq m\}$

Da ciò, da $\langle \widehat{w_e}, \widehat{w_i}\rangle \in \widehat{\Pi}_i'$ e da $\widehat{w_j} \in \Omega_i$ (per i vincoli su $\widehat{w_j}$ imposti dalla relazione di transizione astratta) segue immediatamente che $\langle \widehat{w_e}, \widehat{w_j}\rangle = \langle \widehat{w_h}, \widehat{w_j}\rangle \in \widehat{\Pi}_j^1$.

- Se stm_i è uno statement condizionale o iterativo (`if(`e`)`, `while(`e`)`, `assume(`e`);` oppure `assert(`e`);`), allora

$$\widehat{\tau} = \langle 1, \widehat{w_0}\rangle \widehat{\xrightarrow{\Sigma_0^h}}_P \langle h, \widehat{w_h}\rangle \widehat{\xrightarrow{\Sigma_h^i}}_P \langle i, \widehat{w_i}\rangle \widehat{\xrightarrow{\epsilon}}_P \langle j, \widehat{w_j}\rangle$$

con $\widehat{w_j} = \widehat{w_i}$, $j \in \{Tsucc_P(i), Fsucc_P(i)\}$ e $\langle h, \widehat{w_h}\rangle = \langle e, \widehat{w_e}\rangle$. Si consideri il caso in cui $j = Tsucc_P(i)$ (il caso in cui $j = Fsucc_P(i)$ si dimostra in maniera analoga). Dalla definizione di $\widehat{\Rightarrow}$, esiste $\widehat{\Pi}^1$ tale che $\widehat{\Pi}' \widehat{\Rightarrow} \widehat{\Pi}^1$ (e quindi $\widehat{\Pi}^0 \widehat{\Rightarrow}^* \widehat{\Pi}^1$) con

$$\widehat{\Pi}_j^1 = \widehat{\Pi}_j' \cup \{\langle \widehat{w_e}, \widehat{w_i}\rangle \in \widehat{\Pi}_i' \mid d \in \overline{\widehat{w_i}}(e) \text{ per qualche } d \neq 0\}$$

Poichè $j = Tsucc_P(i)$, dalla definizione della relazione di transizione astratta si ha che $d \in \overline{\widehat{w_i}}(e)$ per qualche $d \neq 0$. Da ciò e da $\langle \widehat{w_e}, \widehat{w_i} \rangle \in \widehat{\Pi}'_i$ segue immediatamente che $\langle \widehat{w_h}, \widehat{w_j} \rangle \in \widehat{\Pi}^1_j$.

- Se stm_i è una chiamata a procedura $pr\,\mathtt{(e)\,;}$, allora

$$\widehat{\tau} = \langle 1, \widehat{w_0} \rangle \widehat{\xrightarrow{\Sigma^e_0}}_P \langle e, \widehat{w_e} \rangle \widehat{\xrightarrow{\Sigma^i_e}}_P \langle i, \widehat{w_i} \rangle \widehat{\xrightarrow{\mathrm{CALL}(\widehat{RetPt_P}(i), \widehat{w})}}_P \langle j, \widehat{w_j} \rangle$$

con $j = First_P(pr)$, $\widehat{w_h} = \widehat{w_j}$ e $\langle e, \widehat{w_e} \rangle \widehat{\xrightarrow{\Sigma^i_e}}_P \langle i, \widehat{w_i} \rangle$ percorso astratto valido dello stesso livello e $\langle \widehat{w_e}, \widehat{w_i} \rangle$ un path edge astratto incidente in i. Quindi, per ipotesi induttiva, esiste $\widehat{\Pi}'$ tale che $\widehat{\Pi}^0 \widehat{\Rightarrow}^* \widehat{\Pi}'$ e $\langle \widehat{w_e}, \widehat{w_i} \rangle \in \widehat{\Pi}'_i$. Dalla definizione di $\widehat{\Rightarrow}$, esiste $\widehat{\Pi}^1$ tale che $\widehat{\Pi}' \widehat{\Rightarrow} \widehat{\Pi}^1$ (e quindi $\widehat{\Pi}^0 \widehat{\Rightarrow}^* \widehat{\Pi}^1$) con

$$\widehat{\Pi}^1_j = \widehat{\Pi}'_j \cup \{ \langle \widehat{w'_j}, \widehat{w'_j} \rangle \mid \langle \widehat{w'_e}, \widehat{w'_i} \rangle \in \widehat{\Pi}'_i,\ \widehat{w'_j}(\mathbf{y}) \in \overline{\widehat{w'_i}}(\mathbf{e}),\ \widehat{w'_j}(\mathbf{g}) = \widehat{w'_i}(\mathbf{g}), \\ \mathbf{g} = Globals_{\widehat{P}} \text{ e } \mathbf{y} = Formals_{\widehat{P}}(First_P(pr)) \}$$

Inoltre, dalla definizione della relazione di transizione astratta, si ha che $\widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g})$, dove $\mathbf{g} = Globals_{\widehat{P}}$ e $\widehat{w_j}(\mathbf{y}) \in \overline{\widehat{w_i}}(\mathbf{e})$, con $\mathbf{y} = Formals_{\widehat{P}}(First_P(pr))$. Da ciò e da $\langle \widehat{w_e}, \widehat{w_i} \rangle \in \widehat{\Pi}'_i$ segue immediatamente che $\langle \widehat{w_h}, \widehat{w_j} \rangle = \langle \widehat{w_j}, \widehat{w_j} \rangle \in \widehat{\Pi}^1_j$.

- Se $i = Exit_P(pr)$, allora

$$\widehat{\tau} = \langle 1, \widehat{w_0} \rangle \widehat{\xrightarrow{\Sigma^h_0}}_P \langle h, \widehat{w_h} \rangle \widehat{\xrightarrow{\Sigma^k_h}}_P \langle k, \widehat{w_k} \rangle \widehat{\xrightarrow{\mathrm{CALL}(\widehat{RetPt_P}(k), \widehat{w})}}_P \langle e, \widehat{w_e} \rangle \widehat{\xrightarrow{\Sigma^i_e}}_P \\ \langle i, \widehat{w_i} \rangle \widehat{\xrightarrow{\mathrm{RET}(j, \widehat{w})}}_P \langle j, \widehat{w_j} \rangle$$

con $e = First_P(pr)$ e $j \in Succ_P(i)$. Inoltre $\langle h, \widehat{w_h} \rangle \widehat{\xrightarrow{\Sigma^k_h}}_P \langle k, \widehat{w_k} \rangle$ e $\langle e, \widehat{w_e} \rangle \widehat{\xrightarrow{\Sigma^i_e}}_P \langle i, \widehat{w_i} \rangle$ sono percorsi astratti validi dello stesso livello e pertanto $\langle \widehat{w_h}, \widehat{w_k} \rangle$ e $\langle \widehat{w_e}, \widehat{w_i} \rangle$ sono path edge astratti incidenti in k e i rispettivamente. Quindi, per ipotesi induttiva, esiste $\widehat{\Pi}''$ tale che $\widehat{\Pi}^0 \widehat{\Rightarrow}^* \widehat{\Pi}''$ e $\langle \widehat{w_h}, \widehat{w_k} \rangle \in \widehat{\Pi}''_k$. Inoltre, attraverso l'applicazione ripetuta della relazione $\Rightarrow$ rispetto ai nodi che si trovano tra k ed i nel percorso $\widehat{\pi}$, si ha che esiste $\widehat{\Pi}'$ tale che $\widehat{\Pi}'' \widehat{\Rightarrow}^* \widehat{\Pi}'$ (e quindi anche $\widehat{\Pi}^0 \widehat{\Rightarrow}^* \widehat{\Pi}'$) con $\langle \widehat{w_e}, \widehat{w_i} \rangle \in \widehat{\Pi}'_i$. Per la monitonicità di $\Rightarrow$ si ha che $\widehat{\Pi}''_i \subseteq \widehat{\Pi}'_i$ per ogni $i \in N_P$, pertanto vale $\langle \widehat{w_h}, \widehat{w_k} \rangle \in \widehat{\Pi}''_k \subseteq \widehat{\Pi}'_k$ e dunque esiste $\widehat{\Pi}'$ tale che $\widehat{\Pi}^0 \widehat{\Rightarrow}^* \widehat{\Pi}'$ e tale che $\langle \widehat{w_h}, \widehat{w_k} \rangle \in \widehat{\Pi}'_k$ e $\langle \widehat{w_e}, \widehat{w_i} \rangle \in \widehat{\Pi}'_i$. Dalla definizione di $\widehat{\Rightarrow}$, esiste $\widehat{\Pi}^1$ tale

che $\widehat{\Pi}'\widehat{\Rightarrow}\widehat{\Pi}^1$ (e quindi $\widehat{\Pi}^0\widehat{\Rightarrow}^*\widehat{\Pi}^1$) con

$$\begin{aligned}\widehat{\Pi}^1_j = \widehat{\Pi}_j \cup \{\langle \widehat{w_h}, \widehat{w_j}\rangle \mid \langle \widehat{w_h}, \widehat{w_k}\rangle \in \widehat{\Pi}'_k,\ \langle \widehat{w_e}, \widehat{w_i}\rangle \in \widehat{\Pi}'_i,\ \widehat{w_e}(\mathbf{y}) \in \overline{\widehat{w_k}}(\mathbf{e}),\\ \widehat{w_e}(\mathbf{g}) = \widehat{w_k}(\mathbf{g}),\ \widehat{w_j}(\mathbf{z}) = \widehat{w_k}(\mathbf{z}),\ \widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g}),\\ j = RetPt_P(k),\ \mathbf{y} = Formals_{\widehat{P}}(First_P(pr)),\\ \mathbf{z} = Locals_{\widehat{P}}(k) \text{ e } \mathbf{g} = Globals_{\widehat{P}}\}\end{aligned}$$

se $j \in Succ_P(i)$. Inoltre, dalla definizione della relazione di transizione astratta per la chiamata a procedura, si ha che $\widehat{w_e}(\mathbf{g}) = \widehat{w_k}(\mathbf{g})$, dove $\mathbf{g} = Globals_{\widehat{P}}$ e $\widehat{w_e}(\mathbf{y}) \in \overline{\widehat{w_k}}(\mathbf{e})$, con $\mathbf{y} = Formals_{\widehat{P}}(First_P(pr))$. Invece, dalla relazione di transizione astratta per i, si ha che $\widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g})$, dove $\mathbf{g} = Globals_{\widehat{P}}$ e $\widehat{w_j}(\mathbf{z}) = \widehat{w_k}(\mathbf{z})$, con $\mathbf{z} = Locals_{\widehat{P}}(j)$. Da ciò, da $\langle \widehat{w_h}, \widehat{w_k}\rangle \in \widehat{\Pi}'_k$ e da $\langle \widehat{w_e}, \widehat{w_i}\rangle \in \widehat{\Pi}'_i$ segue immediatamente che $\langle \widehat{w_h}, \widehat{w_j}\rangle \in \widehat{\Pi}^1_j$.

□

Corollario 7 *Sia $\widehat{\Pi}^0$ definito come sopra, allora per ogni $i \in N_P$, i è raggiungibile se e solo se $\widehat{\Pi}_i \neq \emptyset$ per qualche $\widehat{\Pi}$ tale che $\widehat{\Pi}^0\widehat{\Rightarrow}^*\widehat{\Pi}$*

3.3.2 Procedura simbolica di model checking per astrazioni di programmi lineari con array

Verrà presentata di seguito una versione simbolica della relazione $\widehat{\Rightarrow}$ definita nella precedente sezione. Tale relazione è quella effettivamente utilizzata dal tool Eureka per la costruzione dei path edge associati ai nodi.

Prima, però, analogamente a quanto fatto per i programmi lineari, bisogna codificare gli statement del programma concreto lineare con array in formule dell'aritmetica lineare che ne catturino la semantica astratta. Mentre per i programmi lineari il problema era rappresentato esclusivamente dalle espressioni condizionali ($b\texttt{?}e_1\texttt{:}e_2$), ora bisogna eliminare dalle espressioni degli statement anche tutti gli acessi ad array $a\texttt{[}e\texttt{]}$ e codificarli utilizzando le variabili astratte che si riferiscono all'elemento dell'array a indicizzato dall'espressione e. Ad esempio, se $a\texttt{[}x\texttt{+1]}$ occorre all'interno dell'espressione e, e sia $\theta \in \theta(a)$ tale che $\overline{\widehat{w}}(x\texttt{+1}) = \overline{\widehat{w}}(\theta)$ in base alla valutazione corrente $\widehat{w}$, allora si vuole codificare, all'interno dell'espressione e, l'accesso ad array $a\texttt{[}x\texttt{+1]}$ con la variabile astratta a_θ. E' possibile però che esista più di una variabile $\theta \in \theta(a)$ la cui valutazione è compatibile con quella dell'espressione $x\texttt{+1}$ e quindi si vuole catturare il fatto che esistono k differenti variabili $\theta_{i_1}, \ldots, \theta_{i_k} \in \theta(a)$ tali che le rispettive variabili astratte $a_{\theta_{i_1}}, \ldots, a_{\theta_{i_k}}$ rappresentano lo stesso elemento dell'array a. Pertanto, mantenendo l'analogia con quanto fatto per i programmi lineari, si definisce la relazione

$e \to (B, E)$, che associa, ad ogni espressione del programma concreto e ad ogni variabile scalare del dominio astratto, un insieme B di condizioni ed un insieme E di espressioni tali che, se si verificano tutte le condizioni in B, la valutazione dell'espressione e è equivalente alla valutazione di ciascuna espressione $ne \in E$. Gli insiemi B e E, inoltre, sono tali che nessun espressione in $E \cup B$ contiene sottoespressioni condizionali nè accessi ad array. Sia $\widehat{V} = V_P \cup \Theta \cup \{a_\theta \mid a \in A_P, \theta \in \theta(a)\}$ l'insieme di variabili scalari appartenenti al dominio astratto, si definisce tale relazione come la più piccola relazione tale che $e \to (\emptyset, \{e\})$ per ogni $e \in \widehat{V} \cup \mathcal{D} \cup \{\mathtt{u}\}$ e chiusa rispetto alle seguenti regole di inferenza:

$$\frac{e_1 \to (B_1, E_1) \qquad e_2 \to (B_2, E_2)}{(e_1 \ op \ e_2) \to (B_1 \cup B_2, \{ne_1 \ op \ ne_2 \mid ne_1 \in E_1, \ ne_2 \in E_2\})}$$

se $op \in \{\mathtt{*}, \mathtt{+}, \mathtt{<}, \mathtt{<=}, \mathtt{>}, \mathtt{>=}, \mathtt{==}, \mathtt{!=}\}$.

$$\frac{b \to (B, nB) \qquad e_1 \to (B_1, E_1)}{(b?e_1 : e_2) \to (B \cup B_1 \cup nB^+, E_1)} \qquad \frac{b \to (B, nB) \qquad e_2 \to (B_2, E_2)}{(b?e_1 : e_2) \to (B \cup B_2 \cup nB^-, E_2)}$$

dove $nB^+ = \{b^+ \mid b \in nB\}$, $nB^- = \{b^- \mid b \in nB\}$ e $b^+(b^-)$ è $b\mathtt{!=0}$ ($b\mathtt{==0}$, rispettivamente) se b è un'espressione lineare e b ($\mathtt{!}b$, rispettivamente) se b è un'espressione booleana. Infine c'è il seguente schema di regole per eliminare le occorrenze di accessi ad array. Siano $\emptyset \subseteq \chi \subseteq \theta(a)$, $\mu_\chi = \{a_\theta \mid \theta \in \chi\}$, $\eta_\chi(E) = \bigcup_{ne \in E} \eta_\chi(ne)$ e $\eta_\chi(ne) = \{ne\mathtt{==}\theta \mid \theta \in \chi\} \cup \{ne\mathtt{!=}\theta \mid \theta \in \theta(a) \setminus \chi\}$

$$\frac{e \to (B, E)}{a[e] \to (B \cup \eta_\chi(E), \mu_\chi)} \, \emptyset \subset \chi \subseteq \theta(a) \qquad \frac{e \to (B, E)}{a[e] \to (B \cup \eta_\chi(E), \{u\})} \, \chi = \emptyset$$

Intuitivamente, si vuole catturare il fatto che l'i-esimo elemento dell'array a può essere modellato da tante variabili astratte a_θ quante sono le variabili $\theta \in \theta(a)$ la cui valutazione è uguale ad i. Pertanto, espressioni che includono riferimenti ad elementi di a sono equivalenti a tutte le espressioni ottenute sostituendo l'accesso ad array $a\mathtt{[}e\mathtt{]}$ con una delle variabili astratte che modella proprio l'elemento di a indicizzato da e. L'insieme χ rappresenta un generico sottoinsieme di variabili $\theta \in \theta(a)$. A condizione che tutte le variabili appartenenti a χ siano equivalenti ad e (o meglio alla sua codifica, ottenuta tramite l'applicazione induttiva dello stesso processo) e che tutte le altre variabili di $\theta(a)$ non coincidano con e (tale condizione è catturata dall'insieme $\eta_\chi(E)$), l'accesso ad array è modellato da tutte e sole le variabili a_θ tali che $\theta \in \chi$ (contenute nell'insieme μ_χ). Se $\chi = \emptyset$, allora nessuna variabile astratta a_θ modella l'elemento dell'array indicizzato da e, pertanto il valore dell'accesso

ad array è individuato da u ovvero è scelto non deterministicamente all'interno del dominio $\mathcal{D}$ e la condizione affinchè si verifichi tale situazione è catturata ancora dall'insieme $\eta_\chi(E)$ che, nel caso in cui $\chi = \emptyset$ contiene le disuguaglianze tra le variabili $\theta \in \theta(a)$ e le espressioni ne che rappresentano la codifica di e. Ad esempio, si consideri lo statement di assegnamento y=3+$a[x]$;, si ha che l'accesso ad array $a[x]$ occorre all'interno di un'espressione dello statement. Sia inoltre $\theta(a) = \{\theta_1, \theta_2\}$, allora valgono le seguenti relazioni $a[x] \rightarrow (\{\theta_1 = x, \theta_2 = x\}, \{a_{\theta_1}, a_{\theta_2}\})$, $a[x] \rightarrow (\{\theta_1 = x, \theta_2 \neq x\}, \{a_{\theta_1}\})$, $a[x] \rightarrow (\{\theta_1 \neq x, \theta_2 = x\}, \{a_{\theta_2}\})$ e $a[x] \rightarrow (\{\theta_1 \neq x, \theta_2 \neq x\}, \{\mathtt{u}\})$, che codificano l'accesso ad array, mentre per la codifica dell'intera espressione 3+$a[x]$, valgono le seguenti relazioni (3+$a[x]$) $\rightarrow (\{\theta_1 = x, \theta_2 = x\}, \{3+a_{\theta_1}, 3+a_{\theta_2}\})$, (3+$a[x]$) $\rightarrow (\{\theta_1 = x, \theta_2 \neq x\}, \{3+a_{\theta_1}\})$, (3+$a[x]$) $\rightarrow (\{\theta_1 \neq x, \theta_2 = x\}, \{3+a_{\theta_2}\})$ e (3+$a[x]$) $\rightarrow (\{\theta_1 \neq x, \theta_2 \neq x\}, \{3+\mathtt{u}\})$

Può essere mostrato che se $e \rightarrow (B, E)$, allora le espressioni lineari $ne \in E$ e le espressioni lineari booleane $nb \in B$ non contengono espressioni condizionali nè accessi ad array.

A questo punto vanno ridefinite le funzioni α e β, precedentemente definite in sezione 3.2.1, in modo da considerare anche l'eventuale presenza di accessi ad array negli statement. Pertanto, per la codifica di espressioni che rappresentano la guardia di uno statement condizionale, si utilizzano le funzioni $\widehat{\beta}^+$ e $\widehat{\beta}^-$, definite come segue:

$$\widehat{\beta}^+(e) = \bigvee \left\{ \exists U.(\bigwedge_{ne \in E} ne^+ \wedge \bigwedge B)^* \mid e \rightarrow (B, E) \right\}$$

$$\widehat{\beta}^-(e) = \bigvee \left\{ \exists U.(\bigwedge_{ne \in E} ne^- \wedge \bigwedge B)^* \mid e \rightarrow (B, E) \right\}$$

Siano ora y una variabile scalare e e un'espressione lineare con array, per la codifica della semantica di un assegnamento ad una variabile scalare si definisce:

$$\widehat{\alpha}(y, e) = \bigvee \left\{ \exists U.(\bigwedge_{ne \in E} y = ne \wedge \bigwedge B)^* \mid e \rightarrow (B, E) \right\}$$

che può essere generalizzata come segue per consentire la codifica di assegnamenti paralleli[5]:

[5]Da notare che, mentre per i programmi lineari astratti, nei quali gli statement di assegnamento parallelo erano necessari per l'astrazione degli statement di assegnamento ad array, nel caso di programmi lineari con array concreti gli statement di assegnamento parallelo possono essere esclusi dalla sintassi senza alcuna perdita di espressività del lin-

$$\widehat{\alpha}(\mathbf{y},\mathbf{e}) = \bigvee \left\{ \bigwedge_{i=1}^{k} (\exists U.(\bigwedge_{ne \in E_i} y_i = ne \wedge \bigwedge B_i)^*) \mid e_i \rightarrow (B_i, E_i) \,\mathrm{con}\, i = 1, \ldots, k \right\}$$

con $\mathbf{y}$ ed $\mathbf{e}$ k-uple di variabili scalari ed espressioni rispettivamente.

Infine, sia a una variabile array di P e siano e_1 ed e_2 espressioni, per la codifica di un assegnamento ad array $a\texttt{[}e_1\texttt{]=}e_2\texttt{;}$, si definisce:

$$\widehat{\alpha}(\overline{\mathbf{a}}, \mathbf{a}, a, e_1, e_2) = \bigvee \left\{ \exists U.(\bigwedge B \wedge \bigwedge B_2 \wedge \bigwedge_{\overline{a}_i \in E \cap \overline{\mathbf{a}}, ne_2 \in E_2} \overline{a}_i = ne_2 \wedge \bigwedge_{\overline{a}_i \in \overline{\mathbf{a}} \setminus E} \overline{a}_i = a_i)^* \mid a[e_1] \rightarrow (B, E) \text{ e } e_2 \rightarrow (B_2, E_2) \right\}$$

Si utilizzano due tuple di variabili $\overline{\mathbf{a}}$ e $\mathbf{a}$ che contengono le variabili dell'insieme $\widehat{A}_P(a) = \{a_\theta \mid \theta \in \theta(a)\}$ per ogni variabile array a del programma P. La tupla $\overline{\mathbf{a}}$ modella il valore delle variabili dopo l'assegnamento, mentre $\mathbf{a}$ modella i valori che le variabili assumevano precedentemente all'esecuzione dello statement. Pertanto, la formula che codifica la semantica di un assegnamento ad array prevede, per ogni possibile codifica dell'accesso ad array $a\texttt{[}e_1\texttt{]}$ (rappresentata dagli insiemi B ed E) e per ogni possibile codifica dell'espressione e_2 (insiemi B_2 ed E_2), la congiunzione delle condizioni ($\bigwedge B \wedge \bigwedge B_2$) e la congiunzione delle equivalenze tra tutte le variabili che modellano l'accesso ad array e quelle che modellano e_2 ($\bigwedge_{\overline{a}_i \in E \cap \overline{\mathbf{a}}, ne_2 \in E_2} \overline{a}_i = ne_2$). Infine tutte le variabili appartenenti a $\overline{\mathbf{a}}$ che non modellano l'accesso ad array vengono uguagliate al vecchio valore ($\bigwedge_{\overline{a}_i \in \overline{\mathbf{a}} \setminus E} \overline{a}_i = a_i$).

A titolo d'esempio si consideri di nuovo lo statement di assegnamento $y\texttt{=3+}a\texttt{[}x\texttt{];}$. La traduzione della sua semantica astratta in aritmetica lineare è la seguente:

$\widehat{\alpha}(y', \texttt{3+}a\texttt{[}x\texttt{]}) = (y' = 3 + a_{\theta_1} \wedge y' = 3 + a_{\theta_2} \wedge \theta_1 = x \wedge \theta_2 = x) \vee (y' = 3 + a_{\theta_1} \wedge \theta_1 = x \wedge \theta_2 \neq x) \vee (y' = 3 + a_{\theta_2} \wedge \theta_1 \neq x \wedge \theta_2 = x) \vee \exists u_1.(y' = 3 + u \wedge \theta_1 \neq x \wedge \theta_2 \neq x)$

Si consideri inoltre lo statement di assegnamento ad array $a\texttt{[}x\texttt{]=}y\texttt{;}$, con $\theta(a) = \{\theta_1, \theta_2\}$ e $\mathbf{a} = \langle a_{\theta_1}, a_{\theta_2} \rangle$ (e dunque si ha che $\mathbf{a}' = \langle a'_{\theta_1}, a'_{\theta_2} \rangle$). Di seguito è mostrata la fomula che codifica in aritmetica lineare la sua semantica, ottenuta attraverso l'istanziazione della formula $\widehat{\alpha}$ appena definita:

guaggio. La codifica di assegnamenti paralleli in aritmetica lineare è necessaria però per codificare la semantica astratta delle chiamate a procedura.

$\widehat{\alpha}(\mathbf{a}', \mathbf{a}, a', x, y) = (\theta_1 = x \wedge \theta_2 = x \wedge a'_{\theta_1} = y \wedge a'_{\theta_2} = y) \vee (\theta_1 = x \wedge \theta_2 \neq x \wedge a'_{\theta_1} = y \wedge a'_{\theta_2} = a_{\theta_2}) \vee (\theta_1 \neq x \wedge \theta_2 = x \wedge a'_{\theta_2} = y \wedge a'_{\theta_1} = a_{\theta_1}) \vee (\theta_1 \neq x \wedge \theta_2 \neq x \wedge a'_{\theta_1} = a_{\theta_1} \wedge a'_{\theta_2} = a_{\theta_2})$.

Il seguente lemma, che costituisce l'adattamento del lemma 2 alle funzioni $\widehat{\alpha}$, $\widehat{\beta}^+$ e $\widehat{\beta}^-$, appena definite, afferma formalmente la relazione tra la codifica simbolica e la semantica delle espressioni lineari.

Lemma 3 (a) *Sia y una variabile scalare ed e un'espressione lineare (booleana) con array, tale che y non occorre in e, allora per ogni valutazione $\widehat{w}$ valgono le seguenti relazioni:*

- $\models_{\widehat{w}} \widehat{\beta}^+(e)$ *se e solo se* $d \in \overline{\widehat{w}}(e)$, *per qualche* $d \neq 0$;
- $\models_{\widehat{w}} \widehat{\beta}^-(e)$ *se e solo se* $0 \in \overline{\widehat{w}}(e)$;
- *se e è un'espressione lineare con array, allora* $\overline{\widehat{w}}(e) = \{d \in \mathcal{D} \mid\models_{\widehat{w}[d/y]} \widehat{\alpha}(y, e)\}$.

(b) *Sia a una variabile array, e_1 ed e_2 espressioni lineari con array. Sia inoltre $\theta(a) = \{\theta^a_1, \ldots, \theta^a_m\}$ e si denotino con $\mathbf{a} = \langle a_{\theta^a_1}, \ldots, a_{\theta^a_m}\rangle$ e $\overline{\mathbf{a}} = \langle \overline{a}_{\theta^a_1}, \ldots, \overline{a}_{\theta^a_m}\rangle$ due m-uple di variabili tali che nessuna variabile in $\overline{\mathbf{a}}$ è presente in $\mathbf{a}$ e viceversa e tali che nessuna variabile in $\overline{\mathbf{a}}$ nè in $\mathbf{a}$ occorre in e_1 nè in e_2. Sia infine $D_l = \{\overline{d_l} \in \mathcal{D} \mid\models_{\widehat{w}[d_1/\overline{a}_{\theta^a_1}, \ldots, d_m/\overline{a}_{\theta^a_m}]} \widehat{\alpha}(\overline{\mathbf{a}}, \mathbf{a}, a, e_1, e_2)$ per qualche $d_1, \ldots, d_m \in \mathcal{D}$ con $\overline{d_l} = d_l\}$, allora per ogni valutazione $\widehat{w}$ e per ogni $1 \leq l \leq m$ vale una delle seguenti relazioni:*

- $D_l \subseteq \overline{\widehat{w}}(e_2)$ *se* $\overline{\widehat{w}}(e_1 \texttt{==} \theta^a_l) = \{1\}$;
- $D_l = \{\widehat{w}(a_{\theta^a_l})\}$ *se* $\overline{\widehat{w}}(e_1 \texttt{==} \theta^a_l) = \{0\}$;
- $D_l \subseteq \overline{\widehat{w}}(e_2) \cup \{\widehat{w}(a_{\theta^a_l})\}$ *se* $\overline{\widehat{w}}(e_1 \texttt{==} \theta^a_l) = \{0, 1\}$

Dimostrazione (a) La prova procede per induzione sulla struttura dell'espressione e. Il caso base (per e costante, variabile o `u`) è immediato. Pertanto qui si mostra solo il passo induttivo. La prova procede per casi secondo la struttura di e_2.

- $e = e_1$ *op* e_2, con *op* $\in \{\texttt{+}, \texttt{*}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{==}, \texttt{!=}\}$. Sia $\widehat{w}$ una valutazione arbitraria e sia $\overline{d}$ un generico elemento di $\overline{\widehat{w}}(e)$ ($\overline{d} \in \overline{\widehat{w}}(e)$). Per definizione di $\overline{\widehat{w}}(e)$, $\overline{d} = d_1$ *op* d_2 per qualche $d_1 \in \overline{\widehat{w}}(e_1)$ e per qualche $d_2 \in \overline{\widehat{w}}(e_2)$. Poichè $e = e_1$ *op* e_2, allora sia e_1 che e_2 sono espressioni lineari con array e quindi per ipotesi induttiva vale che:

$$\overline{\widehat{w}}(e_i) = \{d \in \mathcal{D} \mid\models_{\widehat{w}[d/z_i]} \widehat{\alpha}(z_i, e_i)\} \text{ per } i = 1, 2$$

Quindi dalla definizione di $\widehat{\alpha}(z_i, e_i)$, si ha che per $i = 1, 2$

$$\models_{\widehat{w}[d_i/z_i]} \exists U_i.(\bigwedge_{ne_i \in \overline{E}_i} z_i = ne_i \wedge \bigwedge \overline{B}_i)^*$$

per qualche $e_i \rightarrow (\overline{B}_i, \overline{E}_i)$. Si assume senza perdita di generalità che $z_1 \neq z_2$ e che $U_1 \cap U_2 = \emptyset$ (altrimenti si possono rinominare le variabili in $U_2 \cup \{z_2\}$). Poichè nessuna variabile di U_1 occorre libera in $(\bigwedge_{ne_2 \in \overline{E}_2} z_2 = ne_2 \wedge \bigwedge \overline{B}_2)^*$ (perchè le uniche variabili libere sono quelle in U_2 e si è supposto $U_1 \cap U_2 = \emptyset$) e nessuna variabile di U_2 occorre libera in $(\bigwedge_{ne_1 \in \overline{E}_1} z_1 = ne_1 \wedge \bigwedge \overline{B}_1)^*$, allora segue che

$$\models_{\widehat{w}[d_1/z_1, d_2/z_2]} \exists U_1.\exists U_2.(\bigwedge_{ne_1 \in \overline{E}_1} z_1 = ne_1 \wedge \bigwedge_{ne_2 \in \overline{E}_2} z_2 = ne_2 \wedge \bigwedge \overline{B}_1 \wedge \bigwedge \overline{B}_2)^*$$

Sia $U = U_1 \cup U_2$, allora l'equazione precedente è equivalente a

$$\models_{\widehat{w}[d_1/z_1, d_2/z_2]} \exists U.(\bigwedge_{ne_1 \in \overline{E}_1} z_1 = ne_1 \wedge \bigwedge_{ne_2 \in \overline{E}_2} z_2 = ne_2 \wedge \bigwedge \overline{B}_1 \wedge \bigwedge \overline{B}_2)^* \quad (3.1)$$

Dalle definizioni di e^+ e e^- segue che

$$d_1 \; op \; d_2 \neq 0 \text{ se e solo se}$$

$$\models_{\widehat{w}[d_1/z_1, d_2/z_2]} \exists U.((z_1 \; op \; z_2)^+ \wedge \bigwedge_{ne_1 \in \overline{E}_1} z_1 = ne_1 \wedge \bigwedge_{ne_2 \in \overline{E}_2} z_2 = ne_2 \wedge \bigwedge \overline{B}_1 \wedge \bigwedge \overline{B}_2)^*$$

e

$$d_1 \; op \; d_2 = 0 \text{ se e solo se}$$

$$\models_{\widehat{w}[d_1/z_1, d_2/z_2]} \exists U.((z_1 \; op \; z_2)^- \wedge \bigwedge_{ne_1 \in \overline{E}_1} z_1 = ne_1 \wedge \bigwedge_{ne_2 \in \overline{E}_2} z_2 = ne_2 \wedge \bigwedge \overline{B}_1 \wedge \bigwedge \overline{B}_2)^*$$

Poichè $\overline{d} = d_1 \; op \; d_2$ e nè z_1 nè z_2 occorrono in nessuna espressione in $\overline{E}_1 \cup \overline{E}_2 \cup \overline{B}_1 \cup \overline{B}_2$, allora valgono le seguenti equazioni:

$$\models_{\widehat{w}} \exists U.(\bigwedge_{ne_1 \in \overline{E}_1, ne_2 \in \overline{E}_2} (ne_1 \; op \; ne_2)^+ \wedge \bigwedge \overline{B}_1 \wedge \bigwedge \overline{B}_2)^* \text{ se e solo se } \overline{d} \neq 0$$

e

$$\models_{\widehat{w}} \exists U.(\bigwedge_{ne_1 \in \overline{E}_1, ne_2 \in \overline{E}_2} (ne_1 \; op \; ne_2)^- \wedge \bigwedge \overline{B}_1 \wedge \bigwedge \overline{B}_2)^* \text{ se e solo se } \overline{d} = 0$$

Poichè $e = e_1\ op\ e_2$, allora $e \rightarrow (\overline{B}, \overline{E})$, con $\overline{B} = \overline{B}_1 \cup \overline{B}_2$ ed $\overline{E} = \{ne_1\ op\ ne_2 \mid ne_1 \in \overline{E}_1, ne_2 \in \overline{E}_2\}$. Pertanto le due equazioni precedenti sono equivalenti, rispettivamente, a

$$\models_{\widehat{w}} \exists U.(\bigwedge_{ne \in \overline{E}} ne^+ \wedge \bigwedge \overline{B})^* \text{ se e solo se } \overline{d} \neq 0 \tag{3.2}$$

e

$$\models_{\widehat{w}} \exists U.(\bigwedge_{ne \in \overline{E}} ne^- \wedge \bigwedge \overline{B})^* \text{ se e solo se } \overline{d} = 0 \tag{3.3}$$

Poichè $\overline{d}$ è stato scelto in maniera arbitraria tra gli elementi di $\overline{\widehat{w}}(e)$, allora si conclude che per ogni $\overline{d} \in \overline{\widehat{w}}(e)$ esiste una coppia $(\overline{B}, \overline{E})$ tale che $e \rightarrow (\overline{B}, \overline{E})$ e per cui valgono le equazioni 3.2 e 3.3. Ma per la definizione di β^+ (β^-), se vale $\models_{\widehat{w}} \exists U.(\bigwedge_{ne \in \overline{E}} ne^+ \wedge \bigwedge \overline{B})^*$ ($\models_{\widehat{w}} \exists U.(\bigwedge_{ne \in \overline{E}} ne^- \wedge \bigwedge \overline{B})^*$, rispettivamente) per qualche coppia $(\overline{B}, \overline{E})$ tale che $e \rightarrow (\overline{B}, \overline{E})$ allora vale $\models_{\widehat{w}} \widehat{\beta}^+(e)$ ($\models_{\widehat{w}} \widehat{\beta}^-(e)$, rispettivamente). Pertanto si conclude che se esiste $\overline{d} \in \overline{\widehat{w}}(e)$, con $\overline{d} \neq 0$, allora vale $\models_{\widehat{w}} \widehat{\beta}^+(e)$ e se $0 \in \overline{\widehat{w}}(e)$, allora vale $\models_{\widehat{w}} \widehat{\beta}^-(e)$. Inoltre, se e è un'espressione lineare con array (cioè $op \in \{\texttt{*},\texttt{+}\}$), allora dalla 3.1 segue che

$$\models_{\widehat{w}[d_1/z_1, d_2/z_2]} \exists U.(\bigwedge_{ne_1 \in \overline{E}_1, ne_2 \in \overline{E}_2} z_1\ op\ z_2 = ne_1\ op\ ne_2 \wedge \bigwedge \overline{B}_1 \wedge \bigwedge \overline{B}_2)^*$$

e da quest'ultima equazione si ottiene

$$\models_{\widehat{w}[\overline{d}/z]} \exists U.(\bigwedge_{ne_1 \in \overline{E}_1, ne_2 \in \overline{E}_2} z = ne_1\ op\ ne_2 \wedge \bigwedge \overline{B}_1 \wedge \bigwedge \overline{B}_2)^*$$

dove z è una nuova variabile che non occorre in nessuna delle espressioni in $\overline{E}_1 \cup \overline{E}_2 \cup \overline{B}_1 \cup \overline{B}_2$. Ma poichè $\overline{B} = \overline{B}_1 \cup \overline{B}_2$ ed $\overline{E} = \{ne_1\ op\ ne_2 \mid ne_1 \in \overline{E}_1, ne_2 \in \overline{E}_2\}$, l'equazione precedente è equivalente a

$$\models_{\widehat{w}[\overline{d}/z]} \exists U.(\bigwedge_{ne \in \overline{E}} z = ne \wedge \bigwedge \overline{B})^*$$

Dunque, dalla definizione di $\widehat{\alpha}$, si conclude immediatamente che $\models_{\widehat{w}[\overline{d}/z]} \widehat{\alpha}(z, e)$ e quindi $\overline{d} \in \{d \in \mathcal{D} \mid \models_{\widehat{w}[d/z]} \widehat{\alpha}(z, e)\}$.

Per l'altro verso della dimostrazione si assume che $\models_{\widehat{w}} \widehat{\beta}^{+}(e)$, $\models_{\widehat{w}} \widehat{\beta}^{-}(e)$ e che $\overline{d} \in \{d \in \mathcal{D} \mid \models_{\widehat{w}[d/z]} \widehat{\alpha}(z,e)\}$. Pertanto, dalle definizioni di $\widehat{\beta}^{+}$ e $\widehat{\beta}^{-}$ si ha che

$$\models_{\widehat{w}} \exists U.(\bigwedge_{ne \in \overline{E}} ne^{+} \wedge \bigwedge \overline{B})^{*} \text{ per qualche } (\overline{B},\overline{E}) \text{ tale che } e \rightarrow (\overline{B},\overline{E})$$

e

$$\models_{\widehat{w}} \exists U.(\bigwedge_{ne \in \overline{E}} ne^{-} \wedge \bigwedge \overline{B})^{*} \text{ per qualche } (\overline{B}',\overline{E}') \text{ tale che } e \rightarrow (\overline{B}',\overline{E}')$$

Poichè $e = e_1\ op\ e_2$, allora esistono $(\overline{B}_1,\overline{E}_1)$ e $(\overline{B}_2,\overline{E}_2)$ tali che $e_1 \rightarrow (\overline{B}_1,\overline{E}_1)$ e $e_2 \rightarrow (\overline{B}_2,\overline{E}_2)$ con $\overline{B} = \overline{B}_1 \cup \overline{B}_2$ ed $\overline{E} = \{ne_1\ op\ ne_2 \mid ne_1 \in \overline{E}_1, ne_2 \in \overline{E}_2\}$ ed esistono $(\overline{B}'_1,\overline{E}'_1)$ e $(\overline{B}'_2,\overline{E}'_2)$ tali che $e_1 \rightarrow (\overline{B}'_1,\overline{E}'_1)$ e $e_2 \rightarrow (\overline{B}'_2,\overline{E}'_2)$ con $\overline{B}' = \overline{B}'_1 \cup \overline{B}'_2$ ed $\overline{E}' = \{ne_1\ op\ ne_2 \mid ne_1 \in \overline{E}'_1, ne_2 \in \overline{E}'_2\}$. Pertanto le due equazioni precedenti sono equivalenti, rispettivamente, a

$$\models_{\widehat{w}} \exists U.(\bigwedge_{ne_1 \in \overline{E}_1, ne_2 \in \overline{E}_2} (ne_1\ op\ ne_2)^{+} \wedge \bigwedge \overline{B}_1 \wedge \bigwedge \overline{B}_2)^{*} \tag{3.4}$$

e

$$\models_{\widehat{w}} \exists U.(\bigwedge_{ne_1 \in \overline{E}'_1, ne_2 \in \overline{E}'_2} (ne_1\ op\ ne_2)^{-} \wedge \bigwedge \overline{B}'_1 \wedge \bigwedge \overline{B}'_2)^{*} \tag{3.5}$$

Dalla 3.4, devono esistere $d_1, d_2 \in \mathcal{D}$ tali che $d_1\ op\ d_2 \in \overline{\widehat{w}}(e)$ e tali che $d_1\ op\ d_2 \neq 0$. Dalla 3.5, invece, devono esistere $d'_1, d'_2 \in \mathcal{D}$ tali che $d'_1\ op\ d'_2 \in \overline{\widehat{w}}(e)$ e tali che $d'_1\ op\ d'_2 = 0$

Poichè $\overline{d} \in \{d \in \mathcal{D} \mid \models_{\widehat{w}[d/z]} \widehat{\alpha}(z,e)\}$, allora si ha che $\models_{\widehat{w}[\overline{d}/z]} \widehat{\alpha}(z,e)$ e, dalla definizione di $\widehat{\alpha}$ vale che

$$\models_{\widehat{w}[\overline{d}/z]} \exists U.(\bigwedge_{ne \in \overline{E}''} z = ne \wedge \bigwedge \overline{B}'')^{*}$$

per qualche $(\overline{B}'',\overline{E}'')$ tale che $e \rightarrow (\overline{B}'',\overline{E}'')$. Esistono $(\overline{B}''_1,\overline{E}''_1)$ e $(\overline{B}''_2,\overline{E}''_2)$ tali che $e_1 \rightarrow (\overline{B}''_1,\overline{E}''_1)$ e $e_2 \rightarrow (\overline{B}''_2,\overline{E}''_2)$, con $\overline{B}'' = \overline{B}''_1 \cup \overline{B}''_2$ e $\overline{E}'' = \{ne_1\, op\, ne_2 \mid ne_1 \in \overline{E}''_1, ne_2 \in \overline{E}''_2\}$ e quindi l'equazione precedente è equivalente a

$$\models_{\widehat{w}[\overline{d}/z]} \exists U.(\bigwedge_{ne_1 \in \overline{E}''_1, ne_2 \in \overline{E}''_2} z = ne_1\ op\ ne_2 \wedge \bigwedge \overline{B}''_1 \wedge \bigwedge \overline{B}''_2)^{*}$$

da cui si ottiene

$$\models_{\widehat{w}[\overline{d}/z]} \exists U.(\bigwedge_{ne_1\in\overline{E}_1'',ne_2\in\overline{E}_2''} z{=}ne_1 \ op\ ne_2 \wedge \bigwedge \overline{B}_1'')^* \wedge \exists U.(\bigwedge_{ne_1\in\overline{E}_1'',ne_2\in\overline{E}_2''} z{=}ne_1 \ op\ ne_2 \wedge \bigwedge \overline{B}_2'')^*$$

Siano ora z_1 e z_2 due nuove variabili. Dall'equazione precedente devono esistere $d_1'', d_2'' \in \mathcal{D}$ con $\overline{d} = d_1''\ op\ d_2''$ tali che

$$\models_{\widehat{w}[\overline{d}/z,d_1''/z_1,d_2''/z_2]} z{=}z_1 \ op\ z_2 \wedge \exists U.(\bigwedge_{ne_1\in\overline{E}_1''} z_1{=}ne_1 \wedge \bigwedge_{ne_2\in\overline{E}_2''} z_2{=}ne_2 \wedge \bigwedge \overline{B}_1'')^* \wedge \exists U.(\bigwedge_{ne_1\in\overline{E}_1''} z{=}ne_1 \wedge \bigwedge_{ne_2\in\overline{E}_2''} z_2{=}ne_2 \wedge \bigwedge \overline{B}_2'')^*$$

L'equazione precedente implica che

$$\models_{\widehat{w}[d_1''/z_1]} \widehat{\alpha}(z_1, e_1) \qquad \text{e} \qquad \models_{\widehat{w}[d_2''/z_2]} \widehat{\alpha}(z_2, e_2)$$

Per ipotesi induttiva $d_i \in \overline{\widehat{w}}(e_i)$ per $i = 1, 2$. Quindi, dalla definizione di $\overline{\widehat{w}}$, $\overline{d} = d_1''\ op\ d_2'' \in \overline{\widehat{w}}(e)$.

- $e = b?e_1 : e_2$, dove e_1 ed e_2 sono espressioni lineari con array. Sia $\widehat{w}$ una valutazione arbitraria e sia $\overline{d}$ un generico elemento di $\overline{\widehat{w}}(e)$ ($\overline{d} \in \overline{\widehat{w}}(e)$). Per ipotesi induttiva, si sa che $\overline{\widehat{w}}(e_i) = \{d \in \mathcal{D} \mid \models_{\widehat{w}[d/z_i]} \widehat{\alpha}(z_i, e_i)\}$ per $i = 1, 2$, $\models_{\widehat{w}} \widehat{\beta}^+(b)$ se e solo se esiste $d' \in \overline{\widehat{w}}(b)$, con $d' \neq 0$ e $\models_{\widehat{w}} \widehat{\beta}^-(b)$ se e solo se $0 \in \overline{\widehat{w}}(b)$. Dalla definizone di $\overline{\widehat{w}}$, vale alternativamente $\overline{d} \in \overline{\widehat{w}}(e_1)$ e $d' \neq 0 \in \overline{\widehat{w}}(b)$ oppure $\overline{d} \in \overline{\widehat{w}}(e_2)$ e $0 \in \overline{\widehat{w}}(b)$. Si considera solo il primo caso (la dimostrazione per il secondo è simile). Quindi, $\overline{d} \in \overline{\widehat{w}}(e_1)$ e $d' \neq 0 \in \overline{\widehat{w}}(b)$. Poichè $\overline{d} \in \overline{\widehat{w}}(e_1)$, per ipotesi induttiva vale $\models_{\widehat{w}[\overline{d}/z_1]} \widehat{\alpha}(z_1, e_1)$. Dalla definizione di $\widehat{\alpha}$ si ha che

$$\models_{\widehat{w}[\overline{d}/z_1]} \exists U_1.(\bigwedge_{ne_1\in\overline{E}_1} z_1 = ne_1 \wedge \bigwedge \overline{B}_1)^*$$

per qualche coppia $(\overline{B}_1, \overline{E}_1)$ tale che $e_1 \to (\overline{B}_1, \overline{E}_1)$. Poichè esiste $d' \in \overline{\widehat{w}}(b)$, con $d' \neq 0$, allora per ipotesi induttiva vale

$$\models_{\widehat{w}} \widehat{\beta}^+(b)$$

Dalla definizione di $\widehat{\beta}^+$, si ha che

$$\models_{\widehat{w}} \exists U'.(\bigwedge_{nb\in\overline{nB}'} nb^+ \wedge \bigwedge \overline{B}')$$

per qualche coppia $(\overline{B}', \overline{nB}')$ tale che $b \rightarrow (\overline{B}', \overline{nB}')$. Ancora, senza perdita di generalità, si può assumere $U_1 \cap U' = \emptyset$. Poichè nessuna variabile di U_1 occorre in libera in $(\bigwedge_{nb \in \overline{nB}'} nb^+ \wedge \bigwedge \overline{B}')^*$ e nessuna variabile di U' occorre libera in $(\bigwedge_{ne_1 \in \overline{E}_1} z_1 = ne_1 \wedge \bigwedge \overline{B}_1)^*$, allora si ottiene

$$\models_{\widehat{w}[\overline{d}/z_1]} \exists U_1.\exists U'.(\bigwedge_{ne_1 \in \overline{E}_1} z_1 = ne_1 \wedge \bigwedge \overline{B}_1 \wedge \bigwedge_{nb \in \overline{nB}'} nb^+ \wedge \bigwedge \overline{B}')^*$$

Ora, sotto l'assunzione fatta, esiste la coppia $(\overline{B}, \overline{E})$ tale che $e \rightarrow (\overline{B}, \overline{E})$, con $\overline{B} = \overline{B}' \cup \overline{B}_1 \cup \overline{nB}'^+$ e con $\overline{E} = \overline{E}_1$. Quindi, la formula precedente è equivalente a

$$\models_{\widehat{w}[\overline{d}/z_1]} \exists U_1.\exists U'.(\bigwedge_{ne \in \overline{E}} z_1 = ne \wedge \bigwedge \overline{B})^*$$

e dalla definizione di $\widehat{\alpha}$, segue che $\models_{\widehat{w}[\overline{d}/z_1]} \widehat{\alpha}(z_1, e)$ e quindi $\overline{d} \in \{d \in \mathcal{D} \mid \models_{\widehat{w}[d/z_1]} \widehat{\alpha}(z_1, e)\}$. Inoltre, dalla definizione di e^+ e dall'equazione precedente si ottiene

$$\models_{\widehat{w}} \exists U_1.\exists U'.(\bigwedge_{ne \in \overline{E}} ne^+ \wedge \bigwedge \overline{B})^* \text{ se e solo se } \overline{d} \neq 0$$

Ma $\models_{\widehat{w}} \exists U_1.\exists U'.(\bigwedge_{ne \in \overline{E}} ne^+ \wedge \bigwedge \overline{B})^*$ implica $\models_{\widehat{w}} \widehat{\beta}^+(e)$ e quindi si può concludere che se esiste un elemento $\overline{d} \in \overline{\widehat{w}}(e)$, con $\overline{d} \neq 0$, allora $\models_{\widehat{w}} \widehat{\beta}^+(e)$.

Per l'altro verso della dimostrazione, si consideri $\overline{d} \in \{d \in \mathcal{D} \mid \models_{\widehat{w}[d/z]} \widehat{\alpha}(z, e)\}$. Allora vale

$$\models_{\widehat{w}[\overline{d}/z]} \widehat{\alpha}(z, e)$$

ovvero

$$\models_{\widehat{w}[\overline{d}/z]} \exists U.(\bigwedge_{ne \in \overline{E}} z = ne \wedge \bigwedge \overline{B})^*$$

per qualche coppia $(\overline{B}, \overline{E})$ tale che $e \rightarrow (\overline{B}, \overline{E})$. Ci sono due casi:

- $\overline{E} = \overline{E_1}$ e $\overline{B} = \overline{B}_1 \cup \overline{B}' \cup \overline{nB}^+$ per qualche $(\overline{B}_1, \overline{E}_1)$, $(\overline{B}', \overline{nB})$ tali che $e_1 \rightarrow (\overline{B}_1, \overline{E}_1)$ e $b \rightarrow (\overline{B}', \overline{nB})$
- $\overline{E} = \overline{E_2}$ e $\overline{B} = \overline{B}_1 \cup \overline{B}' \cup \overline{nB}^-$ per qualche $(\overline{B}_2, \overline{E}_2)$, $(\overline{B}', \overline{nB})$ tali che $e_2 \rightarrow (\overline{B}_2, \overline{E}_2)$ e $b \rightarrow (\overline{B}', \overline{nB})$

Si considera solo il primo caso (la prova per il secondo si ottiene in maniera analoga). Quindi l'equazione precedente è equivalente a

$$\models_{\widehat{w}[\overline{d}/z]} \exists U.(\bigwedge_{ne_1 \in \overline{E}_1} z = ne_1 \wedge \bigwedge \overline{B}_1 \wedge \bigwedge \overline{B}' \wedge \bigwedge \overline{nB}^+)^* \tag{3.6}$$

da cui si ha

$$\models_{\widehat{w}[\overline{d}/z]} \exists U.(\bigwedge_{ne_1 \in \overline{E}_1} z = ne_1 \wedge \bigwedge \overline{B}_1)^* \wedge \exists U.(\bigwedge \overline{B}' \wedge \bigwedge \overline{nB}^+)^*$$

Dai due congiunti si ottiene $\models_{\widehat{w}[\overline{d}/z]} \widehat{\alpha}(z, e_1)$ e $\models_{\widehat{w}} \widehat{\beta}^+(b)$ e per ipotesi induttiva $\overline{d} \in \overline{\widehat{w}}(e_1)$ ed esiste $d' \in \overline{\widehat{w}}(b)$, con $d' \neq 0$. Poichè per la definizione di $\overline{\widehat{w}}$ vale $\overline{\widehat{w}}(e_1) \subseteq \overline{\widehat{w}}(e)$, allora si conclude che $\overline{d} \in \overline{\widehat{w}}(e)$. Inoltre dalla 3.6 si ottengono

$$\models_{\widehat{w}} \exists U.(\bigwedge_{ne_1 \in \overline{E}_1} ne_1^+ \wedge \bigwedge \overline{B}_1 \wedge \bigwedge \overline{B}' \wedge \bigwedge \overline{nB}^+)^* \text{ se e solo se } \overline{d} \neq 0$$

e

$$\models_{\widehat{w}} \exists U.(\bigwedge_{ne_1 \in \overline{E}_1} ne_1^- \wedge \bigwedge \overline{B}_1 \wedge \bigwedge \overline{B}' \wedge \bigwedge \overline{nB}^+)^* \text{ se e solo se } \overline{d} = 0$$

da cui si conclude che $\models_{\widehat{w}} \widehat{\beta}^+(e)$ implica che esiste $\overline{d} \in \overline{\widehat{w}}(e)$, con $d \neq 0$ e che $\models_{\widehat{w}} \widehat{\beta}^-(e)$ implica che $0 \in \overline{\widehat{w}}(e)$.

- $e = a[e_1]$, dove a è una variabile array ed e_1 è un'espressione lineare con array. Sia $\widehat{w}$ una valutazione arbitraria e sia $\overline{d}$ un generico elemento di $\overline{\widehat{w}}(e)$ ($\overline{d} \in \overline{\widehat{w}}(e)$). Si distinguono due casi:

  1. $\overline{\widehat{w}}(e_1) \subseteq \theta_1(a)$, allora $\overline{d} = \widehat{w}(a_{\theta'})$ per qualche $\theta' \in \theta(a)$ tale che $\widehat{w}(\theta') \in \overline{\widehat{w}}(e_1)$. Per ogni a_θ tale che $\overline{d} = \widehat{w}(a_\theta)$ vale $\overline{d} \in \{\widehat{w}(a_\theta)\} = \overline{\widehat{w}}(a_\theta)$. Poichè a_θ è una variabile induttiva, allora per il caso base vale

     $$\models_{\widehat{w}[\overline{d}/z]} \widehat{\alpha}(z, a_\theta)$$

     per ogni a_θ tale che $\overline{d} = \widehat{w}(a_\theta)$, ovvero

     $$\models_{\widehat{w}[\overline{d}/z]} \exists U'.(\bigwedge_{a_\theta \in \overline{A}} z = a_\theta)^* \tag{3.7}$$

     dove $\overline{A} = \{a_\theta \mid \overline{d} = \widehat{w}(a_\theta)\}$. Inoltre, poiche $\overline{\widehat{w}}(e_1) \subseteq \theta_1(a)$, allora esiste $(\overline{B}_1, \overline{E}_1)$ tale che $e_1 \to (\overline{B}_1, \overline{E}_1)$ e tale che vale

     $$\models_{\widehat{w}} \exists U_1.(\bigwedge \overline{B}_1 \wedge \bigwedge \eta_\chi(\overline{E}_1))^* \tag{3.8}$$

per qualche $\emptyset \subset \chi \subseteq \theta(a)$ tale che $\theta' \in \chi$ perchè esiste una valutazione di e_1 che coincide con quella di θ'. Pertanto, siano $\overline{B} = \overline{B}_1 \cup \eta_\chi(\overline{E}_1)$ e $\overline{E} = \mu_\chi$, allora $e \rightarrow (\overline{B}, \overline{E})$. Inoltre vale $\mu_\chi \subseteq \overline{A}$ infatti tutte le variabili $\theta \in \chi$ hanno la stessa valutazione e pertanto anche tutte le variabili $a_\theta \in \mu_\chi$ hanno la stessa valutazione e, visto che $\theta' \in \chi$, allora $a_{\theta'} \in \mu_\chi$ dove $\widehat{w}(a_{\theta'}) = \overline{d}$. Quindi per ogni $a_\theta \in \mu_\chi$, $\widehat{w}(a_\theta) = \overline{d}$ e ciò implica che $a_\theta \in \overline{A}$. Pertanto dalle equazioni 3.7 e 3.8 segue che

$$\models_{\widehat{w}[\overline{d}/z]} \exists U'.\exists U_1.(\bigwedge_{ne \in \overline{E}} z = ne \wedge \bigwedge \overline{B})^*$$

da cui $\models_{\widehat{w}[\overline{d}/z]} \widehat{\alpha}(z, e)$ e si conclude che $\overline{d} \in \{d \in \mathcal{D} \mid \models_{\widehat{w}[d/z]} \widehat{\alpha}(z, e)\}$. Inoltre dall'equazione precedente si ha

$$\models_{\widehat{w}} \exists U'.\exists U_1.(\bigwedge_{ne \in \overline{E}} ne^+ \wedge \bigwedge \overline{B})^* \text{ se e solo se } \overline{d} \neq 0$$

e

$$\models_{\widehat{w}} \exists U'.\exists U_1.(\bigwedge_{ne \in \overline{E}} ne^- \wedge \bigwedge \overline{B})^* \text{ se e solo se } \overline{d} = 0$$

da cui si ha che se esiste $\overline{d} \in \overline{\widehat{w}}(e)$, con $\overline{d} \neq 0$, allora $\models_{\widehat{w}} \widehat{\beta}^+(e)$ e se $0 \in \overline{\widehat{w}}(e)$, allora $\models_{\widehat{w}} \widehat{\beta}^-(e)$.
Per l'altro verso della dimostrazione si assume $\overline{d} \in \{d \in \mathcal{D} \mid \models_{\widehat{w}[d/z]} \widehat{\alpha}(z, e)\}$, pertanto vale $\models_{\widehat{w}[\overline{d}/z]} \widehat{\alpha}(z, e)$, ovvero

$$\models_{\widehat{w}[\overline{d}/z]} \exists U.(\bigwedge_{ne \in \overline{E}} z = ne \wedge \bigwedge \overline{B})^* \tag{3.9}$$

per qualche $(\overline{B}, \overline{E})$ tale che $e \rightarrow (\overline{B}, \overline{E})$. Dalla definizione di $\overline{\widehat{w}}$ può valere alternativamente $\overline{\widehat{w}}(e) = \{\widehat{w}(a_\theta) \mid \theta \in \theta(a) \text{ e } \widehat{w}(\theta) \in \overline{\widehat{w}}(e_1)\}$ se $\overline{\widehat{w}}(e_1) \subseteq \theta_1(a)$ oppure $\overline{\widehat{w}}(e) = \mathcal{D}$ altrimenti. Nel secondo caso ($\overline{\widehat{w}}(e) = \mathcal{D}$) vale banalmente che $\overline{d} \in \overline{\widehat{w}}(e)$. Si consideri dunque il caso in cui $\overline{\widehat{w}}(e) = \{\widehat{w}(a_\theta) \mid \theta \in \theta(a) \text{ e } \widehat{w}(\theta) \in \overline{\widehat{w}}(e_1)\}$ e $\overline{\widehat{w}}(e_1) \subseteq \theta_1(a)$. Poichè $\overline{\widehat{w}}(e_1) \subseteq \theta_1(a)$, allora non si può verificare $\overline{B} = \overline{B}_1 \cup \eta_\chi(\overline{E}_1)$ per qualche $(\overline{B}_1, \overline{E}_1)$, tale che $e_1 \rightarrow (\overline{B}_1, \overline{E}_1)$ e con $\chi = \emptyset$. Pertanto deve valere $\overline{B} = \overline{B}_1 \cup \eta_\chi(\overline{E}_1)$ e $\overline{E} = \mu_\chi$ per qualche $(\overline{B}_1, \overline{E}_1)$ tale che $e_1 \rightarrow (\overline{B}_1, \overline{E}_1)$ e per qualche $\emptyset \subset \chi \subseteq \theta(a)$. L'equazione 3.9 è equivalente a

$$\models_{\widehat{w}[\overline{d}/z]} \exists U.(\bigwedge_{a_\theta \in \mu_\chi} z = a_\theta \wedge \bigwedge \overline{B}_1 \wedge \bigwedge \eta_\chi(\overline{E}_1))^*$$

Ma $\models_{\widehat{w}[\overline{d}/z]} \exists U.(\bigwedge_{a_\theta \in \mu_\chi} z = a_\theta)^*$ implica $\models_{\widehat{w}[\overline{d}/z]} \widehat{\alpha}(z, a_\theta)$ per ogni $a_\theta \in \mu_\chi$, pertanto, per il caso base, $\overline{d} \in \overline{\widehat{w}}(a_\theta) = \{\widehat{w}(a_\theta)\} \subseteq \overline{\widehat{w}}(e)$.

2. $\overline{\widehat{w}}(e_1) \not\subseteq \theta_1(a)$, allora esiste $(\overline{B}_1, \overline{E}_1)$ tale che $e_1 \to (\overline{B}_1, \overline{E}_1)$ e tale che $e \to (\overline{B}_1 \cup \eta_\chi(\overline{E}_1), \{\mathtt{u}\})$, con $\chi = \emptyset$. Pertanto vale

$$\models_{\widehat{w}[\overline{d}/z]} \exists U.(z = \mathtt{u} \wedge \bigwedge \overline{B}_1 \wedge \bigwedge \eta_\chi(\overline{E}_1))^*$$

e quindi $\overline{d} \in \{d \in \mathcal{D} \mid \models_{\widehat{w}[d/z]} \widehat{\alpha}(z, e)\}$ e la dimostrazione proegue in maniera analoga ai casi precedenti.

(b) La prova procede per induzione sulla struttura dell'espressione e_2. Il caso base (per e_2 costante, variabile o $\mathtt{u}$) è immediato. Pertanto qui si mostra solo il passo induttivo. La prova procede per casi secondo la struttura di e_2.

- $e_2 = e_{2,1}\ op\ e_{2,2}$, con $op \in \{\mathtt{+}, \mathtt{*}, \mathtt{<}, \mathtt{<=}, \mathtt{>}, \mathtt{>=}, \mathtt{==}, \mathtt{!=}\}$. Si consideri un generico elemento $\overline{d} \in D_l$. Per la definizione di D_l esistono $d_1, \ldots, d_m \in \mathcal{D}$ tali che $d_l = \overline{d}$ e tali che

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta_1^a}, \ldots, d_m/\overline{a}_{\theta_m^a}]} \widehat{\alpha}(\overline{\mathbf{a}}, \mathbf{a}, a, e_1, e_2)$$

ovvero

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta_1^a}, \ldots, d_m/\overline{a}_{\theta_m^a}]} \exists U.(\bigwedge \overline{B} \wedge \bigwedge \overline{B}_2 \wedge \bigwedge_{\overline{a}_i \in \overline{E} \cap \overline{\mathbf{a}}, ne_2 \in \overline{E}_2} \overline{a}_i = ne_2 \wedge \bigwedge_{\overline{a}_i \in \overline{\mathbf{a}} \setminus \overline{E}} \overline{a}_i = a_i)^*$$

per qualche coppia $(\overline{B}, \overline{E})$ tale che $a\,[e_1] \to (\overline{B}, \overline{E})$ e per qualche coppia $(\overline{B}_2, \overline{E}_2)$ tale che $e_2 \to (\overline{B}_2, \overline{E}_2)$. Attraverso un passo di regola di inferenza si ha che $\overline{B} = \overline{B}_1 \cup \eta_\chi(\overline{E}_1)$ ed $\overline{E} = \mu_\chi$ per qualche coppia $(\overline{B}_1, \overline{E}_1)$ tale che $e_1 \to (\overline{B}_1, \overline{E}_1)$ e per qualche $\emptyset \subset \chi \subseteq \theta(a)$ oppure si ha che $\overline{E} = \{\mathtt{u}\}$ se $\chi = \emptyset$. Pertanto la formula precedente è equivalente a

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta_1^a}, \ldots, d_m/\overline{a}_{\theta_m^a}]} \exists U.(\bigwedge \overline{B}_1 \wedge \bigwedge \eta_\chi(\overline{E}_1) \wedge \bigwedge \overline{B}_2 \wedge \bigwedge_{\overline{a}_i \in \overline{E} \cap \overline{\mathbf{a}}, ne_2 \in \overline{E}_2} \overline{a}_i = ne_2 \wedge \bigwedge_{\overline{a}_i \in \overline{\mathbf{a}} \setminus \overline{E}} \overline{a}_i = a_i)^*$$

Si distinguono 3 casi:

1. se $\overline{\widehat{w}}(e_1 \mathtt{==} \theta_l^a) = \{1\}$, allora poichè deve valere

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta_1^a}, \ldots, d_m/\overline{a}_{\theta_m^a}]} \exists U.(\bigwedge \eta_\chi(\overline{E}_1))^*$$

allora deve valere anche $(ne_1 == \theta_l^a) \in \eta_\chi(\overline{E}_1)$ per ogni $ne_1 \in \overline{E}_1$ e quindi per la definizione di η_χ si ha che $\theta_l^a \in \chi$, da cui $a_{\theta_l^a} \in \mu_\chi = \overline{E}$ e dunque si ha che $\overline{a}_{\theta_l^a} \in \overline{E} \cap \overline{\mathbf{a}}$ e di conseguenza vale

$$\models_{\widehat{w}[d_l/\overline{a}_{\theta_l^a}]} \exists U.(\bigwedge_{ne_2 \in \overline{E}_2} \overline{a}_{\theta_l^a} = ne_2 \wedge \bigwedge \overline{B}_2)^*$$

Poichè $e_2 = e_{2,1}\ op\ e_{2,2}$, sia $e_{2,1}$ che $e_{2,2}$ sono espressioni lineari con array e si ha che $\overline{B}_2 = \overline{B}_{2,1} \cup \overline{B}_{2,2}$ e $\overline{E}_2 = \{ne_{2,1}\ op\ ne_{2,2} \mid ne_{2,1} \in \overline{E}_{2,1}$ e $ne_{2,2} \in \overline{E}_{2,2}\}$ per qualche $(\overline{B}_{2,1}, \overline{E}_{2,1})$, $(\overline{B}_{2,2}, \overline{E}_{2,2})$ tali che $e_{2,1} \rightarrow (\overline{B}_{2,1}, \overline{E}_{2,1})$ e $e_{2,2} \rightarrow (\overline{B}_{2,2}, \overline{E}_{2,2})$. Quindi la formula precedente è equivalente a

$$\models_{\widehat{w}[d_l/\overline{a}_{\theta_l^a}]} \exists U.(\bigwedge_{ne_{2,1} \in \overline{E}_{2,1}, ne_{2,2} \in \overline{E}_{2,2}} \overline{a}_{\theta_l^a} = ne_{2,1}\ op\ ne_{2,2} \wedge \bigwedge \overline{B}_{2,1} \wedge \bigwedge \overline{B}_{2,2})^*$$

da cui si ottiene

$$\begin{aligned}\models_{\widehat{w}[d_l/\overline{a}_{\theta_l^a}]} \exists U.(&\bigwedge_{ne_{2,1} \in \overline{E}_{2,1}, ne_{2,2} \in \overline{E}_{2,2}} \overline{a}_{\theta_l^a} = ne_{2,1}\ op\ ne_{2,2} \wedge \bigwedge \overline{B}_{2,1})^* \wedge \\ \wedge \exists U.(&\bigwedge_{ne_{2,1} \in \overline{E}_{2,1}, ne_{2,2} \in \overline{E}_{2,2}} \overline{a}_{\theta_l^a} = ne_{2,1}\ op\ ne_{2,2} \wedge \bigwedge \overline{B}_{2,2})^*\end{aligned}$$

Siano z_1, z_2 due nuove variabili. Dalla formula precedente, devono esistere $\overline{d}_1, \overline{d}_2 \in \mathcal{D}$, con $d_l = \overline{d}_1\ op\ \overline{d}_2$ e tali che

$$\begin{aligned}\models_{\widehat{w}[d_l/\overline{a}_{\theta_l^a}, \overline{d}_1/z_1, \overline{d}_2/z_2]} &\overline{a}_{\theta_l^a} = z_1\ op\ z_2 \wedge \\ &\wedge \exists U.(\bigwedge_{ne_{2,1} \in \overline{E}_{2,1}} z_1 = ne_{2,1} \wedge \bigwedge_{ne_{2,2} \in \overline{E}_{2,2}} z_2 = ne_{2,2} \wedge \bigwedge \overline{B}_{2,1})^* \wedge \\ &\wedge \exists U.(\bigwedge_{ne_{2,1} \in \overline{E}_{2,1}} z_1 = ne_{2,1} \wedge \bigwedge_{ne_{2,2} \in \overline{E}_{2,2}} z_2 = ne_{2,2} \wedge \bigwedge \overline{B}_{2,2})^*\end{aligned}$$

L'equazione precedente implica che

$$\models_{\widehat{w}[\overline{d}_1/z_1]} \widehat{\alpha}(z_1, e_{2,1}) \qquad \text{e} \qquad \models_{\widehat{w}[\overline{d}_2/z_2]} \widehat{\alpha}(z_2, e_{2,2})$$

Per la parte a) del lemma, $\overline{d}_i \in \overline{\widehat{w}}(e_{2,i})$ (per $i = 1, 2$). Quindi, dalla definizione di $\overline{\widehat{w}}(e_2)$, $\overline{d} = d_l = \overline{d}_1\ op\ \overline{d}_2 \in \overline{\widehat{w}}(e_2)$

2. se $\overline{\widehat{w}}(e_1 == \theta_l^a) = \{0\}$, allora poichè deve valere

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta_1^a}, \ldots, d_m/\overline{a}_{\theta_m^a}]} \exists U.(\bigwedge \eta_\chi(\overline{E}_1))^*$$

allora deve valere anche $(ne_1\,\texttt{!=}\theta_l^a) \in \eta_\chi(\overline{E}_1)$ per ogni $ne_1 \in \overline{E}_1$ e quindi per la definizione di η_χ si ha che $\theta_l^a \notin \chi$, da cui $a_{\theta_l^a} \notin \mu_\chi = \overline{E}$ e dunque si ha che $\overline{a}_{\theta_l^a} \in \overline{\mathbf{a}} \setminus \overline{E}$ e di conseguenza vale

$$\models_{\widehat{w}[d_l/\overline{a}_{\theta_l^a}]} \exists U.(\overline{a}_{\theta_l^a} = a_{\theta_l^a})^*$$

Poichè $a_{\theta_l^a} \in \widehat{V}$ è una variabile scalare, allora vale $a_{\theta_l^a} \rightarrow (\emptyset, \{a_{\theta_l^a}\})$ e quindi la formula precedente implica che

$$\models_{\widehat{w}[d_l/\overline{a}_{\theta_l^a}]} \widehat{\alpha}(\overline{a}_{\theta_l^a}, a_{\theta_l^a})$$

Per la parte a) del lemma $\overline{d} = d_l \in \overline{\widehat{w}}(a_{\theta_l^a}) = \{\widehat{w}(a_{\theta_l^a})\}$. Poichè $\overline{\widehat{w}}(a_{\theta_l^a}) = \{\widehat{w}(a_{\theta_l^a})\}$ è costituito da un solo elemento allora per un generico elemento $\overline{d} \in D_l$ vale che $\overline{d} = \widehat{w}(a_{\theta_l^a})$, pertanto si conclude che $D_l = \{\widehat{w}(a_{\theta_l^a})\}$.

3. se $\overline{\widehat{w}}(e_1\texttt{==}\theta_l^a) = \{0, 1\}$, allora poichè deve valere

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta_1^a},\ldots,d_m/\overline{a}_{\theta_m^a}]} \exists U.(\bigwedge \eta_\chi(\overline{E}_1))^*$$

allora vale anche $(ne_1\texttt{==}\theta_l^a) \in \eta_\chi(\overline{E}_1)$ per ogni $ne_1 \in \overline{E}_1$ oppure $(ne_1\,\texttt{!=}\theta_l^a) \in \eta_\chi(\overline{E}_1)$ per ogni $ne_1 \in \overline{E}_1$. Nel primo caso dalla definizione di η_χ si ha che $\theta_l^a \in \chi$, da cui $a_{\theta_l^a} \in \mu_\chi = \overline{E}$ e dunque si ha che $\overline{a}_{\theta_l^a} \in \overline{E} \cap \overline{\mathbf{a}}$. Nel secondo caso dalla definizione di η_χ si ha che $\theta_l^a \notin \chi$, da cui $a_{\theta_l^a} \notin \mu_\chi = \overline{E}$ e dunque si ha che $\overline{a}_{\theta_l^a} \in \overline{\mathbf{a}} \setminus \overline{E}$. La prova procede analizzando i due casi separatemente ognuno dei quali corrisponde ad uno dei due casi dimostrati in precedenza.

- $e_2 = b\texttt{?}e_{2,1}\texttt{:}e_{2,2}$, con $e_{2,1}$ ed $e_{2,2}$ espressioni lineari con array. Si consideri un generico elemento $\overline{d} \in D_l$. Per la definizione di D_l esistono $d_1, \ldots, d_m \in \mathcal{D}$ tali che $d_l = \overline{d}$ e tali che

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta_1^a},\ldots,d_m/\overline{a}_{\theta_m^a}]} \widehat{\alpha}(\overline{\mathbf{a}}, \mathbf{a}, a, e_1, e_2)$$

ovvero

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta_1^a},\ldots,d_m/\overline{a}_{\theta_m^a}]} \exists U.(\bigwedge \overline{B} \wedge \bigwedge \overline{B}_2 \wedge \bigwedge_{\overline{a}_i \in \overline{E} \cap \overline{\mathbf{a}}, ne_2 \in \overline{E}_2} \overline{a}_i = ne_2 \wedge \bigwedge_{\overline{a}_i \in \overline{\mathbf{a}} \setminus \overline{E}} \overline{a}_i = a_i)^*$$

per qualche coppia $(\overline{B}, \overline{E})$ tale che $a\,\texttt{[}e_1\texttt{]} \rightarrow (\overline{B}, \overline{E})$ e per qualche coppia $(\overline{B}_2, \overline{E}_2)$ tale che $e_2 \rightarrow (\overline{B}_2, \overline{E}_2)$. Attraverso un passo di regola di inferenza si ha che $\overline{B} = \overline{B}_1 \cup \eta_\chi(\overline{E}_1)$ ed $\overline{E} = \mu_\chi$ per qualche coppia

$(\overline{B}_1, \overline{E}_1)$ tale che $e_1 \rightarrow (\overline{B}_1, \overline{E}_1)$ e per qualche $\emptyset \subset \chi \subseteq \theta(a)$ oppure si ha che $\overline{E} = \{\mathtt{u}\}$ se $\chi = \emptyset$. Pertanto la formula precedente è equivalente a

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta^a_1},\ldots,d_m/\overline{a}_{\theta^a_m}]} \exists U.(\bigwedge \overline{B}_1 \wedge \bigwedge \eta_\chi(\overline{E}_1) \wedge \bigwedge \overline{B}_2 \wedge \bigwedge_{\overline{a}_i \in \overline{E} \cap \overline{\mathbf{a}}, ne_2 \in \overline{E}_2} \overline{a}_i = ne_2 \wedge \bigwedge_{\overline{a}_i \in \overline{\mathbf{a}} \setminus \overline{E}} \overline{a}_i = a_i)^*$$

Si distinguono 3 casi:

1. se $\overline{\widehat{w}}(e_1 \mathtt{==} \theta^a_l) = \{1\}$, allora poichè deve valere

   $$\models_{\widehat{w}[d_1/\overline{a}_{\theta^a_1},\ldots,d_m/\overline{a}_{\theta^a_m}]} \exists U.(\bigwedge \eta_\chi(\overline{E}_1))^*$$

   allora deve valere anche $(ne_1 \mathtt{==} \theta^a_l) \in \eta_\chi(\overline{E}_1)$ per ogni $ne_1 \in \overline{E}_1$ e quindi per la definizione di η_χ si ha che $\theta^a_l \in \chi$, da cui $a_{\theta^a_l} \in \mu_\chi = \overline{E}$ e dunque si ha che $\overline{a}_{\theta^a_l} \in \overline{E} \cap \overline{\mathbf{a}}$ e di conseguenza vale

   $$\models_{\widehat{w}[d_l/\overline{a}_{\theta^a_l}]} \exists U.(\bigwedge_{ne_2 \in \overline{E}_2} \overline{a}_{\theta^a_l} = ne_2 \wedge \bigwedge \overline{B}_2)^*$$

   Poichè $e_2 = b\mathtt{?}e_{2,1} : e_{2,2}$ si ha che $\overline{B}_2 = \overline{B}_b \cup \overline{B}_{2,1} \cup \overline{nB}^+$ e $\overline{E}_2 = \overline{E}_{2,1}$ oppure $\overline{B}_2 = \overline{B}_b \cup \overline{B}_{2,2} \cup \overline{nB}^-$ e $\overline{E}_2 = \overline{E}_{2,2}$ per qualche $(\overline{B}_b, \overline{nB})$, $(\overline{B}_{2,1}, \overline{E}_{2,1})$, $(\overline{B}_{2,2}, \overline{E}_{2,2})$ tali che $b \rightarrow (\overline{B}_b, \overline{nB})$, $e_{2,1} \rightarrow (\overline{B}_{2,1}, \overline{E}_{2,1})$ e $e_{2,2} \rightarrow (\overline{B}_{2,2}, \overline{E}_{2,2})$. Si considera solo il primo caso (per il secondo caso la prova è analoga), pertanto si ha che $\overline{B}_2 = \overline{B}_b \cup \overline{B}_{2,1} \cup \overline{nB}^+$ e $\overline{E}_2 = \overline{E}_{2,1}$ per qualche $(\overline{B}_b, \overline{nB})$, $(\overline{B}_{2,1}, \overline{E}_{2,1})$ tali che $b \rightarrow (\overline{B}_b, \overline{nB})$, $e_{2,1} \rightarrow (\overline{B}_{2,1}, \overline{E}_{2,1})$. Quindi la formula precedente è equivalente a

   $$\models_{\widehat{w}[d_l/\overline{a}_{\theta^a_l}]} \exists U.(\bigwedge_{ne_{2,1} \in \overline{E}_{2,1}} \overline{a}_{\theta^a_l} = ne_{2,1} \wedge \bigwedge \overline{B}_b \wedge \bigwedge \overline{B}_{2,1} \wedge \bigwedge \overline{nB}^+)^*$$

   da cui si ottiene

   $$\models_{\widehat{w}[d_l/\overline{a}_{\theta^a_l}]} \exists U.(\bigwedge_{ne_{2,1} \in \overline{E}_{2,1}} \overline{a}_{\theta^a_l} = ne_{2,1} \wedge \bigwedge \overline{B}_{2,1})^* \wedge \exists U.(\bigwedge \overline{B}_b \wedge \bigwedge \overline{nB}^+)^*$$

   Dai due congiunti si ottengono $\models_{\widehat{w}[d_l/\overline{a}_{\theta^a_l}]} \widehat{\alpha}(\overline{a}_{\theta^a_l}, e_{2,1})$ e $\models_{\widehat{w}} \widehat{\beta^+}(b)$ e per il caso a) del lemma si ha che $\overline{d} = d_l \in \overline{\widehat{w}}(e_{2,1})$ e che esiste $d' \in \overline{\widehat{w}}(b)$, con $d' \neq 0$ e quindi $\overline{\widehat{w}}(e_{2,1}) \subseteq \overline{\widehat{w}}(e_2)$. Pertanto si conclude che $\overline{d} \in \overline{\widehat{w}}(e_2)$.

2. se $\overline{\widehat{w}}(e_1\texttt{==}\theta_l^a) = \{0\}$, allora poichè deve valere

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta_1^a},\ldots,d_m/\overline{a}_{\theta_m^a}]} \exists U.(\bigwedge \eta_\chi(\overline{E}_1))^*$$

allora deve valere anche $(ne_1\texttt{!=}\theta_l^a) \in \eta_\chi(\overline{E}_1)$ per ogni $ne_1 \in \overline{E}_1$ e quindi per la definizione di η_χ si ha che $\theta_l^a \notin \chi$, da cui $a_{\theta_l^a} \notin \mu_\chi = \overline{E}$ e dunque si ha che $\overline{a}_{\theta_l^a} \in \overline{\mathbf{a}} \setminus \overline{E}$ e di conseguenza vale

$$\models_{\widehat{w}[d_l/\overline{a}_{\theta_l^a}]} \exists U.(\overline{a}_{\theta_l^a} = a_{\theta_l^a})^*$$

La dimostrazione procede in maniera analoga a quella per il caso $e_2 = e_{2,1}\ op\ e_{2,2}$.

3. se $\overline{\widehat{w}}(e_1\texttt{==}\theta_l^a) = \{0,1\}$, allora poichè deve valere

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta_1^a},\ldots,d_m/\overline{a}_{\theta_m^a}]} \exists U.(\bigwedge \eta_\chi(\overline{E}_1))^*$$

allora vale anche $(ne_1\texttt{==}\theta_l^a) \in \eta_\chi(\overline{E}_1)$ per ogni $ne_1 \in \overline{E}_1$ oppure $(ne_1\texttt{!=}\theta_l^a) \in \eta_\chi(\overline{E}_1)$ per ogni $ne_1 \in \overline{E}_1$. Nel primo caso dalla definizione di η_χ si ha che $\theta_l^a \in \chi$, da cui $a_{\theta_l^a} \in \mu_\chi = \overline{E}$ e dunque si ha che $\overline{a}_{\theta_l^a} \in \overline{E} \cap \overline{\mathbf{a}}$. Nel secondo caso dalla definizione di η_χ si ha che $\theta_l^a \notin \chi$, da cui $a_{\theta_l^a} \notin \mu_\chi = \overline{E}$ e dunque si ha che $\overline{a}_{\theta_l^a} \in \overline{\mathbf{a}} \setminus \overline{E}$. La prova procede analizzando i due casi separatemente ognuno dei quali corrisponde ad uno dei due casi dimostrati in precedenza.

- $e_2 = a\texttt{[}e_{2,1}\texttt{]}$, con $e_{2,1}$ espressione lineare con array. Si consideri un generico elemento $\overline{d} \in D_l$. Per la definizione di D_l esistono $d_1, \ldots, d_m \in \mathcal{D}$ tali che $d_l = \overline{d}$ e tali che

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta_1^a},\ldots,d_m/\overline{a}_{\theta_m^a}]} \widehat{\alpha}(\overline{\mathbf{a}}, \mathbf{a}, a, e_1, e_2)$$

ovvero

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta_1^a},\ldots,d_m/\overline{a}_{\theta_m^a}]} \exists U.(\bigwedge \overline{B} \wedge \bigwedge \overline{B}_2 \wedge \bigwedge_{\overline{a}_i \in \overline{E} \cap \overline{\mathbf{a}}, ne_2 \in \overline{E}_2} \overline{a}_i = ne_2 \wedge \bigwedge_{\overline{a}_i \in \overline{\mathbf{a}} \setminus \overline{E}} \overline{a}_i = a_i)^*$$

per qualche coppia $(\overline{B}, \overline{E})$ tale che $a\texttt{[}e_1\texttt{]} \rightarrow (\overline{B}, \overline{E})$ e per qualche coppia $(\overline{B}_2, \overline{E}_2)$ tale che $e_2 \rightarrow (\overline{B}_2, \overline{E}_2)$. Attraverso un passo di regola di inferenza si ha che $\overline{B} = \overline{B}_1 \cup \eta_\chi(\overline{E}_1)$ ed $\overline{E} = \mu_\chi$ per qualche coppia $(\overline{B}_1, \overline{E}_1)$ tale che $e_1 \rightarrow (\overline{B}_1, \overline{E}_1)$ e per qualche $\emptyset \subset \chi \subseteq \theta(a)$ oppure si ha che $\overline{E} = \{\texttt{u}\}$ se $\chi = \emptyset$. Pertanto la formula precedente è equivalente a

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta_1^a},\ldots,d_m/\overline{a}_{\theta_m^a}]} \exists U.(\bigwedge \overline{B}_1 \wedge \bigwedge \eta_\chi(\overline{E}_1) \wedge \bigwedge \overline{B}_2 \wedge \bigwedge_{\overline{a}_i \in \overline{E} \cap \overline{\mathbf{a}}, ne_2 \in \overline{E}_2} \overline{a}_i = ne_2 \wedge \bigwedge_{\overline{a}_i \in \overline{\mathbf{a}} \setminus \overline{E}} \overline{a}_i = a_i)^*$$

Si distinguono 3 casi:

1. se $\overline{\widehat{w}}(e_1\texttt{==}\theta^a_l) = \{1\}$, allora poichè deve valere

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta^a_1},\ldots,d_m/\overline{a}_{\theta^a_m}]} \exists U.(\bigwedge \eta_\chi(\overline{E}_1))^*$$

allora deve valere anche $(ne_1\texttt{==}\theta^a_l) \in \eta_\chi(\overline{E}_1)$ per ogni $ne_1 \in \overline{E}_1$ e quindi per la definizione di η_χ si ha che $\theta^a_l \in \chi$, da cui $a_{\theta^a_l} \in \mu_\chi = \overline{E}$ e dunque si ha che $\overline{a}_{\theta^a_l} \in \overline{E} \cap \overline{\mathbf{a}}$ e di conseguenza vale

$$\models_{\widehat{w}[d_l/\overline{a}_{\theta^a_l}]} \exists U.(\bigwedge_{ne_2 \in \overline{E}_2} \overline{a}_{\theta^a_l} = ne_2 \wedge \bigwedge \overline{B}_2)^* \tag{3.10}$$

Poichè $e_2 = a[\![e_{2,1}]\!]$ e poichè $\chi \neq \emptyset$ (perchè $\theta^a_l \in \chi$) si ha che $\overline{B}_2 = \overline{B}_{2,1} \cup \eta_\chi(\overline{E}_{2,1})$ e $\overline{E}_2 = \mu_\chi$ per qualche $(\overline{B}_{2,1}, \overline{E}_{2,1})$ tale che $e_{2,1} \rightarrow (\overline{B}_{2,1}, \overline{E}_{2,1})$ e per qualche $\emptyset \subset \chi \subseteq \theta(a)$. La formula 3.10 è equivalente a

$$\models_{\widehat{w}[d_l/\overline{a}_{\theta^a_l}]} \exists U.(\bigwedge_{a'_\theta \in \mu_\chi} \overline{a}_{\theta^a_l} = a'_\theta \wedge \bigwedge \overline{B}_{2,1} \wedge \bigwedge \eta_\chi(\overline{E}_{2,1}))^*$$

Ma $\models_{\widehat{w}[d_l/\overline{a}_{\theta^a_l}]} \exists U.(\bigwedge_{a'_\theta \in \mu_\chi} \overline{a}_{\theta^a_l} = a'_\theta)^*$ implica $\models_{\widehat{w}[d_l/\overline{a}_{\theta^a_l}]} \widehat{\alpha}(\overline{a}_{\theta^a_l}, a_\theta)$ per ogni $a_\theta \in \mu_\chi$, pertanto, per il caso base, $\overline{d} = d_l \in \overline{\widehat{w}}(a_\theta) = \{\widehat{w}(a_\theta)\} \subseteq \overline{\widehat{w}}(e)$.

2. se $\overline{\widehat{w}}(e_1\texttt{==}\theta^a_l) = \{0\}$, allora poichè deve valere

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta^a_1},\ldots,d_m/\overline{a}_{\theta^a_m}]} \exists U.(\bigwedge \eta_\chi(\overline{E}_1))^*$$

allora deve valere anche $(ne_1\texttt{!=}\theta^a_l) \in \eta_\chi(\overline{E}_1)$ per ogni $ne_1 \in \overline{E}_1$ e quindi per la definizione di η_χ si ha che $\theta^a_l \notin \chi$, da cui $a_{\theta^a_l} \notin \mu_\chi = \overline{E}$ e dunque si ha che $\overline{a}_{\theta^a_l} \in \overline{\mathbf{a}} \setminus \overline{E}$ e di conseguenza vale

$$\models_{\widehat{w}[d_l/\overline{a}_{\theta^a_l}]} \exists U.(\overline{a}_{\theta^a_l} = a_{\theta^a_l})^*$$

La dimostrazione procede in maniera analoga a quelle per i casi precedenti.

3. se $\overline{\widehat{w}}(e_1\texttt{==}\theta^a_l) = \{0,1\}$, allora poichè deve valere

$$\models_{\widehat{w}[d_1/\overline{a}_{\theta^a_1},\ldots,d_m/\overline{a}_{\theta^a_m}]} \exists U.(\bigwedge \eta_\chi(\overline{E}_1))^*$$

allora vale anche $(ne_1\texttt{==}\theta^a_l) \in \eta_\chi(\overline{E}_1)$ per ogni $ne_1 \in \overline{E}_1$ oppure $(ne_1\texttt{!=}\theta^a_l) \in \eta_\chi(\overline{E}_1)$ per ogni $ne_1 \in \overline{E}_1$. Nel primo caso dalla

definizione di η_χ si ha che $\theta_l^a \in \chi$, da cui $a_{\theta_l^a} \in \mu_\chi = \overline{E}$ e dunque si ha che $\overline{a}_{\theta_l^a} \in \overline{E} \cap \overline{\mathbf{a}}$. Nel secondo caso dalla definizione di η_χ si ha che $\theta_l^a \notin \chi$, da cui $a_{\theta_l^a} \notin \mu_\chi = \overline{E}$ e dunque si ha che $\overline{a}_{\theta_l^a} \in \overline{\mathbf{a}} \setminus \overline{E}$. La prova procede analizzando i due casi separatemente ognuno dei quali corrisponde ad uno dei due casi dimostrati in precedenza.

□

Con $\widehat{\mathcal{A}}(P)$ si denota la classe di insiemi di ADLC indicizzata sui nodi del grafo N_P tale che $\widehat{\mathcal{A}}_i(P)$ è l'insieme di ADLC di arietà $\mid InScope_{\widehat{P}}(i) \mid$ per ogni $i \in N_P$. Siano $\Delta, \Delta^1 \in \widehat{\mathcal{A}}(P)$[6], si definisce la relazione binaria $\Delta \widehat{\rightarrow} \Delta^1$ come segue. Sia $i \in N_P$, $\Delta_j^1 = \Delta_j$ per ogni $j \notin Succ_P(i)$ e

- se i corrisponde ad uno statement del tipo `skip` o ad un `return;`, allora

$$\Delta^1_{sSucc_P(i)} = \Delta_{sSucc_P(i)} \sqcup \Delta_i$$

- se i corrisponde ad uno statement di assegnamento del tipo y`=`e`;`, allora

$$\Delta^1_{sSucc_P(i)} = \Delta_{sSucc_P(i)} \sqcup \lambda \mathbf{x}\mathbf{x}''.\exists \mathbf{x}'.(\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \widehat{\alpha}(y'', e') \sqcap \mathbf{z}'' = \mathbf{z}')$$

  dove $\mathbf{x} = InScope_{\widehat{P}}(i)$ e $\mathbf{z} = InScope_{\widehat{P}}(i) \setminus \{y\}$.

- se i corrisponde ad uno statement di assegnamento del tipo a`[`e_1`]=`e_2`;`, allora

$$\Delta^1_{sSucc_P(i)} = \Delta_{sSucc_P(i)} \sqcup \lambda \mathbf{x}\mathbf{x}''.\exists \mathbf{x}'.(\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \widehat{\alpha}(\mathbf{a}'', \mathbf{a}', a'', e_1', e_2') \sqcap \mathbf{z}'' = \mathbf{z}')$$

  dove $\mathbf{x} = InScope_{\widehat{P}}(i)$, $\mathbf{a} = \{a_\theta \mid \theta \in \theta(a)\}$ e $\mathbf{z} = InScope_{\widehat{P}}(i) \setminus \mathbf{a}$.

- se i corrisponde ad uno statement condizionale o iteraticvo (`if(`b`)`, `while(`b`)`, `assume(`b`);` oppure `assert(`b`);`), allora

$$\Delta^1_{Tsucc_P(i)} = \Delta_{Tsucc_P(i)} \sqcup \lambda \mathbf{x}\mathbf{x}'.(\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \widehat{\beta}^+(b'))$$

$$\Delta^1_{Fsucc_P(i)} = \Delta_{Fsucc_P(i)} \sqcup \lambda \mathbf{x}\mathbf{x}'.(\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \widehat{\beta}^-(b'))$$

  con $\mathbf{x} = InScope_{\widehat{P}}(i)$

[6] Anche in questo caso con un abuso di notazione si semplifica l'espressione $\Delta_i \in \widehat{\mathcal{A}}_i(P)$ per ogni $i \in N_P$ con l'espressione $\Delta \in \widehat{\mathcal{A}}(P)$

- se i corrisponde ad una chiamata a procedura del tipo pr`(e);`, allora

$$\Delta^1_{sSucc_P(i)} = \Delta_{sSucc_P(i)} \sqcup \lambda \mathbf{w}\mathbf{w}''.(\exists \mathbf{x}\mathbf{x}'.(\Delta_i(\mathbf{x},\mathbf{x}') \sqcap \widehat{\alpha}(\mathbf{f}'',\mathbf{e}') \sqcap \\ \sqcap \mathbf{g}'' = \mathbf{g}') \sqcap \mathbf{w} = \mathbf{w}'')$$

  con $\mathbf{x} = InScope_{\widehat{P}}(i)$, $\mathbf{g} = Globals_{\widehat{P}}$, $\mathbf{w} = InScope_{\widehat{P}}(sSucc_P(i))$ e $\mathbf{f} = Formals_{\widehat{P}}(First_P(pr))$.

- se i corrisponde ad un vertice $Exit_P(pr)$, allora

$$\Delta^1_j = \Delta_j \sqcup \lambda \mathbf{w}\mathbf{w}''.\exists \mathbf{w}'''.(\exists \mathbf{x}'\mathbf{z}''.(\Delta_k(\mathbf{w},\mathbf{w}''') \sqcap \Delta_i(\mathbf{x}',\mathbf{x}'') \sqcap \widehat{\alpha}(\mathbf{f}',\mathbf{e}''') \sqcap \\ \sqcap \mathbf{g}' = \mathbf{g}''') \sqcap \mathbf{l}'' = \mathbf{l}''')$$

  con $j \in Succ_P(i)$, k tale che $stm_k = pr(\mathbf{e})$; e $RetPt_P(k) = j$. Inoltre vale che $\mathbf{w} = InScope_{\widehat{P}}(k)$, $\mathbf{f} = Formals_{\widehat{P}}(First_P(pr))$, $\mathbf{l} = Locals_{\widehat{P}}(k)$, $\mathbf{x} = InScope_{\widehat{P}}(i)$, $\mathbf{z} = Locals_{\widehat{P}}(i)$ e $\mathbf{g} = Globals_{\widehat{P}}$.

Sia $\Delta \in \widehat{\mathcal{A}}(P)$, si definisce $[\![\Delta]\!]$ come la classe di insiemi indicizzata sull'insieme N_P dei nodi del CFG tale che $\forall i \in N_P$ $[\![\Delta]\!]_i \in [\![\Delta]\!]$ e $[\![\Delta]\!]_i = [\![\Delta_i]\!]$. La correttezza e completezza dell'algoritmo deriva dal seguente teorema, che rappresenta la versione del teorema 5 adattata alla versione astratta $\widehat{\rightarrow}$ della relazione $\rightarrow$.

Teorema 8 (Correttezza e Completezza) *Sia P un programma lineare con array e siano $\Delta, \Delta^1 \in \widehat{\mathcal{A}}(P)$. Vale che $\Delta \widehat{\rightarrow} \Delta^1$ se e soltanto se $[\![\Delta]\!] \widehat{\Rightarrow} [\![\Delta^1]\!]$*

Dimostrazione E' sufficiente mostrare che attraverso la sostituzione di ogni ADLC δ con $[\![\delta]\!]$ nella definizione di $\widehat{\rightarrow}$, si ottiene la definizione di $\widehat{\Rightarrow}$. Sia $i \in N_P$. Se $j \notin Succ_P(i)$, allora $\Delta^1_j = \Delta_j$ diventa $[\![\Delta^1_j]\!] = [\![\Delta_j]\!]$ e quindi $[\![\Delta^1]\!]_j = [\![\Delta]\!]_j$. Se $j \in Succ_P(i)$, allora la prova procede per casi a seconda del tipo dello statement stm_i.

- Se stm_i è un `return;` oppure uno `skip`, allora $\Delta^1_{sSucc_P(i)} = \Delta_{sSucc_P(i)} \sqcup \Delta_i$ diventa

$$[\![\Delta^1]\!]_{sSucc_P(i)} = [\![\Delta]\!]_{sSucc_P(i)} \cup [\![\Delta]\!]_i$$

- Se stm_i è un assegnamento a variabile scalare y`=`e`;`, allora $\Delta^1_{sSucc_P(i)} = \Delta_{sSucc_P(i)} \sqcup \Delta^*$ e occorre mostrare che

$$[\![\Delta^*]\!] = \{\langle \widehat{w_e}, \widehat{w_i}[d/y] \rangle \mid \langle \widehat{w_e}, \widehat{w_i} \rangle \in [\![\Delta]\!]_i, d \in \overline{\widehat{w_i}}(e)\}$$

dove $\Delta^* = \lambda \mathbf{x}\mathbf{x}''.\exists \mathbf{x}'.(\Delta_i(\mathbf{x},\mathbf{x}') \sqcap \widehat{\alpha}(y'',e') \sqcap \mathbf{z}'' = \mathbf{z}')$ con $\mathbf{x} = InScope_{\widehat{P}}(i)$ e $\mathbf{z} = InScope_{\widehat{P}}(i) \setminus \{y\}$. Per definizione $[\![\Delta^*]\!]$ è uguale a

$$\{\langle \widehat{w_e}, \widehat{w_j}\rangle \mid\models_{\widehat{w_e} \cup \widehat{w_j''}} \exists \mathbf{x}'.(\Delta_i(\mathbf{x},\mathbf{x}') \sqcap \widehat{\alpha}(y'',e') \sqcap \mathbf{z}'' = \mathbf{z}')\}$$

che è equivalente per il lemma 3 a

$$\{\langle \widehat{w_e}, \widehat{w_j}\rangle \mid \langle \widehat{w_e}, \widehat{w_i}\rangle \in [\![\Delta]\!]_i, \widehat{w_j}(y) \in \overline{\widehat{w_i}}(e), \widehat{w_j}(\mathbf{z}) = \widehat{w_i}(\mathbf{z})\}$$

la quale può infine essere semplificata con

$$\{\langle \widehat{w_e}, \widehat{w_i}[d/y]\rangle \mid \langle \widehat{w_e}, \widehat{w_i}\rangle \in [\![\Delta]\!]_i, d \in \overline{\widehat{w_i}}(e)\}$$

- Se stm_i è un assegnamento ad array $a\texttt{[}e_1\texttt{]=}e_2\texttt{;}$, allora $\Delta^1_{sSucc_P(i)} = \Delta_{sSucc_P(i)} \sqcup \Delta^*$ e occorre mostrare che

$$[\![\Delta^*]\!] = \{\langle \widehat{w_e}, \widehat{w_j}\rangle \mid \langle \widehat{w_e}, \widehat{w_i}\rangle \in [\![\Delta]\!]_i \text{ e } \widehat{w_j} \in \Omega_i\}$$

con $\Omega_i = \{\widehat{w} \mid \widehat{w} = \widehat{w_i}[d_1/a_{\theta^a_1}, \ldots, d_m/a_{\theta^a_m}]$ dove

$\theta(a) = \{\theta^a_1, \theta^a_2, \ldots, \theta^a_m\}$	
$d_l \in \overline{\widehat{w_i}}(e_2)$	se $\overline{\widehat{w_i}}(e_1\texttt{==}\theta^a_l) = \{1\}$
$d_l = \widehat{w_i}(a_{\theta^a_l})$	se $\overline{\widehat{w_i}}(e_1\texttt{==}\theta^a_l) = \{0\}$
$d_l \in \overline{\widehat{w_i}}(e_2) \cup \{\widehat{w_i}(a_{\theta^a_l})\}$	altrimenti
con $1 \leq l \leq m\}$	

dove $\Delta^* = \lambda \mathbf{x}\mathbf{x}''.\exists \mathbf{x}'.(\Delta_i(\mathbf{x},\mathbf{x}') \sqcap \widehat{\alpha}(\mathbf{a}'',\mathbf{a}',a'',e_1,e_2) \sqcap \mathbf{z}'' = \mathbf{z}')$ con $\mathbf{x} = InScope_{\widehat{P}}(i)$ e $\mathbf{z} = InScope_{\widehat{P}}(i) \setminus \{a_\theta \mid \theta \in \theta(a)\}$ e con $\mathbf{a} = \langle a_{\theta^a_1}, \ldots, a_{\theta^a_m}\rangle$. Per definizione $[\![\Delta^*]\!]$ è uguale a

$$\{\langle \widehat{w_e}, \widehat{w_j}\rangle \mid\models_{\widehat{w_e} \cup \widehat{w_j''}} \exists \mathbf{x}'.(\Delta_i(\mathbf{x},\mathbf{x}') \sqcap \widehat{\alpha}(\mathbf{a}'',\mathbf{a}',a'',e_1,e_2) \sqcap \mathbf{z}'' = \mathbf{z}')\}$$

Quindi esiste una valutazione $\widehat{w_i'}$, definita su $\mathbf{x}'$ tale che $\langle \widehat{w_e}, \widehat{w_i}\rangle \in [\![\Delta]\!]_i$, $\widehat{w_j}(\mathbf{z}) = \widehat{w_i}(\mathbf{z})$ e tale che per ogni $1 \leq l \leq m$ e $\theta^a_l \in \theta(a)$, $\widehat{w_j}(a_{\theta^a_l}) \in D_l$, con $D_l = \{\overline{d_l} \in \mathcal{D} \mid\models_{\widehat{w}[d_1/\overline{a}_{\theta^a_1}, \ldots, d_m/\overline{a}_{\theta^a_m}]} \widehat{\alpha}(\overline{\mathbf{a}}, \mathbf{a}, a, e_1, e_2)$ per qualche $d_1, \ldots, d_m \in \mathcal{D}$ con $\overline{d_l} = d_l\}$. Ma per il lemma 3 vale che

$D^l \subseteq \overline{\widehat{w_i}}(e_2)$	se $\overline{\widehat{w_i}}(e_1\texttt{==}\theta^a_l) = \{1\}$
$D_l = \{\widehat{w_i}(a_{\theta^a_l})\}$	se $\overline{\widehat{w_i}}(e_1\texttt{==}\theta^a_l) = \{0\}$
$D_l \subseteq \overline{\widehat{w_i}}(e_2) \cup \{\widehat{w_i}(a_{\theta^a_l})\}$	altrimenti

Pertanto si ha che $\widehat{w_j} \in \Omega_i$ e quindi la formula precedente risulta equivalente a

$$\{\langle \widehat{w_e}, \widehat{w_j} \rangle \mid \langle \widehat{w_e}, \widehat{w_i} \rangle \in [\![\Delta]\!]_i, \widehat{w_j} \in \Omega_i\}$$

- Se stm_i è uno statement iterativo o condzionale (`if(b)`, `while(b)`, `assume(b);` oppure `assert(b);`), allora $\Delta^1_{Tsucc_P(i)} = \Delta_{Tsucc_P(i)} \sqcup \Delta^*$ (il caso di $\Delta^1_{Fsucc_P(i)}$ è simmetrico) e occorre mostrare che

$$[\![\Delta^*]\!] = \{\langle \widehat{w_e}, \widehat{w_i} \rangle \in [\![\Delta]\!]_i \mid d \in \overline{\widehat{w_i}}(b) \text{ per qualche } d \neq 0\}$$

dove $\Delta^* = \lambda \mathbf{x}\mathbf{x}'.(\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \widehat{\beta}^+(b'))$ con $\mathbf{x} = InScope_{\widehat{P}}(i)$. Per definizione $[\![\Delta^*]\!]$ è uguale a

$$\{\langle \widehat{w_e}, \widehat{w_j} \rangle \mid\models_{\widehat{w_e} \cup \widehat{w'_j}} \Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \widehat{\beta}^+(b')\}$$

dove $j = Tsucc_P(i)$. Dalla definizione, $\models_{\widehat{w_e} \cup \widehat{w'_j}} \Delta_i(\mathbf{x}, \mathbf{x}')$ se e solo se $\langle \widehat{w_e}, \widehat{w_j} \rangle \in [\![\Delta_i]\!]$, quindi la formula precedente è equivalente a

$$\{\langle \widehat{w_e}, \widehat{w_i} \rangle \mid \langle \widehat{w_e}, \widehat{w_i} \rangle \in [\![\Delta_i]\!] \text{ e } \models_{\widehat{w_e} \cup \widehat{w'_i}} \widehat{\beta}^+(b')\}$$

la quale per il lemma 3 diventa

$$\{\langle \widehat{w_e}, \widehat{w_i} \rangle \in [\![\Delta]\!]_i \mid d \in \overline{\widehat{w_i}}(b) \text{ per qualche } d \neq 0\}$$

- Se stm_i è una chiamata a procedura `pr(e);`, allora $\Delta^1_{sSucc_P(i)} = \Delta_{sSucc_P(i)} \sqcup \Delta^*$ e occorre mostrare che

$$[\![\Delta^*]\!] = \{\langle \widehat{w_j}, \widehat{w_j} \rangle \mid \langle \widehat{w_e}, \widehat{w_i} \rangle \in [\![\Delta]\!]_i,\ \widehat{w_j}(\mathbf{y}) \in \overline{\widehat{w_i}}(\mathbf{e}),\ \widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g}),\ \mathbf{g} = Globals_{\widehat{P}} \text{ e } \mathbf{y} = Formals_{\widehat{P}}(First_P(pr))\}$$

dove $j = sSucc_P(i)$ e $\Delta^* = \lambda \mathbf{w}\mathbf{w}''.(\exists \mathbf{x}\mathbf{x}'.(\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \widehat{\alpha}(\mathbf{y}'', \mathbf{e}') \sqcap \mathbf{g}'' = \mathbf{g}') \sqcap \mathbf{w} = \mathbf{w}'')$ con $\mathbf{x} = InScope_{\widehat{P}}(i)$, $\mathbf{g} = Globals_{\widehat{P}}$, $\mathbf{w} = InScope_{\widehat{P}}(j)$ e $\mathbf{y} = Formals_{\widehat{P}}(First_P(pr))$. Per definizione $[\![\Delta^*]\!]$ è uguale a

$$\{\langle \widehat{w_h}, \widehat{w_j} \rangle \mid\models_{\widehat{w_h} \cup \widehat{w''_j}} \exists \mathbf{x}\mathbf{x}'.(\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \widehat{\alpha}(\mathbf{y}'', \mathbf{e}') \sqcap \mathbf{g}'' = \mathbf{g}') \sqcap \mathbf{w} = \mathbf{w}''\}$$

Per l'ultimo congiunto ($\mathbf{w} = \mathbf{w}''$) nel DLC della formula preceedente, la stessa è equivalente a

$$\{\langle \widehat{w_j}, \widehat{w_j} \rangle \mid\models_{\widehat{w_j} \cup \widehat{w''_j}} \exists \mathbf{x}\mathbf{x}'.(\Delta_i(\mathbf{x}, \mathbf{x}') \sqcap \widehat{\alpha}(\mathbf{y}'', \mathbf{e}') \sqcap \mathbf{g}'' = \mathbf{g}')\}$$

che è equivalente per il lemma 3 a

$$\{\langle \widehat{w_j}, \widehat{w_j} \rangle \mid \langle \widehat{w_e}, \widehat{w_i} \rangle \in [\![\Delta]\!]_i,\ \widehat{w_j}(\mathbf{y}) \in \overline{\widehat{w_i}}(\mathbf{e}) \text{ e } \widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g})\}$$

- Se $i = Exit_P(pr)$, allora $\Delta^1_j = \Delta_j \sqcup \Delta^*$ dove $j \in Succ_P(i)$ e occorre mostrare che

$$[\![\Delta^*]\!] = \{\langle \widehat{w_e}, \widehat{w_j}\rangle \mid \langle \widehat{w_e}, \widehat{w_k}\rangle \in [\![\Delta]\!]_k,\ \langle \widehat{w_h}, \widehat{w_i}\rangle \in [\![\Delta]\!]_i,\ \widehat{w_h}(\mathbf{f}) \in \overline{\widehat{w_k}}(\mathbf{e}),\ \widehat{w_h}(\mathbf{g}) = \widehat{w_k}(\mathbf{g}),\ \widehat{w_j}(\mathbf{y}) = \widehat{w_k}(\mathbf{y}) \text{ e } \widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g})\}$$

con k tale che $stm_k =$ `pr(e);` e $RetPt_P(k) = j$ e

$$\Delta^* = \lambda \mathbf{w}\mathbf{w}''.\exists \mathbf{w}'''.(\Delta_k(\mathbf{w}, \mathbf{w}''') \sqcap \mathbf{y}'' = \mathbf{y}''' \sqcap \sqcap \exists \mathbf{x}'\mathbf{z}''.(\Delta_i(\mathbf{x}', \mathbf{x}'') \sqcap \widehat{\alpha}(\mathbf{f}', \mathbf{e}''') \sqcap \mathbf{g}' = \mathbf{g}'''))$$

con $\mathbf{y} = Locals_{\widehat{P}}(k)$, $\mathbf{f} = Formals_{\widehat{P}}(First_P(pr))$, $\mathbf{g} = Globals_{\widehat{P}}$, $\mathbf{w} = InScope_{\widehat{P}}(k)$, $\mathbf{x} = InScope_{\widehat{P}}(i)$, $\mathbf{z} = Locals_{\widehat{P}}(i)$. Per definizione si ha che

$$[\![\Delta^*]\!] = \{\langle \widehat{w_e}, \widehat{w_j}\rangle \mid \models_{\widehat{w_e} \cup \widehat{w''_j}} \exists \mathbf{w}'''.(\Delta_k(\mathbf{w}, \mathbf{w}''') \sqcap \mathbf{y}'' = \mathbf{y}''' \sqcap \sqcap \exists \mathbf{x}'\mathbf{z}''.(\Delta_i(\mathbf{x}', \mathbf{x}'') \sqcap \widehat{\alpha}(\mathbf{f}', \mathbf{e}''') \sqcap \mathbf{g}' = \mathbf{g}'''))\}$$

Quindi una coppia $\langle \widehat{w_e}, \widehat{w_j}\rangle \in [\![\Delta^*]\!]$ se e solo se esiste una valutazione $\widehat{w_k}$ tale che $\langle \widehat{w_e}, \widehat{w_k}\rangle \in [\![\Delta_k]\!]$ con $\widehat{w_j}(\mathbf{y}) = \widehat{w_k}(\mathbf{y})$ (poichè deve valere $\models_{\widehat{w''_j} \cup \widehat{w'''_k}} \mathbf{y}''' = \mathbf{y}''$) e tale che

$$\models_{\widehat{w''_j} \cup \widehat{w'''_k}} \exists \mathbf{x}'\mathbf{z}''.(\Delta_i(\mathbf{x}', \mathbf{x}'') \sqcap \widehat{\alpha}(\mathbf{f}', \mathbf{e}''') \sqcap \mathbf{g}' = \mathbf{g}''')$$

D'altra parte, dalla definizione di $[\![\Delta_i]\!]$ e dal lemma 3, la formula precedente vale se e solo se esistono due valutazioni $\widehat{w_h}$ e $\widehat{w_i}$ tali che $\langle \widehat{w_h}, \widehat{w_i}\rangle \in [\![\Delta_i]\!]$, $\widehat{w_h}(\mathbf{f}) \in \overline{\widehat{w_k}}(\mathbf{e})$, $\widehat{w_h}(\mathbf{g}) = \widehat{w_k}(\mathbf{g})$ (poichè deve valere $\models_{\widehat{w'_h} \cup \widehat{w'''_k}} \mathbf{g}' = \mathbf{g}'''$) e $\widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g})$ (poichè $\mathbf{x}'' = \mathbf{z}'' \cup \mathbf{g}''$ e le variabili $\mathbf{g}''$ non vengono eliminate con la quantificazione esistenziale). Riassumendo, dal precedente ragionamento si ha che

$$[\![\Delta^*]\!] = \{\langle \widehat{w_e}, \widehat{w_j}\rangle \mid \langle \widehat{w_e}, \widehat{w_k}\rangle \in [\![\Delta]\!]_k,\ \langle \widehat{w_h}, \widehat{w_i}\rangle \in [\![\Delta]\!]_i,\ \widehat{w_h}(\mathbf{f}) \in \overline{\widehat{w_k}}(\mathbf{e}),\ \widehat{w_h}(\mathbf{g}) = \widehat{w_k}(\mathbf{g}),\ \widehat{w_j}(\mathbf{y}) = \widehat{w_k}(\mathbf{y}) \text{ e } \widehat{w_j}(\mathbf{g}) = \widehat{w_i}(\mathbf{g})\}$$

come richiesto dall'enunciato.

□

Si denoti con $\widehat{\rightarrow}^*$ la chiusura riflessiva e transitiva della relazione $\widehat{\rightarrow}$ e siano $\Theta = \{\theta_1, \ldots, \theta_n\}$ e $e_{\theta_i} = \Theta_{ass}(\theta_i)$ per $i = 1, \ldots, n$. Sia inoltre $\Delta^0 \in \widehat{\mathcal{A}}(P)$ definito in modo tale che $\Delta^0_{First_P(main)} = \lambda \mathbf{x}\mathbf{x}'.(\mathbf{x} = \mathbf{x}' \sqcap \widehat{\alpha}(\vec{\theta'}, \mathbf{e}_{\vec{\theta}}))$ con $\vec{\theta} = \langle \theta_1, \ldots, \theta_n\rangle$ ($\vec{\theta}$ è la n-upla contenente tutte le variabili $\theta \in \Theta$), $\mathbf{e}_{\vec{\theta}} = \langle e_{\theta_1}, \ldots, e_{\theta_n}\rangle$ e $\mathbf{x} = InScope_{\widehat{P}}(First_P(main))$ e $\Delta^0_j = \lambda \mathbf{x}\mathbf{x}'.\bot$ con $\mathbf{x} = InScope_{\widehat{P}}(j)$ per ogni $j \in N_P \setminus \{First_P(main)\}$, vale il seguente corollario, che rappresenta la controparte astratta del corollario 6.

Corollario 8 *Sia P un programma lineare con array e sia $\Delta^0 \in \widehat{\mathcal{A}}(P)$ definito come sopra, allora per ogni $i \in N_P$, i è raggiungibile da un percorso astratto valido e ammissibile se e solo se $\Delta_i \not\sqsubseteq \lambda \mathbf{x}\mathbf{x}'.\bot$, per qualche Δ tale che $\Delta^0 \widehat{\rightarrow}^* \Delta$.*

Capitolo 4

Simulazione e raffinamento del modello

In tale capitolo verranno presentati i moduli di simulazione e raffinamento che operano all'interno del ciclo CEGAR.

Come già accennato, se la verifica sul modello astratto restituisce una traccia τ di errore, compito del simulatore è verificare che tale traccia sia eseguibile anche all'interno del programma concreto P. Se τ risulta essere fattibile in P, allora il ciclo termina restituendo la traccia che porta all'errore, altrimenti il modello va raffinato affinchè τ non sia più fattibile neanche nel modello astratto.

4.1 Verifica di fattibilità della traccia

Se la fase di verifica sul modello astratto restituisce una traccia $\tau = i_1, \ldots, i_n$, corrispondente ad una sequenza di nodi tale che esiste un percorso astratto valido e ammissibile $\widehat{\pi} = \langle i_1, \widehat{w_{i_1}} \rangle \xrightarrow{\widehat{\Sigma_{i_1}^{i_n}}}_P \langle i_n, \widehat{w_{i_n}} \rangle$ e tale che $i_1 = First_P(main)$ e i_n è un nodo di errore di P, allora occorre verificare che τ sia anche una traccia eseguibile all'interno del programma concreto P, ovvero che esista anche un percorso concreto valido inizializzato $\pi = \langle i_1, w_{i_1} \rangle \xrightarrow{\Sigma_{i_1}^{i_n}}_P \langle i_n, w_{i_n} \rangle$, che conduca al nodo di errore i_n attraverso la sequenza di nodi τ.

Tale operazione viene performata dal simulatore. Il problema viene ridotto al problema di determinare la soddisfacibilità di un insieme di formule libere da quantificatori (chiamate *trace formulae*) nella teoria decidibile risultante dalla combinazione di aritmetica lineare e teoria degli array.

Per aritmetica lineare si intende l'aritmetica standard (su un dominio numerico $\mathcal{D}$) con addizione ($+$) e usuali operatori relazionali (cioè $<, \leq, >, \geq$,

$=, \neq$), ma senza la moltiplicazione (la moltiplicazione di una costante per una variabile, $c * x$, è consentita ma è solo un'abbreviazione per l'espressione lineare $x + \ldots + x$ con c occorrenze della variabile x).

La teoria degli array che noi consideriamo modella gli array come strutture dati che rappresentano associazioni di elementi a un insieme di indici. Diversamente dagli array utilizzati nei comuni linguaggi di programmazione, gli array modellati dalla teoria degli array non hanno necessariamente dimensione finita. Siano INDEX, ELEM e ARRAY i tipi utilizzati per indici, elementi e array, rispettivamente, e siano $select : ARRAY \times INDEX \rightarrow ELEM$ e $store : ARRAY \times INDEX \times ELEM \rightarrow ARRAY$ due simboli funzionali. Si assuma inoltre che il linguaggio della teoria degli array includa un costruttore di termini condizionale che consente termini della forma $(w?t_1 : t_2)$, per ogni formula w e termini t_1 e t_2. Allora la seguente formula è una presentazione concisa della teoria degli array:

$$\forall a, i, j, e.select(store(a, i, e), j) = (j = i?e : select(a, j)) \tag{4.1}$$

Di seguito l'aritmetica lineare verrà denotata con $\mathcal{T}_0$ e l'unione dell'aritmetica lineare con la teoria degli array con $\mathcal{T}_1$.

Sia $stm_{i_1}, \ldots, stm_{i_n}$ la sequenza di statement associati con la traccia $\tau = i_1, \ldots, i_n$. La sequenza di statement è tradotta in *Single Assignment Form* [ABM07], cioè le variabili del programma sono rinominate in modo tale che ogni variabile è assegnata esattamente una volta nel programma risultante. La rinomina avviene come segue. Sia v una variabile di programma e i un'istruzione del programma, si definisce la funzione $\alpha(v, i)$ come il numero di assegnamenti fatti ad i prima dell'istruzione i. Sia e un'espressione del programma P che occorre nell'istruzione i, con $\varrho(e)$ si denota l'espressione ottenuta da e attraverso la sostituzione di ogni variabile v in e con $v_{\alpha(v,i)}$. Ogni assegnamento a una variabile della forma x`=`e`;` nell'istruzione i è sostituito dall'assegnamento $x_{\alpha(x,i)+1}$`=`$\varrho(e)$`;`. Ogni assegnamento ad array a`[`e_1`]=`e_2`;` è sostituito dall'assegnamento $a_{\alpha(a,i)+1}$`[`$\varrho(e_1)$`]=`$\varrho(e_2)$`;`. Ogni condizione c che occorre all'interno della guardia di un'istruzione condizionale o iterativa è sostituita dall'espressione $\varrho(c)$.

Per ogni espressione o guardia di istruzione condizionale e, si denoti, con e' l'espressione ottenuta a partire da e attraverso la sostituzione di ogni accesso ad array a`[`e`]` con l'espressione $select(a, e)$ appartenente alla teoria degli array e ogni operatore relazionale `<=,>=,!=` con $\leq, \geq, \neq$, rispettivamente, ed ogni operatore booleano `!,&&,||` con $\neq, \wedge, \vee$, rispettivamente.

L'insieme di trace formulae per τ rispetto a P è l'insieme di formule libere da quantificatori $\Phi(\tau, P) = \bigcup_{k=1}^{n} \phi(stm_{i_k})$, dove $\phi(stm_{i_k})$ è definito in tabella 4.1. Si definisce $\Phi_{\mathcal{T}_i}(\tau, P) = \mathcal{T}_i \cup \Phi(\tau, P)$ per $i = 0, 1$.

stm_{i_k}	$\phi(stm_{i_k})$	Condizione
`if`(c), `while`(c), `assert`(c); o `assume`(c);	$\{c'\}$	se $i_{k+1} = Tsucc_P(i_k)$
`if`(c), `while`(c) o `assert`(c);	$\{\neg(c')\}$	se $i_{k+1} = Fsucc_P(i_k)$
v_{j+1} `=` e;	$\{v_{j+1} = e'\}$	
a_{j+1}`[`e_1`]` `=` e_2;	$\{a_{j+1} = store(a_j, e'_1, e'_2)\}$	
`skip` o `return`;	$\emptyset$	

Tabella 4.1: Codifica degli statement del programma in formule libere da quantificatori

Vale il seguente teorema [ABM07]:

Teorema 9 *Sia P_0 un programma lineare e sia P_1 un programma lineare con array, allora $\tau \in traces(P_i)$ se e solo se $\Phi_{\mathcal{T}_i}(\tau, P)$ è soddisfacibile, per $i = 0, 1$*

Pertanto, il processo di verifica della fattibilità, nel programma concreto, della traccia di errore individuata dal model checker, avviene in due fasi:

1. fase di codifica, durante la quale a partire dalla traccia τ e dal programma concreto P viene generato l'insieme di trace formulae $\Phi(\tau, P)$;

2. fase di decisione, durante la quale l'insieme di formule $\Phi(\tau, P)$, restituito dalla fase di codifica, viene dato in input ad un theorem prover, che ne decide la soddisfacibilità o meno ed, in quest'ultimo caso, restituisce la prova di insoddisfacibilità, cioè la prova per la formula $\Phi_{\mathcal{T}_i}(\tau, P) \vdash \bot$.

4.2 Raffinamento del modello

Se, in seguito al processo di simulazione di una traccia τ all'interno di un programma lineare con array P, $\Phi_{\mathcal{T}_1}(\tau, P)$ risulta essere soddisfacibile, allora, per il teorema precedente $\tau \in traces(P)$ e il processo di verifica termina con la conclusione che il programma concreto non è corretto, o meglio esiste una sua possibile esecuzione, rappresentata dalla traccia τ appunto, che conduce ad un nodo di errore.

Se invece la traccia di errore τ si rivela essere spuria, allora occorre raffinare il modello astratto, affinchè tale traccia non vi appartenga più.

Raffinare il modello astratto significa generare una nuova semantica astratta per P, che verrà denotata con $\widehat{[\![P]\!]}_1$, tale che il percorso $\widehat{\pi}$, che apparteneva alla precedente semantica astratta di P, denotata con $\widehat{[\![P]\!]}_0$ (in simboli $\widehat{\pi} \in \widehat{[\![P]\!]}_0$), non appartenga alla nuova semantica astratta $\widehat{[\![P]\!]}_1$ (in

simboli $\widehat{\pi} \notin \widehat{[\![P]\!]}_1$). In altre parole si vuole generare una nuova semantica per P tale che $\widehat{[\![P]\!]}_1 \subset \widehat{[\![P]\!]}_0$ e tale che $\widehat{\pi} \notin \widehat{[\![P]\!]}_1$.

Costruire una nuova semantica astratta per un programma concreto lineare con array P con le proprietà desiderate equivale a definire una nuova relazione di transizione astratta $\widehat{\xrightarrow{\sigma}}_P$ rispetto alla nuova classe di insiemi indici $I' = \{\theta'(a) \mid a \in A_P\}$ tale che I' rappresenta un'estensione della classe $I = \{\theta(a) \mid a \in A_P\}$ utilizzata nel precedente processo di astrazione, cioè per ogni $a \in A_P$, $\theta(a) \subseteq \theta'(a)$ ed esiste $a \in A_P$ tale che $\theta(a) \subset \theta'(a)$ (in simboli, $I \sqsubset I'$).

Obiettivo del processo di raffinamento è proprio determinare la classe di insiemi di indici I' tale che $I \sqsubset I'$ e tale che la semantica astratta di P costruita rispetto ad I' non includa più percorsi $\widehat{\pi}$ cui è associata la traccia di esecuzione spuria τ. A tale scopo si sfrutta la prova di inconsistenza della traccia spuria, individuando gli elementi di array da cui la prova dipende. Pertanto, il risultato del processo di raffinamento è rappresentato dall'insieme I', che verrà dato in input alla successiva fase di astrazione, nella prossima iterazione CEGAR.

Si fa notare ancora una volta che il modulo di refiner preesistente a tale lavoro di tesi restituiva al processo di astrazione una classe di insiemi di indici numerici $\{R(a) \mid a \in A_P\}$ con $R(a) \subseteq \{0, 1, \ldots, dim(a)-1\}$ per ogni $a \in A_P$, ignorando eventuali indici che non riusciva a ridurre a numerali. In seguito all'introduzione della nuova strategia di astrazione descritta nel presente elaborato, in particolare all'interno del capitolo 2, capace di gestire anche indici in forma di espressione lineare con array, il comportamento del refiner è stato modificato in modo tale da restituire anche gli indici che non si riescono a ridurre a numerali e che restano pertanto sottoforma di espressione lineare con array. In conclusione, sebbene il refiner restituisca una classe di indici $I' = \{\theta'(a) \mid a \in A_P\} \cup \{R'(a) \mid a \in A_P\}$, che comprende sia indici numerici che indici sottoforma di espressione[1], all'interno della presente tesi l'attenzione verrà posta maggiormente sugli indici-espressione, che rappresentano l'aspetto più significativo del presente lavoro, facendo notare che l'estensione al caso generale è piuttosto immediata (per maggiori informazioni sulla precedente strategia di astrazione, effettuata esclusivamente su indici numerici, si faccia riferimento al lavoro in [Cas06]).

[1] In seguito denotati con la locuzione indici-espressione oppure semplicemente indici

4.3 Meccanismo di raffinamento in Eureka

Il meccanismo di raffinamento di Eureka si basa sulla trasformazione, attraverso una manipolazione puramente sintattica, della prova in un'altra equivalente che contenga, però, informazioni sugli elementi di array da inserire nel prossimo processo di astrazione.

Per ogni accesso ad array (formula di tipo $select(a, e)$) presente in un sequente foglia della prova di inconsistenza Π, un nuovo predicato $Q(e, a)$, con il quale si indica la relazione di appartenenza dell'indice e all'insieme di indici di a rispetto cui astrarre, viene aggiunto alle premesse del sequente e trasmesso fino alla radice attraverso l'applicazione delle stesse regole di inferenza applicate nella prova originale.

In seguito alla trasformazione si ottiene una nuova prova Π', il cui sequente radice, che in Π era rappresentato da una formula del tipo $\Phi_{\mathcal{T}_1}(\tau, P) \vdash \bot$, assume ora il seguente aspetto: $\Phi_{\mathcal{T}_1}(\tau, P), Q(e_1, a_{k_1}), \ldots, Q(e_n, a_{k_n}) \vdash \bot$.

A questo punto, se si indica con $A_P = \{a_1, \ldots, a_s\}$ l'insieme degli array del programma P e con $\overline{Q} = \{\langle e_l, a_{k_l}\rangle \mid 1 \leq l \leq n\}$ l'insieme di coppie $\langle e, a\rangle$ tali che $Q(e, a)$ compare tra le premesse del sequente radice di Π', gli insiemi di indici computati dal refiner e che saranno utilizzati nel processo di costruzione del nuovo modello astratto sono definiti come segue: $\theta'(a_j) = \theta(a_j) \cup \{e_l | \langle e_l, a_{k_l}\rangle \in \overline{Q}$ e $k_l = j\}$ per ogni $1 \leq j \leq s$.

Si consideri, a titolo di esempio, la prova Π di inconsistenza, costruita a partire dal semplice programma P in tabella 4.2. La procedura di model checking sull'astrazione iniziale di P, costruita rispetto ad $I = \{\theta(a) \mid a \in A_P\}$ con $\theta(a) = \emptyset$ per ogni $a \in A_P$, restituisce la traccia di errore $\tau =$ `1,2,3,0`.

Nella tabella è mostrato il codice del programma, la traduzione in formule della successione di statement corrispondenti alla traccia che porta al fallimento dell'assert e la prova di inconsistenza di tale insieme di formule.

La prova dipende da quattro sequenti foglia, tre dei quali appartengono all'insieme delle Trace Formulae, mentre il quarto è rappresentato dall'istanziazione della formula 4.1, appartenente alla teoria degli array. Si nota la presenza di formule di tipo $select()$ all'interno del sequente foglia $\vdash select(a_1, i_1) \neq 4$ e all'interno di $\vdash$ (4.1)[2]. Innanzitutto, il refiner aggiunge i predicati $Q(i_1, a_1)$ alle premesse del sequente $\vdash select(a_1, i_1) \neq 4$ e $Q(i_1, a_0)$ alle premesse di $\vdash$ (4.1). Dopodichè riapplica tutte le regole di inferenza con cui è stata costruita la prova, propagando i predicati Q, introdotti nelle foglie, fino alla radice. Si può osservare come nel caso di applicazione di

[2]Da notare che i sequenti, che dovrebbero essere della forma $\Phi_{\mathcal{T}_1}(\tau, P) \vdash \varphi$, sono stati semplificati, al solo scopo di allegerire la sintassi, omettendo il predecessore $\Phi_{\mathcal{T}_1}(\tau, P)$

Programma P	τ	$\rightarrow$	**Trace Formulae**
`main() {`			
`  int i;`			
`  int a[6];`			
	[0]		
[1] `  i = 3;`	[1]	$\rightarrow$	$i_1 = 3;$
[2] `  a[3] = 4;`	[2]	$\rightarrow$	$a_1 = store(a_0, 3, 4)$
[3] `  assert(a[i] == 4);`	[3]	$\rightarrow$	$select(a_1, i_1) \neq 4$
`}`			

Prova di inconsistenza Π

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash select(a_1, i_1) \neq 4 \quad \vdash a_1 = store(a_0, 3, 4)}{\vdash select(store(a_0, 3, 4), i_1) \neq 4} \quad \vdash (4.1)}{\vdash (i_1 = 3 ? 4 : select(a_0, i_1)) \neq 4} \quad \vdash i_1 = 3}{\vdash 4 \neq 4}}{\vdash \bot}$$

Tabella 4.2: Esempio di prova Π di inconsistenza dell'insieme di formule che codificano una traccia di errore spuria di un programma P

una regola di sostituzione, la sostituzione è applicata, se possibile, anche agli indici dei predicati Q, introducendo nuovi predicati a partire da quelli già presenti in cui l'indice viene sostituito compatibilmente alla regola applicata. Ad esempio nel penultimo passo di inferenza della prova si applica la sostituzione di i_1 con 3 e tale sostituzione provoca l'aggiunta dei predicati $Q(3, a_1)$ e $Q(3, a_0)$ a partire da $Q(i_1, a_1)$ e $Q(i_1, a_0)$, attraverso la sostituzione dell'indice. Il risultato dell'intero processo è mostrato in tabella 4.3.

Come si può notare dal confronto delle prove Π e Π', si tratta praticamente della stessa prova, con l'aggiunta, nella prova costruita dal refiner, della relazione di appartenenza tra indici e rispettivi array, che verrà restituita al nuovo processo di astrazione.

Infatti, l'ultima fase del processo di raffinamento consiste nell'analisi del sequente radice, con lo scopo di estrarre gli indici rappresentati e aggiungerli all'insieme di indici degli array con cui sono in relazione di appartenenza Q.

Nell'esempio trattato, all'interno del sequente radice compaiono l'indice 3 e l'indice i_1 associati entrambi all'array a. Tuttavia l'unico indice aggiunto alla lista dell'array a sarebbe l'indice 3, infatti l'indice i_1 non viene aggiunto

$$\dfrac{\dfrac{\dfrac{Q(i_1,a_1)\vdash select(a_1,i_1)\neq 4 \quad \vdash a_1 = store(a_0,3,4)}{Q(i_1,a_1)\vdash select(store(a_0,3,4),i_1)\neq 4} \qquad Q(i_1,a_0)\vdash (4.1)}{Q(i_1,a_1),Q(i_1,a_0)\vdash (i_1=3?4:select(a_0,i_1))\neq 4} \qquad \vdash i_1 = 3}{Q(3,a_1),Q(3,a_0),Q(i_1,a_1),Q(i_1,a_0)\vdash 4\neq 4}$$
$$Q(3,a_1),Q(3,a_0),Q(i_1,a_1),Q(i_1,a_0)\vdash \bot$$

Tabella 4.3: Esempio di prova Π' ottenuta dal refiner a partire dalla prova Π di inconsistenza di una traccia spuria

perchè si è riusciti a ridurlo ad un numerale attraverso il processo di ricostruzione della prova. Pertanto, il processo di raffinamento restituirà l'indice 3 associato all'array a.

Dal precedente esempio emerge l'utilità della ricostruzione della prova che consente, attraverso l'applicazione delle regole di inferenza (in particolare di regole di sostituzione o comunque regole che sfruttano eventuali uguaglianze tra termini, come ad esempio la regola di transitività) ai nuovi sequenti arrichiti dei predicati Q, di aggiungere nuovi indici necessari e magari ridurre a numerali alcuni indici-espressione.

Tuttavia quest'operazione comporta un doppio attraversamento, uno in fase discendente (dalla radice alle foglie) e uno in fase ascendente (dalle foglie alla radice), dell'albero che la rappresenta. Tale operazione grava non poco sulle prestazioni di un'iterazione del ciclo CEGAR. Quindi se si riuscisse a trovare un nuovo modo per semplificare le espressioni, e quindi le variabili in esse coinvolte, in numerali, si potrebbe evitare la costosa attività di attraversamento e ricostruzione della prova.

In realtà, prima che il procedimento termini, avviene anche un ulteriore tentativo di riduzione ad un numerale di tutte le espressioni che non si è riusciti a ridurre in fase di ricostruzione della prova, attraverso la chiamata alla funzione $simplify$ del theorem prover, che cerca appunto di semplificare un'espressione in un numerale, utilizzando l'insieme di formule $\Phi_{\mathcal{T}_1}(\tau, P)$.

Si consideri ora il programma di tabella 4.4.

La procedura di model checking della prima iterazione, effettuata sul modello iniziale, costruito ripetto all'insieme vuoto di indici, restituisce la traccia spuria `1,2,3,4,5,6,7,8,6,7,8,6,7,8,6,9,10,0`, che termina nel fallimento dell'assert. Il refiner restituisce come indice su cui astrarre l'espressione y_2, cioè la variabile y rinominata dopo il suo secondo assegnamento, che non si riesce a ridurre a numerale durante la ricostruzione della prova. Pertanto il processo di astrazione riceverà in ingresso l'espressione y associata al vettore a. Verranno create la variabile $\theta \in \theta(a)$, che verrà assegnata proprio con l'espressione y, e la variabile a_θ, che servirà a modellare l'elemento di a

```
      main() {
        int x,y
        int a[3];

[1]     y = 0;
[2]     x = 0;
[3]     x = u;
[4]     y = x;
[5]     x = 0;
[6]     while (x < 3) {
[7]       a[x] = 3;
[8]       x = x+1;
        }
[9]     if ( (0 <= y) && (y < 3) ) {
[10]      assert(a[y] == 3);
        }
      }
```

Tabella 4.4: Esempio di programma per illustrare il comportamento del refiner

indicizzato dall'espressione y. A questo punto sorge un ulteriore problema, ovvero quello di stabilire dove effettuare l'assegnamento della variabile θ con l'espressione y. Inserire l'assegnamento $\theta = y$ nel primo nodo del main, sarebbe scorretto, infatti si vuole raffinare sul valore assunto dall'espressione y dopo il suo secondo assegnamento, che avviene nel nodo `4`.

Il refiner individua tale nodo isolando un frammento della traccia significativo per la prova di inconsistenza. Infatti, non tutte le formule ottenute dalla traduzione degli statement della traccia vengono utilizzate nella prova. Pertanto, si prende l'insieme di assunzioni (formule corrispondenti a statement della traccia) usate, cioè di assunzioni da cui dipende la prova e si estrae il più piccolo frammento di traccia che contiene tutti gli statement corrispondenti ad assunzioni di tale insieme.

Ritornando al precedente esempio di esecuzione, le assunzioni usate dalla prova di inconsistenza sono incluse nel frammento di traccia `4,5,6,7,8,6,7,8,6,7,8,6,9,10`. A questo punto avviene una nuova traduzione in Single Assignment Form sul nuovo frammento di traccia e si ha che l'espressione y_2 corrisponde a y_1 nella nuova rinomina effettuata sul nuovo frammento di traccia. Attraverso la funzione $simplify()$ del theorem prover y_1 viene ridotta a x_0 (che corrisponde ad x_2 nella rinomina effettuata sull'intera traccia)

ed in conclusione il refiner restituisce l'espressione x con cui inizializzare θ e il nodo `4` (il primo del frammento di traccia calcolato) in cui effettuare l'assegnamento.

Capitolo 5

Implementazione

In questo capitolo verranno illustrati i dettagli implementativi del tool Eureka, ponendo particolare attenzione alle librerie esterne con le quali il tool interagisce.

5.1 Architettura di Eureka

Lo strumento Eureka è un prototipo in grado di verificare un ampio frammento dei programmi C: i programmi lineari con array già definiti nella sezione 2.1. Esso è basato sul paradigma CEGAR, pertanto è composto da quattro moduli principali: Astrattore, Model Checker, Simulatore e Refiner. In figura 5.1 vengono rappresentati i moduli di Eureka e come essi interagiscono tra di loro all'interno del ciclo CEGAR.

Prima dell'esecuzione del ciclo c'è bisogno di una fase di pre-processing in cui il programma sorgente viene parsificato allo scopo di costruire le strutture dati principali su cui lavorare, in particolare viene creato il Control Flow Graph del programma e gli insiemi che forniscono informazioni sulla visibilità delle variabili per ogni istruzione. Allo stato attuale sono riconosciuti come validi i principali costrutti del linguaggio C, con l'esclusione dei puntatori, e con l'elemento aggiuntivo costituito dal simbolo di indefinito.

L'Astrattore riceve in input il CFG del programma lineare con array P generato dal parser ed un insieme classe di insiemi di indici array (inizialmente vuoti) in base ai quali sarà generata una semantica astratta $\widehat{[\![P]\!]}$ su un dominio lineare per P. Tale semantica verrà data in ingresso al Model Checker insieme al CFG di P. Il Model Checker verifica la raggiungibilità di un'istruzione corrispondente ad un nodo di errore sul modello astratto appena ricevuto. Se nessuno nodo di errore viene raggiunto, allora il programma è privo di errori (SAFE) e il ciclo termina, altrimenti viene restituita la trac-

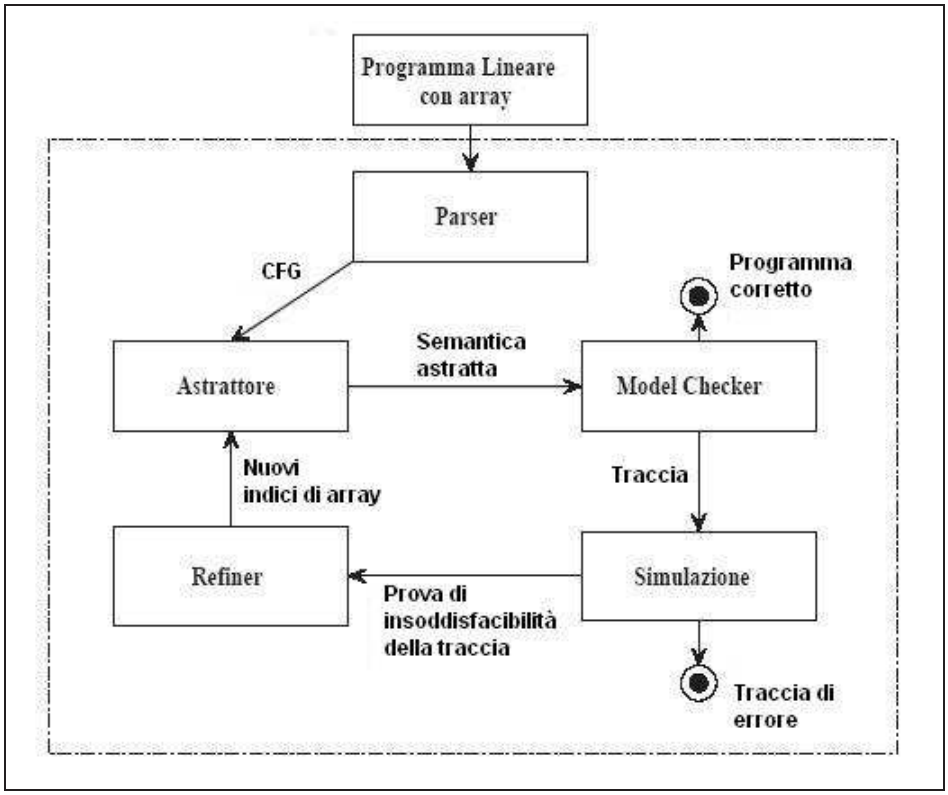


Figura 5.1: Moduli di Eureka

cia d'esecuzione τ. Quest'ultima è analizzata dal Simulatore che effettua un controllo di fattibilità della traccia τ nel programma concreto P. Se τ risulta essere una traccia reale (eseguibile all'interno del programma concreto P), allora viene restituita e il ciclo termina, altriment la traccia è spuria ed il Simulatore fornisce in ingresso al Refiner una prova di insoddisfacibilità Π dell'insieme di Trace Formulae ad essa associato. Il Refiner si occuperà di individuare, utilizzando la dimostrazione di insoddisfacibilità Π, gli indici di array rispetto cui astrarre affinchè la τ non sia più eseguibile nel nuovo modello astratto. Il risultato di questa analisi fornisce l'input per una nuova fase di astrazione che dà il via ad una nuova iterazione del ciclo CEGAR. Lo pseudo-codice dell'intero processo è mostrato in tabella 5.1.

In figura 5.2, invece, appare il diagramma delle classi con i metodi principali, relativo ai quattro moduli di Eureka.

```
   doCegar(cfg)
1     arrInds = ∅
2     arrIndsTerm = ∅
      do
3         cfg = abstract(cfg,arrInds,arrIndsTerm)
4         cfg = buildTauAbs(cfg)
5         trace = verify(cfg)
6         if trace = NULL // nessun nodo di errore è stto raggiunto
7             return SAFE
          end
8         spuria = simulate(cfg,trace)
9         if spuria = false // trace è una traccia eseguibile nel programma concreto
10            return trace
          end
11        ⟨arrInds′, arrIndsTerm′⟩ = refine(cfg,trace,arrInds,arrIndsTerm)
      end
```

Tabella 5.1: Pseudo-codice del ciclo CEGAR

5.2 Modulo per l'astrazione

Il principale metodo pubblico della classe Abstractor + *abstrac*() che si occupa di definire la transizione astratta, rispetto ad una classe di indici I, per ogni statement del programma P.

Funzione *abstract*

- INPUT: cfg, il Control Flow Graph del programma lineare con array P, $arrInds$ e $arrIndsTerm$, due strutture dati che codificano la classe $\{R(a) \mid a \in A_P\}$ di indici numerici e la classe $\{\theta(a) \mid a \in A_P\}$ di indici-espressione, rispettivamente, rispetto cui astrarre;

- OUTPUT: genera la nuova relazione di transizione astratta e quindi la nuova semantica astratta $\widehat{[\![P]\!]}$ per P rispetto alla classe di insiemi di indici $\{R(a) \mid a \in A_P\} \cup \{\theta(a) \mid a \in A_P\}$.

La funzione esamina tutti gli statement associati ai nodi del CFG ed effettua le operazione necessarie per la definizione della nuova relazione di transizione astratta definita su un dominio lineare senza array associata ad ognuno di essi.

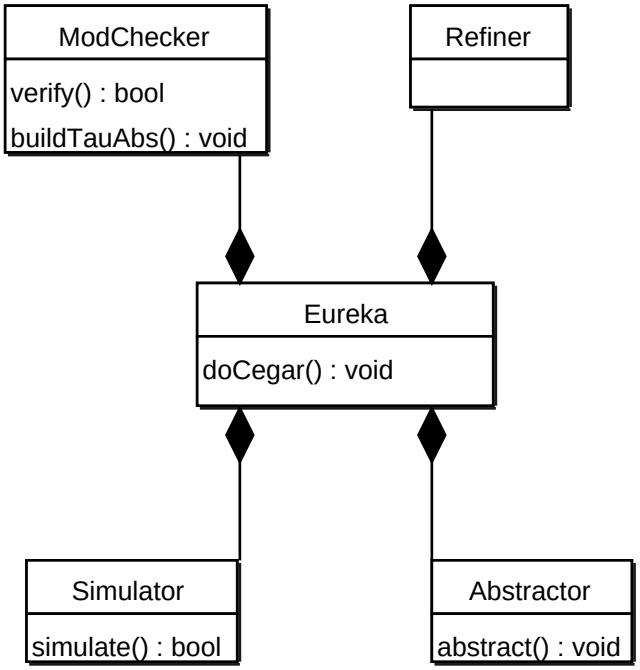


Figura 5.2: Diagramma delle classi relativo ai principali moduli di Eureka

---

abstract(cfg, $arrInds$, $arrIndsTerm$)

```
1    vet = preProcessingAbsArray(arrInds,arrIndsTerm)
2    per ogni nodo n ∈ cfg
3        per ogni assegnamento θ=e; ∈ n
4            ⟨τ_T, τ_F⟩ = abstractExpression(θ=e;,vet);
5            accoda(τ_θ,τ_T);
         end
6        setTauTeta(nodo,τ_θ)
7        ⟨τ_T, τ_F⟩ = abstractExpression(getExpr(n),vet);
8        setTauTrue(nodo,τ_T)
9        setTauFalse(nodo,τ_F)
     end
```

La funzione esegue una prima elaborazione (linea [1]) nella quale genera uno schema di vincoli in forma normale disgiuntiva parametrizzato sugli indici in ingresso alla fase di astrazione. Successivamente, per ogni nodo del CFG, viene computata la sequenza di relazioni di transizione astratta per

tutti gli assegnamenti a variabili θ che avvengono nel nodo ([3-4]); tale sequenza viene memorizzata in un vettore ([5]), che viene associato al nodo del CFG ([6]). Il passo successivo consiste nel calcolo della relaziona di tranzione astratta (per il ramo vero ed eventualmente per il ramo falso) per lo statement concreto del nodo ([7]) e nella memorizzazione di quest'ultima nel nodo del CFG ([8-9]). La funzione di transizione è calcolata dalla funzione *abstractExpression*, che esegue un'istanziazione, dipendente dallo statement di cui si vuole costruire la relazione di transizione, dello schema di vincoli generato dalla funzione *preProcessingAbsArray* ([1]). Vengono illustrate di seguito le funzioni *preProcessingAbsArray* e *abstractExpression*.

Funzione *preProcessingAbsArray*

La funzione accetta in input gli indici numeri ($arrInds$) e sottoforma di espressione ($arrIndsTerm$) rispetto cui astrarre. Si tratta di due strutture dati che mantengono una corrispondenza tra ogni array del programma e un vettore di indici. Il risultato della procedura è un'altra struttura dati che contiene, per ogni array del programma, una matrice di vincoli (condizioni booleane) e variabili θ e costanti.

Sia a un array del programma e siano $arrInds[a] = \langle\, i_1, \ldots, i_n \,\rangle$ e $arrIndsTerm[a] = \langle e_1, \ldots, e_m \rangle$ gli insiemi di indici numerici e di indici-espressioni associati ad a, rispettivamente. Inizialmente la funzione genera le matrici A_a a partire da $arrInds[a]$ e B_a a partire da $arrIndsTerm[a]$ cosi formate:

- A_a contiene, per ogni $i_l \in arrInds[a]$ una riga della forma $\langle indice = i_l, i_l \rangle$ e un'ulteriore riga del tipo $\langle indice \neq i_1, indice \neq i_2, \ldots, indice \neq i_n, u \rangle$;

- B_a è costituita da una riga per ogni $\chi \in 2^{\theta(a)}$. In particolare, sia $\chi = \{\theta_{i_1}, \ldots, \theta_{i_s}\} \neq \emptyset$ e sia $\theta(a) \setminus \chi = \{\theta_{j_1}, \ldots, \theta_{j_r}\}$ una riga di B_a è rappresentata dal seguente vettore: $\langle indice = \theta_{i_1}, \ldots, indice = \theta_{i_s}, indice \neq \theta_{j_1}, \ldots, indice \neq \theta_{j_r}, \theta_{i_1}, \ldots, \theta_{i_s} \rangle$. Esiste poi un'ulteriore riga di B_a corrispondente a $\chi = \emptyset$, $\theta(a) = \theta(a) \setminus \chi = \{\theta_{k_1}, \ldots, \theta_{k_h}\}$ della forma $\langle indice \neq \theta_{k_1}, \ldots, indice \neq \theta_{k_h}, u \rangle$.

Si può notare che ogni riga delle due matrici contiene una serie di vincoli booleani seguiti da un numero, da una serie di variabili θ_i oppure dal simbolo di indefinito. Intuitivamente le due matrici catturano come vengono modellati gli accessi ad elementi di array nel dominio astratto. Gli elementi di ogni riga possono vanno interpretati come in congiunzione tra di loro, mentre le righe sono in disgiunzione tra di loro. Si ha pertanto che tali matrici rappresentano

una formula in forma normale disgiuntiva. I vincoli booleani rappresentano delle condizioni (si può notare che la congiunzione delle condizioni di una riga è mutuamente esclusiva rispetto alla congiunzione delle condizioni di ogni altra riga della matrice), la cui validità implica che un accesso ad array della forma $a[indice]$ sia modellato dalla variabile astratta a_i se dopo le condizioni nella riga compare un intero (matrice A_a), da tutte le variabili astratte a_{θ_i} per ogni θ_i che compare nella riga (matrice B_a). Se invece le condizioni sono seguite dal simbolo di indefinito, allora l'elemento dell'array indicizzato da *indice* non è modellato nel programma astratto. A partire dalle matrici A_a e B_a, viene calcolata la matrice C_a equivalente alla congiunzione delle formule in forma normale disgiuntiva rappresentate dalle due matrici. La congiunzione avviene concatenando ogni riga di A_a con ogni riga di B_a. Per ogni array $a \in A_P$, si ottiene dunque la matrice C_a costituita da un numero di righe pari al prodotto tra il numero di righe di A_a $(R(a)+1)$ e il numero di righe di B_a $(2^{|\theta(a)|})$. Può capitare, per come viene costruita C_a, che una sua riga contenga, dopo la sequenza di condizioni, una sequenza costituita sia dal simbolo u che da un intero o una sequenza di $theta_i$ oppure da due simboli u. Ciò non è corretto infatti, per l'interpretazione data ad ogni riga di C_a (ovvero se la sequenza di condizioni è verificata allora l'accesso ad $a[indice]$ è modellato dalla sequenza di variabili a_e con e appartenente alla riga e uguale ad un numero oppure ad una variabile θ_i oppure non è modellato affatto) al termine della sequenza di istruzioni ci si aspetta una sequenza di numeri e/o variabili θ_i oppure il solo simbolo inefinito, pertanto prima terminare la funzione elimina tutte le occorrenze inopportune di u da ogni riga di C_a per ogni $a \in A_P$ dopodichè la funzione restituisce la corrispondenza tra ogni array $a \in A_P$ e la relativa matrice C_a, memorizzata nella struttura dati *vet*.

Come già detto la costruzione della relazione di transizione astratta avviene attraverso l'istanziazione dello schema di vincoli rappresentati da C_a, attraverso la sostituzione, all'interno dei vincoli, del generico indice *indice* con l'espressione che indicizza l'accesso ad array (come si vedrà in seguito, si fa in modo che all'interno di ogni statement compaia al più un solo accesso ad array) presente nello statement e la sostituzione della sequenza di interi e variabili con la sequenza di formule che descrivono la relazione di transizione astratta dello statement stesso.

Funzione *abstractExpression*

Viene presentato di seguito lo pseudo - codice relativo alla funzione *abstractExpression*, che calcola la relazione di transizione astratta associata ad uno statement di programma utilizzando la struttura dati *vet* computata dalla funzione *preProcessingAbsArray*.

```
  abstractExpression(stm,vet)
1   ⟨lineare, temp⟩ = divTerm(stm) // stm viene trasformata in modo che contenga
                    // al più una sola occorrenza di array. Le altre occorrenze di array
                    // vengono assegnate a variabili temporanee. lineare si ottiene da stm
                    // sostituiendo le occorrenze di array (eccetto la prima) con le variabili
                    // temporanee a cui sono state assegnate. temp contiene gli
                    // assegnamenti delle occorrenze di array alle variabili temporanee
2   per ogni assegnamento Tmp = a[i] ; ∈ temp
3       single = istance(Tmp = a[i] ;,vet); //genera la relazione di transizione astratta
                    // per l'assegnamento attraverso l'opportuna istanziazione di vet
4       accoda(τ_T,single)
    end
5   res = DNFnormalize(lineare) // lineare viene tradotto in formula booleana in
                    // forma normale disgiuntiva e memorizzato in una matrice
6   res = ArrRemove(res,vet)
7   accoda(τ_T,res)
8   if (stm è un condizionale o iterativo)
9       res = DNFnormalize(neg(lineare)) // la negata della condizione viene
                    // trasformata in forma normale disgiuntiva
10      res = ArrRemove(res,vet)
11      accoda(τ_F,res)
    end
12  return ⟨τ_T, τ_F⟩
```

La funzione dapprima trasforma lo statement in uno contenente al più una sola occorrenza di array ([1]): ogni occorrenza di array $a[i]$ nello statement, eccetto la prima, viene sosituita da una diversa variabile temporanea Tmp e per ognuna di esse viene inserito nella struttra *temp* l'assegnamento Tmp=$a[i]$;. In seguito vengono calcolate e memorizzate nel vettore τ_T le relazioni di transizione astratta per ognuno di tali assegnamenti attraverso la funzione *istance* ([2-4]), che istanzia in maniera opportuna lo schema di vincoli *vet*. Infine la funzione $DNFnormalize$ converte lo statement *lineare* in una formula booleana in forma normale disgiuntiva, i cui congiunti e disgiunti vengono memorizzati in una matrice ([5]), data in input alla funzione $ArrRemove$ che, sfruttando sempre un'opportuna istanziazione di *vet*, calcola la relazione di transizione astratta dello statement *lineare* ([6]), anch'essa memorizzata nel vettore τ_T ([7]). Se *stm* è uno statement condizionale o iterativo, allora la funzione calcola anche la relazione di transizione per il ramo falso ([8-11]). Infine viene restituita in output la coppia di relazioni di transizione per il ramo vero e per il ramo falso ([12]).

Verranno illustrate ora le funzioni *istance* e $ArrRemove$, che effettivamente computano la relazione di transizione astratta.

Funzione *istance*

Viene riportato di seguito lo pseudo-codice della funzione:

```
  istance(Tmp=a[e1];,vet)
1    C_a = vet[a]
2    for i = 1 to numRighe(C_a)
3        for j = 1 to numColonne(C_a)
4            if (C_a[i][j] è un vincolo del tipo indice op e)
5                C'_a[i][j] = (e1 op e)
6            elseif (C_a[i][j] è il simbolo indefinito)
7                C'_a[i][j] = (Tmp = u)
8            elseif (C_a[i][j] è una variabile θ_l)
9                C'_a[i][j] = (Tmp = a_θl)
10           elseif (C_a[i][j] è una costante intera c)
11               C'_a[i][j] = (Tmp = a_c)
             end
         end
     end
12   return C'
```

Funzione *ArrRemove*

Viene riportato di seguito lo pseudo-codice della funzione:

```
  ArrRemove(res, vet)
1    if è uno statement di assegnamento // res contiene una sola riga e una sola colonna
                                        // res[1][1] = (lVal = rVal)
2        if lVal = a[e1] // assegnamento ad array
3            C_a = vet[a]
4            for i = 1 to numRighe(C_a)
5                for j = 1 to numColonne(C_a)
6                    if (C_a[i][j] è un vincolo del tipo indice op e)
7                        accoda(C'_a[i],(e1 op e))
8                    elseif (C_a[i][j] è il simbolo indefinito)
9                        per ogni n ∈ R(a)
10                           accoda(C'_a[i],(a'_n = a_n))
                         end
11                       per ogni θ ∈ R(a)
12                           accoda(C'_a[i],(a'_θ = a_θ))
```

```
                        end
13                  elseif ($C_a[i][j]$ è una variabile $\theta_l$)
14                      per ogni $n \in R(a)$
15                          accoda($C'_a[i]$,$(a'_n = a_n)$)
                        end
16                      per ogni $\theta \in R(a)$
17                          if ($\theta = C_a[i][j]$)
18                              accoda($C'_a[i]$,$(a'_\theta = rVal)$)
                            else
19                              accoda($C'_a[i]$,$(a'_\theta = a_\theta)$)
                            end
                        end
20                  elseif ($C_a[i][j]$ è una costante intera $c$)
21                      per ogni $n \in R(a)$
22                          if ($n = C_a[i][j]$)
23                              accoda($C'_a[i]$,$(a'_n = rVal)$)
                            else
24                              accoda($C'_a[i]$,$(a'_n = a_n)$)
                            end
                        end
25                      per ogni $\theta \in R(a)$
26                          accoda($C'_a[i]$,$(a'_\theta = a_\theta)$)
                        end
                    end
                end
            end
        else // assegnamento a variabile scalare
27          $C'_a$ = istanceGeneral($res$[1][1],$vet$)
        end
28  elseif è uno statement condizinale, iterativo o una chiamata a procedura
29      for $i = 1$ to numRighe($res$)
30          for $j = 1$ to numColonne($res$)
31              $parziale$ = istanceGeneral($res[i][j]$,$vet$)
32              $sub$ = matrixAnd($sub$,$parziale$)
            end
33          $C'_a$ = matrixOr($C'_a$,$sub$)
        end
    end
34  return $C'_a$
```

Il comportamento della funzione dipende dal tipo di statement. Se lo statement è un assegnamento ad array ([2]), allora si istanzia in maniera opportuna la relativa matrice C_a memorizzata in vet in corrispondenza dell'array a coinvolto nello statement (si ricorda che lo statement contiene al più una sola occorrenza di array). L'istanziazione delle condizioni è banale e prevede solo la sostituzione del generico indice con l'espressione che indicizza l'array nello statement ([6-7]). Se invece l'elemento di C_a è un intero c ([20-26]) (una

variabile θ_l ([13-19])), allora significa che se si verificano tutte le condizioni della riga allora l'elemento modificato dallo statement è quello indicizzato da c (da θ_l, rispettivamente) e quindi il valore della variabile astratta che lo modella va aggiornato ([22-23] e [17-18] rispettivamente) mentre il valore di tutte le altre va lasciato inalterato ([24] e [19] rispettivamente). Infine, se l'elemento di C_a è il simbolo di indefinito, allora significa che l'assegnamento non coinvolge elementi di array modellati nell'astrazione, pertanto il valore di tutte le variabili astratte che modellano elementi di array va lasciato inalterato ([9-12]). Nel caso in cui lo statement è un assegnamento a varaibile scalare viene richiamata la funzione *istanceGeneral* che si occuperà di calcolare la relazione di transizione astratta ([27]). Anche nel caso in cui lo statement è di tipo condizionale, iterativo o una chiamata a procedura ([28-33]) si utilizza la procedura *istanceGeneral*, ma in questo caso avvengono più chiamate alla funzione, una per ogni elemento della matrice res ([29-31], nel caso dell'assegnamento res era una matrice 1×1) e le matrici restituite da ogni chiamata vanno combinate insieme con delle operazioni di congiunzione ([32]) e disgiunzione ([33]) attraverso le funzioni *matrixAnd* e *matrixOr*, che eseguono congiunzione e disgiunzione di due matrici in modo tale che il prodotto sia una matrice che rappresenti la congiunzione o disgiunzione, in forma normale disgiuntiva, delle formule rappresentate dalle due matrici in ingresso.

Funzione *istanceGeneral*

Viene fornito infine lo pseudo-codice della funzione *istanceGeneral*.

```
   istanceGeneral(term,vet)
1      if esiste un'occorrenza di array a[e1] in term
2          C_a = vet[a]
3          for i = 1 to numRighe(C_a)
4              for j = 1 to numColonne(C_a)
5                  if C_a[i][j] è un vincolo del tipo indice op e
6                      C'_a[i][j] = (e1 op e)
7                  elseif C_a[i][j] è il simbolo indefinito
8                      C'_a[i][j] = sostituisci l'occorrenza di array a[e1] in term con u
9                  elseif C_a[i][j] è una variabile θ_l
10                     C'[i][j] = sostituisci l'occorrenza di array a[e1] in term con a_θl
11                 elseif C_a[i][j] è una costante intera c
12                     C'[i][j] = sostituisci l'occorrenza di array a[e1] in term con a_c
                   end
```

```
            end
        end
    else // non esiste un'occorrenza di array a[e_1] in term
13      C'[1][1] = term
    end
```

Se è presente un'occorrenza di array $a[e_1]$ in *term* ([1-12]), allora la funzione istanzia la matrice C_a, memorizzata in *vet* in corrispondenza dell'array a, sostituendo all'interno dei vincoli il generico indice *indice* con l'espressione e_1 che indicizza l'occorrenza di array in *term* ([5-6]) e sostituendo i simboli di indefinito ([7-8], le costanti intere c [11-12], le variabili θ_l [9-10]) con l'espressione ottenuta da *term* sostituendo l'occorrenza di array con u (a_c, a_{θ_l}, rispettivamente). L'idea intuitiva deriva dal fatto che la semantica di uno statement è catturata da una sequenza di formule equivalenti ottenute sostituendo all'unica occorrenza di array presente in un'espressione dello statement tutte le possibili variabili astratte che modellano l'elemento dell'array a cui si fa riferimento.

5.3 Modulo di verifica

Il modulo di verifica di Eureka, costituito dal Model Checker, si occupa della traduzione simbolica, tramite l'utilizzo di ADLC, della relazione di transizione astratta τ costruita in fase di astrazione e nel successivo calcolo della raggiungibilità dei nodi ERROR, attraverso il calcolo dei path edge, rappresentati anch'essi tramite ADLC.

In seguito alle modifiche apportate dal presente lavoro è stato necessario modificare, o meglio integrare, all'interno del model checker di Eureka, sia la fase relativa alla costruzione dell'ADLC per la relazione di transizione che quella della verifica della raggiungibilità, per adattarle al nuovo modello astratto. Infatti, come gia illustrato nei capitoli precedenti, il nuovo meccanismo di astrazione si basa sull'introduzione di nuove variabili nel programma astratto, variabili che vanno assegnate in opportuni nodi del CFG. Pertanto è stato necessario inserire frammenti di codice all'interno del model checker che computassero l'ADLC della relazione di transizione astratta anche per gli assegnamenti alle variabili θ e fare in modo che in fase di verifica venissero considerati gli effetti di tali assegnamenti durante la costruzione dei path edge.

Come più volte evidenziato, si utilizzano gli ALDC sia per la rappresentazione della relazione di transizione astratta che per i path edge. Pertanto la fase di verifica, che consiste nell'implementazione della relazione $\widehat{\rightarrow}$ defini-

ta in sezione 3.3.2, viene eseguita attraverso operazioni eseguite sugli ADLC rappresentanti path edge e relazione di transizione astratta.

5.3.1 PPL: The Parma Polyhedra Library

Per la rappresentazione degli ADLC si è deciso di sfruttare la Parma Polyhedra Librery (PPL). Si tratta di una libreria che fornisce astrazione numerica particolarmente adatta alle applicazioni nel campo di analisi e nella verifica dei sistemi complessi. PPL può gestire tutti i poliedri convessi che possono essere definiti come l'intersezione di un numero limitato di iperspazi (aperti o chiusi), ciascuno descritto da un'uguaglianza o da una disuguaglianza (stretta o meno) con i coefficenti razionali. PPL inoltre gestisce una ristretta classe di poliedri che offrono un'interessante alternanza complessità/precisione. La libreria inoltre gestisce powerset finiti di poliedri e problemi di programmazione lineare risolti con una versione in aritmetica esatta dell'algoritmo del simplesso [Bag06].

Poliedri convessi e Powerset di poliedri

Si indica con $\mathbb{R}^n$ lo spazio vettoriale n-dimensionale nel campo dei numeri reali. Si indica inoltre l'insieme di tutti i reali non negativi col simbolo $\mathbb{R}_+$. Per ogni $i \in \{0, 1, \ldots, n-1\}$, v_i identifica l'i-esimo elemento del vettore colonna $\mathbf{v} = (v_0, v_1, \ldots, v_{n-1})^T \in \mathbb{R}^n$. Si denota con $\mathbf{0}$ il vettore appartenente a $\mathbb{R}^n$ chiamato origine, vettore che ha tutte le componenti uguali a zero. Un vettore $v \in \mathbb{R}^n$ può anche essere interpretato come una matrice in $\mathbb{R}^{n\times 1}$ e manipolata con le usuali operazioni definite sulle matrici: somma, prodotto (scalare o matriciale) e trasposizione. Un prodotto scalare per $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$, indicato con $\langle \mathbf{v}, \mathbf{w} \rangle$, è il numero reale

$$\mathbf{v}^T \mathbf{w} = \Sigma_{i=0}^{n} v_i w_i.$$

Per ogni $S_1, S_2 \subseteq \mathbb{R}^n$, la somma di Minkowski di S_1 e S_2 è data da: $S_1 + S_2 = \{\mathbf{v}_1 + \mathbf{v}_2 \mid \mathbf{v}_1 \in S_1, \mathbf{v}_2 \in S_2\}$. Per ogni coppia di vettori $\mathbf{a}, \mathbf{x} \in \mathbb{R}^n$, con $\mathbf{a} \neq \mathbf{0}$, per ogni $b \in \mathbb{R}$ e per ogni $op \in \{=, \geq, >\}$, un vincolo lineare $\langle \mathbf{a}, \mathbf{x} \rangle$ op b definisce:

- un iperpiano affine se è composto da un vincolo di uguaglianza, ovvero se $op \in \{=\}$;
- un semipiano affine topologicamente chiuso se è composto da un vincolo di disuguaglianza, ovvero se $op \in \{\geq\}$;

- un semipiano affine topologicamente aperto se è composto da un vincolo di disuguaglianza stretta, ovvero se $op \in \{>\}$.

Notare che ogni iperpiano $\langle \mathbf{a}, \mathbf{x} \rangle = b$ può essere definito come l'intersezione di due semispazi affini chiusi $\langle \mathbf{a}, \mathbf{x} \rangle \geq b$ e $-\langle \mathbf{a}, \mathbf{x} \rangle \geq -b$. Inoltre notare che, quando $\mathbf{a} = \mathbf{0}$, il vincolo $\langle \mathbf{a}, \mathbf{x} \rangle \geq b$ o è una tautologia o è inconsistente, così che esso definisce o un intero spazio vettoriale $\mathbb{R}^n$ o l'insieme vuoto $\emptyset$. L'insieme $\mathcal{P} \subseteq \mathbb{R}^n$ è un *poliedro non necessariamente chiuso* (NNC Polyhedron) se e soltanto se vale una delle due seguenti condizioni:

- $\mathcal{P}$ può essere espresso come intersezione di un numero finito di semispazi affini (chiusi o aperti) di $\mathbb{R}^n$;
- $n = 0$ e quindi $\mathcal{P} = \emptyset$.

L'insieme di tutti i poliedri NNC su uno spazio vettoriale $\mathbb{R}^n$ è definito come $\mathbb{P}_n$. L'insieme $\mathcal{P} \in \mathbb{P}_n$ è un *poliedro convesso chiuso* se e soltanto se vale una delle seguenti condizioni:

- $\mathcal{P}$ può essere espresso come intersezione di un numero finito di semispazi affini chiusi di $\mathbb{R}^n$;
- $n = 0$ e quindi $\mathcal{P} = \emptyset$.

L'insieme di tutti i poliedri chiusi su uno spazio vettoriale $\mathbb{R}^n$ è indicato da $\mathbb{CP}_n$.

Ordinando i poliedri NNC per inclusione, l'insieme vuoto $\emptyset$ e l'intero spazio vettoriale $\mathbb{R}^n$ sono, rispettivamente, il più piccolo ed il più grande elemento sia di $\mathbb{P}_n$ che di $\mathbb{CP}_n$. Lo spazio vettoriale $\mathbb{R}^n$ è anche chiamato poliedro universo. Un poliedro NNC può essere rappresentato attraverso dei vincoli. Per definizione un poliedro $\mathcal{P} \in \mathbb{P}_n$ è l'insieme di soluzioni di un sistema di vincoli. Usando la notazione matriciale abbiamo:

$$\mathcal{P} \stackrel{def}{=} \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}_1\mathbf{x} = \mathbf{b}_1, \mathbf{A}_2\mathbf{x} \geq \mathbf{b}_2, \mathbf{A}_3\mathbf{x} > \mathbf{b}_3\}$$

dove, per ogni $i \in \{1, 2, 3\}$, $\mathbf{A}_i \in \mathbb{R}^{m_i \times n}$, $\mathbf{b}_i \in \mathbb{R}^{m_i}$ e $m_1, m_2, m_3 \in \mathbb{N}$ sono il numero di uguaglianze, disuguaglianze non strette e disuguaglianze strette, rispettivamente. La dimensione di un poliedro non necessariamente chiuso $\mathcal{P} \in \mathbb{P}_n$ è la dimensione $n \in \mathbb{N}$ del corrispondente spazio vettoriale $\mathbb{R}^n$. Similmente per i vincoli. A meno che sia dichiarato il contrario, tutti i poliedri ed i vincoli di PPL devono obbedire alle seguenti regole di compatibilità sulla dimensione:

- poliedri sono compatibili sulla dimensione se e soltanto se hanno la stessa dimensione;

- il vincolo $\langle \mathbf{a}, \mathbf{x} \rangle$ *op* b, con $op \in \{=, \geq, >\}$ e $\mathbf{a}, \mathbf{x} \in \mathbb{R}^m$, è compatibile sulla dimensione con un poliedro di dimensione n se e soltanto se $m \leq n$;

- un sistema di vincoli è compatibile sulla dimensione con un poliedro se e soltanto se tutti i vincoli nel sistema sono compatibili sulla dimensione con il poliedro stesso.

Un poliedro è detto *razionale* se può essere rappresentato attraverso un sistema di vincoli dove tutti i vincoli hanno coefficienti razionali. Sia D un dominio sul quale è definita la relazione di *entailment* ($\vdash$), in cui è identificabile l'elemento top ($\mathbf{1}$) e l'elemento bottom ($\mathbf{0}$). Un insieme $\mathcal{S} \subseteq D$ è detto *non ridondante* rispetto alla relazione $\vdash$ se e soltanto se $\mathbf{0} \notin \mathcal{S}$ e per ogni $d_1, d_2 \in \mathcal{S}$, $d_1 \vdash d_2 \Rightarrow d_1 = d_2$. L'insieme di sottoinsiemi finiti non ridondanti su D (rispetto a $\vdash$) è indicato con $\wp^{\vdash}_{fn}(D)$. La funzione $\Omega^{\vdash}_D : \wp_f(D) \rightarrow \wp^{\vdash}_{fn}(D)$, chiamata Ω-riduzione, mappa un sottoinsieme finito di D nella sua controparte non ridondante; essa è definita per $S \in \wp_f(D)$ come segue:

$$\Omega^{\vdash}_D(S) \stackrel{def}{=} S \setminus \{d \in S \mid d = \mathbf{0} \text{ oppure esiste } d' \in S \text{ tale che } d \Vdash d'\}$$

dove $d \Vdash d'$ denota $d \vdash d'$ e $d \neq d'$.

Un dominio di powerset finiti su D è l'insieme di tutti gli insiemi finiti non ridondanti su D, denotato con D_P. Il dominio include l'ordinamento rispetto a $\vdash_P$ definito come segue. Siano $S_1, S_2 \in D_P$, $S_1 \vdash_P S_2$ se e soltanto se per ogni $d_1 \in S_1$ esiste $d_2 \in S_2$ tale che $d_1 \vdash d_2$. Inoltre l'elemento top è dato da $\{\mathbf{1}\}$ mentre l'elemento bottom da $\emptyset$. Il dominio dei powerset di poliedri, indicato con $(\mathbb{P})_P$, è un dominio di powerset finiti definiti sul dominio dei poliedri NNC.

Rappresentazione di ADLC con un Powerset di poliedri

Un vincolo su una o più varibili è rappresentato in PPL con un'espressione lineare. Una congiunzione di espressioni lineari determina un sistema di vincoli la cui soluzione è un poliedro. Una disgiunzione di poliedri rappresenta un powerset di poliedri. La costruzione di powerset di poliedri attraverso PPL è del tutto simile alla costruzione di un ADLC. Per questo motivo e per sfruttare l'enorme potenzialità di PPL si è deciso, nella realizzazione del Model Checker per il tool Eureka, di rappresentare un ADLC attraverso un powerset di poliedri NNC. Tale scelta ci permette di sfruttare la libreria PPL non solo per la rappresentazione degli ADLC ma anche per la realizzazione di tutti gli operatori che manipolano gli ADLC. La costruzione di un ADLC

risulta quindi molto semplice. Ogni espressione lineare è utilizzata come elemento base per la costruzione di sistemi di vincoli, i quali possono essere costruiti a partire da un sistema vuoto ed aggingendo, uno alla volta, tanti congiunti quanti sono i vincoli lineari. Un meccanismo simile è utilizzato per la costruzione di un powerset di poliedri. A partire dal powerset vuoto $\emptyset$ è possibile costruire un qualsiasi altro powerset aggiungendo nuovi elementi (disgiunti) nel powerset. Ogni disgiunto è composto da un sistema di vincoli. Dati un powerset di poliedri $S \in D_P$ ed un elemento (poliedro) $d \in D$, l'operatore *add_disjunct* calcola l'elemento $\Omega^{\top}_D(S \cup \{d\})$. Questo elemento è ancora un powerset. Si utilizzano i powerset di poliedri $\emptyset$ e $\{\mathbf{1}\}$, dove con $\mathbf{1}$ si denota il poliedro universo, per la costruzione degli ADLC $\bot$ (falso) e $\top$ (vero), rispettivamente. Bisogna però considerare che le variabili utilizzate da PPL sono delle variabili interne che, in linea di principio, non sono legate alle variabili del programma lineare e quindi non sono legate alle variabili degli ADLC. Per ovviare a questo problema si è deciso di sfruttare delle tabelle per creare una corrispondenza fra le variabili del programma e le variabili PPL. Ad ogni variabile di programma corrispondono due o più variabili negli ADLC, bisogna considerare infatti, oltre alla variabile stessa (ad es.: x), anche le sue versioni primate (ad es.: x', x'' e x'''). D'altro canto, ogni variabile di PPL corrisponde ad una delle dimensioni sulle quali sono definiti i poliedri. E' possibile quindi creare una corrispondenza fra le variabili degli ADLC e le variabili di PPL. Per far questo si fa uso di tabelle grazie alle quali è possibile memorizzare due mappe, una per l'associazione variabileADLC - variabilePPL ed una per l'associazione inversa[1]. Bisogna tener presente che effettuare operazioni su powerset di poliedri è relativamente costoso, infatti molte operazioni su poliedri hanno complessità quadratica o cubica rispetto alla dimensione su cui sono definiti i poliedri e le operazioni su powerset sono lineari rispetto al numero di poliedri che compongono il powerset stesso. Ovviamente al crescere del numero di variabili, e quindi della dimensione dello spazio su cui è definito il poliedro, cresce anche il tempo necessario ad effettuare ogni singola operazione. Per questo motivo è utile usare il minor numero di variabili PPL nella rappresentazione di un ADLC. Si è deciso, infatti, di utilizzare tante tabelle quanti sono gli scope di un programma. In questo modo ogni tabella conserva un riferimento solo alle variabili visibili all'interno dello scope e si evita di rappresentare variabili che non portano informazione utile, ovvero le variabili non visibili nello scope dello statement. Di conseguenza, il numero di variabili di programma necessari per la rappresentazione di un ADLC è ridotto e quindi è ridotto anche il numero di

[1]Di fatto non sono fondamentali entrambe, ma memorizzare esplicitamente entrambe le corrispondenze risulta più efficiente a discapito di un ragionevole spreco di memoria

variabili PPL necessarie per la rappresentazione del relativo powerset. Notare che due powerset uguali non necessariamente corrispondono a due ADLC uguali. Infatti se questi due powerset fanno riferimento a tabelle differenti probabilmente le variabili coinvolte saranno differenti.

Operazioni su poliedri e powerset di poliedri

Nella sezione 3.2.3 sono state definite alcune operazioni che manipolano gli ADLC necessarie alla fase di verifica. Pertanto l'implementazione del Model Checker ha richiesto l'implementazione di tali operazione.

Sfruttando gli operatori messi a disposizione dalla libreria PPL, sono state implementate le seguenti funzioni, utilizzate in fase di verifica:

- $rename\_vars$: implementa l'operatore di applicazione. Sia $\delta = \lambda \mathbf{x}\mathbf{x}'.D$ un ADLC, la funzione $rename\_vars(\delta, \mathbf{s}, \mathbf{s}')$, dove $\mathbf{s}$ e $\mathbf{s}'$ rappresentano due vettori di variabili, restituisce l'ADLC $\delta_2 = \delta(\mathbf{s}, \mathbf{s}')$;

- $disjunct$: implementa l'operatore di disgiunzione ($\sqcup$) tra ADLC. Siano δ_1 e δ_2 due ADLC, la funzione $disjunct(\delta_1, \delta_2)$ restituisce l'ADLC $\delta_3 = \delta_1 \sqcup \delta_2$;

- $conjunct$: implementa l'operatore di congiunzione ($\sqcap$) tra ADLC. Siano δ_1 e δ_2 due ADLC, la funzione $conjunct(\delta_1, \delta_2)$ restituisce l'ADLC $\delta_3 = \delta_1 \sqcap \delta_2$;

- $Qelim$: implementa l'operatore di eliminazione (quantifier elimination, $\exists$). Sia $\delta = \lambda \mathbf{x}\mathbf{x}'.D$ un ADLC con D DLC contenente delle variabili libere contenute nel vettore $\mathbf{x}''$, allora la funzione $Qelim(\delta, \mathbf{x}'')$ restituisce l'ADLC $\lambda \mathbf{x}\mathbf{x}'.(\exists \mathbf{x}''.D)$;

- $entail$: implementa l'operatore di entailment ($\sqsubseteq$). Siano δ_1 e δ_2 due ADLC. La funzione $entail(\delta_1, \delta_2)$ restituisce valore di verità vero se e solo se vale $\delta_1 \sqsubseteq \delta_2$.

5.3.2 Costruzione dell'ADLC corrispondente alla relazione di transizione e verifica del modello

Ora che sono stati forniti tutti gli strumenti alla base della procedura di verifica della raggiungibilità, è possibile illustrare il funzionamento del modulo di verifica di Eureka. Il Model Checker effettua dapprima la traduzione della relazione di transizione astratta, calcolata in fase di astrazione, in un ADLC generato utilizzando la libreria PPL (funzione $buildTauAbs$, con riferimento

alla tabella 5.1), dopodichè, il calcolo della raggiungibilità avviene nella funzione $verify$. La funzione, a partire dal primo nodo del main, propaga i path edge associati al nodo a tutti i suoi successori, compatibilmente con la relazione di transizione astratta, e la procedura si ripete per ogni nodo verso cui sono stati propagati nuovi path edge. La propagazione dei path edge di un generico nodo i avviene attraverso l'operazione di giunzione (eventualmente ripetuta più volte a seconda del numero di ADLC che occorre giungere) tra l'ADLC che rappresenta i path edge di i (δ_i) e gli ADLC relativi alle relazioni di transizione astratte associate al nodo (ovvero gli ADLC corrispondenti alle relazioni di transizione per tutti gli assegnamenti a variabili θ che avvengono in i, denotati con $\tau_{\theta_1,i}, \ldots, \tau_{\theta_m,i}$, gli ADLC della relazione di transizione dello statement di i, denotato con τ_i ed eventualmente quello relativo al ramo falso, denotato con $\tau_{i_{false}}$, in caso di statement iterativo o condizionale). La giunzione tra due ADLC viene affettuata dalla funzione $join(\delta_1, \delta_2)$, il cui pseudocodice è rappresentato di seguito:

```
   join(δ1,δ2)
1    δ'1 = rename_vars(δ1,x,x'') //rinomina delle variabili primate in doppio primate
2    δ'2 = rename_vars(δ2,x'',x') //rinomina delle variabili normali in doppio primate
                                  //affinchè coincidano con quelle di δ1
3    δ3 = conjunct(δ'1,δ'2) //congiunzione degli ADLC
4    δ3 = Qelim(δ3,x") //eliminazione delle variabili intermedie
5    return δ3
```

La funzione effettua la rinomina delle variabili dei due ADLC affinchè le variabili di arrivo del primo ADLC coincidano con quelle di partenza del secondo ADLC, dopodichè si effettua la congiunzione dei due ADLC ed infine si eliminano le valutazioni intermedie, ormai non più necessarie, delle variabili.

5.4 Simulazione e raffinamento

Se la fase di verifica rileva la presenza di una traccia di errore eseguibile nel modello astratto, occorre verificare la fattibilità della traccia. Come già ampiamento descritto, tale problema viene ridotto al problema di stabilire la soddisfacibilità o meno di un insieme di formule. A tale scopo Eureka si interfaccia anche con un'altro modulo esterno, CVC Lite.

5.4.1 CVC Lite

CVC Lite [Bar] è un Theorem Prover automatico (o validity checker) per logiche del primo ordine, include il supporto per quantificatori, funzoni parziali ed altro. Alcune delle teorie attualmente implementate nel tool comprendono l'aritmetica intera, reale e teoria degli array. Formalmente, CVC Lite risolve il problema della validità (o soddisfacibilità) di una formula in un contesto dato:

$$\Gamma \models \phi$$

dove Γ è un insieme di formule che definiscono il contesto logico, e ϕ è una formula la cui validità deve essere verificata in Γ. Se la formula è valida, CVC Lite produce una dimsotrazione e un sottoinsieme di assunzioni $\Gamma' \subseteq \Gamma$ utilizzate nella prova e tali che $\Gamma' \models phi$. Se la formula non è valida, allora è mostrato un controesempio che falsifica ϕ sottoforma di nuove assunzioni consistenti con Γ. Si potrebbe pensare di utilizzare direttamente CVC Lite per effettuare Model Checking su un programma C, ma codificare un intero programma e osservarne il comportamento è un'impresa non fattibile.

Pertanto all'interno di Eureka, precisamente durante il processo di simulazione, viene codificato solo un sottoinsieme delle istruzioni del programma, quelle che compongono la traccia restituita dal Model Checker. Tali formule vengono convertite in un formato accettato da CVC Lite e asserite all'interno del contesto Γ, al fine di verificarne la consistenza, che corrisponde a verificare la validità della formula $\Gamma \models \bot$.

Tipi Base

CVC Lite utilizza i seguenti tipi base:

- *Expr*: tutte le formule, i teoremi e le dimostrazioni sono oggetti di tipo espressione. Ogni espressioni ha un tipo (che rappresenta l'operazione principale recuperabile con il metodo *getKind*) e un numero di argomenti variabile che sono visti come figli di quell'espressione. Per accedervi si utilizza il metodo *getKids*. CVC Lite fornisce numerosi metodi per la costruzione di espressioni di vario genere come ad esempio: *eqExpr*, *notExpr*, *andExpr*, *impExpr*.

- *Theorem*: questo oggetto rappresenta un teorema ed è corredato quindi di assunzioni, conclusioni e una dimostrazione. Le assunzioni sono oggetti di tipo *Assumptions* e sono a loro volta dei teoremi. Le conclusioni sono espressioni di tipo *Expr* come anche gli oggetti *Proof* rappresentanti le dimostrazioni. L'introduzione di *Assumptions* e *Proof*

al posto di *Expr* è giustificata dal fatto che si evita che i metodi per le espressioni possano modificarli direttamente.

- *Theory*: rappresenta una classe astratta da cui ereditano tutte le teorie implementate in CVC Lite. Ognuna di esse ha le proprie regole di inferenza legate al particolare tipo di teoria. In Eureka, in particolare sono state utilizzate principalmente quella aritmetica $TheoryArith$, oltre alla teoria base $TheoryCore$.

- *TheoremProducer*: altro tipo di classe molto importante che consente di gestire ulteriori regole di inferenza. In particolare in Eureka, per la ricostruzione della dimostrazione, sono stati sfruttati i seguenti:

  - $CommonTheoremProducer$
  - $SearchEngineTheoremProducer$
  - $CoreTheoremProducer$
  - $ArithTheoremProducer$
  - $ArrayTheoremProducer$

- $ValidityChecker$: è la classe che consente di effettuare tutte le operazioni fondamentali; ad esempio il metodo $assertFormula$ consente di inserire una formula nel contesto e *query* che verifica la validità (o insoddisfacibilità) di una o più formule. Inoltre vi sono i metodi per istanziare nuove espressioni e nuovi tipi.

Regole di inferenza

In questa sezione si fornisce una panoramica delle regole di inferenza utilizzate e reimplementate per la ricostruzione della prova in fase di rafinamento (anche se come già detto l'orientamento per il futuro è quello di evitare la ricostruzione della prova). Per questioni di spazio sono mostrate a campione solo una piccola parte delle regole realmente implementate. E' possibile effettuare una suddivisione in base alla teoria a cui le regole appartengono.

CommonTheorem Fanno parte di questa categoria tutte le regole fondamentali per la manipolazioni base tra cui vi sono quelle per la creazione di assunzioni, per la riflessività, la simmetria e la transitività.

- $assumpRule$: trasforma un espressione a in un'assunzione $\frac{}{a \vdash a}$;

- $reflexivityRule$: regola riflessiva $\frac{}{a=a}$ oppure $\frac{}{a \Leftrightarrow a}$;

- *symmetryRule*: regola simmetrica $\frac{a_1=a_2}{a_2=a_1}$;
- *transitivityRule*: transitività $\frac{a_1=a_2 \quad a_2=a_3}{a_1=a_3}$;
- *substitutivityRule*: regola di sostituzione $\frac{c_1=d_1 \wedge ... \wedge c_n=d_n}{op(c_1,...,c_n)=op(d_1,...,d_n)}$;
- *contraddictionRule*: contraddizione $\frac{\Gamma_1 \vdash e \quad \Gamma_2 \vdash \neg e}{\Gamma_1 \cup \Gamma_2 \vdash \bot}$;
- *notnotElim*: eliminazione della doppia negazione $\frac{\Gamma \vdash \neg\neg e}{\Gamma \vdash e}$;
- *implMP*: Modus Ponens dell'implicazione $\frac{\Gamma_1 \vdash e_1 \quad \Gamma_2 \vdash e_2 \Rightarrow e_2}{\Gamma_1 \cup \Gamma 2 \vdash e_2}$;
- *andElim*: eliminazione dell'AND $\frac{\vdash e_1 \wedge ... \wedge e_n}{\vdash e_i}$ con $1 \leq i \leq n$;
- *andIntro*: introduzione dell'AND $\frac{\Gamma_1 \vdash e_1 ... \Gamma_n \vdash e_n}{\bigcup_{i=1}^{n} \Gamma_i \vdash \bigwedge_{i=1}^{n} e_i}$.

CoreTheoremProducer In questa categoria ci sono tutte regole che costruiscono nuovi assiomi che consentono la riscrittura in altre forme delle espressioni in input.

- *rewriteNotIte*: la regola riscrive un espressione condizionale $\neg a?b : c \Leftrightarrow a?c : b$;
- *rewriteIteSame*: semplifica l'espressione condizionale $c?e : e = e$;
- *rewriteIteBool*: eliminazione del condizionale $c?e_1 : e_2 \Leftrightarrow (\neg c \vee e_1) \wedge (c \vee e_2)$;
- *rewriteIteToImp*: trasformazione da condizionale in implicazione $a?b : \top \Leftrightarrow a \Rightarrow b$;
- $iffToIte$: trasforma la doppia implicazione in un'espressione condizionale $a \Leftrightarrow b = a?b : \neg b$.

SearchEngine

- *proofByContradiction*: prova per contraddizione $\frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}$;
- *negIntro*: introduzione della negazione $\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}$;
- *caseSplit*: separazione dei casi $\frac{\Gamma_1, A \vdash C \quad \Gamma_2, \neg A \vdash C}{\Gamma_1 \cup \Gamma_2 \vdash C}$;
- *conflictClause*: clausola conflittuale $\frac{\Gamma, A_1, ..., A_n \vdash \bot}{\Gamma \vdash \neg A_1 \vee ... \vee \neg A_n}$.

ArrayTheorem Le regole in questa categoria effettuano manipolazioni e riscritture delle strutture dati array.

- $rewriteReadWrite$: $read(write(store, index_1, value), index_2) = (index_1 = index_2 ? value : read(store, index_2))$;
- $rewriteRedundantWrite1$: $value = read(store, index) \Rightarrow write(store, index, value) = store$;
- $rewriteRedundantWrite2$: $write(write(store, index, v_1), index, v_2) = write(store, index, v_2)$;
- $interchangeIndices$: $write(write(store, index_1, v_1), index_2, v_2) = write(write(store, index_2, v_2), index_1, (index_1 = index_2 ? v_2, v_1))$.

ArithTheorem Fanno parte di questa categoria le regole per la costruzione di nuovi assiomi dalle espressioni passate in input:

- $uMinusToMult$: $-(e) = (-1) * e$;
- $minusToPlus$: $x - y = x + (-1) * y$;
- $constPredicate$: $e_0\ op\ e_1 \Leftrightarrow \top \mid \bot$, con $op \in \{<, \leq, >, \geq, =\}$;
- $rightMinusLeft$: $e_0\ op\ e_1 \Leftrightarrow e_1 - e_0$, con $op \in \{<, \leq, >, \geq, =\}$;
- $plusPredicate$: $x\ op\ y \Leftrightarrow x + z\ op\ y + z$, con $op \in \{<, \leq, >, \geq, =\}$;
- $multEqn$: $x = y \Leftrightarrow x * z = y * z$, con $z \neq 0$;
- $flipInequality$: $op_1 >|\geq op_2 \Leftrightarrow op_2 <|\leq op1$;
- $lessThanToLe$, se a e b sono interi allora $a < b \Rightarrow a \leq b - 1$ oppure $a + 1 \leq b$;

Conclusioni

Il presente lavoro di tesi è consistito nell'implementazione e formalizzazione di una nuova strategia di astrazione per Eureka, tool di verifica di programmi lineari con array.

Mentre la precedente strategia di astrazione utlizzata in Eureka rappresentava una trasformazione sintattica del programma cocreto in un programma astratto, che non prevedesse la presenza di array, la nuova strategia proposta è esclusivamente a livello semantico. Consiste, cioè, nella definizione di una nuova relazione di transizione astratta e quindi nella costruzione di una nuova interpretazione semantica su un dominio di variabili scalari astratte per il programma sottoposto a verifica.

La strategia proposta è stata implementata in linguaggio C++ ed introdotta all'interno del cilco CEGAR, su cui si basa il tool Eureka, attraverso l'adattamento dei moduli di verifica e raffinamento con cui essa interagisce.

Nel presente elaborato la strategia viene descritta e formalizzata, ne vengono illustrati i benefici che ha apportato al tool e viene presentata una dimostrazione della sua correttezza. Vengono illustrate inoltre le ulteriori modifiche apportate agli altri moduli del tool, in particolare al Model Checker, che è stato adattato affinchè eseguisse la verifica di raggiungibilità utilizzando una nuova interpretazione semantica per il programma da verificare. Vengono esibite dimostrazioni di correttezza e completezza dell'attività di verifica del Model Checker sulla semantica astratta del programma.

I benefici della nuova strategia proposta vanno ricercati innanzitutto in una maggiore efficienza. Infatti, grazie al nuovo meccanismo di astrazione è possibile eliminare un insieme di tracce spurie, anzichè una, all'interno di un'unica iterazione del ciclo CEGAR, con l'introduzione di sole due variabili all'interno del dominio astratto. Tale meccansimo permette dunque di ridurre il numero di iterazioni CEGAR, ma soprattutto presenta notevoli benefici per la fase di verifca, la cui complessità dipende in maniera esponenziale dalla taglia del dominio di variabili astratte.

Anche in vista di una eventuale estensione del tool alla verifica di programmi con puntatori, la nuova strategia appare più adeguata a quella prece-

dentemente utilizzata. Infatti, l'introduzione di variabili astratte che modellano dinamicamente elementi di array rappresenta il primo passo verso una futura introduzione nel linguaggio dei puntatori, che rappresentano proprio accessi dinamici a locazioni di memoria. Se si interpreta l'intera memoria del programma come un grande array e i puntatori come indici dinamici che vi accedono, appare evidente che la teoria proposta nel presente lavoro supporta tale estensione.

Pertanto, eventuali sviluppi futuri potrebbero riguardare l'estensione del dominio di programmi che è possibile verificare con Eureka al dominio di programmi lineari con puntatori. Un'altra possibile direzione verso cui orientarsi per incrementare le prestazioni del tool consiste nell'implementazione di una nuova strategia di raffinamento più effeiciente che non preveda la ricostruzione della prova di inconsistenza della traccia e quindi il doppio attraversamento dell'albero che la rappresenta.

Bibliografia

[ABM05a] A. Armando, M. Benerecetti, and J. Mantovani. Abstracting linear programs with array into linear program. Technical report, Università degli studi di Napoli Federico II ed Università degli studi di Genova, 2005.

[ABM05b] A. Armando, M. Benerecetti, and J. Mantovani. Model checking linear programs with array, 2005.

[ABM06] A. Armando, M. Benerecetti, and J. Mantovani. Abstraction refinement of linear programs with arrays. Technical report, Università degli studi di Napoli Federico II ed Università degli studi di Genova, 2006.

[ABM07] A. Armando, M. Benerecetti, and J. Mantovani. Counterexample-guided Abstraction Refinement for Data-intensive Sequential Programs. *ACM Transactions on Programming Languages and Systems*, pages 1–49, September 2007.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bag06] Roberto Bagnare. *Ppl: The parma polyhedra library*, 2006.

[Bar] Clark Barret. *CVC Lite User Manual*.

[BCM+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10 exp 20 states and beyond. In *LICS*, pages 428–439, 1990.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction for C programs. Technical report, Microsoft Research, 2001.

[BR00a] Thomas Ball and Sriram Rajamani. Boolean programs: A model and process for software analysis. Technical report, Microsoft Research, 2000.

[BR00b] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[CAM04] C.Castellini, A. Armando, and J. Mantovani. Software model checking using linear constraints, 2004.

[Cam06] Andrea Campagnuolo. Progettazione e sviluppo di un model checker per programmi lineari orientato alla verifica del software. Master's thesis, Università degli studi di Napoli Federico II, 2006.

[Cas06] Ciro Cascella. Studio e implementazione di tecniche di astrazione e raffinamento applicate alla verifica di programmi. Master's thesis, Università degli studi di Napoli Federico II, 2006.

[CGJ+00a] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[CGJ+00b] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and Helmuth Veith. Progress on state explosion problem in model checking, 2000.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. Model checking. *MIT Press*, 1999.

[Cla03] Edmund M. Clarke. Counterexample-guided abstraction refinement, 2003.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[HJMS03] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST, 2003.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, 1995.