MPROLOG, it allows much more flexibility of the hanldler, permitting the mark to be set everywhere in the proof tree. The best advantage of that is the realization of a clean mechanism for propagating exceptions. Infact, by means of a raise_exception statement inside an action_predicate and a mark statement somewhere in the proof tree, we achieved propagation in a very simple and explicit way.

References

[CoDu82a] Cocco, Dulli. A Mechanism for Exception Handling and its VerificationRules. Computer Languages, Vol.7, 1982

[CoDu82b] Cocco, Dulli. Costrutti per la gestione delle eccezioni: confronto tra Clu, Chill e Ada. Rivista di Informatica, vol.XII, n.3, 1982.

[GhJa82] Ghezzi, Jazajeri. Programming Language Concepts. Wiley & Sons, Inc., 1982.

[Good75] Goodenough. Exception Handling: Issues and a proposed Notation. C.A.C.M. vol. 18, n.12, 1975.

[Harp86] Harper. Introduction to Standard ML. ECS-LFCS-86-14, nov. 1986

[HMQM86] Harper, MacQueen, Milner. Standard ML. ECS-LFCS-86-2, march 1986.

[Kral87] Krall. Implementation of a high-speed Prolog Interpreter. ACM Proceedings of the SIGPLAN 87 on Interpreters and Interpretative Techniques, St. Paul Minnesota, 1987

[MProlog] MPROLOG Language Reference, Release 1.5, Logicware, Toronto, Nov.1984.

[Prolog1] Prolog-1 8086 Reference manual, Expert Systems Limited, Oxford, Dec.1983.

[Prolog2] Prolog-2, Technical Reference Manual, Expert Systems Limited, Oxford, 1985.

Applicazioni di un Linguaggio Logico + Funzionale di Ordine Superiore

P.G. Bosco, C. Cecchi, C. Moiso

CSELT

Centro Studi E Laboratori Telecomunicazioni

via Reiss Romoli 274 - 10148 Torino

Sommario

Le caratteristiche di IDEAL - un linguaggio integrato logico-funzionale di ordine superiore - sono evidenzate nell'articolo in due momenti: (a) s'illustra la programmazione effettiva di un *simulatore logico*, mettendo cosi` in risalto gli aspetti dei linguaggi funzionali (mancanti in quelli logici) che si rendono necessari nella programmazione dichiarativa; (b) si mostra la "invertibilita`" dei programmi (funzionali) IDEAL, per cui il simulatore logico e` immediatamente utilizzabile come un *fault-finder*, il che mette in luce un aspetto fondamentale dei linguaggi logici (mancante in quelli funzionali) che ancora e` di notevole interesse per la programazione dichiarativa.
IDEAL e` realizzato mediante compilazione in un linguaggio del I ordine, che e` Prolog stesso se si vuol ottenere una valutazione eager (call by value), oppure K-LEAF - un linguaggio logico+funzionale del I ordine - se si desidera ottenere una valutazione lazy (call by need). Il linguagio K-LEAF si traduce in codice WAM (Warren Abstract Machine) mediante una opportuna estensione del procedimento di compilazione e dell'insieme d'istruzioni della WAM, cosi` da implementare una regola di selezione dinamica.

1. Introduzione

La programmazione logica e quella funzionale sono i due stili piu` diffusi di programmazione dichiarativa ed e` tuttora in corso il dibattitito sui rispettivi pro e contro; una soluzione per superare questa discussione e` quella di combinare i due paradigmi e quindi sviluppare un linguaggio che contenga i loro aspetti positivi, eliminandone gli svantaggi.

Tra le caratteristiche peculiari dei due paradigmi, una delle piu` significative differenze tra i linguaggi funzionali e quelli basati sulle clausole di Horn e` la presenza nei primi di costrutti di ordine superiore, un potente strumento che puo` essere sfruttato nella cosiddetta programmazione *in the large*, nel sintetizzare programmi da specifiche, ecc. .

Inoltre, i linguaggi funzionali offrono una varieta` di altri utili concetti di programmazione (come i sistemi di tipi, le differenti strategie di riduzione, ecc.) che

possono essere vantaggiosamente incluse in un sistema integrato.

Sono stati proposti alcuni differenti approcci a questa integrazione. Da una parte i linguaggi logici+funzionali, cioe` i linguaggi logici con alcuni limitati aspetti funzionali: essendo essenzialmente basati sulla logica del prim'ordine con uguaglianza, spesso mancano degli aspetti di ordine superiore. Dall'altra parte i linguaggi funzionali+logici, cioe` linguaggi funzionali arricchiti con caratteristiche logiche, come la capacita` di risolvere equazioni o di invertire funzioni. [Bellia86] contiene una completa analisi delle proposte piu` interessanti.

Il nostro approccio, descritto in [Bellia87], si basa su una gerarchia di linguaggi, al vertice della quale si situa IDEAL [Bosco86], un *superlinguaggio* che combina lo stile funzionale di ordine superiore (basato, come i piu` recenti linguaggi funzionali, sul lambda-calcolo con una teoria di tipi polimorfi) con le clausole di Horn. Il secondo livello, che puo` essere cosiderato un linguaggio intermedio nel processo trasformazionale e` K-LEAF [Bosco87], un linguaggio semanticamente ben fondato basato su una logica di Horn con uguaglianza. Un terzo livello e` rappresentato da una estensione della ben nota Macchina Astratta di Warren (WAM [Warren83]).

Finora le presentazioni di tali linguaggi integrati si sono limitate all'enumerazione delle loro proprieta` astratte, senza fornire esempi significativi di programmazione che giustifichino in modo soddisfacente la necessita` dell' integrazione.

In questo articolo consideriamo un problema di programmazione di una certa difficolta`: la simulazione logica e la diagnosi di guasti in circuiti digitali, problema sottopostoci da ricercatori di quel campo d'applicazioni [Mehring87], con i quali abbiamo potuto discutere i risultati delle sperimentazioni condotte. L'obbiettivo e` mostrare l'adeguatezza del linguaggio L+F IDEAL a certe classi d'applicazioni, sia in termini di espressivita`, sia di efficienza, rispetto ai linguaggi di programmazione convenzionali.

Grazie alle caratteristiche di ordine superiore del linguaggio il simulatore risulta semplice da descrivere (e quindi da programmare) e facile da capire. Infatti i componenti logici possono essere specificati come funzioni tra *stream* di eventi temporali, i componenti base come istanze di schemi generali, istanziati alle caratteristiche della famiglia logica adottata, ed i circuiti complessi come composizioni di funzioni piu` elementari.

A differenza di un simulatore implementato con un linguaggio standard, nel nostro caso e` possibile la conversione in un algoritmo di ricerca per la diagnosi di guasti, semplicemente sfruttando le caratteristiche logiche del linguaggio, cioe` l'invertibilita` e la variabile logica.

Tale conversione e` realizzata come segue: ogni componente logico viene equipaggiato con parameri aggiuntivi che specificano se i propri sottocomponenti funzionano correttamente o no. A questo punto e` ancora possibile effettuare una semplice simulazione, richiedendo l'esecuzione del programma con i suddetti parametri inizializzati a "ok". La diagnosi dei guasti viene invece effettuata fornendo la situazione di input-output del circuito guasto e lasciando che il meccanismo di risoluzione istanzi i parametri aggiuntivi, permettendo cosi` l'identificazione delle sottocomponenti difettose. Una terza modalita` di esecuzione, cioe` fornendo l'output e l'identificazione di possibili componenti difettose e deducendo l'input del circuito, puo` costituire la base per la generazione di configurazioni di test.

L'articolo e` organizzato come segue: nella sezione 2 vengono presentate informalmente la sintassi e la semantica di IDEAL; la sezione 3 illustra la specifica (= l'implementazione) del simulatore logico; la sezione 4 introduce le estensioni atte a trasformare il simulatore in un algoritmo per la diagnosi di guasti e la sezione 5 da` una panoramica sulle fasi compilative ed esecutive del linguaggio IDEAL.

2. Il Linguaggio IDEAL

La componente funzionale di ordine superiore del linguaggio e` una versione di *lambda-calcolo polimorfo*, con tipi universalmente quantificati, nello stile di ML [Milner78], o MIRANDA [Turner85]; la componente logica e` essenzialmente il linguaggio del prim'ordine (Prolog-like) sottostante, cioe` K-LEAF.

L'integrazione della parte logica con quella funzionale e` ottenuta mediante definizioni di funzione che sono *teste* (heads) di clausole Horn, mediante termini funzionali che sono argomenti di predicati, ed infine goals che risultano essere equazioni da risolvere. Un programma e` percio` un insieme di definizioni condizionali di funzione, cioe` clausole della forma:

$$\text{fname}@P_1@...@P_n = \text{term } \{ :\text{- cond } \}.$$

dove $P_1, ..., P_n$ sono patterns, cioe` termini formati da variabili e costruttori, ed in particolare possono essere variabili, nel qual caso sono identici agli usuali parametri formali. Se una funzione *fname* ha una definizione per casi mediante molteplici clausole, viene imposta una *condizione di non ambiguita`* cioe` non deve accadere che due tuple di patterns

$$<P_{i,1},...,P_{i.n}> \text{ e } <P_{j,1},...,P_{j,n}>$$

corrispondenti a clausole diverse i, j, $i \neq j$, siano unificabili.

Una ulteriore condizione deve valere per i patterns, la *linearita` sinistra* , per cui nessuna variabile in un pattern puo` avere ocorrenze multiple. Le due condizioni enunciate sopra garantiscono la confluenza del sistema di riscrittura risultante dalle definizioni di funzione.

Il membro destro dell'uguaglianza, *term*, e` un'espressione di un lambda-calcolo, esteso sintatticamente mediante i costrutti *if-then-else* e *whererec*, *letrec* per la mutua ricorsione, piu` vari costruttori, discriminatori e selettori con le loro regole di riduzione. Inoltre, sono consentite le seguenti forme di lambda-astrazioni condizionali:

$$\lambda P_1.\ \text{term}_1\ \{:\text{- cond}_1\};$$
$$P_2.\ \text{term}_2\ \{:\text{- cond}_2\};$$
$$...$$
$$P_n.\ \text{term}_n\ \{:\text{- cond}_n\}.$$

(dove si assume che i patterns $P_1, ..., P_n$ siano mutuamente disgiunti, cioe` non unificabili).

Come menzionato in precedenza, i patterns sono formati da variabili e costruttori. Oltre a quelli predefiniti (per le liste, i numeri, le tuple, ecc.), possono essere introdotti nuovi costruttori tramite dichiarazioni di tipi polimorfi. I tipi di dato polimorfi definiti da utente sono introdotti mediante un costrutto simile a quello suggerito in [Turner85]:

typename($Typevar_1$,...,$Typevar_n$)::=

constructor$_1$($Typename_{1,1}$,...,$Typename_{1,n1}$);

.....

constructor$_m$($Typename_{m,1}$,...,$Typename_{m,nm}$).

che dichiara l'esistenza di un tipo *typename*, i cui oggetti sono strutture identificate da *constructor*$_1$, ...,*constructor*$_m$, che sono costruttori, ciascuno con gli appropriati tipi degli argomenti.

Il seguente esempio di programma funzionale, tratto da [Turner85] , fornisce un'introduzione alla sintassi del sottoinsieme funzionale di IDEAL. L'informazione di tipo di un'espressione, che e` inferita dal sistema, viene riporatata a seguito dell'espressione dopo il simbolo ":".

Esempio 1

```
foldr@Op@Z = g
    whererec  g@[] = Z,
              g@[A|X] = Op@(A, g@X):
                  :(α # β—>β)—>β—>(list(α)—>β).
```

Definizione alternativa di *foldr*

```
foldr@Op@Z = λ[] .    Z,
             [A|X]. Op@(A, foldr@Op@Z@X).
```

```
product = foldr@(*)@1.          :list(int)—>int
sum = foldr@(+)@0.              :list(int)—>int.
sum@[1,2,3]                     % termine da valutarsi
   :int      6                  % risultato
```

L'uso dei pattern nelle definizioni di funzioni e predicati costituisce un potente strumento di programmazione: le frequentissime definizioni per casi sulla struttura dell'input sono codificate mediante i pattern in modo conciso ed al tempo stesso con espressivita`; nei linguaggi funzionali convenzionali, tale codifica richiede pesanti annidamenti di costrutti if-then-else e di selettori di dati.

Diversamente da altri linguaggi, i patterns in IDEAL non costituiscono un costrutto peculiare aggiunto, piuttosto derivano dall'aver integrato un linguaggio logico con uno funzionale. In effetti, l'uso dei patterns in IDEAL non e` ristretto alla selezione/discriminazione dell'input (come accade in particolare quando un pattern viene confrontato con un parametro attuale); al contrario, i patterns in IDEAL vengono usati mediante *unificazione*, ragion per cui i ruoli di discriminazione/ selezione/ costruzione non sono fissi, ma dipendono dallo stato d'istanziazione dei dati con i quali avviene l'unificazione.

Una condizione puo` essere aggiunta sia nelle definizioni al top-level che nelle lambda-astrazioni condizionali, con l'effetto di restringere il *dominio d'applicazione* di una clausola funzionale. Il dominio d'applicazione di una definizione mediante piu`

clausole alternative e` definito come l'unione dei domini delle singole clausole (tali domini non hanno intersezioni grazie alla condizione di non ambiguita`). Il dominio d'applicazione di una clausola *P=term :- cond* , informalmente parlando, e` l'insieme di tutte le istanze del pattern *P* per cui la condizione *cond* e` verificata. Percio`, affinche` la riscrittura di un redex *F@Arg* possa aver luogo con una data clausola *F@P=term :- cond*, bisogna che l'argomento *Arg* si confronti con il pattern P con risultato positivo, ed i legami di variabile conseguenti soddisfino la condizione *cond*.

Una condizione e` una congiunzione di letterali (literals), cioe` di termini il cui tipo e` quello dei valori di verita` (ad esempio, un predicato applicato ai suoi argomenti ha tipo valore di verita`). Il linguaggio possiede un predicato d'uguaglianza = predefinito. Definizioni di predicato si danno con la stessa sintassi delle funzioni, eccetto che il campo *valore* della definizione e` omesso, essendo implicitamente noto. Un predicato e` definito al top-level percio` mediante una o piu` clausole della forma seguente:

predname@P_1@...@P_n { :- cond}.

dove le restrizioni di non ambiguita` e linearita` sinistra non vengono piu` imposte, dato che da un lato, l'intersezione delle heads non puo` introdurre non-confluenza nei predicati (il membro destro di tutte le riscritture e` identico, essendo il valore di verita` implicito) e, dall'altro, occorrenze multiple della stessa variabile possono essere eliminate mediante trasformazioni sintattiche (che in generale fanno perdere la non ambiguita`, il che non le rende applicabili alle funzioni, ma che per i predicati sono ammesse).

Analogamente alle funzioni, si possono definire predicati anche mediante lambda-astrazioni condizionali, dove il campo *term* e` pero` omesso:

```
λ P1.  {:- cond1 };
  P2.  {:- cond2 };
  ...
  Pn.  {:- condn}.
```

L'estensione degli aspetti di ordine superiore ai predicati rende possibili le definizioni di combinatori su predicati e le definizioni di predicato con *scoping* lessicale, che migliorano notevolmente le caratteristiche di modularita` ed efficacia espressiva dei programmi logici, come e` mostrato dal seguente esempio.

Esempio 2.

```
comb@P@(X,Y) :- P@(Z,X), P@(Z,Y)
   :(α # β—>truth-value) —>β # β —>truth-value.
```

un tipo α —> **truth-value** si puo` abbreviare con [α].

Il tipo dell'espressione precedente e` visualizzato in effetti come [α # β] —>[β # β] .

```
non-disjoint = comb@member
   whererec    member@(X,[X|L]).
               member@(X,[Y|L]) :- member@(X,L)
          :[list(α) # list(α)].

person ::= (a;b;c;d;...).                    ;dichiara il tipo person

brothers = comb@parent
   where  parent@(a,b),.
          parent@(c,d), ... :   [person #person].
```

dove *non-disjoint* e` un predicato che e` soddisfatto da due liste che hanno almeno un elemento in comune, mentre *brothers* ha successo quando gli argomenti hanno un *parent* in comune.

Il programma principale, o *goal*, e` un'applicazione di funzione ad argomenti, cioe`:

?- $f@t_1@...@t_n$ [= Result].

oppure una query logica (esistenzialmente quantificata) :

?- $p@t_1@...@t_m$.

o piu` in generale un'equazione tra lambda-termini:

?- t1 = t2.

Il prossimo esempio mostra la possibilita` di invocare la parte funzionale da un contesto logico: il predicato *member* "chiama" la funzione *map* , la cui applicazione occorre in uno degli argomenti del predicato.

Esempio 3.

```
map([],F) = []
map([X|L],F) = [F@X|map(L,F)]          :list(α) , (α—>β)—>list(β).

member@(X,map([1,2,3], λx.x+1))   ;termine da valutarsi

:int    2;
        3;
        4.
```

i risultati 2,3,4 sono valori "alternativi" che in un ambiente sequenziale sono ottenuti per backtracking.

La componente funzionale puo` invocare la parte logica mediante le definizioni condizionali di funzione che prevedono condizioni logiche nell'antecedente delle clausole.

Come considerazione finale sulla sintassi di IDEAL, precisiamo che l'applicazione funzionale (@) e` in generale un operatore n-ario, con $n \geq 2$, per cui la sintassi astratta di un termine di tipo applicazione e` $@(F, A_1, ..., A_m)$, $m \geq 1$. Quando m=1, il termine $@(F, A)$ puo` essere scritto come $F@A$. Quando $F$ e` una costante, cioe` un simbolo $f$ definito al top-level, l'applicazione $@(f, A_1, ..., A_m)$, puo` essere scritta $f(A_1, ..., A_n)$, come ad esempio nelle definizione di *map* all'esempio 3. Da queste regole segue sintattiche segue che $f@x$ e $f(x)$ sono scritture equivalenti nel caso di funzioni globali f. L'applicazione come operatore n-ario non esclude la possibilita` di applicare funzioni unarie ad n-uple di argomenti. Le due modalita` d'applicazione sono pero` distinte gia` a livello di analisi statica: una funzione f n-aria ha tipo polimorfo $\alpha_1,...,\alpha_n \rightarrow \beta$, mentre una funzione g unaria definita su n-uple ha il tipo $\alpha_1\#...\#\alpha_n \rightarrow \beta$. I due tipi polimorfi non unificano, il che riflette una precisa scelta semantica: gli argomenti di una funzione $\alpha_1, ..., \alpha_n \rightarrow \beta$, infatti, sono intesi costituire delle tuple "piatte", mentre quelli di una funzione $\alpha_1\# ...\# \alpha_n \rightarrow \beta$ sono tuple binarie annidate di profondita` almeno n. Ad esempio, il tipo $\alpha \# \beta \# \gamma$ e` un'istanza di $\alpha \# \beta$, e di fatto le due espressioni (a,b,c) e (a,B) unificano, con B=(b,c), mentre il tipo $\alpha, \beta, \gamma \rightarrow \delta$ non e` un'istanza di $\alpha, \beta \rightarrow \gamma$.

Questa distinzione e` stata introdotta per avere nel linguaggio IDEAL sia l'interpretazione logica (della Horn Clause Logic) dell'applicazione, che e` n-aria, sia quella del lambda-calcolo, dove l'applicazione e` binaria e funzioni n-arie con n >1 sono definite in temini di tuple annidate.

3. Un prototipo di Simulatore Logico scritto in IDEAL

Un simulatore logico e` uno strumento per analizzare il comportamento di circuiti logici. E` basato sulla riproduzione astratta di un'evoluzione dello stato del circuito: il valore continuo del voltaggio di un connettore e` discretizzato (in 0 e 1, o High, Low e Unknown,...) e le porte logiche sono modellate da funzioni sui domini discreti scelti. Il simulatore logico che e` stato sviluppato in IDEAL e` un simulatore *event-driven* , dove un nuovo stato del circuito e` computato solo quando si verifica un evento che puo` modificare quello precedente; inoltre il voltaggio e` stato per semplicita` discretizzato in due soli valori (0 e 1).

Il simulatore e` basato su funzioni operanti su *stream* (liste in generale infinite ) di eventi: un evento e` descritto da una coppia, la cui prima componente e` l'istante in cui l'evento si verifica, mentre la seconda rappresenta lo stato ($0 o $1). Uno stream descrive la storia di un connettore: lo stream d(t0,$1):(t1,$0):(t2,$1):..., dove ti<ti+1, denota un connettore che ha stato $1 da t0 a t1, stato $0 tra t1 e t2,...

Il circuito simulato e` descritto da una funzione IDEAL di tipo:

$$\underbrace{\text{stream(event)\#....\#stream(event)}}_{\text{m volte}} \rightarrow \underbrace{\text{stream(event)\#....\#stream(event)}}_{\text{n volte}}$$

dove *m* e *n* sono, rispettivamente, il numero di ingressi e di uscite del circuito. Il valore di una funzione in corrispondenza di una m-upla di stream rappresentanti le storie degli m connettori di input e` la n-upla di stream che denotano le storie dei connettori di output.

La costruzione di tale funzione e` basata sulla composizione, in conformita` alla topologia del circuito, delle funzioni che simulano il comportamento di una porta logica. Una porta logica e` simulata da una funzione IDEAL da stream di eventi a stream di eventi. La funzione corrispondente ad una porta logica *G* elabora un evento in input in due fasi: durante la prima si computa il nuovo stato del connettore di output di *G* (cioe` la seconda componente del nuovo evento inserito nella stream di output), secondo la funzione booleana associata alla porta logica, mentre nella seconda si valuta l'istante in cui il nuovo stato si verifica (cioe` la prima componente dell'evento), secondo il tempo di ritardo associato alla porta. Le caratteristiche di ordine superiore di IDEAL sono usate per parametrizzare la (funzione che rappresenta la) porta logica rispetto alla funzione booleana computata. Infatti, le componenti logiche fondamentali (come *andgate*, *orgate*,...) sono definite come istanze di schemi generali di porte logiche a due input, le quali, ricevendo in input una funzione booleana restituiscono una funzione tra stream (come si puo` notare dalla definizioni di *gate1* e *gate2*, riportate in seguito nel listing del programma).

Le seguenti definizioni di funzione corrispondono ai circuiti di *halfadder* e di *fulladder*.

```
sum_of((S,C)) = S.          carry_of((S,C)) = C.          % selettori di coppia

halfadder(X, Y) = let sum  = xorgate(X, Y),                % halfadder
                      carry = andgate(X, Y)
                  in (sum, carry).

fulladder(X, Y, C) = let aux1 = halfadder(X, Y),           % fulladder
                         aux2 = halfadder(sum_of(aux1), C),
                         carry = orgate(carry_of(aux1), carry_of(aux2))
                     in (sum_of(aux2), carry).
```

E` immediato notare la corrispondenza uno a uno tra la descrizione funzionale dei circuiti e le funzioni IDEAL usate per la loro simulazione (e la rappresentazione grafica dei circuiti, come ad esempio quelle in figura 1).

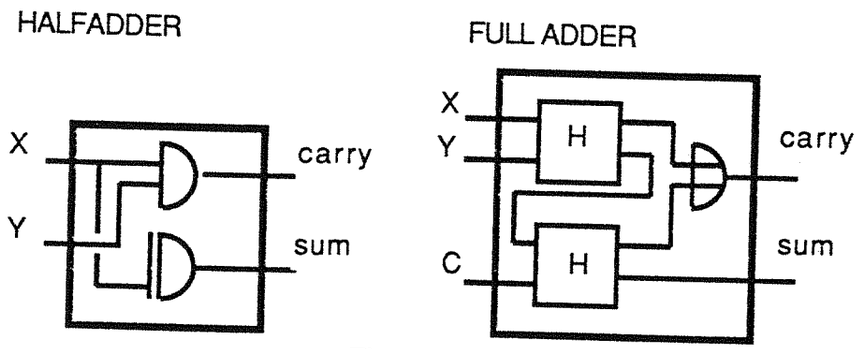

Figura 1

Inoltre, come gia` osservato da diversi ricercatori nel campo dei linguaggi per la descrizione di hardware (i quali hanno inserito nei loro linguaggi elementi di ordine superiore funzionale), la ricorsione puo` essere sfruttata per descrivere sinteticamente circuiti con una struttura regolare e quindi, nel nostro caso, anche per definire i programmi per la loro simulazione. La seguente funzione IDEAL, definita per induzione sul numero di bit, simula il comportamento un sommatore di n-bit (figura 2).

```
adder@0@([X0],[Y0]) =
                let h = halfadder(X0,Y0)
                in ([sum_of(h)],carry_of(h) ).
adder@s(N)@([Xn|X],[Yn|Y]) =
                let add = adder@N@(X,Y),
                    fulladd = fulladder(Xn,Yn,carry_of(add))
                in ([sum_of(fulladd) | sum_of(add)],carry_of(fulladd) ).

int —> list(stream(event)) # list(stream(event)) —> list(stream(event)) # stream(event)
```

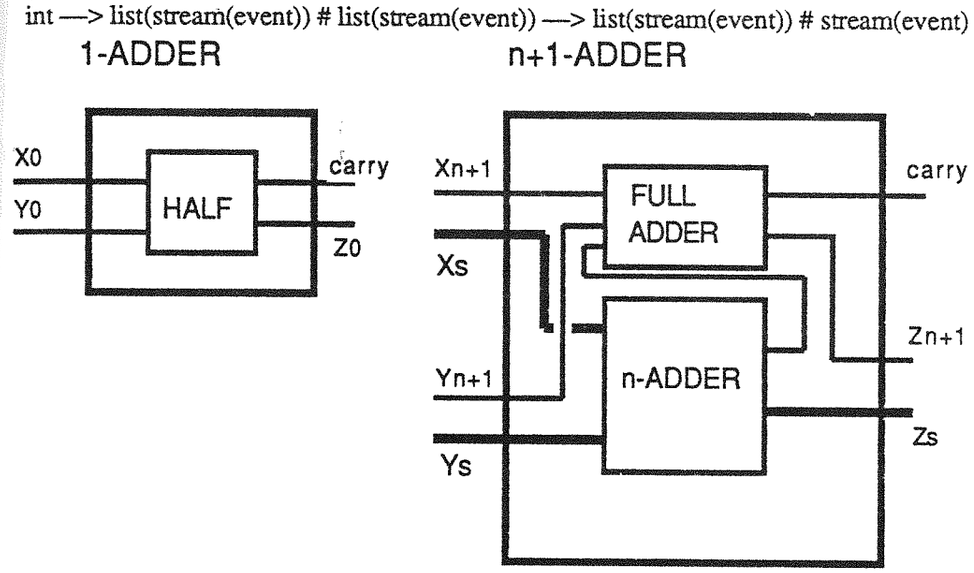

Figura 2

Le seguenti semplici *query* mostrano il comportamento del simulatore logico:

```
?- andgate(d(0,$0):d(2,$1):lnil,
           d(0,$0):d(3,$1):lnil) = R.

   R= d(0,$0):d(2,$0):d(4,$0):d(5,$1):lnil

?- halfadder(d(0,$0):lnil,
             d(0,$0):d(1,$1):lnil) = (Sum, Carry).

   Sum   = d(0,$0):d(1,$0):d(2,$1):lnil
   Carry = d(0, $0):d(2, $0):d(3,$0):lnil
```

```
?- fulladder( d(0,$0):lnil,
              d(0,$0):d(1,$1):lnil,
              d(0,$0):d(2,$1):lnil) = (Sum, Carry).

   Sum   = d(0, $0):d(2, $0):d(3,$0):lnil
   Carry = d(0, $0):d(2, $0):d(4,$0):d(5, $1):lnil
```

Per la simulazione di circuiti ciclici (come, ad esempio, i *flip-flop)*, l'evento speciale *stop* e` inserito al termine degli stream di input, per poter riconoscere la conclusione della simulazione.

Questa sezione e` completata dalla descrizione completa del simulatore logico proposto.

Definizioni di tipo:

```
stream(X) : := lnil  ;  X:stream(X).   % dichiarazione del tipo stream.
value ::=   $0 ; $1.                   % valori booleani.
event ::= d(int, value) ; stop.        % Eventi=coppie di valori bool. e di tempo.
```

Funzioni booleane:

```
bnot@$0 = $1.      bnot@$1 = $0.      % not
band@$0 = λx.$0.   band@$1 = λx.x.    % and
bor@$0 = λx.x.     bor@$1 = λx.$1.    % or
...
```

Le seguenti funzioni sono le componenti logiche di base corrispondenti alle porte logiche fondamentali, tramite le quali si compongono le funzioni per la simulazione; essi sono istanziazioni di porte generiche, *sgate1* and *sgate2*, rispettivamente, di uno o due input.

```
notgate(In) = sgate1(In, bnot, 2).
andgate(In1,In2) = sgate2(In1, In2, band, 2).
...
xorgate(In1,In2) = sgate2(In1, In2, bxor, 2).
```

Le porte generiche inizializzano la porta logica simulata con la corrispondente funzione booleana (*F*) e il tempo di ritardo (*D*); inoltre lo stato dei connettori di input e output e` inizializzato a *$0*.

```
sgate1(In, F, D) = d(0,$0):kdelay(gate1@F@In, D).
sgate2(In1, In2, F, D) = d(0, $0):kdelay(gate2@F@($0,$0)@(In1,In2), D).
```

La funzione *kdelay* aggiunge il tempo di ritardo *D* al valore di tempo indicato negli eventi di input.

```
kdelay(stop:In, D) = stop:lnil.
kdelay(lnil, D) = lnil.
kdelay(d(T,V):In, D) = d(T+D, V):kdelay(In, D).
```


Il comportamento di una generica porta con un solo input e` simulato da *gate1* il cui primo argomento e` la funzione booleana associata, mentre il secondo e` lo stream che denota la storia del connettore di input.

```
gate1@F@stop:L = stop:lnil.
gate1@F@lnil = lnil.
gate1@F@d(T,V):In = d(T,F@V):gate1@F@In.
```

Il comportamento di una generica porta con due conettori di input e` simulato da *gate2*, il cui primo argomento e` la funzione booleana computata, il secondo e` la coppia*(ov1,ov2)* degli stati di connettori di input, mentre l'ultimo e` la coppia di stream che denotano le storie dei due connettori di input.
Schematicamente, siano *d(t1,v1)* e *d(t2,v2)* i primi elementi dei due stream di input:

a) se *t1=t2* allora il nuovo stato del connettore di output e` *F@v1@v2* e lo stato di quelli di input diventa *v1* e *v2*, rispettivamente;
b) se *t1<t2* allora il nuovo stato del connettore di output e` *F@v1@ov2* mentre lo stato del primo input diventa *v1*;
c) il caso *t1>t2* e` simmetrico al precedente;

```
gate2@F@(OV1,OV2)@(stop:L, In2) = stop:lnil :- non_stop(In1).

gate2@F@(OV1,OV2)@(In1, stop:L) = stop:lnil.
gate2@F@(OV1,OV2)@(lnil, In2) = gate22(In2, F, OV1) :- non_end(In2).
                                                        %Fine di stream1.
gate2@F@(OV1,OV2)@(In1, lnil)  = gate21(In1, F, OV2) :- non_stop(In1).
                                                        %Fine di stream2.

gate2@F@(OV1,OV2)@(d(T1,V1):In1, d(T2,V2):In2) =
    if T1=T2 then d(T1, F@V1@V2):gate2@F@(V1,V2)@(In1, In2)      % (a)
    else if T1<T2 then                                           % (b)
        d(T1, F@V1@OV2):gate2@F@(V1, OV2)@(In1,d(T2,V2):In2)
    else d(T2,F@V2@OV1):gate2@F@(OV1,V2)@(d(T1,V1):In1,In2).     % (c)

non_end(d(T,V):L).

non_stop(lnil).
non_stop(L) :- non_end(L).
```

La funzione *gate21* (risp. *gate22*) simula una generica porta a due input con solo il primo (risp. il secondo) input attivo.

```
gate21(stop:L, F, OV) = stop:lnil.
gate21(lnil, F, OV) = lnil.
gate21(d(T,V):In, F, OV) = d(T, F@V@OV):gate21(In, F, OV).

gate22(stop:L, F, OV) = stop:lnil.
gate22(lnil, F, OV) = lnil.
gate22(d(T,V):In, F, OV) = d(T, F@OV@V):gate21(In, F, OV).
```

4. Conversione del Simulatore Logico in un Fault-finder

La rappresentazione di ciascun componente logico viene estesa con un parametro ulteriore d'ingresso che specifica lo stato *ok/faulty* (corretto/difettoso). Se lo stato del componente e` *ok*, si comporta in accordo alle specifiche fornite nella sezione precedente. Lo stato *faulty* e` definito come un termine strutturato, secondo la seguente dichiarazione di tipo:

```
fault ::=   ok ;                          % Ok significa : "nessun guasto".
            up ;                          % Corto circuito: l'uscita e` costantemente $1.
            down ;                        % Connessione interrotta: l'uscita e` sempre $0.
            andg(faults) ; ...            % Guasti nei componenti base:
      ...   xorg(faults) ;
            half(faults,faults)           % Guasti in un semi-addizionatore
            full(faults,faults,faults)    % Guasti in un addizionatore completo.
            n_add(list(faults),faults)    % Guasti in un n-addizionatore.
```

I circuiti complessi hanno descrittori di guasto strutturati, al fine di rappresentare assieme le descrizioni dei guasti dei sotto-componenti; il descrittore di guasto di un circuito e` essenzialmente l'albero di parsing della sua rappresentazione, dove le foglie riportano lo stato delle porte semplici.

Seguono le rappresentazioni IDEAL di componenti logici (eventualmente) guasti (cfr. fig.3).

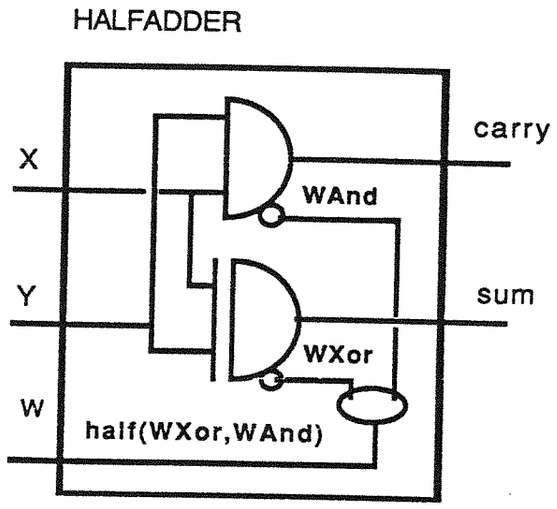

Figura 3. Un semi addizionatore (eventualmente) guasto.

```
xorgate@ok@(X,Y) =            sgate2(X, Y, bxor, 1),  % Come in precedenza.
xorgate@xorg(up)@(X,Y) =      sgate2(X, Y, λx.$1, 1),
xorgate@xorg(down)@(X,Y)=     sgate2(X, Y, λx.$0, 1):
        faults —> stream # stream —> stream.
```



```
halfadder@half(WXor,WAnd)@(X, Y) =
        ( xorgate@WXor@(X,Y),  andgate@Wand@(X,Y) ):

        faults —> stream # stream —> stream # stream.

fulladder@full(WH1, WH2, WOr)@(X,Y,C)=
        let half1  = halfadder@WH1@(X,Y),
            half2  = halfadder@WH2@(sum_of(half1), C),
            carry  = orgate@WOr@(carry_of(half1), carry_of(half2))
        in  ( sum_of(half2),  carry).

adder@0@n_add([], WH)@([X0],[Y0])=
        let h= halfadder@WH@(X0,Y0)  in
         ( [sum_of(h)], carry_of(h) ),

adder@s(N)@n_add([WFull | WRest], WH)@([Xn| Xs],[Yn| Ys])=
 let    nadd       = adder@N@n_add(WRest, WH)@(Xs, Ys),
        fulla      = fulladder@WFull@( Xn , Yn, carry_of(nadd))
 in      ( [sum_of(fulla) | sum_of(nadd)] ,
           carry_of(fulla) ):

int —>
     faults —> list(stream) #  list(stream) —> list(stream) # stream.
```

Il tipo dell'espressione *adder@n* e` un'istanza del seguente tipo polimorfo:

faults —> IN —> OUT

dove *IN*, *OUT* stanno per i tipi rispettivamente dell'input e dell'output di un n-addizionatore. Ogni rappresentazione di circuito *CR* ha un tipo che e` un'istanza di *faults —> IN —> OUT*, percio`, per ogni *CR*, la seguente e` una query IDEAL ben tipata:

?- CR@F@In=Out.

dove *F*, *In*, e *Out* sono espressioni di tipo appropriato. La simulazione di un componente e` ottenuta avendo *F* e *In* istanziati e *Out* non istanziato. La diagnosi di guasti si esegue lasciando *F* non istanziato e specificando invece *In* e *Out*. La generazione di test-pattern richiede invece *F* e *Out* specificati e *In* non istanziato. La tavola in fig. 4 mostra i vari modi d'istanziazione per gli argomenti della query di cui sopra, riportando per ognuno la sua interpretazione. Un'espressione istanziata (rispettivamente, non istanziata) e` annotata con il simbolo `+` (`?`).

Instantiation Modes				Interpretations
?-CR@Fault@Inp=Out				
	+	+	?	Simulation
	?	+	+	Fault-finding
	+	?	+	Test-pattern Generation

Figura 4. Vari modi di input/output per la Rappresentatzone di un Circuito.

La query seguente simula l'elaborazione di un addizionatore a 2 bit guasto, sugli input *[$0,$1]* e *[$1,$1]*. L'addizionatore ha un guasto in entrambi i sottocomponenti: la linea *carry* e` *down* sia nel semi addizionatore che nell'addizionatore completo (fig. 5).

```
?- ( [Z1, Z0], C) = adder@s(0)@fault@([x1, x0], [y1, y0])
                    where
        x1 = d(0,$0):lnil,  x0 = d(0,$1):lnil,
        y1 = d(0,$1):lnil,  y0 = d(0,$1):lnil,
        fault = n_add([full(half(xorg(ok),andg(ok)),
                            half(xorg(ok), andg(ok)),
                            orgate(down))],
                      half(xorg(ok),andg(down))).

C  = d(0,$0):d(2,$0:d(4,$0):d(6,$0):lnil,
Z1 = d(0,$0):d(2,$0):d(4,$1):lnil,
Z0 = d(0,$0):d(2,$0):lnil.
```

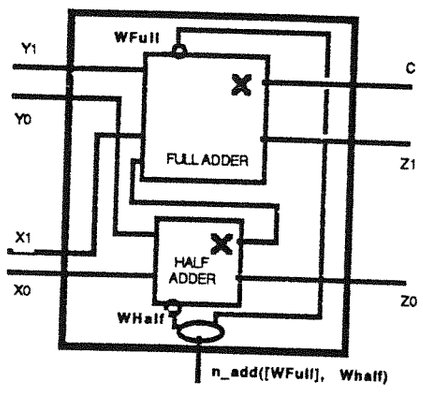

Figura 5. Un addizionatore a 2 bit guasto.

La query successiva determina i guasti in un addizionatore a 2 bit il cui comportamento osservato e` quello della precedente risoluzione. I valori rilevati alle uscite del circuito sono prese dal risultato della simulazione precedente; in un'applicazione realistica, invece, essi risulterebbero dai rilevamenti in un circuito effettivo.

```
?- adder@s(0)@What@ ([x1, x0], [y1, y0])= ( [z1, z0] , c)
                    where
        x1, x0, y1, y0 , z1, z0 , c : valori della query precedente.

What =  n_add([full( half(xorg(ok),andg(ok)),
                     half(xorg(ok),andg(ok)),
                     orgate(ok))],
              half(xorg(ok),andg(down))).
What =  n_add([full( half(xorg(ok),andg(ok)),
                     half(xorg(ok),andg(ok)),
                     orgate(down))],
              half(xorg(ok),andg(down))).
...
```

Le risposte successive non sono riportate. Per evitare l'esame di un gran numero di possibilita`, si puo` procedere ponendo una query di diagnosi con un descrittore di guasto parzialmente specificato. In effetti, osservando le prime risposte alla query precedente, si puo` ipotizzare che la porta *and* del semi addizionatore sia necessariamente guasta, poiche` i valori prodotti sono tutti istanze del seguente schema:

```
n_add([WFull],  half(xorg(Wx),andg(down))).
```

Per verificare la validita` dell'ipotesi, la query viene riformulata con il seguente descrittore di guasto parziale:

```
What = n_add([WFull],  half(xorg(Wx),andg(ok))
```

dove il descrittore della porta *and* e` stata posta a *ok*. Se la query fallisce, la porta *and* in questione e` necessariamente guasta quindi va rimpiazata, cosicche` l'analisi possa procedere alla ricerca di ulteriori guasti.

5. Stato attuale della implementazione di IDEAL e attivita` future

I programmi IDEAL sono trasformati in clausole di Horn: le caratteristiche di ordine superiore sono eliminate per mezzo di una variante del *lambda-lifting*: questa tecnica e` stata sviluppata per compilare programmi funzionali in un insieme di *supercombinatori* [Hughes83], i quali possono essere considerati regole di riscrittura del prim'ordine, facilmente trasformabili in clausole Prolog [Bosco86] o K-LEAF [Bellia87]. Il lambda-lifting e` stato esteso per trattare gli aspetti logici di IDEAL (pattern, variabili logiche e condizioni).

Sono state sviluppate due implementazioni di IDEAL: la prima (che trasforma IDEAL in Prolog e, quindi, esegue i programmi cosi` ottenuti) adotta una strategia eager (cioe` call-by-value) come default; il confronto di questa implementazione con alcuni

sistemi LISP compilati/interpretati e` soddisfacente. La seconda implementazione, che esegue computazioni lazy, e` ottenuta trasformando IDEAL in K-LEAF ed eseguendo i programmi risultanti secondo la strategia outermost di K-LEAF [Bosco87], dopo averli compilati in una estensione della WAM.

Infatti, si e` dovuto estendere la WAM, sviluppata per il Prolog, per trattare la natura dinamica della strategia di selezione adottata in K-LEAF. Questa seconda implementazione, piu` fedele alla semantica del linguaggio, e` in uno stato prototipale. Anche nel caso lazy si prevedono risultati soddisfacenti: stimiamo che, avendo a disposizione la versione finale dell'esecutore K-LEAF, la versione *invertibile* del simulatore possa elaborare circa 1000 eventi al secondo su un MicroVax, cioe` tre volte piu` lentamente rispetto ad un simulatore "industriale" *non-invertibile* (scritto in C) [Mehring87] eseguito sulla stessa macchina.

Grazie all'ortogonalita` tra le estensioni per la strategia di selezione outermost e quelle per realizzare il parallelismo OR, e` abbastanza immediato estendere il sistema Prolog OR-parallelo sviluppato in CSELT [Giandonato87] per sfruttare il parallelismo OR presente in K-LEAF. Inoltre abbiamo introdotto un modello di parallelismo AND ristretto (alla riduzione funzionale), cosi` da ottenere una combinazione trattabile tra parallelismo OR e quello AND, coerente con l'integrazione logico-funzionale.

6. Conclusioni

L'implementazione del simulatore logico sfrutta la maggior parte delle caratteristiche di IDEAL, un linguaggio logico + funzionale di ordine superiore.

Per quanto riguarda la componente funzionale, oltre alla notazione basata sulla composizione funzionale (la quale risulta spesso essere piu` naturale che la congiunzione di atomi dei linguaggi basati sulle clausole di Horn), i principali aspetti di questo paradigma sfruttati nell'esempio proposto sono:

- l'ordine superiore: la sua presenza, assieme ad un sistema di tipi polimorfi, favorisce la modularizzazione, che ha permesso la definizione di componenti generiche (come le funzioni *sgate1* e *sgate2* nel nostro caso);
- valutazione lazy: per la simulazione di circuiti con cicli, la lazy evaluation e` indispensabile: infatti, le funzioni IDEAL che simulano tali circuiti possono essere intese (secondo l'interpretazione a processi dei programmi logici e/o funzionali [Levi86]) come reti cicliche di processi, la cui esecuzione richede questa strategia di valutazione.

Gli aspetti logici di IDEAL, che praticamente non sono stati impiegati nell'implementazione del simulatore, diventano essenziali nella sua esecuzione in modo *invertito* cosi` da ottenere un *fault-finder* (o un generatore di *pattern*). Infatti, possiamo ottenere gratuitamente questi due *nuovi* algoritmi, grazie alla presenza in IDEAL delle seguenti caratteristiche logiche:

- le funzioni possono essere usate in modo invertito, grazie alla presenza di variabili logiche (che permettono di specificare input parzialmente istanziati) e all'uso dell'unificazione, al posto del *pattern-matching*, come meccanismo di attivazione di funzioni/predicati;
- le computazioni basate sulla ricerca di soluzioni, rese possibili dal carattere nondeterministico delle computazioni logiche.

L'esempio che abbiamo presentato illustra quindi l'efficacia espressiva dei linguaggi logico-funzionali. Oltre ad essere formalismi di specifica eseguibile e, quindi, adatti per la prototipazione, possono essere usati come linguaggi finali per alcune applicazioni. Infatti, la nostra esperienza nelle implementazioni di IDEAL ci porta ad affermare che questa classe di linguaggi puo` essere efficientemente eseguita, sia sequenzialemente che in parallelo, assumendo come punto di partenza l'ormai consolidata compilazione dei linguaggi logici, cioe` quella che ha per *target* la WAM (con alcune estensioni per trattare l'esecuzione degli aspetti non standard), e costruendo su questa, sempre tramite compilazione, macchine astratte piu` complesse, preservando l'efficienza della macchina di base.

Ringraziamenti

Questo lavoro e` stato parzialmente finanziato dalla CEE nell'ambito del Progetto ESPRIT n.415 "Parallel Architectures and Languages for A.I.P.: a VLSI-directed approach".

Riferimenti

[Bellia86] **M. Bellia and G. Levi**, The relation between logic and functional languages: A survey, Journal of Logic Programming 3 (1986),217-236.

[Bellia87] **M. Bellia, P.G. Bosco, E. Giovannetti, G. Levi, C. Moiso and C. Palamidessi**, A two-level approach to logic plus functional programming integration, Proc. PARLE Conference, LNCS 258 (Springer-Verlag, 1987), 374-393.

[Bosco86] **P.G. Bosco and E. Giovannetti**, IDEAL: An Ideal DEductive Applicative Language, Proc. 1986 Symp. on Logic Programming (IEEE Comp. Society Press, 1986), 89-94.

[Bosco87] **P.G. Bosco, E. Giovannetti, G. Levi, C. Moiso and C. Palamidessi**, A complete semantic characterization of K-LEAF, a logic language with partial functions, Proc. 1987 Symp. on Logic Programming (IEEE Comp. Society Press, 1987), 318-327.

[Giandonato87] **G. Giandonato and G. Sofi**, An OR-parallel language for a class of AI applications, Proc. GULP'88 (Roma, Maggio 1988).

[Hughes83] **J. Hughes**, The design and implementation of programming languages, PhD Thesis, University of Oxford (1983).

[Levi86] **G. Levi**, Logic programming: the foundations, the approach and the role of concurrency, in Current Trends in Concurrency, LNCS 224 (Springer-Verlag, 1986), 396-441.

[Mehring87] **P. Mehring and E. Aposporidis**, Multi-level simulator for VLSI: an

overview, Proc. PARLE Conference, LNCS 258 (Springer-Verlag, 1987), 446-460.

[Milner78] **R. Milner**, A theory of type polymorphism in programming, Journal of Computer and System Sciences 17, 3 (Dic. 1978), 348-375.

[Turner85] **D.A. Turner**, MIRANDA: a non-strict functional language with polymorphic types, Proc. 2nd Conf. on Functional Programming and Computer Architecture, LNCS 201 (Springer-Verlag, 1985), 1-16.

[Warren83] **D. H. D. Warren**, An Abstract Prolog Instruction Set, Technical Note 309, SRI International (Ott.1983).

Un sistema esperto con architettura a blackboard per la fusione dati

Antonio Brogi[3], Riccardo Filippi[1],Mauro Gaspari[2],Franco Turini[3]

1- Selenia S.p.A.,2- Delphi S.p.A.,3- Dipartimento di Informatica
Corso Italia, 40, 56100 Pisa, Italy. Tel. +39.50.510253. Telex 590291 DIPISA I

Sommario

In questo articolo viene presentato un prototipo di sistema esperto per la fusione dati basato su un' architettura a blackboard. Il problema della fusione dati e' quello di considerare insieme dati provenienti da sensori di tipo diverso tra loro o da altre sorgenti di informazione.

In particolare e' stata trattata la fusione dei dati provenienti da un sensore radar, uno infrarosso con informazioni note a priori. Il processo di fusione avviene su dati generati da un programma simulatore.

Le caratteristiche dell' architettura a blackboard hanno consentito di scomporre tutta la conoscenza di dominio in moduli cooperanti (sorgenti di conoscenza) e di mantenere separato questo tipo di conoscenza da quella di controllo. Nel processo di costruzione delle ipotesi e' stato usato il ragionamento approssimato; in particolare e' stata applicata la teoria di Dempster-Shafer. Il linguaggio utilizzato per l' implementazione e' il NIP "Edinburgh Prolog" e la macchina su cui e' stato sviluppato il sistema e' una SUN 3/180.