From Program Transformation to Program Verification

Maurizio Proietti, CNR-IASI, Rome, Italy
joint work with
Emanuele De Angelis, CNR-IASI, Rome, Italy
Fabio Fioravanti, Univ. Chieti-Pescara, Italy

VPT2020@ETAPS2021 dedicated to Alberto Pettorossi

Automated Program Derivation 1970s-90s

- Automated theorem proving, declarative languages, program synthesis from formal specifications, automatic programming.
- Program transformation as a means of developing programs in FP and LP:

  Fold/Unfold, Partial Evaluation/Program Specialization, Supercompilation, Tupling, Deforestation, Continuation Passing Style, Compiling control, …

1

Deriving correct and efficient programs via transformations

Program transformation

Rules + Strategies Approach [Burstall-Darlington]: Derive programs by separating correctness from efficiency issues.

1. Correctness is guaranteed by semantics-preserving transformation **rules**.

Initial program $P_0 \rightarrow P_1 \rightarrow \ldots \rightarrow P_n$ Final program

$$Sem(P_0) = Sem(P_1) = \ldots = Sem(P_n)$$

2. Efficiency obtained by suitable **strategies** of application of the rules:

$$Cost(P_0) \geq Cost(P_n)$$

List reversal

P: reverse([ ],[ ]).
reverse([X|Xs],Ys) ← reverse(Xs,R), append(R,[X],Ys).

append([ ],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) ← append(Xs,Ys,Zs).

Goal:
← reverse(Xs,Zs).

Source of inefficiency: existential list R occurs in the body but not in the head

Deriving an efficient list reversal program

Define ('Eureka')

rev-acc(Xs,A,Ys) ← reverse(Xs,R), append(R,A,Ys).

Deriving an efficient list reversal program

Define (‘Eureka’)

rev-acc(Xs,A,Ys) ← reverse(Xs,R), append(R,A,Ys).

Unfold

rev-acc([ ],A,A).

rev-acc([X|Xs],A,Ys) ← reverse(Xs,T), append(T,[X],R), append(R,A,Ys).

Deriving an efficient list reversal program

Define ('Eureka')

rev-acc(Xs,A,Ys) ← reverse(Xs,R), append(R,A,Ys).

Unfold

rev-acc([ ],A,A).

rev-acc([X|Xs],A,Ys) ← reverse(Xs,T), append(T,[X],R), append(R,A,Ys).

Laws (Associativity and Neutral element of append)

rev-acc([ ],A,A).

rev-acc([X|Xs],A,Ys) ← reverse(Xs,T), append(T,[X|A],Ys).

← reverse(Xs,Ys), append(Ys,[ ],Zs). % append(Ys,[ ],Zs) → Xs=Ys

Deriving an efficient list reversal program

Define (‘Eureka’)

rev-acc(Xs,A,Ys) ← reverse(Xs,R), append(R,A,Ys).

Unfold

rev-acc([ ],A,A).

rev-acc([X|Xs],A,Ys) ← reverse(Xs,T), append(T,[X],R), append(R,A,Ys).

Laws (Neutral element and Associativity of append)

rev-acc([ ],A,A).

rev-acc([X|Xs],A,Ys) ← reverse(Xs,T), append(T,[X|A],Ys).

← reverse(Xs,Ys), append(Ys,[ ],Zs).

Fold

rev-acc([ ],A,A).

rev-acc([X|Xs],A,Ys) ← rev-acc(Xs,[X|A],Ys).

← rev-acc(Xs,[ ],Zs).

No unnecessary, existential list

2

Theorem Proving via unfold/fold transformations

Proving laws via U/F

Law (Neutral elem.): $\forall Xs,Ys.\ append(Xs,[\ ],Ys) \rightarrow Xs=Ys.$

Proving laws via U/F

Law (Neutral elem.): $\forall$Xs,Ys. append(Xs,[ ],Ys) → Xs=Ys.

Translation into Goal: false ← append(Xs,[ ],Ys), Xs=\=Ys. %Existential vars

Proving laws via U/F

Law (Neutral elem.): $\forall$Xs,Ys. append(Xs,[ ],Ys) $\rightarrow$ Xs=Ys.

Translation into Goal: false $\leftarrow$ append(Xs,[ ],Ys), Xs=\=Ys. %Existential vars

Define negp $\leftarrow$ append(Xs,[ ],Ys), Xs=\=Ys.

Proving laws via U/F

Law (Neutral elem.): ∀Xs,Ys. append(Xs,[ ],Ys) → Xs=Ys.

Translation into Goal: false ← append(Xs,[ ],Ys), Xs=\=Ys. %Existential vars

Define negp ← append(Xs,[ ],Ys), Xs=\=Ys.

Unfold append ~~negp ← [ ]=\=[ ].~~ %Delete: UNSAT body
negp ← append(Xs,[ ],Ys), [X|Xs]=\=[X|Ys].

Proving laws via U/F

Law (Neutral elem.):	∀Xs,Ys. append(Xs,[ ],Ys) → Xs=Ys.
Translation into Goal:	false ← append(Xs,[ ],Ys), Xs=\=Ys. %Existential vars
Define	negp ← append(Xs,[ ],Ys), Xs=\=Ys.
Unfold append	~~negp ← [ ]=\=[ ].~~ %Delete: UNSAT body negp ← append(Xs,[ ],Ys), [X|Xs]=\=[X|Ys].
Unfold =\=	negp ← append(Xs,[ ],Ys), Xs=\=Ys.

Proving laws via U/F

Law (Neutral elem.):	∀Xs,Ys. append(Xs,[ ],Ys) → Xs=Ys.	
Translation into Goal:	false ← append(Xs,[ ],Ys), Xs=\=Ys.	%Existential vars
Define	negp ← append(Xs,[ ],Ys), Xs=\=Ys.	
Unfold append	~~negp ← [ ]=\=[ ].~~ negp ← append(Xs,[ ],Ys), [X|Xs]=\=[X|Ys].	%Delete: UNSAT body
Unfold =\=	negp ← append(Xs,[ ],Ys), Xs=\=Ys.	
Fold	false ← negp. negp ← negp.	Final clauses: No existential vars

Proving laws via U/F

Law (Neutral elem.): ∀Xs,Ys. append(Xs,[ ],Ys) → Xs=Ys.

Translation into Goal: false ← append(Xs,[ ],Ys), Xs=\=Ys. %Existential vars

Define: negp ← append(Xs,[ ],Ys), Xs=\=Ys.

Unfold append:
~~negp ← [ ]=\=[ ].~~ %Delete: UNSAT body
negp ← append(Xs,[ ],Ys), [X|Xs]=\=[X|Ys].

Unfold =\=: negp ← append(Xs,[ ],Ys), Xs=\=Ys.

Fold:
false ← negp.
negp ← negp.

Final clauses: No existential vars

The final clauses are SAT (negp is false in the least H. model): Law holds!

U/F works as quantifier elimination

First Order Theorem Proving via U/F

The elimination of existential variables from (Constraint) Logic Programs with negation allows the proof of full first order logic.

All lists of integers have an upper bound

P: member (X,[Y|L]) $\leftarrow$ X=Y % set of Horn clauses with
member (X,[Y|L]) $\leftarrow$ member (X,L) % constraints (CLP aka CHCs)

φ : $\forall L\ \exists U\ \forall X\ (\ member(X,L) \rightarrow X \leq U\ )$ % closed first order formula

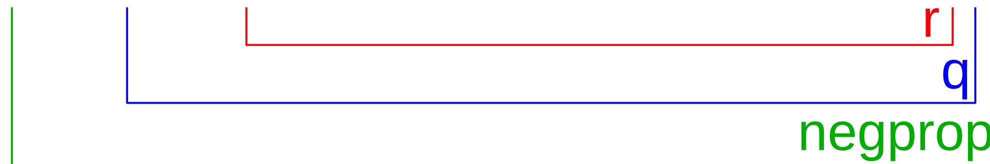
M(P): least **D**-model of P % **D** is the interpretation of the constraints

$$M(P) \models \varphi\ ?$$

Lloyd-Topor transformation

φ : $\forall L\ \exists U\ \forall X\ (\ member(X,L) \rightarrow X \leq U\ )$

$f \leftarrow \neg\exists L\ \neg\exists U\ \neg\exists X\ (\ member(X,L) \land \neg X \leq U\ )$



Lloyd-Topor transformation + addition of a list(L) atom for each list variable L:

CF: f ← ¬ negprop

negprop ← list(L), ¬ q(L)

q(L) ← list(L), ¬ r(L,U)

r(L,U) ← X > U, list(L), member(X,L)

- P U CF stratified program with Perfect model M(P U CF)
- existential variables

$M(P) \models \varphi$ iff $M(P \cup CF) \models f$

Elimination of Existential Variables by Unfold/Fold transformation

CF: $f \leftarrow \neg$ negprop
negprop $\leftarrow$ list(L), $\neg$ q(L)
q(L) $\leftarrow$ list(L), $\neg$ r(L,U)
r(L,U) $\leftarrow$ X > U, list(L), member(X,L)

Unfold/Fold →

Prop: $f \leftarrow \neg$negprop
negprop $\leftarrow$ q1
q1 $\leftarrow$ q1

- The Unfold/Fold transformation eliminates the existential variables
- The transformation rules and strategies are extended to handle negation
- $M(P) \models \varphi$ iff $M(P \cup CF) \models f$ iff $M(Prop) \models f$
- Upper bound example:

  $M(Prop) = \{f\} \Rightarrow M(P) \models \forall L\ \exists U\ \forall X\ (\ member(X,L) \rightarrow X \leq U\ )$

3

Program verification

Via CHC transformation

Program verification

- Constrained Horn clauses (CHCs) are a suitable logical formalism for reasoning about program correctness;
- Program semantics can be formally specified via CHCs;
- Many verification problems can be encoded as a satisfiability problem for CHCs;
- Many CHC tools for verfying satisfiability (CHC solvers) are available: Spacer/Z3, Eldarica, …
- CHC solvers handle many constraint theories: Linear Integer/Real Arithmetic, Booleans, Arrays, Bit-vectors, ADTs, …

List partitioning

- Functional fragment of Scala with pre/postconditions (require/ensuring)
- Partitioning a list of integers wrt a positive integer pivot

```
def partition(x: Nat, l: List[Nat]): (List[Nat], List[Nat]) = {
  l match {
    case Nil() => (Nil[Nat](), Nil[Nat]())
    case Cons(y, ys) =>
      val (l1, l2) = partition(x, ys)
      if (x > y) { (Cons(y, l1), l2) }
      else       { (l1, Cons(y, l2)) }
  }
} ensuring { res =>
  all_grt(x, res._1) && all_leq(x, res._2)   // partition postcondition
}
```

Assertion predicates

- Pre/postconditions are defined via Boolean-valued functions

```
def all_grt(x: Nat, l: List[Nat]): Boolean = {
  l match {
    case Nil() => true
    case Cons(y, ys) if (x =< y) => false
    case Cons(y, ys) if (x > y) => all_grt(x, ys)
 }

def all_leq(x: Nat, l: List[Nat]): Boolean = {
  l match {
    case Nil() => true
    case Cons(y, ys) if (x > y) => false
    case Cons(y, ys) if (x =< y) => all_leq(x, ys)
  }
}
```

Contract specification

- Contract is valid if

```
             require                             ensuring
∀x,res. pre(x)  &&  f(x)==res  ==>  post(x,res)
```

- Contract for partition (`pre(x)` is `true`)

```
∀x,l,l1,l2.
  partition(x,l)==(l1,l2) ==> all_grt(x,l1) && all_leq(x,l2)
```

CHC translation of functions

- f(X,Y) is the translation of ‘f(X) evaluates to Y’ (in *call-by-value* semantics)
- Linear Integer Arithmetic (LIA) and Boolean (Bool) constraints

PartitionCls

```
partition(X,[ ],[ ],[ ]).
partition(X,[Y|Ys],[Y|L1s],L2s) ← X>Y, Y>=0, partition(X,Ys,L1s,L2s).
partition(X,[Y|Ys],L1s,[Y|L2s]) ← X=<Y, X>=0, partition(X,Ys,L1s,L2s).

all_grt(X,[ ],B) ← X>=0, B=tt.                   % tt stands for true
all_grt(X,[Y|Ys],B) ← X=<Y, X>=0, B=ff.          % ff stands for false
all_grt(X,[Y|Ys],B) ← X>Y, Y>=0, all_grt(X,Ys,B).

all_leq(X,[ ],B) ← X>=0, B=tt.
all_leq(X,[Y|Ys],B) ← X>Y, Y>=0, B=ff.
all_leq(X,[Y|Ys],B) ← X=<Y, X>=0, all_leq(X,Ys,B).
```

CHC translation of contract

- The contract for partition

```
∀x,l,l1,l2.
  partition(x,l)==(l1,l2)  ==>  all_grt(x,l1) && all_leq(x,l2)
```

- is translated into the goals:

false ← B=ff, partition(X,L,L1,L2), all_grt(X,L1,B). (G1)

false ← B=ff, partition(X,L,L1,L2), all_leq(X,L2,B). (G2)

- Contract is valid iff *PartitionCls* U {G1,G2} is satisfiable.

List removal

- *PartitionCls* U {G1,G2} is satisfiable
  but state-of-the-art solvers (Eldarica, Spacer/Z3) are not able to prove satisfiability: the model cannot expressed as a quantifier-free formula in TheoryOfLists U LIA U Bool.
- Elimination of list arguments, followed by CHC solving will do.

List removal

- false ← B=ff, partition(X,L,L1,L2), all_leq(X,L2,B). (G2)

List removal

- false ← B=ff, partition(X,L,L1,L2), all_leq(X,L2,B).   (G1)
- Define
  pl(X,B) ← partition(X,L,L1,L2), all_leq(X,L2,B).   %Bool and Nat in the head

List removal

- false ← B=ff, partition(X,L,L1,L2), all_leq(X,L2,B). (G1)
- Define
  pl(X,B) ← partition(X,L,L1,L2), all_leq(X,L2,B). %Bool and Nat in the head
- Unfold definition of pl
  pl(A,B) ← A=tt, B>=0.
  ~~pl(A,B) ← B=<C, B>=0, partition(B,D,E,F), B>C, A=ff.~~ %Delete
  pl(A,B) ← B=<C, B>=0, partition(B,D,E,F), all_leq(B,F,A).

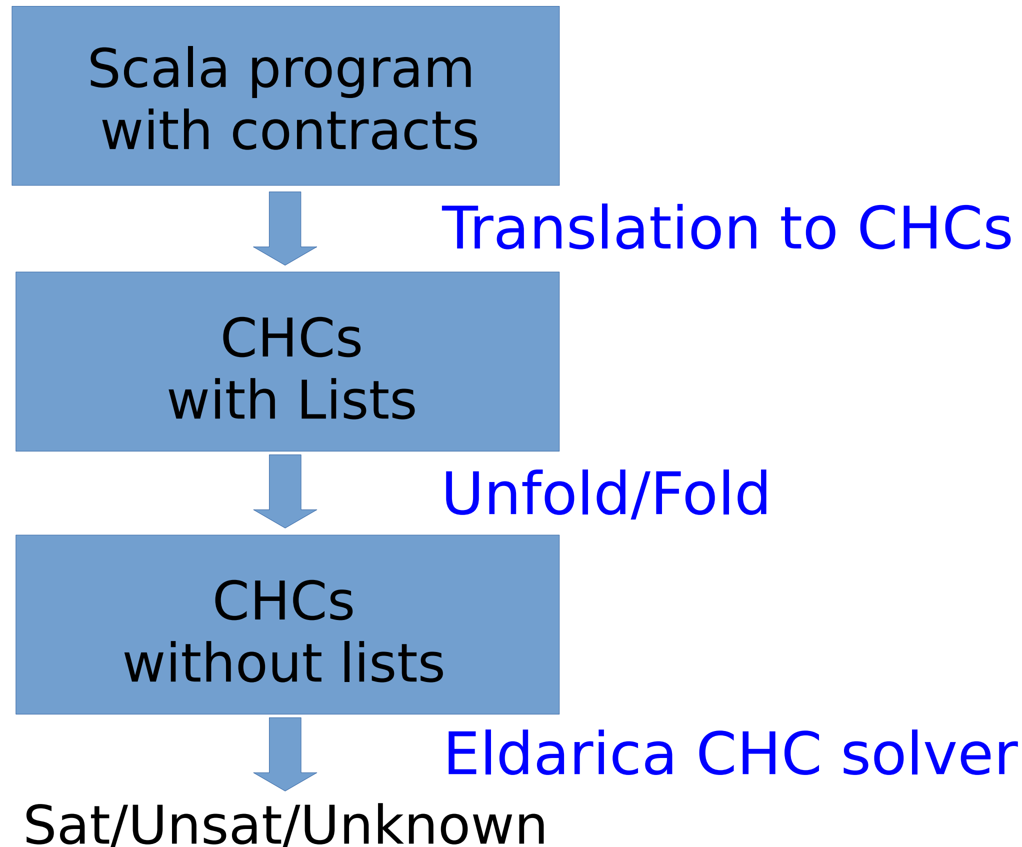
List removal

- false ← B=ff, partition(X,L,L1,L2), all_leq(X,L2,B). (G1)
- Define
  pl(X,B) ← partition(X,L,L1,L2), all_leq(X,L2,B). %Bool and Nat in the head
- Unfold definition of pl
  pl(A,B) ← A=tt, B>=0.
  ~~pl(A,B) ← B=<C, B>=0, partition(B,D,E,F), B>C, A=ff.~~ %Delete
  pl(A,B) ← B=<C, B>=0, partition(B,D,E,F), all_leq(B,F,A).
- Fold twice: No lists
  false ← A=ff, pl(A,B).
  pl(A,B) ← A=tt, B>=0.
  pl(A,B) ← B>=0, pl(A,B).

List removal

- false ← B=ff, partition(X,L,L1,L2), all_leq(X,L2,B). (G1)
- Define
  pl(X,B) ← partition(X,L,L1,L2), all_leq(X,L2,B). %Bool and Nat in the head
- Unfold definition of pl
  pl(A,B) ← A=tt, B>=0.
  ~~pl(A,B) ← B=<C, B>=0, partition(B,D,E,F), B>C, A=ff.~~ %Delete
  pl(A,B) ← B=<C, B>=0, partition(B,D,E,F), all_leq(B,F,A).
- Fold twice: No lists
  false ← A=ff, pl(A,B).
  pl(A,B) ← A=tt, B>=0.
  pl(A,B) ← B>=0, pl(A,B).
- Eldarica proves SAT and computes a model:
  pl(A,B) ← A=tt, B>=0.

CHC verification via list removal



The method can be extended to generic ADTs.

Quicksort

```
def quicksort(l: List[Nat]): List[Nat] = {
  l match {
    case Nil() => Nil[Nat]()
    case Cons(x, xs) =>
      val (ys,zs) = partition(x, xs)
      append(quicksort(ys), Cons(x, quicksort(zs)))
  }
} ensuring { res =>
    forall((a: Nat) => all_grt(a,l) ==> all_grt(a,res)) &&
    forall((a: Nat) => all_leq(a,l) ==> all_leq(a,res)) &&
    isSorted(0,res) &&
    forall((a: Nat) => count(a,l) == count(a,res))
}
```

- Plus definition of `partition` and `append` (list concatenation).

Assertion functions for quicksort

```
def count(a: Nat, l: List[Nat]): Nat = {
  l match {
    case Nil() => 0
    case Cons(x, xs) => if (x==a) { count(a,xs)+1 } else { count(a,xs) }
  }
}

def isSorted(a: Nat, l: List[Nat]): Boolean = {
  l match {
    case Nil() => true
    case Cons(x,xs) => if (a<=x) isSorted(x,xs) else false
  }
}
```

- `count(a,l)==` number of occurrences of a in list `l`
- `isSorted(a,l) == true` iff a is a lower bound of `l` and `l` is ordered

Parameterized Catamorphisms

```
def pCata(p:A, l:List[A]): B = {
  match l {
    case Nil() => c
    case Cons(x,xs) => g(p,x,pCata(h(p,x),xs))
  }
```

Total functions defined by structural recursion on lists.

Contracts for quicksort

- $\forall$a,l,res. `quicksort(l)==res && all_grt(a,l) ==> all_grt(a,res))`
- $\forall$a,l,res. `quicksort(l)==res && all_leq(a,l) ==> all_leq(a,res))`
- $\forall$l,res. `quicksort(l)==res ==> isSorted(0,res)`
- $\forall$a,l,res. `quicksort(l)==res ==> count(a,l)==count(a,res)`
- plus contracts for `partition` and `append`.

CHC translation of functions

%Program *QuicksortCls*

```
quicksort([ ],[ ]).
quicksort([X|Xs],Ys) ← X>=0,
   partition(X,Xs,Littles,Bigs),
   quicksort(Littles,Ls), quicksort(Bigs,Bs),
   append(Ls,[X|Bs],Ys).
append([ ],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) ← X>=0, append(Xs,Ys,Zs).
```

\+ clauses for partition

%Parameterized Catamorphisms

```
count(X,[ ],N) ← X>=0, N=0.
count(X,[Y|Ys],N) ← X>=0, X=Y, N=M+1, count(X,Ys,M).
count(X,[Y|Ys],N) ← X>=0, Y>=0, X=\=Y, N=M, count(X,Ys,M).

isSorted(A,[ ],B) ← A>=0, B=tt.
isSorted(A,[X|Xs],B) ← X>=0, A>X, B=ff.
isSorted(A,[X|Xs],B) ← A>=0, A=<X, isSorted(X,Xs,B).
```

CHC translation of contracts

Contract for quicksort

false ← B1=tt, B2=ff, quicksort(B,C), all_grt(A,B,B1), all_grt(A,C,B2). (G3)

false ← B1=tt, B2=ff, quicksort(B,C), all_leq(A,B,B1), all_leq(A,C,B2). (G4)

false ← B1=ff, quicksort(L,S), isSorted(0,S,B1). (G5)

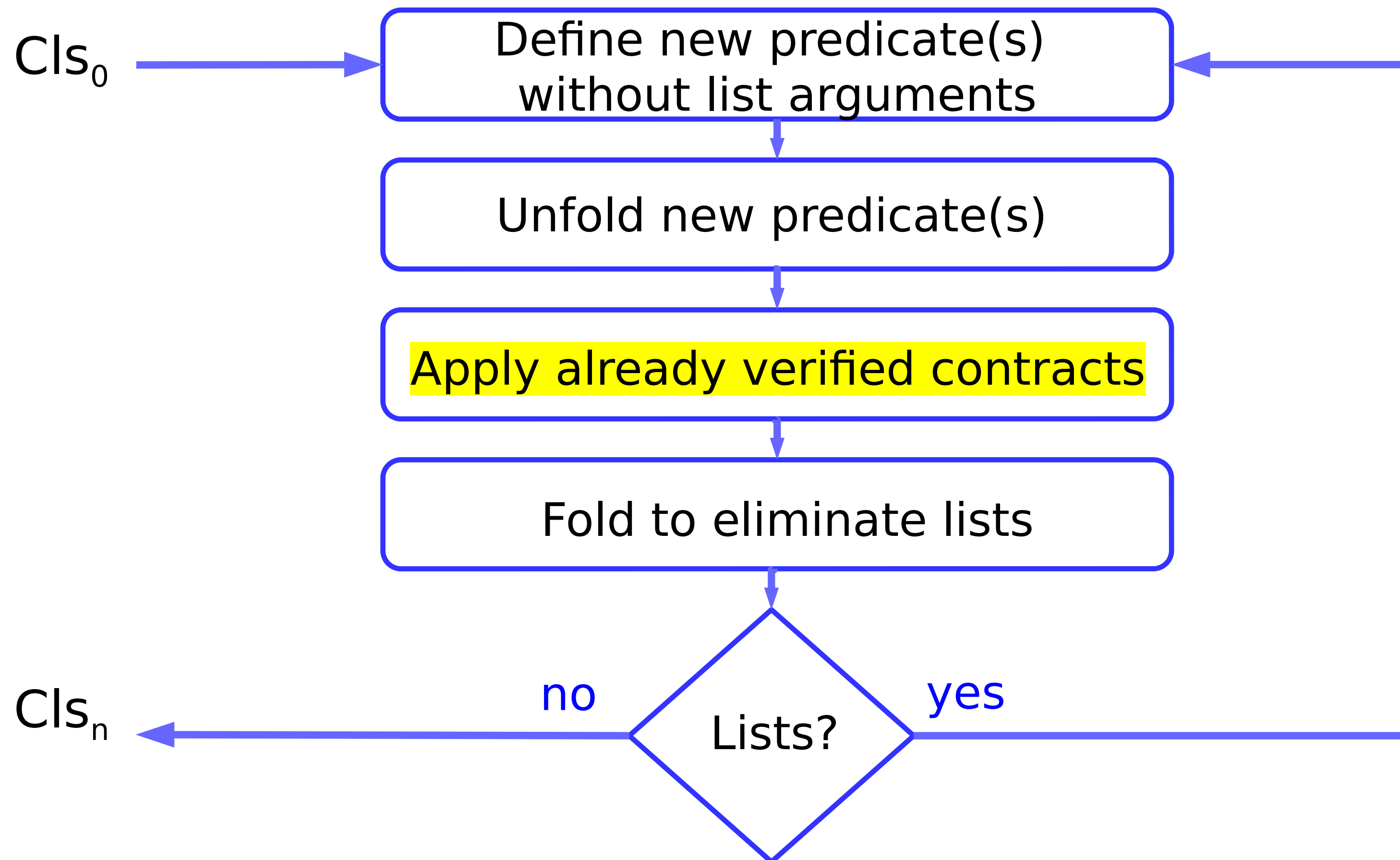
false ← N1=/=N2, quicksort(L,S), count(X,L,N1), count(X,S,N2). (G6)

Plus translation of contracts for partition and append

CHC translation of contracts

- false ← B1=ff, quicksort(L,S), isSorted(0,S,B1). (G5)
- Contract is valid iff QuicksortCls U {G5} is satisfiable
- CHC solvers are not able to prove satisfiability.
- The **Stainless** verifier for Scala is not able to prove the validity of this contract

List Removal Algorithm



- Soundness: If Cls_n satisfiable then Cls_0 satisfiable

List removal: Define

- false ← B1=ff, quicksort(L,S), isSorted(0,S,B1). (G5)

List removal: Define

- false ← B1=ff, quicksort(L,S), isSorted(0,S,B1). (G5)
- Define new predicate without list arguments

  qss(B1) ← quicksort(L,S), isSorted(0,S,B1).

List removal: Unfold

- false ← B1=ff, quicksort(L,S), isSorted(0,S,B1). (G5)

- Define new predicate without list arguments

  qss(B1) ← quicksort(L,S), isSorted(0,S,B1).

- Unfold quicksort

  qss(A) ← A=tt.
  qss(A) ← B>=0,
  partition(B,C,D,E),
  quicksort(D,F),
  quicksort(E,G),
  append(F,[B|G],H),
  isSorted(0,H,A).

List removal: Apply contracts

- false ← B1=ff, quicksort(L,S), isSorted(0,S,B1). (G5)

- Define new predicate without list arguments

  qss(B1) ← quicksort(L,S), isSorted(0,S,B1).

- Apply already verified contracts G1--G4

```
qss(A) ← A=tt.
qss(A) ← B>=0,
      partition(B,C,D,E), all_grt(B,D,tt), all_leq(B,E,tt),      % G1, G2
      quicksort(D,F), isSorted(0,F,B1), all_grt(B,F,tt),         % G3
      quicksort(E,G), isSorted(0,G,B2), all_leq(B,G,tt),         % G4
      append(F,[B|G],H),
      isSorted(0,H,A).
```

List removal: Fold

- false ← B1=ff, qss(B1). (G5)

- Define new predicate without list arguments

  qss(B1) ← quicksort(L,S), isSorted(0,S,B1).

- Fold using definition of qss

```
qss(A) ← A=tt.
qss(A) ← B>=0,
    partition(B,C,D,E), all_grt(B,D,tt), all_leq(B,E,tt),
    qss(B1), isSorted(0,F,B1), all_grt(B,F,tt),
    qss(B2), isSorted(0,G,B2), all_leq(B,G,tt),
    append(F,[B|G],H),
    isSorted(0,H,A).
```

List Removal: Final Clauses

Define:

```
a(B,X,Y,Z,T,U,B1,B2,A) ←
    isSorted(X,F,B1), all_grt(B,F,T),
    isSorted(Y,G,B2), all_leq(B,G,U),
    append(F,[B|G],H), isSorted(Z,H,A).
```

Unfold/Fold …

```
false ← B1=ff, qss(B1).
qss(A) ← A=tt.
qss(A) ←
    B>=0,
    qss(B1),
    qss(B2),
    a(B,0,0,0,tt,tt,B1,B2,A).

+ clauses (without lists) for  a
```

Eldarica proves these clauses SAT

The contract is valid

Conclusions

- An old idea in logic program transformation, elimination of ‘unnecessary’ (existential) variables [Pettorossi&Proietti 1991], has found new applications in automated theorem proving and program verification.

- Future work: Construction of tools that support CHC Solvers and Program Verifiers.