Lemma Generation for Horn Clauses Satisfiability

Emanuele De Angelis, University of Pescara, Italy

Fabio Fioravanti, University of Pescara, Italy

Maurizio Proietti, CNR-IASI, Rome, Italy

Alberto Pettorossi, University of Rome "Tor Vergata", Italy

VPT 2019, Genova, Italy April 2nd, 2019

Background

1. **Proofs of properties of programs**
   via satisfiability of Constrained Horn Clauses (CHCs):
   (i) translation of programs to CHCs, and
   (ii) check of satisfiability using CHC solvers (such as Eldarica, Z3, ...)

2. **If programs manipulate structured types such as lists and trees,**
   the CHC solvers are usually not so effective. Then,
   - either add induction principles [Reynolds-Kuncak '15, Unno et al. '17]
   - or derive by transformation equisatisfiable CHCs without structured types
     using the Elimination Algorithm [De Angelis et al. '18]

Background and Contribution

1. **Proofs of properties of programs**
   via satisfiability of Constrained Horn Clauses (CHCs):
   (i) translation of programs to CHCs and
   (ii) check of satisfiability using CHC solvers (such as Eldarica, Z3, ...)

2. **If programs manipulate structured types such as lists and trees,**
   the CHC solvers are usually not so effective. Then,
   - either add induction principles [Reynolds-Kuncak '15, Unno et al. '17]
   - or derive by transformation equisatisfiable CHCs without structured types
     using the Elimination Algorithm [De Angelis et al. '18]

3. **"Difference Predicates" to enhance the Elimination Algorithm.**

Property to verify

∀ l. sumlist l = sumlist (insertionSort l)

Property *Sum*

OCaml Program

```
type list = Nil | Cons of int * list

let rec sumlist l =
   match l with
      | Nil -> 0
      | Cons(x,xs) -> x + sumlist xs

let rec ins i l =
  match l with
     | Nil -> Cons(i,Nil)
     | Cons(x,xs) -> if i<=x then Cons(i,Cons(x,xs)) else Cons(x,ins i xs)

let rec insertionSort l =
  match l with
     | Nil -> Nil
     | Cons(x,xs) -> ins x (insertionSort xs)
```

Translation into Constrained Horn Clauses

$\forall$l. sumlist l = sumlist (insertionSort l)

Property *Sum*

```
1. false :- M=\=N, sumlist(L,M), insertionSort(L,SL), sumlist(SL,N).
```

OCaml Program

```
let rec sumlist l =
    match l with
      | Nil -> 0
      | Cons(x,xs) -> x + sumlist xs
```

```
2. sumlist([],0).
3. sumlist([X|Xs],M) :- M=X+N, sumlist(Xs,N).
```

```
let rec ins i l =
    match l with
      | Nil -> Cons(i,Nil)
      | Cons(x,xs) -> if i<=x then Cons(i,Cons(x,xs)) else Cons(x,ins i xs)
```

```
4. ins(I,[],[I]).
5. ins(I,[X|Xs],[I,X|Xs]) :- I=<X.
6. ins(I,[X|Xs],[X|Ys]) :- I>X, ins(I,Xs,Ys).
```

```
let rec insertionSort l =
    match l with
      | Nil -> Nil
      | Cons(x,xs) -> ins x (insertionSort xs)
```

```
7. insertionSort([],[]).
8. insertionSort([X|Xs],SL) :- insertionSort(Xs,SXs), ins(X,SXs,SL).
```

LIA = Linear Integer Arithmetic : M=X+N, X= 2Y+3, …, ~~X=YZ+1~~, …

Satisfiability of Constrained Horn Clauses

Fact. Property *Sum* holds iff clauses 1-8 are satisfiable.

A satisfiability check is sufficient to check the property.

Fact. Satisfiability is undecidable and not semidecidable.

Definition. LIA-solvability is satisfiability via models definable in Linear Integer Aritmetic.

Fact. LIA-solvability is undecidable and semidecidable.

LIA-Solvability of Constrained Horn Clauses

```
                                                    -- Increase by 1 and by 2
false :- X≥N, Y≤X, p(X,Y,N).
p(X1,Y2,N) :- X<N, X1=X+1, Y2=Y+2, p(X,Y,N).
p(X,Y,N) :- X=0, Y=0, N≥1.
```

LIA-solvable: $p(X,Y,N) \equiv (Y>X,\ X\geq 0,\ N\geq 1) \vee (N>X,\ Y\geq X,\ X\geq 0)$

LIA-Solvability of Constrained Horn Clauses

```
                                        -- Commutativity of product

false :-  P≠Q, prod(X,Y,P), prod(Y,X,Q).
prod(0,Y,0).
prod(X1,Y,P1) :- X1=X+1, P1=P+Y, prod(X,Y,P).
```

Not LIA-solvable.

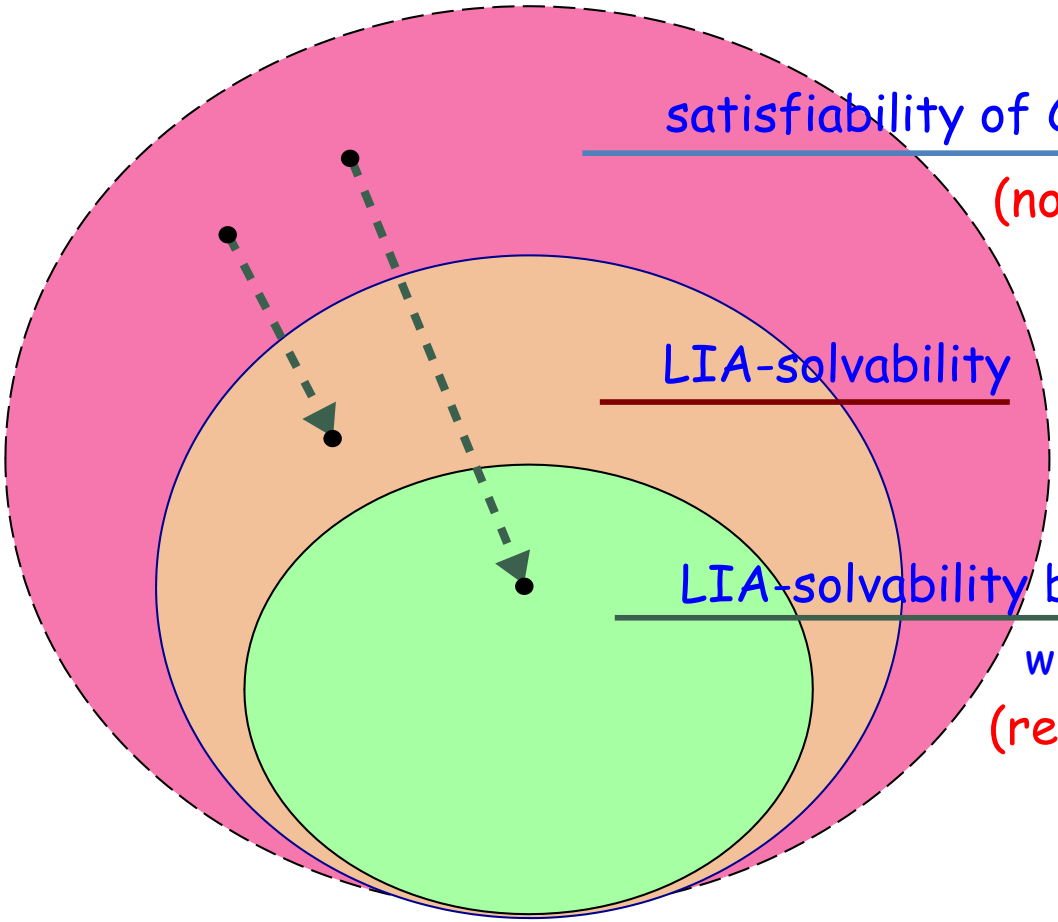
LIA-Solvability by CHC solvers

Sometimes LIA-solvability can be shown by CHC solvers.

LIA-solvability of clauses 1-8 cannot be shown by CHC solvers
because they do not have induction on lists.

Our objective: to avoid list variables, while preserving satisfiability.

From satisfiability to LIA-solvability



satisfiability of CHCs = proof of property

(not r.e.)

LIA-solvability

(r.e.)

LIA-solvability by a fixed CHC solver (e.g., Eldarica)

within a fixed time

(rec)

Elimination of lists

Elimination of lists, i.e. "  ", by applying the Elimination Algorithm.

The Elimination Algorithm $\mathscr{E}$.

Input: A set $Cls \cup Gs$, where Cls is a set of definite clauses and Gs is a set of goals;

Output: A set $TransfCls$ of clauses such that: (i) $Cls \cup Gs$ is satisfiable iff $TransfCls$ is satisfiable, and (ii) every clause in $TransfCls$ has basic types.

$Defs := \emptyset$; $InCls := Gs$; $TransfCls := \emptyset$;

while $InCls \neq \emptyset$ **do**

$Define\text{-}Fold(Defs, InCls, NewDefs, FldCls)$; -- introduce new definitions and fold.

$Unfold(NewDefs, Cls, UnfCls)$; -- unfold the new definitions.

$Replace(UnfCls, Cls, RCls)$;

$Defs := Defs \cup NewDefs$; $InCls := RCls$; $TransfCls := TransfCls \cup FldCls$;

NOTE: Only integer or boolean variables in the head of new definitions.

Applying the Elimination Algorithm (1)

```
 1. false :- M=\=N, sumlist(L,M), insertionSort(L,SL), sumlist(SL,N).
```

define:

```
 9. new1(M,N) :-sumlist(L,M), insertionSort(L,SL), sumlist(SL,N).
```

($\in Defs$)

fold 1 using 9:

```
10. false :- M=\=N, new1(M,N).
```

Applying the Elimination Algorithm (2)

9. `new1(M,N) :-sumlist(L,M), insertionSort(L,SL), sumlist(SL,N).` ($\in$ *Defs*)

unfold 9 w.r.t. `sumlist(L,M)`:

11. `new1(0,0).`
12. `new1(M1,N1) :- M1=H+M, sumlist(T,M), insertionSort(T,ST), ins(H,ST,SU), sumlist(SU,N1).`
(12.1) (12.2) (12.3) (12.4)

unfold 12 w.r.t. `sumlist(T,M)`: ...

Applying the Elimination Algorithm (3)

9. `new1(M,N) :- sumlist(L,M), insertionSort(L,SL), sumlist(SL,N).` $(\in Defs)$

---

unfold 12 w.r.t. `sumlist(T,M)`:

12.1 `new1(H,H).`

12.2 `new1(M1,N1) :- M1=H+Ma, Ma=Xa+Na,`
`sumlist(Xsa,Na), insertionSort([Xa|Xsa],ST), ins(H,ST,SU), sumlist(SU,N1).`

define:

`new2(M1,N1) :- sumlist(Xsa,Na), insertionSort([Xa|Xsa],ST), ins(H,ST,SU), sumlist(SU,N1).`

... the Elimination Algorithm does not terminate.

Difference Predicate: Embed

9. `new1(M,N) :- sumlist(L,M), insertionSort(L,SL), sumlist(SL,N).`

definition for folding

12. `new1(M1,N1) :- M1=H+M, sumlist(T,M), insertionSort(T,ST), ins(H,ST,SU), sumlist(SU,N1).`

clause to be folded

-- Stop unfolding --

Difference Predicate: Embed, Rename

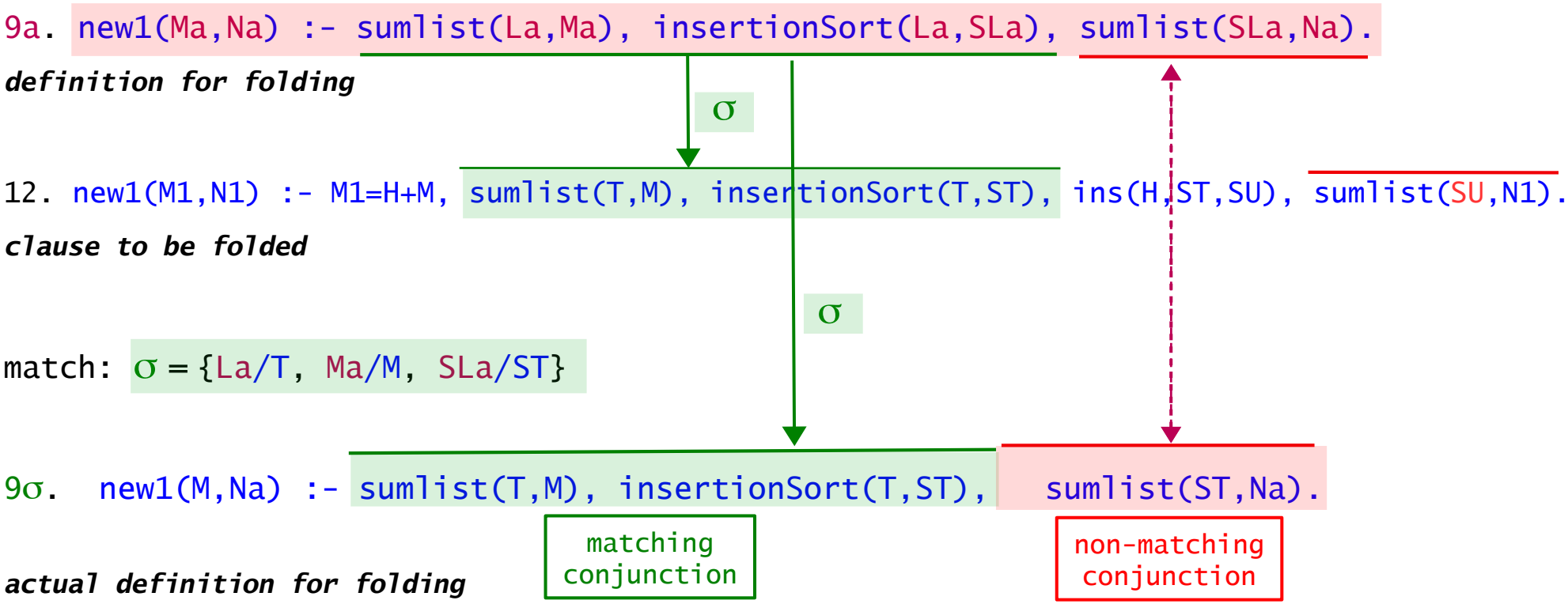
9a. `new1(Ma,Na) :- sumlist(La,Ma), insertionSort(La,SLa), sumlist(SLa,Na).`

definition for folding

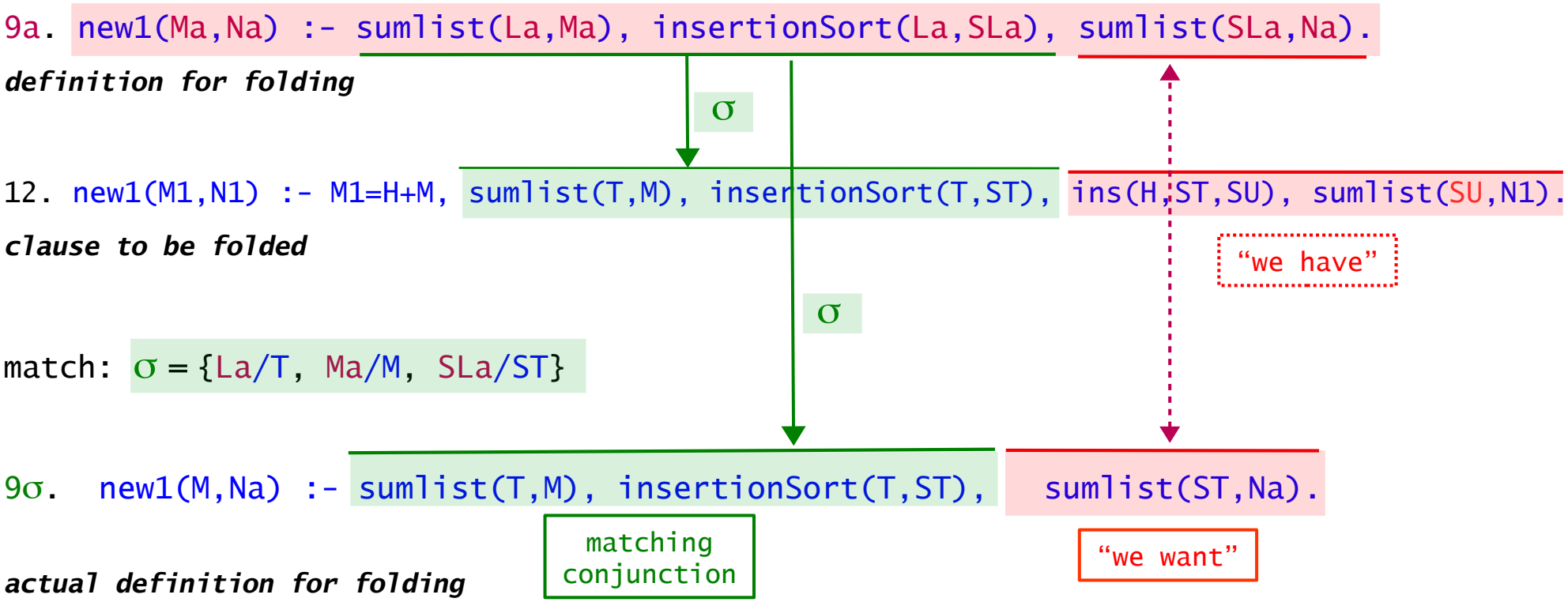
12. `new1(M1,N1) :- M1=H+M, sumlist(T,M), insertionSort(T,ST), ins(H,ST,SU), sumlist(SU,N1).`

clause to be folded

Difference Predicate: Embed, Rename, Match



Difference Predicate: Embed, Rename, Match



Difference Predicate: Embed, Rename, Match

9a. new1(Ma,Na) :- sumlist(La,Ma), insertionSort(La,SLa), sumlist(SLa,Na).

definition for folding

σ

12. new1(M1,N1) :- M1=H+M, sumlist(T,M), insertionSort(T,ST), ins(H,ST,SU), sumlist(SU,N1).

clause to be folded

"we have"

σ

match: σ = {La/T, Ma/M, SLa/ST}

9σ. new1(M,Na) :- sumlist(T,M), insertionSort(T,ST), sumlist(ST,Na).

matching conjunction

"we want"

actual definition for folding

introduce the difference predicate:

13. diff(H,Na,N1) :- ins(H,ST,SU), sumlist(SU,N1), sumlist(ST,Na).

integer variables

"we have"

"we want"

Difference Predicate: Replace, Fold

```
12.  new1(M1,N1) :- M1=H+M, sumlist(T,M), insertionsort(T,ST), ins(H,ST,SU), sumlist(SU,N1).
```

"we have"

replace:

```
12r. new1(M1,N1) :- M1=H+M, sumlist(T,M), insertionsort(T,ST), sumlist(ST,Na), diff(H,Na,N1).
```

"we want" diff

fold :

```
12f. new1(M1,N1) :- M1=H+M, new1(M,Na), diff(H,Na,N1).
```

Concluding ...

```
13. diff(H,Na,N1) :- ins(H,ST,SU), sumlist(SU,N1),  sumlist(ST,Na).
```

unfold ins, sumlist and
fold using diff:

```
14. diff(H,0,H).
15. diff(H,Na,N1) :- H=<X, Na=X+N2, N1=H+Na, sumlist(Xs,N2).
16. diff(H,Na,N1) :- H>X,  Na=X+N2, N1=X+N3, diff(H,N2,N3).
```

define:
    new2(N):- sumlist(Xs,N). (by eliminating the list variable Xs)

unfold/fold using new2:

```
17. new2(0).
18. new2(N) :- N=X+N1, new2(N1).
```

Final clauses and their LIA model

Final Clauses **without list variables**:

```
10.  false :- M=\=N, new1(M,N).

11.  new1(0,0).
12f. new1(M1,N1) :- M1=H+M, new1(M,Na), diff(H,Na,N1).


14.  diff(H,0,H).
15f. diff(H,Na,N1) :- H=<X, Na=X+N2, N1=H+Na, new2(N2).
16.  diff(H,Na,N1) :- H>X,  Na=X+N2, N1=X+N3, diff(H,N2,N3).
17.   new2(0).
18.   new2(N) :- N=X+N1, new2(N1).
```

Their LIA model (found by Eldarica):

```
new2(N)        ≡  true
new1(M,N)      ≡   M=N
diff(H,Na,N1)  ≡  N1=H+Na
```

Thus, Property *Sum* holds.

Comments on the Difference Predicates technique

1. Renaming

2. Matching

3. Correctness of the Replacement

4. Lemma Discovery

1. Renaming

Renaming is performed once only.

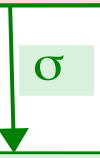
The other steps do not require any renaming-apart of the clauses.

Easy mechanization of the technique.

2. Matching

In general, various matchings are possible:

```
9a.  new1(Ma,Na) :- sumlist(La,Ma), insertionSort(La,SLa), sumlist(SLa,Na).
```




```
12.  new1(M1,N1) :- M1=H+M, sumlist(T,M), insertionSort(T,ST), ins(H,ST,SU), sumlist(SU,N1).
```

match: $\sigma = \{La/T,\ Ma/M,\ SLa/ST\}$

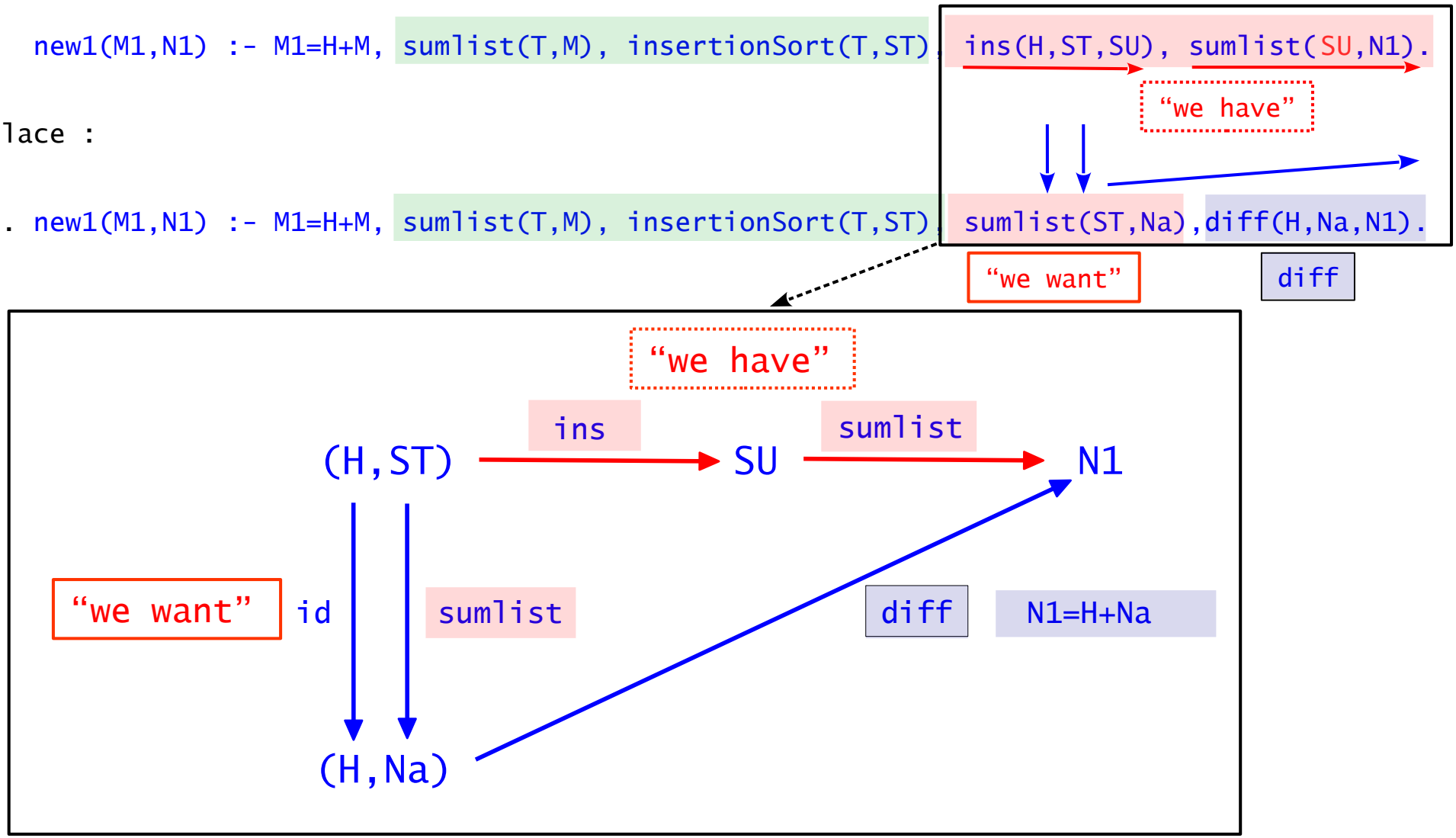
---- We need a strategy to find "a good matching".

3. Correctness of the Replacement (1)

12. new1(M1,N1) :- M1=H+M, sumlist(T,M), insertionSort(T,ST), ins(H,ST,SU), sumlist(SU,N1).

replace :

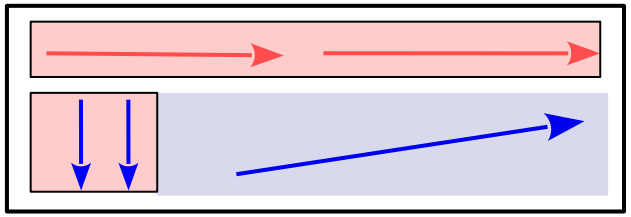
12r. new1(M1,N1) :- M1=H+M, sumlist(T,M), insertionSort(T,ST), sumlist(ST,Na),diff(H,Na,N1).

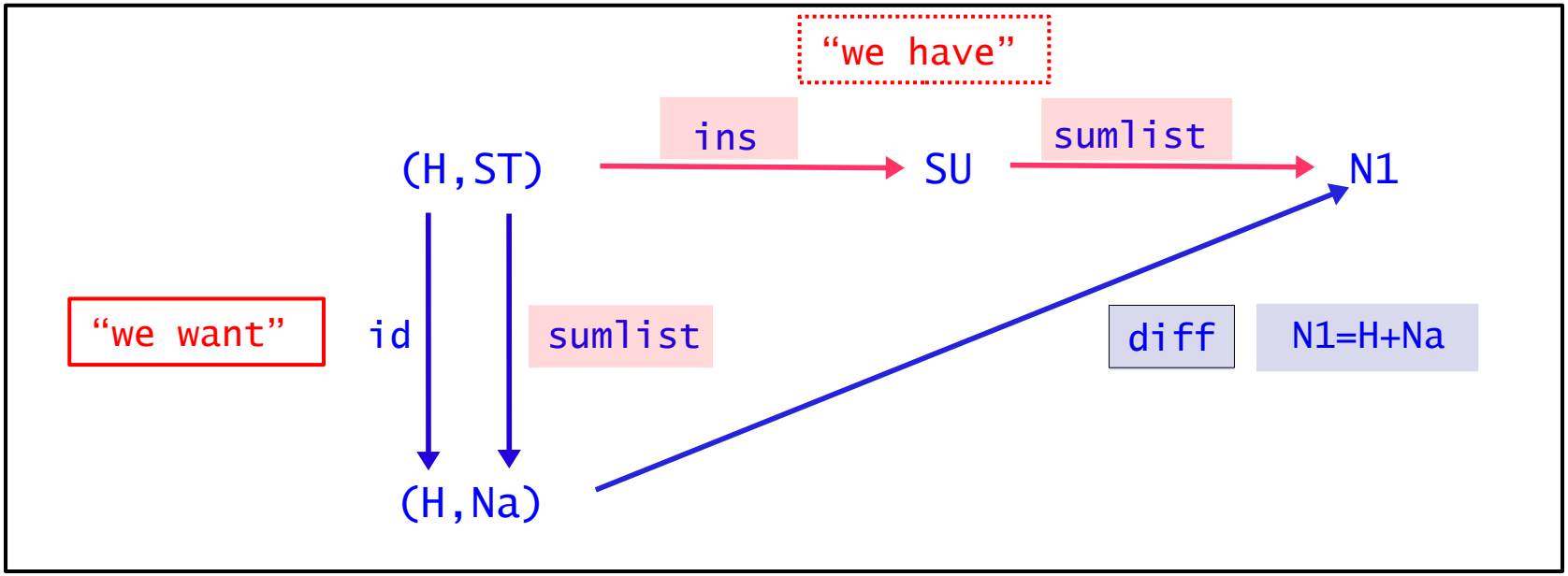


3. Correctness of the Replacement (2)

replace: C

Cr = C − "we have" + "we want" + diff





Theorem. If diff is a function, then C is satisfiable iff Cr is satisfiable.

Theorem. C is satisfiable, if Cr is satisfiable.

4. Lemma Discovery (1)

Theorem. ∀ l. sumlist l = sumlist (insertionSort l)

Proof. By structural induction.

l = []

sumlist [] ? = sumlist (insertionSort []) =

= {definition of insertionSort} =

= sumlist []

l = [h|t]

sumlist [h|t] ? = sumlist (insertionSort [h|t]) =

= sumlist (ins h (insertionSort t)) =

= {lemma: ∀ h,l. sumlist (ins h l) = h + sumlist l} =

= h + sumlist (insertionSort t) =

= {induction hypothesis} =

= h + sumlist t =

= {definition of sumlist} =

= sumlist [h|t]

4. Lemma Discovery (2)

Lemma. ∀ h,l. sumlist (ins h l) = h + sumlist l.

13. diff(H,Na,N1) :- ins(H,ST,SU), sumlist(SU,N1), sumlist(ST,Na).

ins H ST = SU, sumlist SU = N1, sumlist ST = Na ⟶ diff(H,Na,N1)

ins H ST = SU, sumlist SU = N1, sumlist ST = Na ⟶ N1 = H + Na (LIA model)

sumlist (ins H ST) = N1 ⟶ H + (sumlist ST) = N1

sumlist (ins H ST) = H + (sumlist ST)