Astrazione: dalla Teoria alla Realizzazione di un Abstract Proof Checker *

Roberto Sebastiani

Sommario

I processi di astrazione, nelle loro varie forme, vengono utilizzati quotidianamente nel ragionamento e nella soluzione di problemi afferenti a svariati campi dell'attività umana. Vista la sua indubbia utilità pratica, la nozione di astrazione è stata recentemente oggetto di formalizzazione in termini logico-matematici. In questo articolo discutiamo la realizzazione di un *abstract proof checker*, un sistema interattivo per la dimostrazione automatica di teoremi in grado di operare "per astrazione", secondo uno schema operativo che ricalca quello usato dagli esseri umani (matematici, progettisti, etc.). Tale sistema possiede una solida fondazione logica, in quanto si appoggia su una teoria formale dell'astrazione, generale e ricca di risultati formali. L'abstract proof checker qui discusso presenta significativi vantaggi rispetto a un proof checker tradizionale: l'uso del meccanismo di astrazione permette sia di ridurre sensibilmente il costo computazionale del processo globale di dimostrazione, sia di generare prove più leggibili, comprensibili, e quindi più facilmente ispezionabili da parte dell'utente che interagisce, "guidandolo", con il sistema.

L'abstract proof checker presentato si propone inoltre come un utile strumento per verificare le potenzialità applicative dell'astrazione in quei settori dell'intelligenza artificiale, quali la pianificazione e il ragionamento di senso comune, a cui la nozione di astrazione è connaturata.

1 Introduzione

Per *processo di astrazione* si intende generalmente una sequenza di operazioni che, data una rappresentazione di un determinato problema, ne generi una rappresentazione "semplificata" mediante l'eliminazione di dettagli che, almeno in prima istanza, non siano rilevanti alla soluzione del problema. La rappresentazione così ottenuta deve inoltre essere tale che la sua soluzione costituisca una falsariga da seguire per la soluzione del problema originale.

I processi di astrazione, nelle loro varie forme, vengono usati quotidianamente nel ragionamento e nella soluzione di problemi. Per esempio, il tipico procedimento seguito dai matematici nella dimostrazione di teoremi prevede prima di tutto l'individuazione

*Questo articolo riassume ed amplia i risultati del lavoro di tesi svolto dall'autore all'interno del Gruppo di Ragionamento Meccanizzato dell'IRST di Trento, sotto la guida di Fausto Giunchiglia. Si ringraziano tutti i componenti del gruppo per il costante aiuto, Toby Walsh per le numerose spiegazioni e, soprattutto, Fausto Giunchiglia per l'attenta e costante supervisione. Un ulteriore ringraziamento va anche alle persone che hanno letto questo articolo e suggerito miglioramenti, come Chiara Ghidini, Fabrizio Sebastiani e Paolo Traverso.

di una "traccia" della dimostrazione cercata; una volta trovata questa traccia, il matematico la elabora e sviluppa in tutti i suoi singoli passaggi e dettagli. Analogamente, un ingegnere elettronico, nel progettare un circuito digitale o un grosso programma software, traccia dapprima uno "schema a blocchi" del sistema da realizzare, rimandando a un secondo momento la progettazione delle singole parti componenti.

L'applicabilità della nozione di astrazione a questa molteplicità di problemi ha fatto sì che varie tecniche riconducibili a tale nozione siano state proposte e utilizzate in svariati campi dell'intelligenza artificiale, quali la pianificazione [Sac74], il ragionamento di senso comune [Hob85], la dimostrazione automatica di teoremi (o *theorem proving*) [Pla81, Ten87] e il ragionamento per analogia [Ble90].

Con la *Teoria dell'Astrazione* [GW89a, GW92b] Giunchiglia e Walsh introducono una definizione formale di astrazione che cattura la quasi totalità degli esempi di astrazione riportati in letteratura e risulta essere più generale di ogni precedente definizione (ad es. [Pla80, Pla81, Ten87]). Inoltre, quasi tutti i risultati teorici precedentemente riportati in letteratura vengono sussunti da risultati più generali di tale teoria [GW92b].

Nella Teoria dell'Astrazione viene introdotta la nozione di *abstract proof checking*[1], un processo di theorem proving fondato sull'astrazione. Tale processo si articola in una sequenza di fasi: *scelta di un'astrazione adatta allo scopo, applicazione di tale astrazione alla formula che si intende dimostrare (goal), dimostrazione della versione "astratta" (e quindi semplificata) del goal, uso della prova "astratta" come falsariga per la costruzione di una prova del goal originale.* La complessità della dimostrazione del goal astratto può risultare di notevolmente minore di quella del goal originale [Kor87, GW91b]. La riduzione della dimostrazione di un teorema alla dimostrazione di una sua versione astratta può quindi sia ridurre sensibilmente il costo computazionale del processo globale di dimostrazione, sia contribuire alla generazione di prove più comprensibili e quindi meglio ispezionabili.

[1]Nella letteratura del theorem proving l'espressione "proof checking" non è usata nel senso di "verifica di correttezza di una dimostrazione", ma nel senso di "dimostrazione interattiva di un teorema". A questo uso ci atterremo anche nel presente articolo.

In questo articolo presentiamo la realizzazione di un *abstract proof checker* (APC), come parte del sistema interattivo di deduzione automatica `GETFOL`[2]. Questo APC fornisce un ambiente del tutto generale per l'uso di processi di astrazione in applicazioni di theorem proving, utilizzando la Teoria dell'Astrazione come riferimento teorico.

L'astrazione da utilizzarsi in una particolare sessione di abstract proof checking è strettamente dipendente dal goal che si intende dimostrare, e non può quindi essere cablata nel sistema. Nell'APC che qui discutiamo essa è completamente definibile dall'utente: egli può infatti definire i sistemi formali in cui lavorare (linguaggio, assiomi, apparato deduttivo, ...), *definire* l'astrazione da utilizzare, farla *applicare* ai vari oggetti formali che partecipano alla rappresentazione del problema (simboli, assiomi, goal, ...) e successivamente controllare, in modo più o meno automatico, i vari passi della deduzione.

Restringendoci all'ambito del theorem proving, possiamo affermare che l'APC costituisce la prima realizzazione *effettiva* di un sistema di deduzione basato su una teoria dell'astrazione. Esso permette di costruire una dimostrazione per raffinamenti successivi, iniziando dalla prova di una rappresentazione semplificata del goal. Questo processo cattura effettivamente la tecnica

[2] `GETFOL` [GT91] è una re-implementazione ed estensione del sistema `FOL` [Wey77, Wey80].

usata usualmente nella matematica per costruire dimostrazioni [Pol45]. La possibilità concessa all'utente di definire da sè il processo di astrazione e la flessibilità del meccanismo di definizione conferiscono all'APC la generalità auspicata dalla teoria, consentendo di prevedere un suo utilizzo anche al di fuori del ristretto ambito del theorem proving: esso può infatti costituire un utile strumento per studiare le potenzialità dei meccanismi di astrazione nei diversi campi dell'intelligenza artificiale sopra citati, oltre a realizzare processi generalmente non concepiti come astrazioni [Mel87, Seb91].

L'idea di APC si inserisce all'interno di un approccio *interattivo* alla deduzione automatica: la definizione dell'astrazione e la scelta delle strategie nelle varie fasi di theorem proving sono lasciate all'utente. Utilizzando sequenzialmente una serie di comandi, egli può dare delle definizioni, applicare regole più o meno complesse e inferire risultati intermedi, effettuando una serie di scelte che "guidano" il sistema alla ricerca della soluzione cercata. In questo modo l'intervento umano "intelligente" permette di ridurre considerevolmente il numero di passi di deduzione, evitando, almeno parzialmente, i problemi di esplosione combinatoria tipici dei meccanismi di deduzione completamente automatici. Il livello di interattività del processo deduttivo diminuisce man mano che vengono definiti nuovi comandi e nuove regole più complessi e potenti: l'esistenza di librerie predefinite di (prototipi di) astrazioni, la possibilità di generare automaticamente il linguaggio del sistema formale "astratto" [Seb91], l'uso di decisori automatici per particolari sottoproblemi [GG88] e di altri strumenti ("definitions", "semantic attachment" ..., [GT91]) permettono di ridurre notevolmente e progressivamente l'intervento umano all'interno di tali deduzioni.

L'articolo è organizzato come segue. Allo scopo di rendere la trattazione "self-contained", nella Sezione 2 daremo una breve descrizione della nozione di APC. Approfondiremo inoltre alcuni aspetti di questa nozione che riteniamo particolarmente significativi, come la correttezza e completezza del processo di astrazione, l'indipendenza da ogni forma di controllo e le caratteristiche di interattività di questo approccio. Nella Sezione 3 presenteremo le varie funzionalità dell'APC che abbiamo realizzato, evidenziando le varie fasi del processo di astrazione che esso realizza. In particolare, evidenzieremo alcuni aspetti relativi alla generalità e comodità della dichiarazione di astrazioni e ai vari livelli possibili di interattività nelle varie fasi del processo. Infine, nella Sezione 4, presenteremo un semplice esempio, tratto dalla letteratura, da cui risulteranno evidenti le caratteristiche principali dell'intero processo di abstract proof checking. La Sezione 5 conclude, discutendo alcune interessanti linee di ricerca di cui il presente lavoro crea i presupposti.

2 Dedurre per astrazione

Per rappresentare formalmente la nozione di astrazione, introduciamo due definizioni [GW92b]:

Definizione 1 (Sistema Formale) *: Un sistema formale assiomatico Σ è una tripla $\langle \Lambda, \Omega, \Delta \rangle$, dove Λ è un linguaggio logico del primo ordine, Ω è un insieme di assiomi $\in \Lambda$ e Δ è un insieme di regole d'inferenza.*

In tutto l'articolo sottintendiamo l'aggettivo "*assiomatico*" quando parliamo di un sistema formale. Inoltre, rifacendoci alla terminologia del sistema `GETFOL`, useremo la parola *contesto* come sinonimo di *sistema formale.*

Definizione 2 (Astrazione) *: Un' astrazione $f : \Sigma_1 \Rightarrow \Sigma_2$ è una tripla $\langle \Sigma_1, \Sigma_2, f_\Lambda \rangle$, dove*

$$\Sigma_1 = \langle \Lambda_1, \Omega_1, \Delta_1 \rangle,\ \Sigma_2 = \langle \Lambda_2, \Omega_2, \Delta_2 \rangle$$

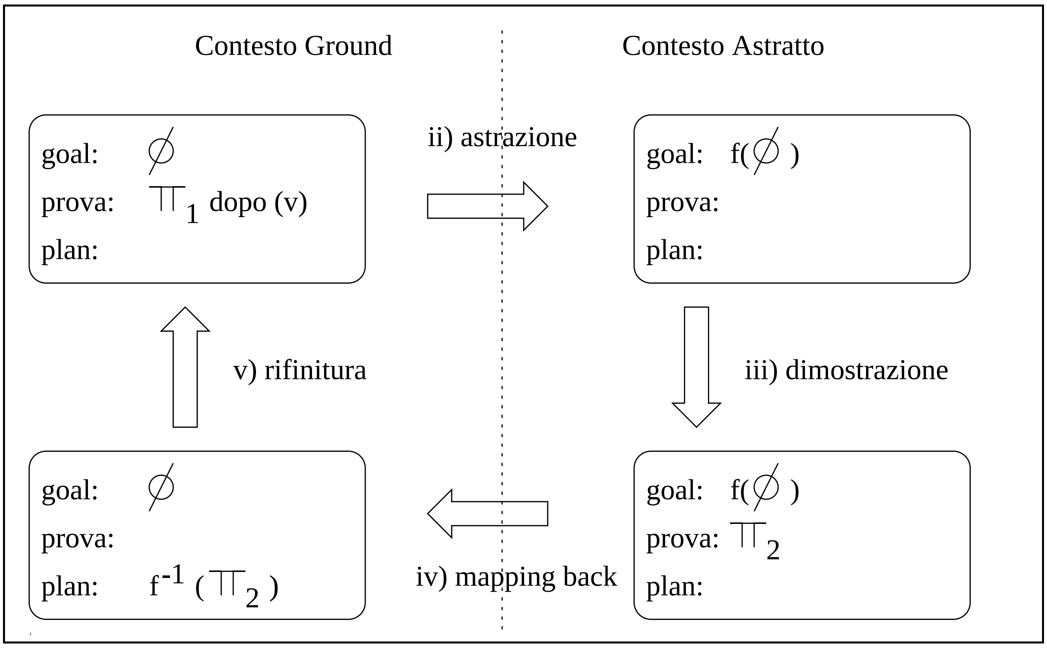

Figura 1: I principali passi di abstract proof checking.

sono due sistemi formali, detti rispettivamente "ground" ed "astratto", e

$$f_\Lambda : \Lambda_1 \rightarrow \Lambda_2$$

è una funzione computabile, suriettiva e totale tra i due linguaggi.

Il processo di "*abstract proof checking*" si sviluppa secondo una sequenza di fasi distinte (Figura 1):

(i) **Scelta e definizione di un'astrazione** $f : \Sigma_1 \Rightarrow \Sigma_2$ adatta allo scopo. Questa fase comprende la definizione, manuale o automatica, dei due contesti (linguaggio, assiomi,...) e della funzione f_Λ.

(ii) **Applicazione dell'astrazione** f al goal $\varphi \in \Lambda_1$. Questa operazione genera un nuovo goal $f_\Lambda(\varphi) \in \Lambda_2$. Allo stesso modo è possibile applicare f agli assiomi ground.

(iii) **Dimostrazione**: Si costruisce una prova Π_2 di $f_\Lambda(\varphi)$ nello spazio astratto Σ_2.

(iv) **Mapping back**: La prova Π_2 viene "*mappata all'indietro*", cioè viene generato in Σ_1 un albero di formule Π_1' tale che $f_\Lambda(\Pi_1') = \Pi_2$. Tale albero, detto "*proof plan*", costituisce uno *schema* della dimostrazione cercata. Poiché f_Λ non è in generale iniettiva, un proof plan potrà contenere delle formule parametriche.

(v) **Rifinitura**: Partendo dal proof plan Π_1', si costruisce, se possibile, una dimostrazione Π_1 di φ, istanziando opportunamente eventuali formule parametriche ed aggiungendo i pezzi di dimostrazione mancanti.

All'interno della fase (v), ogni singola operazione di "rifinitura" di un proof plan è consentita solo se dà origine ad inferenze logicamente corrette : la fase (v) può essere considerata conclusa solo se il proof plan rifinito risulta essere una prova della formula richiesta. In questo modo il processo di abstract proof checking è *corretto*, in quanto può originare solo dimostrazioni logicamente corrette di formule ground. Possiamo banalmente affermare che tale processo è anche *completo*, nel senso che, dato un qualsiasi teorema φ di Σ_1, esiste (almeno) un'astrazione tale che si possa dimostrare φ per abstract proof checking (si pensi all'uso dell'astrazione identità).

In generale un'astrazione viene scelta "*ad hoc*" per semplificare la dimostrazione del goal, e quindi può eliminare informazione indispensabile alla deduzione di molti altri teoremi. Una scelta oculata dell'astrazione da applicare è perciò determinante per la buona riuscita del processo di deduzione, in quanto tramite tale scelta l'utente "guida" il sistema nella creazione di una rappresentazione semplificata del teorema. Alcuni criteri di scelta euristici sono descritti in [Seb91].

Il processo di abstract proof checking è indipendente da ogni possibile forma di controllo che l'utente voglia introdurre all'interno delle singole fasi. Per esempio, la dimostrazione del teorema astratto potrebbe semplicemente essere svolta da un uomo con carta e penna, oppure da un sistema di deduzione completamente automatico (es. Resolution). Analogamente, la funzione f_Λ potrebbe essere descritta semplicemente a carta e penna, oppure realizzata mediante un opportuno algoritmo di controllo cablato, o ancora descritta mediante un qualche linguaggio di programmazione. Nel nostro caso, abbiamo formalizzato e realizzato l'APC all'interno di un sistema di deduzione *interattivo*, `GETFOL`. All'utente vengono forniti alcuni comandi per realizzare le varie fasi del processo di abstract proof checking, concedendogli una serie di scelte sul grado di interattività dell'azione. Ad esempio, egli può generare manualmente il contesto astratto, oppure affidarsi parzialmente o totalmente ad un meccanismo di generazione automatica; può definire personalmente l'algoritmo di controllo della funzione f_Λ, oppure utilizzare apposite librerie predefinite; può infine generare la prova "astratta" del teorema passo per passo, o utilizzare alcuni decisori automatici interni al sistema. Il progressivo sviluppo di comandi che realizzino funzionalità sempre più complesse contribuisce ad incrementare ulteriormente il livello di automazione del processo.

Allo stato di realizzazione attuale del sistema (aprile 1992) l'utente è in grado di utilizzare pienamente le prime tre fasi del processo di abstract proof checking: creazione dei contesti Σ_1 e Σ_2 e definizione della funzione di astrazione f_Λ (fase (i)), applicazione di f_Λ al goal (fase (ii)), dimostrazione della versione "astratta" del goal in una "normale" sessione di lavoro in `GETFOL` (fase (iii)). Tutti i comandi per effettuare il mapping back (fase (iv)) e per manipolare i proof plan (fase (v)) sono ancora in fase di sviluppo. Anche se alcuni risultati sono già stati ottenuti, questi strumenti sono tuttora a livello di prototipo e non verranno qui presentati. Per questo motivo la creazione della prova ground è provvisoriamente realizzata mediante un uso "manuale" di `GETFOL`. Questo significa che, attualmente, l'utente può generare una prova astratta ed utilizzarla come riferimento per trovare manualmente una dimostrazione del teorema originale (cfr. Sezione 4).

Al di là di ogni considerazione sugli sviluppi futuri del sistema, possiamo notare come non sia sempre necessario generare la prova ground: alcune astrazioni sono tali che la validità del goal astratto garantisca la validità del goal originale [GW92b]. Nella Sezione 3 presenteremo un esempio di tale situazione.

3 L'Abstract Proof Checker

Secondo la Definizione 2, un'astrazione $f : \Sigma_1 \Rightarrow \Sigma_2$ consiste di due sistemi formali e una generica funzione f_Λ computabile, suriettiva e totale tra linguaggi logici del primo ordine. Per poter realizzare l'abstract proof checker abbiamo ideato e definito un *linguaggio formale* con cui rappresentare f_Λ, tale da catturare la nozione di astrazione in tutta la sua generalità. A tal scopo abbiamo scelto un formalismo ricorsivo che permette, come vedremo, di separare la descrizione dell'algoritmo di controllo di f_Λ dalle trasformazioni che agiscono sui simboli dell'alfabeto.

Per comprendere questi concetti, analizziamo ora le singole fasi del processo di abstract proof checking.

3.1 Definizione dei contesti

`GETFOL` è un sistema *multicontestuale*: l'utente può creare diversi sistemi formali autonomi, detti *contesti*, ciascuno dei quali dotato di un proprio linguaggio e di propri assiomi. Esiste in ogni istante un *contesto corrente*: tutte le operazioni effettuate dall'utente andranno ad interessare solo tale contesto, e nessun oggetto ad esso esterno (simboli alfabetici, assiomi, ecc.) potrà essere utilizzato o modificato direttamente.

Supponiamo, ad esempio, che l'utente voglia creare un sistema formale di nome `C0` per rappresentare un circuito elettronico digitale di tipo "Full-Adder" (somma con riporto di 2 bit e precedente riporto), il quale utilizza a sua volta due sotto-circuiti di tipo "Half-Adder" (somma semplice di due bit con riporto), come in Figura 2. Per prima cosa dovrà generare il linguaggio del contesto `C0`: dichiarerà quindi alcuni simboli costanti di funzione ("`And`", "`Or`", "`Not`") che rappresentano le principali porte logiche, due predicati "`HalfAdder`" e "`FullAdder`" che descrivono i due circuiti digitali, ed alcune variabili. Dovrà poi definire un insieme di assiomi [3] che descrivano il comporta-

[3] Per brevità omettiamo la descrizione assiomatica delle funzioni `And`, `Or` e `Not`, e del fatto che ammettiamo nel dominio unicamente i valori booleani 0 e 1.

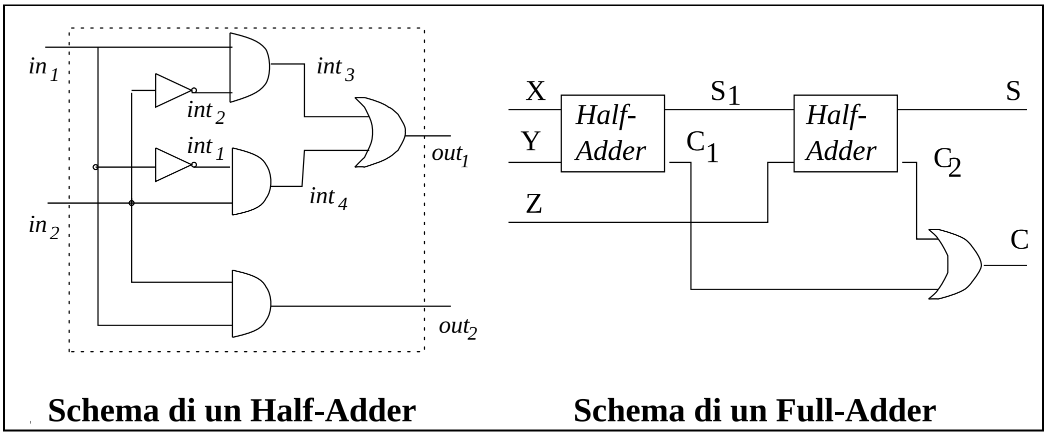

Figura 2: Rappresentazione "astratta" di un circuito Full-Adder.

mento di tali funzioni e predicati [4]:

```
C0> DECLARE FUNCONST And Or 2;
And is a Funconst
Or is a Funconst

C0> DECLARE FUNCONST Not 1;
Not is a Funconst

C0> DECLARE PREDCONST HalfAdder
                       FullAdder 8;
HalfAdder is a Predconst
FullAdder is a Predconst

C0> DECLARE INDVAR
x y z c1 c2 c s1 s int1 int2 int3
          int4 int5 int6 int7 int8;
x is an Indvar
y is an Indvar
....
int8 is an Indvar
```

[4] In tutta la trattazione descriviamo la chiamata ai comandi di GETFOL usando il font ttype, con caratteri maiuscoli per le parole chiave. Ogni comando di GETFOL inizia dopo il prompt, dato dal nome del contesto corrente seguito da un ">", e termina con un ";". Ad esso segue la risposta del sistema, che in alcuni casi ometteremo.

```
C0> AXIOM half_adder : HalfAdder(in1,
   in2,out1,out2,int1,int2,int3,int4)
   iff (out1 = Or(int3,int4) and
        int3 = And(in1,int2) and
        int4 = And(in2,int1) and
        int2 = Not(in2)      and
        int1 = Not(in1)      and
        out2 = And(in1,in2)  );

C0> AXIOM full_adder : FullAdder(x,
              y,z,s,c,s1,c1,c2) iff
(exists int1 int2 int3 int4.
 HalfAdder(x,y,s1,c1,int1,int2,int3,
                                int4)
and exists int5 int6 int7 int8.
 HalfAdder(s1,z,s,c2,int5,int6,int7,
                                int8)
and c = Or(c1,c2));
....
```

A questo punto l'utente ha generato il sistema formale ground Σ_1. Ora potrebbe creare il contesto astratto in modo analogo, generando cioè un nuovo contesto vuoto C1, dichiarando una serie di simboli e definendo nuovi assiomi. Nella Sezione 3.3 illustreremo un metodo alternativo molto più rapido, completamente automatico.

3.2 Definizione di un'astrazione

Scopo di questa sotto-sezione è quello di illustrare il procedimento di definizione di astrazioni, ed in particolare di descrivere i costrutti sintattici mediante i quali l'utente può descrivere una funzione f_Λ.

Nella Figura 2 viene rappresentato graficamente un processo di astrazione molto usato in elettronica: nella descrizione di un circuito Full-Adder ogni occorrenza del sotto-circuito di sinistra viene rappresentato da una scatola opaca, di cui sono note solamente le relazioni ingresso-uscita. In questo modo viene eliminata l'informazione relativa alla rappresentazione interna del sotto-circuito Half-Adder, descritta nella parte sinistra della figura. È possibile formalizzare questo processo [Mel87] definendo una particolare astrazione (che chiameremo `circuiti`) che trasformi ogni occorrenza della generica formula "$\exists\ \underline{int}\ .\ HalfAdder(\underline{in}, \underline{out}, \underline{int})$" [5] nella formula atomica: "$BlackBox(\underline{in}, \underline{out})$", che utilizza un predicato di arietà minore. Quest'ultimo non contiene il vettore degli stati interni $\underline{int}$ dell'Half-Adder, e quindi verrà descritto da un assioma più semplice, come vedremo più avanti. L'astrazione `circuiti` può essere definita, per esteso, nel seguente modo:

```
C0> ABSTRACTION DECLARE circuiti
GROUND C0
ABSTRACT C1
FLAMBDA
 MAPS
  INDS
  SENTS
  FUNS
  PREDS:
   HalfAdder -> BlackBox ARITY 4
 RULES
  ABS(exists $int1 $int2 $int3 $int4.
  HalfAdder($in1,$in2,$out1,$out2,
          $int1,$int2,$int3,$int4))
 := MAP[HalfAdder](ABS($in1),
   ABS($in2),ABS($out1),ABS($out2))
 ABS(forall $x. $A) :=
             forall MAP[$x].ABS($A)
 ....
 ABS($A imp $B) := ABS($A) imp
                              ABS($B)
 ABS(not $A)  := not ABS($A)
 ABS($P1($t)) := MAP[$P1](ABS($t))
 ABS($P2($t1,$t2)) := MAP[$P2]
                (ABS($t1),ABS($t2))
 ....
 ABS($t1=$t2) := ABS($t1)=ABS($t2)
 ABS($f1($t ) := MAP[$f1](ABS($t))
 ABS($x)      := MAP[$x];
```

[5] Per migliorare la leggibilità le sequenze di variabili del tipo $int_1, \ldots, int_4$ sono state rappresentate mediante un unico vettore di variabili $\underline{int}$, scritto in caratteri sottolineati.

Nella dichiarazione vengono inizialmente riportati i nomi dei due contesti, `C0` e `C1`, seguiti dalla descrizione di f_Λ. Quest'ultima è divisa nettamente in due parti: la descrizione delle mappe sui simboli dell'alfabeto (`MAPS`) e quella dell'algoritmo di controllo (`RULES`).

L'**algoritmo di controllo** della funzione f_Λ viene descritto mediante una lista di *regole ricorsive*, scritte in un linguaggio completamente dichiarativo, affine a quello della logica dei predicati del primo ordine. Ad esempio, la terza regola:

```
ABS($A and $B) := ABS($A) and ABS($B)
```

è facilmente interpretabile come:

$$f_\Lambda(A\ \wedge\ B)\ =\ f_\Lambda(A) \wedge f_\Lambda(B), \qquad (1)$$

mentre:

```
ABS( $P1($t) ) := MAP[$P1] (ABS($t))
```

ha il significato di:

$$f_\Lambda(\ P_1(t)\ )\ =\ [f_\Lambda(P_1)]\ \ (\ f_\Lambda(t)\ ). \qquad (2)$$

"`MAP[$P1]`" effettua una chiamata alle mappe tra predicati. In generale, gli operatori "`ABS`" effettuano una chiamata ricorsiva alle regole , mentre i "`MAP`" richiamano le mappe dei simboli (`MAPS`). Il linguaggio fa inoltre uso di una serie di meta-parametri cablati "`$`": "`$A`", "`$B`" sono istanziabili a intere formule, "`$t`" a termini, "`$P`" e "`$f`" rispettivamente a predicati e funzioni, "`$x`", "`$y`" a variabili individuali, ecc.

Per definire l'astrazione `circuiti`, è necessario aggiungere una regola del tipo:

$$f_\Lambda(\exists \underline{int}.HalfAdder(\underline{in}, \underline{out}, \underline{int})\ ) \ = \ [f_\Lambda(HalfAdder)]\ (f_\Lambda(\underline{in}),\ f_\Lambda(\underline{out})) \quad (3)$$

la quale contiene un esplicito riferimento al predicato ground `HalfAdder` (regola dipendente dal contesto) e rappresenta quindi un'eccezione alle regole descritte dall'algoritmo.

Le **mappe sui simboli** dell'alfabeto vengono descritte mediante una sequenza di coppie "*simbolo ground* $\longrightarrow$ *simbolo astratto*", raggruppate secondo il tipo sintattico del simbolo (individui, proposizioni, funzioni, predicati). La coppia "`HalfAdder -> BlackBox ARITY 4`" significa che ogni occorrenza del predicato "`HalfAdder`", di arietà 8, viene astratta in un occorrenza del predicato "`BlackBox`", di arietà 4. Essa viene esplicitamente chiamata dalla parte destra "`MAP[HalfAdder](...)`" della prima regola della definizione. Per default, tutti i simboli che non occorrono esplicitamente nelle `MAPS` vengono astratti in una copia di sè stessi. Nel nostro caso, l'unico simbolo che subisce una trasformazione risulta essere il predicato `HalfAdder`.

La dichiarazione *diretta* di un'astrazione è una operazione estremamente lunga e noiosa. In genere, è preferibile definire un'astrazione in modo *indiretto*, istanziando un opportuno *prototipo di astrazione*. Ogni prototipo è costituito da una lista di regole predefinite , come la (1) e la (2), dove non compaiono simboli propri di un particolare contesto. I prototipi rappresentano quindi opportuni algoritmi di controllo predefiniti. Per esempio, l'astrazione `circuiti` può essere definita in modo molto più semplice istanziando un particolare prototipo presente in libreria, chiamato `logic_pres`, il cui algoritmo mantiene inalterata la struttura logica di ogni espressione, limitandosi ad astrarre i simboli dell'alfabeto:

```
C0> ABSTRACTION INSTANTIATE
          logic_pres TO circuiti BY
GROUND C0
ABSTRACT C1
FLAMBDA
 MAPS
  INDS
  SENTS
  FUNS
  PREDS:
   HalfAdder -> BlackBox ARITY 4
 NEWRULES
  ABS(exists $int1 $int2 $int3 $int4.
   HalfAdder($in1,$in2,$out1,$out2,
       $int1,$int2,$int3,$int4)) :=
  MAP[HalfAdder](ABS($in1),ABS($in2),
             ABS($out1),ABS($out2));
```

Nella definizione indiretta è sufficiente elencare le "`MAPS`" e aggiungere le eventuali regole "eccezione" all'algoritmo di controllo richiamato dal prototipo: nel nostro caso la regola (3). Un prototipo di astrazione descrive un algoritmo di controllo per f_Λ in modo indipendente da ogni contesto, e può quindi essere usato per realizzare una gran quantità di astrazioni differenti. Per esempio, il solo `logic_pres` può generare funzioni che collassino gruppi di costanti in una unica, o che astraggano gruppi di predicati simili in uno dal significato più generale ("maestro" e "professore" in "insegnante"), ecc.. Esso quindi cattura, da solo, tutte le astrazioni che si limitano a collassare simboli (ad es. [Hob85, Imi87, Ten87]). La possibilità di definire nuovi prototipi [Seb91] consente inoltre all'utente di inventare nuovi l'algoritmi di controllo per la funzione f_Λ.

3.3 Applicazione di un'astrazione e dimostrazione astratta

Generare manualmente il contesto astratto Σ_2 potrebbe essere operativamente molto oneroso. Sarebbe necessario infatti dichiara-re tutti i simboli dell'alfabeto $\mathcal{A}_2$ e generare i nuovi assiomi, come si è fatto per Σ_1. Il sistema permette all'utente di creare $\mathcal{A}_2$ in modo automatico , utilizzando tutte le informazioni contenute in f_Λ(si noti infatti che $\mathcal{A}_2 = f_\Lambda(\mathcal{A}_1)$). Egli può generare il contesto vuoto C1 ed utilizzare un primo comando di applicazione delle astrazioni, ABSTRACT ALPHABET:

```
C0> MAKECONTEXT C1;
You have created the context: C1

C0> ABSTRACT ALPHABET BY circuiti;
I am switching from the current
                         context to: C1
x is an Indvar.
y is an Indvar.
....
int8 is an Indvar.
And is a Funconst.
Or is a Funconst.
Not is a Funconst.
BlackBox is a Predconst.
FullAdder is a Predconst.
I am switching from the current
                         context to: C0
```

Il comando ABSTRACT ALPHABET "salta" nel contesto C1 e dichiara una serie di nuovi simboli (variabili individuali, funzioni, predicati) all'interno di tale contesto. Si noti la presenza del predicato BlackBox di arietà 4 al posto di HalfAdder. Allo stesso modo, è possibile generare automaticamente parte (o tutti) degli assiomi del contesto astratto ($\Omega_2 = f_\Lambda(\Omega_1)$). Nel nostro caso, si può applicare l'astrazione circuiti all'assioma che descrive il predicato FullAdder e generare poi manualmente un nuovo assioma che descriva il predicato BlackBox, cioè le specifiche ingresso-uscita del circuito Half-Adder[6].

```
C0> ABSTRACT AXIOM full_adder BY
     circuiti TO simpler_full_adder;
I am switching from the current
                         context to: C1
simpler_full_adder :
FullAdder(x,y,z,s,c,s1,c1,c2) iff
((BlackBox(x,y,s1,c1)  and
  BlackBox(s1,z,s,c2)) and
 (c = Or(c1,c2)))
is a new axiom in 'C1'.

C1> AXIOM black_box :
BlackBox(in1,in2,out1,out2) iff
( (out1 = 0 iff in1 = in2) and
  out2 = And(in1,in2) );
```

La fase (i) del processo si può dire conclusa. È stata generata una rappresentazione molto semplificata del problema, che non tiene conto degli stati interni <u>*int*</u> di ogni sotto-circuito "Half-Adder": gli assiomi simpler_full_adder e black_box risultano essere molto più semplici dei corrispondenti assiomi ground, full_adder e half_adder, descritti precedentemente.

Supponiamo ora che l'utente voglia trovare una dimostrazione del teorema "FullAdder(1,1,1,1,1,0,1,0)", che significa "se tutti tre gli ingressi dell Full-Adder sono a livello "1", anche le sue due uscite lo sono, per opportuni valori dei suoi stati interni" (Figura 2). Egli dovrà tornare nel contesto ground e applicare l'astrazione circuiti a tale formula (fase (ii)):

```
C1> SWITCHCONTEXT C0;
You are now using context: C0

C0> ABSTRACT GOAL
FullAdder(1,1,1,1,1,0,1,0) BY circuiti;
I am switching from the current
                         context to: C1
FullAdder(1,1,1,1,1,0,1,0)
is the mapped goal from 'C0'.
```

In questo particolare caso, la formula astratta risulta essere uguale a quella ground

[6] Come in precedenza, l'astrazione degli assiomi che descrivono i valori booleani "0" ed "1" e le funzioni "And", "Or" e "Not" non viene riportata per brevità; essa genera assiomi assolutamente identici ai corrisponenti assiomi ground.

($f_\Lambda(\varphi) = \varphi$). Dimostrare il teorema in C1 (fase (iii)) equivale ora a verificare la semplice formula:

$$(0 = 0 \leftrightarrow 1 = 1) \ \wedge \ (1 = And(1,1)) \wedge$$
$$(1 = 0 \leftrightarrow 0 = 1) \ \wedge \ (0 = And(0,1)) \wedge$$
$$(1 = Or(1,0)).$$

Tale operazione risulta essere piuttosto semplice in GETFOL. In particolare, il decisore interno TAUTEQ [GG88] è in grado di verificarla in un unico passaggio. Nel contesto C0 sarebbe stato necessario dimostrare una formula molto più complessa:

$$\begin{aligned}
&\exists\ int_1\ int_2\ int_3\ int_4\ .\\
&\qquad (0 = Or(int_3, int_4) \wedge\\
&\qquad int_3 = And(1, int_2) \wedge\\
&\qquad int_4 = And(1, int_1) \wedge\\
&\qquad int_2 = Not(1) \wedge\\
&\qquad int_1 = Not(1) \wedge\\
&\qquad 1 = And(1,1)) \wedge\\
&\exists\ int_1\ int_2\ int_3\ int_4\ .\\
&\qquad (1 = Or(int_3, int_4) \wedge\\
&\qquad int_3 = And(0, int_2) \wedge\\
&\qquad int_4 = And(1, int_1) \wedge\\
&\qquad int_2 = Not(1) \wedge\\
&\qquad int_1 = Not(0) \wedge\\
&\qquad 0 = And(0,1)) \wedge\\
&1 = Or(1,0).
\end{aligned}$$

Nessuno dei decisori di GETFOL è riuscito a verificare direttamente tale formula.

Come accennato nella Sezione 2, dato il particolare tipo di astrazione, in questo caso non è necessario dimostrare la formula ground (fasi (iv) e (v)), in quanto la prova del teorema astratto $f_\Lambda(\varphi)$ garantisce la validità del teorema originale φ [GW92b].

4 Un esempio completo

Descriviamo ora un intero processo di abstract proof checking servendoci di un semplice esempio [Ten87]. L'utente crea dapprima un contesto World, definendo due oggetti a e b, rispettivamente di tipo "bottiglia" e "damigiana", e dichiarando che tutte le bottiglie sono sollevabili e tutte le damigiane sono trascinabili:

```
WORLD> DECLARE INDVAR   x ;
x is an Indvar

WORLD> DECLARE INDCONST a b;
a is an Indconst
b is an Indconst

WORLD> DECLARE PREDCONST Bottiglia
Damigiana Trascinabile Sollevabile 1;
Bottiglia is a Predconst
.....
Sollevabile is a Predconst

WORLD> AXIOM bottiglia :
                      Bottiglia(a);
WORLD> AXIOM damigiana :
                      Damigiana(b);
WORLD> AXIOM sollevabile :
 forall x.
 (Bottiglia(x) imp Sollevabile(x));
WORLD> AXIOM trascinabile :
forall x.
(Damigiana(x) imp Trascinabile(x));
```

Supponiamo ora che l'utente voglia trovare una dimostrazione del teorema "$Sollevabile(a) \wedge Trascinabile(b)$". Egli può osservare che, scambiando le terne di simboli $\langle\ a, Bottiglia, Sollevabile\ \rangle$ e $\langle\ b, Damigiana, Trascinabile\ \rangle$, il problema rimane invariato. Questo fatto gli suggerisce di generare una rappresentazione semplificata del problema, all'interno di un contesto astratto AbsWorld, collassando le due terne in una unica, dal significato più generale: $\langle\ c, Recipiente, Spostabile\ \rangle$. Allo scopo definisce l'astrazione collapse:

```
WORLD> MAKECONTEXT AbsWorld;
You have created the empty context:
                           AbsWorld

WORLD> ABSTRACTION INSTANTIATE
         logic_pres TO collapse BY
 GROUND   World
 ABSTRACT AbsWorld
 FLAMBDA
  MAPS
```

```
   INDS :
    a -> c
    b -> c
   SENTS
   FUNS
   PREDS :
    bottiglia -> recipiente
    damigiana -> recipiente
    trascinabile -> spostabile
    sollevabile  -> spostabile;

WORLD> ABSTRACT ALPHABET BY
                          collapse;
I am switching from the current
               context to: AbsWorld
c is an Indconst.
....
Spostabile is a Predconst.
I am switching from the current
                  context to: World

WORLD> ABSTRACT AXIOM bottiglia BY
            collapse TO recipiente;
I am switching from the current
               context to: AbsWorld
recipiente : Recipiente(c)
is a new axiom in 'AbsWorld'.

ABSWORLD> SWITCHCONTEXT World;
You are now using context: World

WORLD> ABSTRACT AXIOM sollevabile
         BY collapse TO spostabile;
I am switching from the current
               context to: AbsWorld
spostabile : forall x.
  (Recipiente(x) imp Spostabile(x))
is  a new axiom in 'AbsWorld'.

ABSWORLD> SWITCHCONTEXT World;
You are now using context: World
```

La fase (i) può dirsi conclusa. A questo punto l'utente potrà applicare l'astrazione `collapse` al goal (fase (ii)), ottenendone una nuova rappresentazione nel contesto `AbsWorld`:

```
WORLD> ABSTRACT GOAL Sollevabile(a)
   and Trascinabile(b) BY collapse;
I am switching from the current
               context to: AbsWorld
Spostabile(c) and Spostabile(c)
is the mapped goal from 'World'.
```

Il nuovo goal, "$\mathit{Spostabile}(c) \wedge \mathit{Spostabile}(c)$", è facilmente dimostrabile utilizzando i nuovi assiomi `recipiente` e `spostabile`, come evidenziato in Figura 3. In `GETFOL`, tale prova si può costruire in tre passaggi (fase (iii)), all'interno del contesto astratto `AbsWorld`:

```
ABSWORLD> ALLE spostabile c;
1   Recipiente(c) imp Spostabile(c)

ABSWORLD> IMPE recipiente 1;
2   Spostabile(c)

ABSWORLD> ANDI 2 2;
3   Spostabile(c) and Spostabile(c)
```

A questo punto, l'utente utilizzerà la prova astratta come riferimento per costruire una dimostrazione del teorema originale. A questo proposito, potrà utilizzare due volte lo schema (proof plan):

$$\dfrac{Q(Y) \qquad \dfrac{\forall x.(Q(x) \supset P(x))}{Q(Y) \supset P(Y)}\,\forall\mathrm{E}}{P(Y)}\,\supset\mathrm{E} \quad ,$$

istanziando la terna dei meta-parametri $\langle Y, P, Q\rangle$ a $\langle a, \mathit{Sollevabile}, \mathit{Bottiglia}\rangle$ e $\langle b, \mathit{Trascinabile}, \mathit{Damigiana}\rangle$ (fasi (iv) e (v)). Otterrà quindi la prova ground di Figura 3. Non essendo ancora disponibile in `GETFOL` alcuno strumento automatico per il mapping back [GW91a, GW92c], tale tecnica dovrà essere applicata manualmente:

```
ABSWORLD> switchcontext World;
You are now using context: World

WORLD> ALLE sollevabile a;
```

$$\dfrac{\dfrac{\mathit{Recipiente}(c) \qquad \dfrac{\forall x.(\mathit{Recipiente}(x) \supset \mathit{Spostabile}(x))}{\mathit{Recipiente}(c) \supset \mathit{Spostabile}(c)}\,\forall\mathrm{E}}{\mathit{Spostabile}(c)}\,\supset\mathrm{E}}{\mathit{Spostabile}(c) \wedge \mathit{Spostabile}(c)}\,\wedge\mathrm{I}$$

$$\dfrac{\dfrac{\mathit{Bott.}(a) \qquad \dfrac{\forall x.(\mathit{Bott.}(x) \supset \mathit{Soll.}(x))}{\mathit{Bott.}(a) \supset \mathit{Soll.}(a)}\,\forall\mathrm{E}}{\mathit{Soll.}(a)}\,\supset\mathrm{E} \qquad \dfrac{\mathit{Dam.}(b) \qquad \dfrac{\forall x.(\mathit{Dam.}(x) \supset \mathit{Trasc.}(x))}{\mathit{Dam.}(b) \supset \mathit{Trasc.}(b)}\,\forall\mathrm{E}}{\mathit{Trasc.}(b)}\,\supset\mathrm{E}}{\mathit{Sollevabile}(a) \wedge \mathit{Trascinabile}(b)}\,\wedge\mathrm{I}$$

Figura 3: Confronto tra la prova astratta e la corrispondent prova ground.

```
1  Bottiglia(a) imp Sollevabile(a)

WORLD> IMPE bottiglia 1;
2  Sollevabile(a)

WORLD> ALLE trascinabile b;
3  Damigiana(b) imp Trascinabile(b)

WORLD> IMPE damigiana 3;
4  Trascinabile(b)

WORLD> ANDI 2 4;
5  Sollevabile(a) and Trascinabile(b)
```

Le coppie di istruzioni 1-2 e 3-4 ripetono fedelmente lo schema della dimostrazione astratta, utilizzando istanze diverse.

5 Conclusioni e lavoro futuro

In questo articolo abbiamo presentato la formalizzazione e realizzazione di un abstract proof checker, nella convinzione che esso presenti significativi vantaggi rispetto ai *theorem prover* “tradizionali” e costituisca un utile strumento per studiare le potenzialità applicative dell'astrazione in vari settori dell'intelligenza artificiale. In particolare, abbiamo ideato un linguaggio formale sufficientemente generale con cui l'utente può definire i processi di astrazione: questo fatto conferisce all'APC una notevole generalità.

Punto focale del sistema è la fusione delle nozioni di *astrazione* e di *proof checking*: da un lato, l'uso di astrazioni in theorem proving consente di ridurre sensibilmente il costo computazionale del processo globale di deduzione; dall'altro, l'*interattività* del sistema permette di sfruttare la guida “intelligente” dell'utente, evitando, almeno parzialmente, i problemi di esplosione combinatoria tipici dei meccanismi di deduzione completamente automatici.

Come prossima fase nello sviluppo dell'APC, cercheremo di ridurre progressivamente il livello di interattività nelle operazioni di mapping back e rifinitura (fasi (iv) e (v)), formalizzando e realizzando alcuni strumenti che introducano nuove forme di automatismo in tali operazioni. In particolare, vogliamo completare lo sviluppo dei comandi di mapping back e di un ambiente che consenta di manipolare proof plan (cfr. Sezione 2).

Dal punto di vista strettamente applicativo, vogliamo investigare le potenzialità del sistema, cercando di utilizzarlo sia nella soluzione di alcuni problemi presi dalla letteratura ("mutilated chess board", "problema dei quattro colori", ecc.), sia nella dimostrazione di teoremi complessi. La dimostrazione "abstract-checked" del primo teorema di Gödel, proposta informalmente da A.Bundy e ripresa da T.Walsh, rappresenta uno stimolante caso di studio.

Bibliografia

[Ble90] W.W. Bledsoe. A precondition prover for analogy. CS Dept. memo, 1990.

[GG88] F. Giunchiglia and E. Giunchiglia. Building complex derived inference rules: a decider for the class of prenex universal-existential formulas. In *Proc. 7th ECAI*, 1988.

[GT91] F. Giunchiglia and P. Traverso. `GETFOL` User Manual - `GETFOL` version 1. Manual 9109-09, IRST, Trento, Italy, 1991.

[GW89a] F. Giunchiglia and T. Walsh. Abstract Theorem Proving. In *Proc. 11th IJCAI*, 1989.

[GW89b] F. Giunchiglia and T. Walsh. Abstracting into inconsistent spaces (or the false proof problem). In *Proc. AI*IA 89*, 1989.

[GW91a] F. Giunchiglia and T. Walsh. Mapping back. Technical Report 9111-02, IRST, Trento, Italy, 1991.

[GW91b] F. Giunchiglia and T. Walsh. Using abstraction. In *Proc. of the 8th Conference of the Society for the Study of Artificial Intelligence and Simulation of Behaviour*, Leeds, UK, 1991.

[GW92a] F. Giunchiglia and T. Walsh. The inevitability of inconsistent abstract spaces. *To appear in: Journal of Automated Reasoning*, 1992.

[GW92b] F. Giunchiglia and T. Walsh. A Theory of Abstraction. *Artificial Intelligence*, 56(2-3):323–390, 1992.

[GW92c] F. Giunchiglia and T. Walsh. Tree subsumption: Reasoning with outlines. In *Proc. 10th ECAI*, pages 77–81, Vienna, Austria, 1992.

[Hob85] J.R. Hobbs. Granularity. In *Proc. 9th IJCAI*, pages 432–435, 1985.

[Imi87] T. Imielinski. Domain abstraction and limited reasoning. In *Proc. 10th IJCAI*, pages 997–1003, 1987.

[Kor87] R.E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88, 1987.

[Mel87] T.F. Melham. Abstraction mechanisms for hardware verification. Technical Report 106, University of Cambridge, Computer Laboratory, 1987.

[Pla80] D.A. Plaisted. Abstraction mappings in mechanical theorem proving. In *5th Conference on Automated Deduction*, pages 264–280. Proc. of the 5th Conference on Automated Deduction, 1980.

[Pla81] D.A. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16:47–108, 1981.

[Pol45] G. Polya. *How to Solve It.* Princeton University Press, Princeton, NJ, 1945.

[Sac74] E.D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[Seb91] R. Sebastiani. L'astrazione in un dimostratore automatico interattivo: definizione e realizzazione.

Technical Report 9112-24, IRST, Trento, Italy, 1991. Thesis.

[Ten87] J.D. Tenenberg. Preserving Consistency across Abstraction Mappings. In *Proc. 10th IJCAI*, pages 1011–1014, 1987.

[Wey77] R.W. Weyhrauch. A Users Manual for FOL. Technical Report STAN-CS-77-432, Computer Science Department, Stanford University, 1977.

[Wey80] R.W. Weyhrauch. Prolegomena to a Theory of Mechanized Formal Reasoning. *Artificial Intelligence*, 13(1):133–176, 1980.