Pure, declarative, and constructive arithmetic relations

Declarative pearl

Oleg Kiselyov
FNMOC
oleg@pobox.com

William E. Byrd
Daniel P. Friedman
Indiana University
{webyrd,dfried}@cs.indiana.edu

Chung-chieh Shan
Rutgers University
ccshan@cs.rutgers.edu

FLOPS
14 April 2008

What?

```
?- add(X,Y,[l,o,l]).
```

What?

```
?- add(X,Y,[l,o,l]).    X = [l,o,l], Y = []       ;
                        X = []     , Y = [l,o,l]  ;
                        X = [l]    , Y = [o,o,l]  ;
                        X = [o,o,l], Y = [l]      ;
                        X = [l,l]  , Y = [o,l]    ;
                        X = [o,l]  , Y = [l,l]    ; fail.
```

What?

```
?- add(X,Y,5).
X = 5 , Y = 0 ;
X = 0 , Y = 5 ;
X = 1 , Y = 4 ;
X = 4 , Y = 1 ;
X = 3 , Y = 2 ;
X = 2 , Y = 3 ; fail.
```

- Pure Prolog: no negation, no cut, no `var/1`
- Evaluation (and re-evaluation) terminates under SLD (and under SLD with interleaving)

What?

```
?- add(X,Y,5).      X = 5 , Y = 0 ;
                    X = 0 , Y = 5 ;
                    X = 1 , Y = 4 ;
                    X = 4 , Y = 1 ;
                    X = 3 , Y = 2 ;
                    X = 2 , Y = 3 ; fail.

?- add(2,3,Z).      Z = 5 ; fail.

?- add(X,3,5).      X = 2 ; fail.

?- add(2,Y,5).      Y = 3 ; fail.

?- add(2,3,5).      true ; fail.
```

- Pure Prolog: no negation, no cut, no `var/1`
- Evaluation (and re-evaluation) terminates under SLD (and under SLD with interleaving)

What?

- Pure Prolog: no negation, no cut, no `var/1`
- Evaluation (and re-evaluation) terminates under SLD (and under SLD with interleaving)

What?

```
?- add(2,Y,Z).
```

- Pure Prolog: no negation, no cut, no `var/1`
- Evaluation (and re-evaluation) terminates under SLD (and under SLD with interleaving)

What?

```
?- add([o,l],Y,Z).

   Y = []          , Z = [o,l]          ;
   Y = [l]         , Z = [l,l]          ;
   Y = [o,l]       , Z = [o,o,l]        ;
   Y = [o,o,G|H]   , Z = [o,l,G|H]      ;
   Y = [o,l,l]     , Z = [o,o,o,l]      ;
   Y = [o,l,o,G|H] , Z = [o,o,l,G|H]    ; ...
```

- Pure Prolog: no negation, no cut, no `var/1`
- Evaluation (and re-evaluation) terminates under SLD (and under SLD with interleaving)

What?

```
?- add(2,Y,Z).
```

```
Y = 0                    , Z = 2                    ;
Y = 1                    , Z = 3                    ;
Y = 2                    , Z = 4                    ;
Y = 4 × G + 8 × H        , Z = 2 + 4 × G + 8 × H    ;
Y = 6                    , Z = 8                    ;
Y = 2 + 8 × G + 16 × H   , Z = 4 + 8 × G + 16 × H   ; ...
```

- Pure Prolog: no negation, no cut, no `var/1`
- Evaluation (and re-evaluation) terminates under SLD (and under SLD with interleaving)

What?

```
?- add(2,Y,Z).
```

```
Y = 0                    , Z = 2                    ;
Y = 1                    , Z = 3                    ;
Y = 2                    , Z = 4                    ;
Y = 4 × G + 8 × H        , Z = 2 + 4 × G + 8 × H    ;
Y = 6                    , Z = 8                    ;
Y = 2 + 8 × G + 16 × H , Z = 4 + 8 × G + 16 × H ; ...
```

```
?- add(X,3,Z).
```

```
?- add(X,Y,Z).
```

- Pure Prolog: no negation, no cut, no `var/1`
- Evaluation (and re-evaluation) terminates under SLD (and under SLD with interleaving)

What?

- Addition and subtraction

$$\texttt{add}(X, Y, Z) \iff Z = X + Y$$

- Multiplication and division with remainder

$$\texttt{mul}(X, Y, Z) \iff Z = XY$$
$$\texttt{div}(N, M, Q, R) \iff N = MQ + R < M(Q + 1)$$

- Exponentiation and logarithm with remainder

$$\texttt{log}(N, M, Q, R) \iff N = M^Q + R < M^{Q+1}$$

- Pure Prolog: no negation, no cut, no `var/1`
- Evaluation (and re-evaluation) terminates under SLD (and under SLD with interleaving)

Why?

We like declarative programming!

How far can we go while staying pure,
without deferring evaluation in constraints?

Why?

We like declarative programming!

How far can we go while staying pure,
without deferring evaluation in constraints?

> "Forgive me for asking a crass and naïve question—
> but what is the point of devising a machine that
> cannot be built in order to prove that there are certain
> mathematical statements that cannot be proved?
> Is there any practical value in all this?"
>
> —*Breaking the Code*

Useful in an implementation of the functional-logic programming language Curry.

How?

- **Unary addition**
  - **Prove properties of *solution sets***
  - No sharing or conjoined goals

Unary multiplication
- Reorder subgoals to avoid left recursion
- Reorder arguments to avoid infinite solution set
- Handle boundary cases separately

Binary addition

Binary multiplication
- Bound intermediate shape by surrounding arguments

Unary addition is append

```
add([],X,X).
add([u|X],Y,[u|Z]) :- add(X,Y,Z).
```

Unary addition is append

```
add([],X,X).
add([u|X],Y,[u|Z]) :- add(X,Y,Z).
```

```
?- add(X,Y,[u,u,u,u,u]).
   X = []           , Y = [u,u,u,u,u] ;
   X = [u]          , Y = [u,u,u,u]   ;
   X = [u,u]        , Y = [u,u,u]     ;
   X = [u,u,u]      , Y = [u,u]       ;
   X = [u,u,u,u]    , Y = [u]         ;
   X = [u,u,u,u,u], Y = []            ; fail.
```

Unary addition is append

```
add([],X,X).
add([u|X],Y,[u|Z]) :- add(X,Y,Z).
```

```
?- add(X,Y,[u,u,u,u,u]).
   X = []           , Y = [u,u,u,u,u] ;
   X = [u]          , Y = [u,u,u,u]   ;
   X = [u,u]        , Y = [u,u,u]     ;
   X = [u,u,u]      , Y = [u,u]       ;
   X = [u,u,u,u]    , Y = [u]         ;
   X = [u,u,u,u,u], Y = []            ; fail.
```

```
?- add([u],Y,Z).   Z = [u|Y].
```

Unary addition is append

```
add([],X,X).
add([u|X],Y,[u|Z]) :- add(X,Y,Z).
```

```
?- add(X,Y,[u,u,u,u,u]).
   X = []          , Y = [u,u,u,u,u] ;
   X = [u]         , Y = [u,u,u,u]   ;
   X = [u,u]       , Y = [u,u,u]     ;
   X = [u,u,u]     , Y = [u,u]       ;
   X = [u,u,u,u]   , Y = [u]         ;
   X = [u,u,u,u,u], Y = []           ; fail.
```

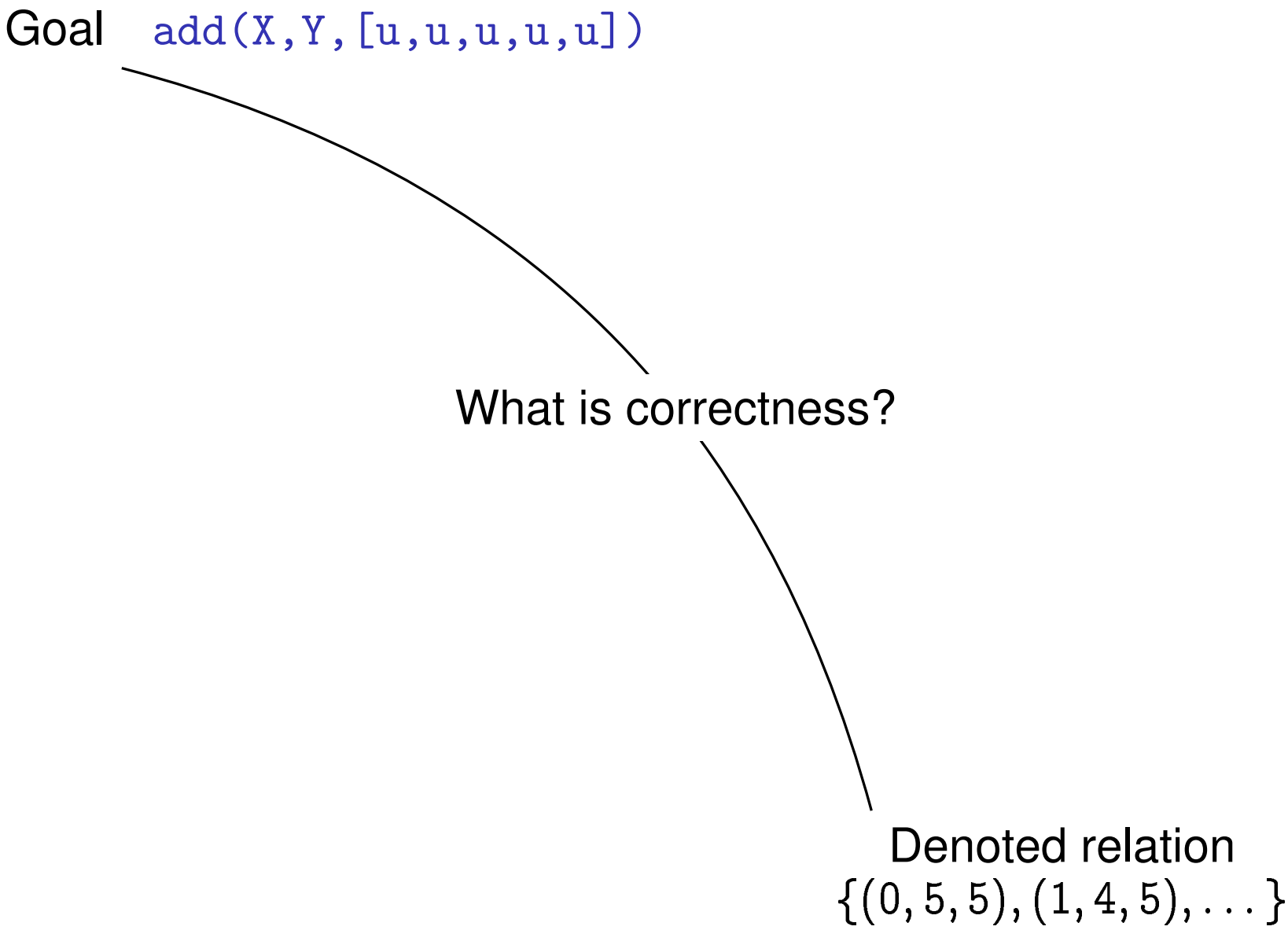
```
?- add([u],Y,Z).   Z = [u|Y].
```

```
?- add(X,[u],Z).   X = []    , Z = [u]     ;
                   X = [u]   , Z = [u,u]   ;
                   X = [u,u], Z = [u,u,u] ; ...
```

Unary addition is append

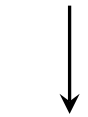
```
add([],X,X).
add([u|X],Y,[u|Z]) :- add(X,Y,Z).

?- add(X,Y,[u,u,u,u,u]).
   X = []          , Y = [u,u,u,u,u] ;
   X = [u]         , Y = [u,u,u,u]   ;
   X = [u,u]       , Y = [u,u,u]     ;
   X = [u,u,u]     , Y = [u,u]       ;
   X = [u,u,u,u]   , Y = [u]         ;
   X = [u,u,u,u,u], Y = []           ; fail.

?- add([u],Y,Z).   Z = [u|Y].

?- add(X,[u],Z).   X = []    , Z = [u]     ;
                   X = [u]   , Z = [u,u]   ;
                   X = [u,u], Z = [u,u,u] ; ...

?- add(X,[u],X).   % Diverges
```

Solution sets



Solution sets

Goal `add(X,Y,[u,u,u,u,u])`

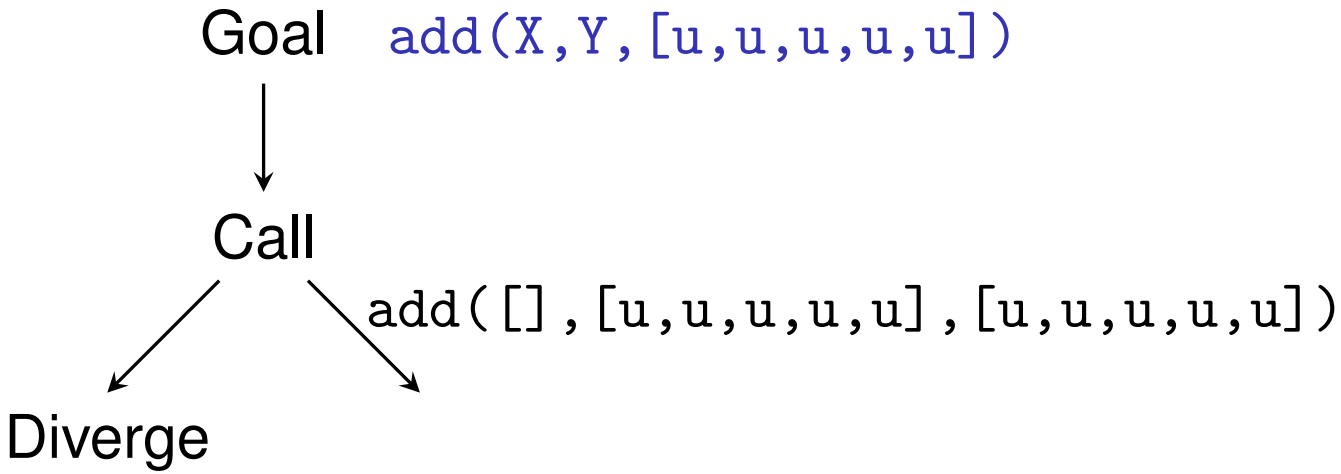


Call

Denoted relation

$\{(0,5,5),(1,4,5),\ldots\}$

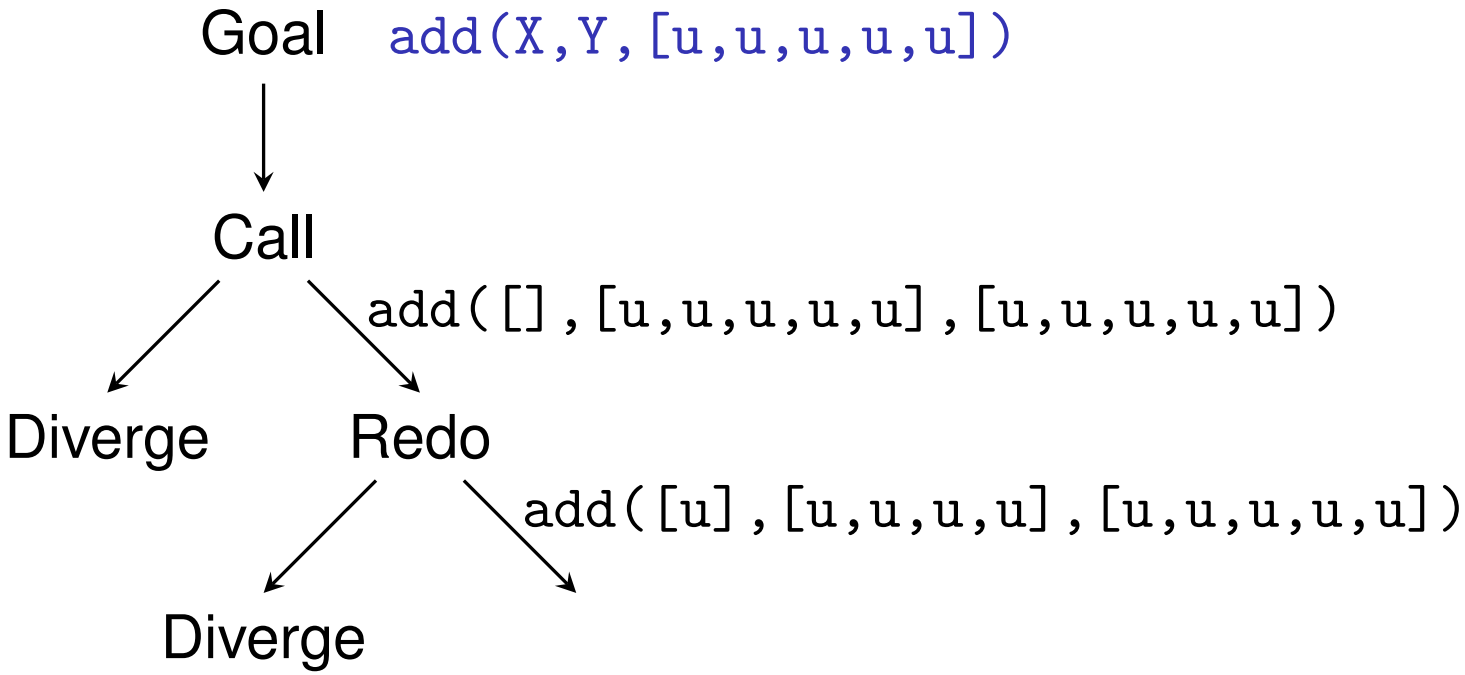
Solution sets



Denoted relation

$\{(0,5,5),(1,4,5),\ldots\}$

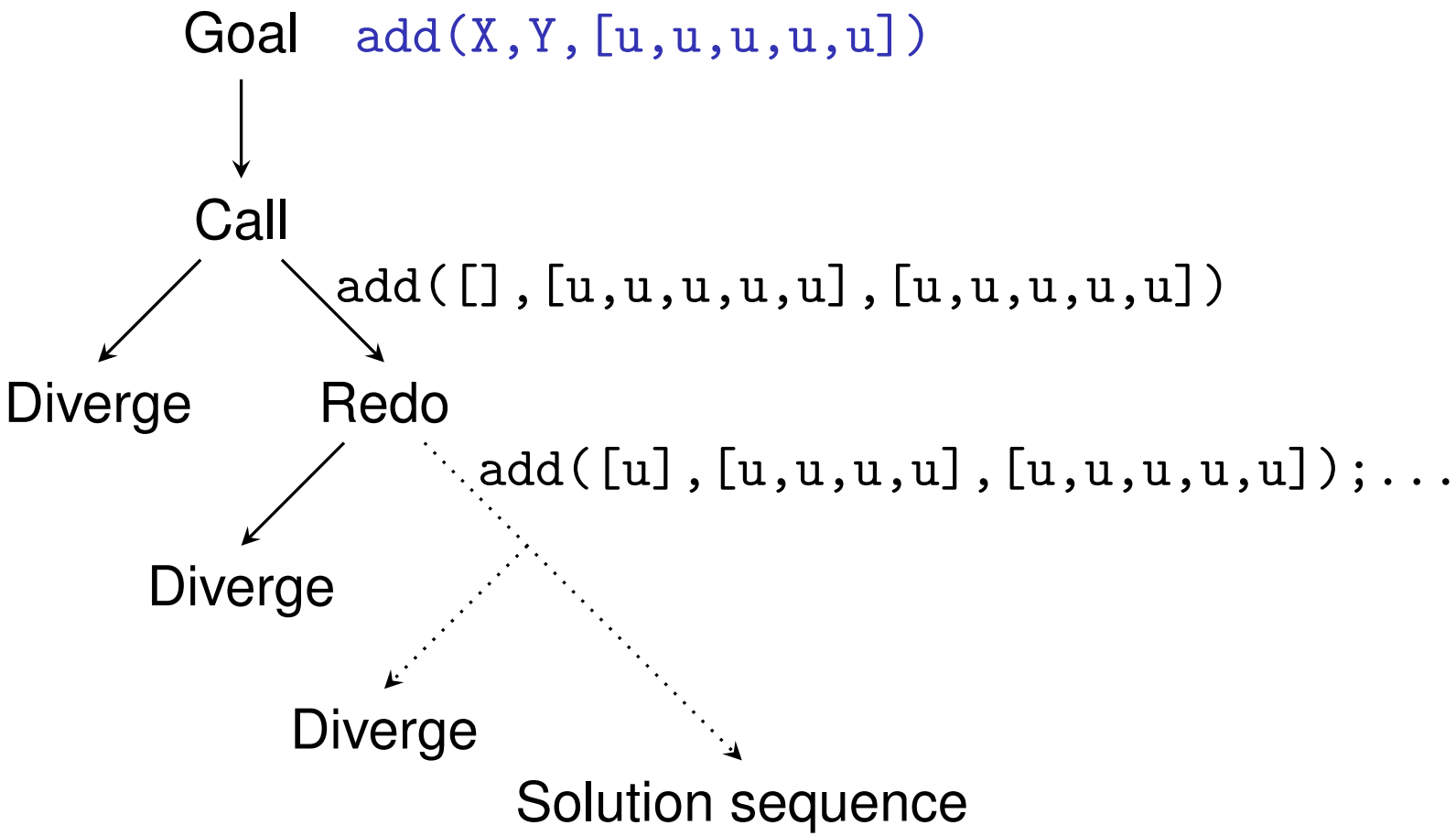
Solution sets



Denoted relation

$\{(0,5,5),(1,4,5),\ldots\}$

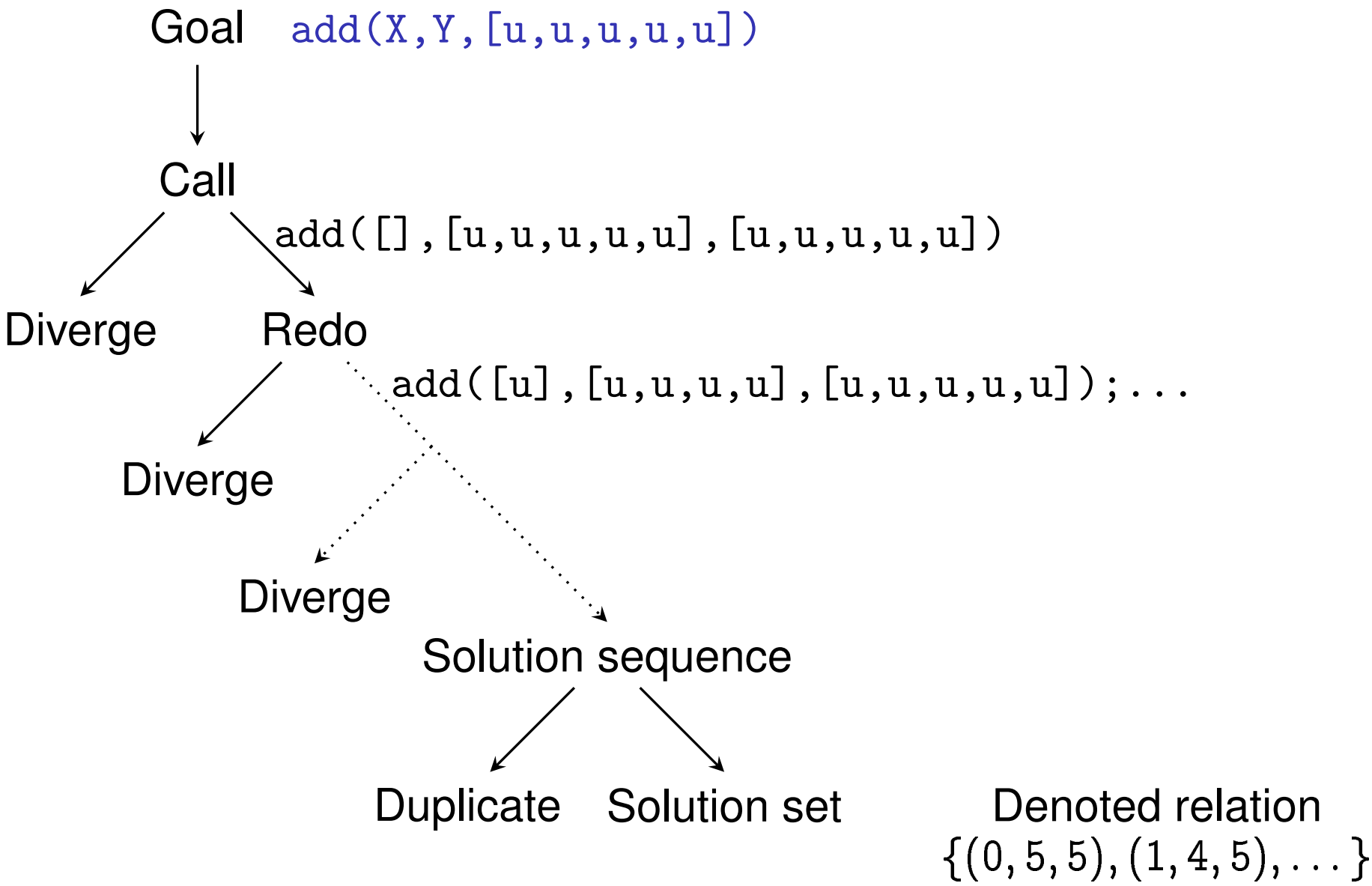
Solution sets



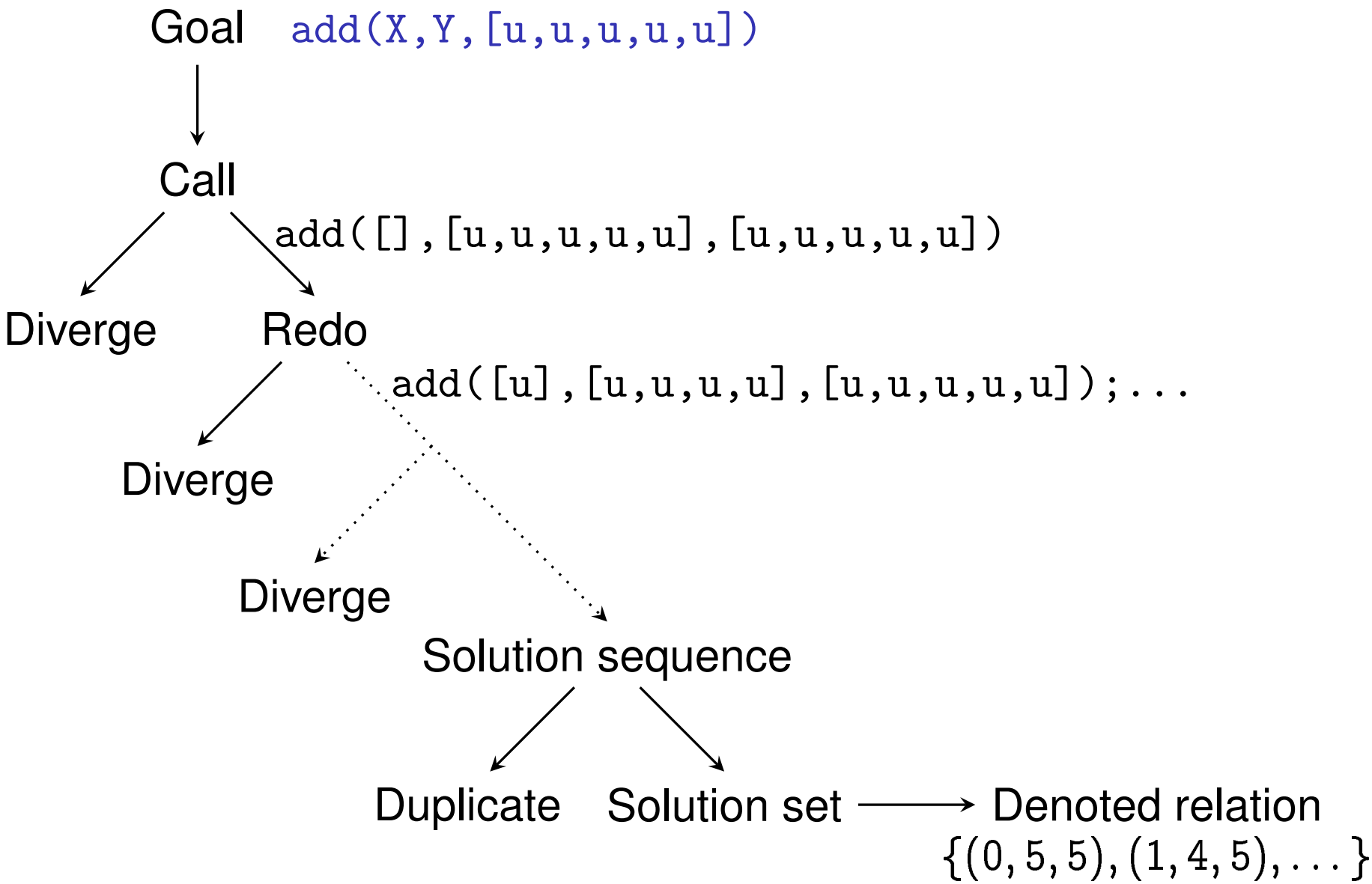
Denoted relation

$\{(0,5,5),(1,4,5),\ldots\}$

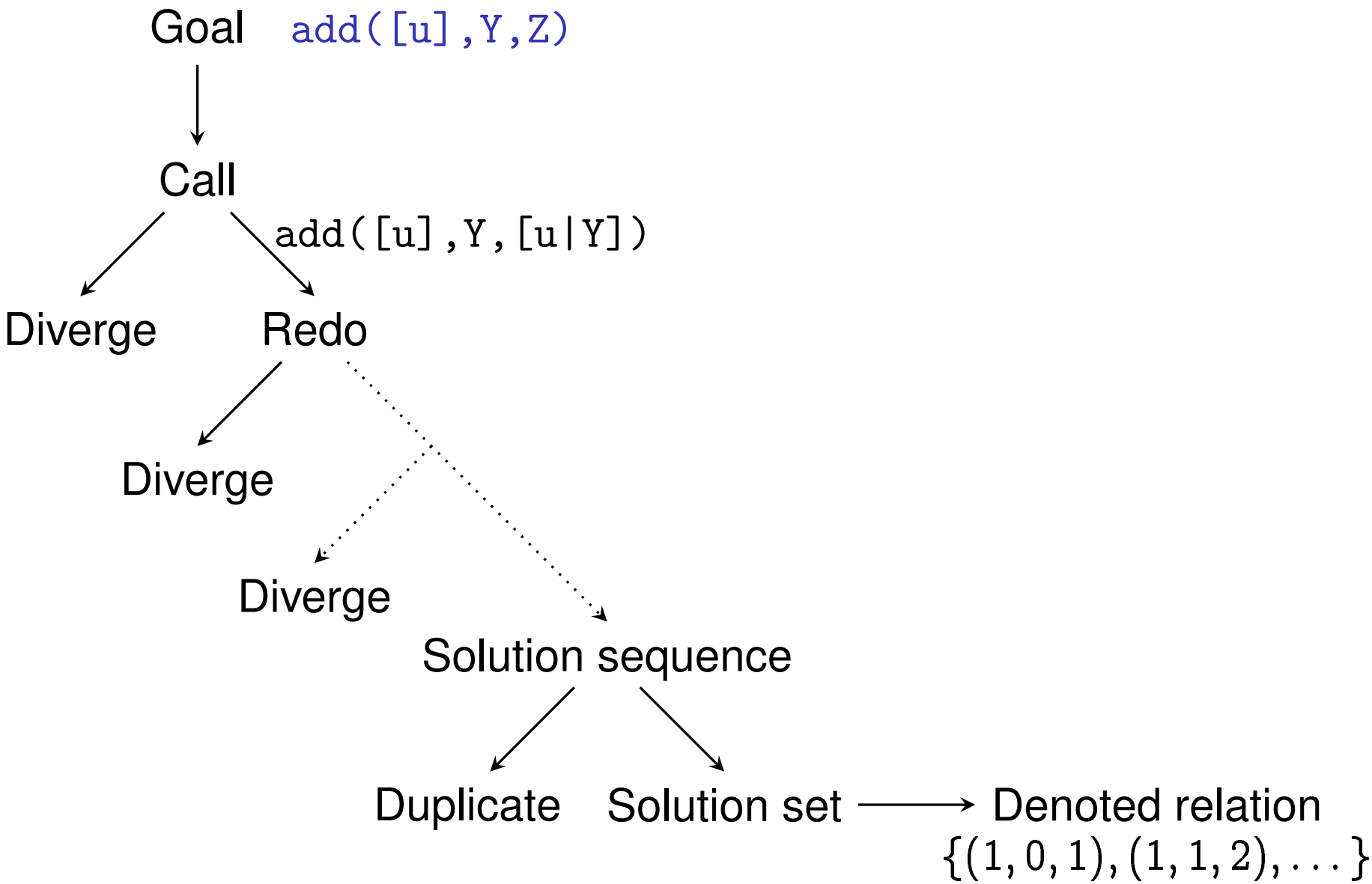
Solution sets



Solution sets



Solution sets



What? More precisely

Any goal that calls one of our arithmetic predicates denotes the correct relation.

What? More precisely

Any goal that calls one of our arithmetic predicates
with no sharing among its arguments
denotes the correct relation.

What? More precisely

Any goal that calls one of our arithmetic predicates
with no sharing among its arguments
denotes the correct relation.

What? More precisely

Any goal that calls one of our arithmetic predicates
with no sharing among its arguments
denotes the correct relation under SLD with interleaving.

How?

Unary addition

Prove properties of *solution sets*

No sharing or conjoined goals

- **Unary multiplication**
  - Reorder subgoals to avoid left recursion
  - Reorder arguments to avoid infinite solution set
  - Handle boundary cases separately

Binary addition

Binary multiplication

Bound intermediate shape by surrounding arguments

First try: direct encoding

```
mul([],_,[]).

mul([u|X],Y,Z) :- mul(X,Y,Z1), add(Z1,Y,Z).
```

```
?- mul(3,2,Z).    Z = 6.

?- mul(X,2,Z).    X = 0, Z = 0 ; X = 1, Z = 2 ;
                  X = 2, Z = 4 ; X = 3, Z = 6 ; ...
```

First try: direct encoding

```
mul([],_,[]).

mul([u|X],Y,Z) :- mul(X,Y,Z1), add(Z1,Y,Z).
```

```
?- mul(3,2,Z).    Z = 6.

?- mul(X,2,Z).    X = 0, Z = 0 ; X = 1, Z = 2 ;
                  X = 2, Z = 4 ; X = 3, Z = 6 ; ...

?- mul(X,2,6).    X = 3 ; % Diverges

?- mul(X,2,5).    % Diverges
```

Reorder subgoals to avoid left recursion (let Z bound Y)

```
mul([],_,[]).

mul([u|X],Y,Z) :- mul(X,Y,Z1), add(Z1,Y,Z).
```

```
?- mul(3,2,Z).    Z = 6.

?- mul(X,2,Z).    X = 0, Z = 0 ; X = 1, Z = 2 ;
                  X = 2, Z = 4 ; X = 3, Z = 6 ; ...

?- mul(X,2,6).    X = 3 ; % Diverges

?- mul(X,2,5).    % Diverges
```

Reorder subgoals to avoid left recursion (let Z bound Y)

```
mul([],_,[]).

mul([u|X],Y,Z) :- add(Z1,Y,Z), mul(X,Y,Z1).
```

```
?- mul(3,2,Z).
```

```
?- mul(X,2,Z).    X = 0, Z = 0 ; X = 1, Z = 2 ;
                  X = 2, Z = 4 ; X = 3, Z = 6 ; ...
```

```
?- mul(X,2,6).
```

```
?- mul(X,2,5).
```

Reorder subgoals to avoid left recursion (let Z bound Y)

```
mul([],_,[]).

mul([u|X],Y,Z) :- add(Z1,Y,Z), mul(X,Y,Z1).
```

```
?- mul(3,2,Z).
```

```
?- mul(X,2,Z).    X = 0, Z = 0 ; X = 1, Z = 2 ;
                  X = 2, Z = 4 ; X = 3, Z = 6 ; ...
```

```
?- mul(X,2,6).    X = 3 ; fail.
```

```
?- mul(X,2,5).    fail.
```

Reorder arguments to avoid infinite solution set

```
mul([],_,[]).

mul([u|X],Y,Z) :- add(Z1,Y,Z), mul(X,Y,Z1).
```

```
?- mul(3,2,Z).    Z=6 ; % Diverges
?- mul(X,2,Z).    X=0, Z=0 ; X=1, Z=2 ;
                  X=2, Z=4 ; X=3, Z=6 ; ...
?- mul(X,2,6).    X=3 ; fail.
?- mul(X,2,5).    fail.
```

Reorder arguments to avoid infinite solution set

```
mul([],_,[]).

mul([u|X],Y,Z) :- add(Y,Z1,Z), mul(X,Y,Z1).
```

```
?- mul(3,2,Z).    Z = 6.

?- mul(X,2,Z).    X = 0, Z = 0 ; X = 1, Z = 2 ;
                  X = 2, Z = 4 ; X = 3, Z = 6 ; ...

?- mul(X,2,6).    X = 3 ; fail.

?- mul(X,2,5).    fail.
```

Handle boundary cases separately

```
mul([],_,[]).

mul([u|X],Y,Z) :- add(Y,Z1,Z), mul(X,Y,Z1).
```

```
?- mul(3,2,Z).    Z=6.

?- mul(X,2,Z).    X=0, Z=0 ; X=1, Z=2 ;
                  X=2, Z=4 ; X=3, Z=6 ; ...

?- mul(X,2,6).    X=3 ; fail.

?- mul(X,2,5).    fail.

?- mul(X,Y,6).    % Diverges
```

Handle boundary cases separately

```
mul([],_,[]).
mul([u|X],[],[]).
mul([u|X],[u|Y],Z) :- add([u|Y],Z1,Z), mul(X,[u|Y],Z1).
```

```
?- mul(3,2,Z).    Z=6.

?- mul(X,2,Z).    X=0, Z=0 ; X=1, Z=2 ;
                  X=2, Z=4 ; X=3, Z=6 ; ...

?- mul(X,2,6).    X=3 ; fail.

?- mul(X,2,5).    fail.

?- mul(X,Y,6).    X=6, Y=1 ; X=3, Y=2 ;
                  X=2, Y=3 ; X=2, Y=3 ; fail.
```

How?

Unary addition

Prove properties of *solution sets*

No sharing or conjoined goals

Unary multiplication

Reorder subgoals to avoid left recursion

Reorder arguments to avoid infinite solution set

Handle boundary cases separately

- **Binary addition—Straightforward ripple-carry adder**

Binary multiplication

Bound intermediate shape by surrounding arguments

Binary representation

A list of bits (`0` or `1`) that does not end with `0`

```
pos([_|_]).
gtl([_,_|_]).
```

Binary representation

A list of bits (`0` or `1`) that does not end with `0`

```
pos([_|_]).
gtl([_,_|_]).
```

Binary representation

A list of bits (`0` or `1`) that does not end with `0`

```
pos([_|_]).
gtl([_,_|_]).
```

```
?- add(2,Y,Z).
```

Binary representation

A list of bits (`o` or `1`) that does not end with `o`

```
pos([_|_]).
gtl([_,_|_]).
```

```
?- add([u,u],Y,Z).
```

```
   Z = [u,u|Y].
```

Binary representation

A list of bits (`o` or `1`) that does not end with `o`

```
pos([_|_]).
gt1([_,_|_]).
```

```
?- add([o,1],Y,Z).
```

```
   Y = []           , Z = [o,1]       ;
   Y = [1]          , Z = [1,1]       ;
   Y = [o,1]        , Z = [o,o,1]     ;
   Y = [o,o,G|H]    , Z = [o,1,G|H]   ;
   Y = [o,1,1]      , Z = [o,o,o,1]   ;
   Y = [o,1,o,G|H]  , Z = [o,o,1,G|H] ; ...
```

How?

Unary addition

Prove properties of *solution sets*

No sharing or conjoined goals

Unary multiplication

Reorder subgoals to avoid left recursion

Reorder arguments to avoid infinite solution set

Handle boundary cases separately

Binary addition—Straightforward ripple-carry adder

- **Binary multiplication**

  Bound intermediate shape by surrounding arguments

Bound intermediate shape by surrounding arguments

```
mul([]    , Y , []    ).
mul(X     , [], []    ) :- pos(X).
mul([1]   , Y , Y     ) :- pos(Y).
mul([o|X], Y , [o|Z]) :- pos(Y), pos(X), pos(Z),
                         mul(X,Y,Z).
mul([1|X], Y , Z     ) :- pos(Y), pos(X), gt1(Z),
                         mul(X,Y,Z1), add([o|Z1],Y,Z).
```

Bound intermediate shape by surrounding arguments

```
mul([]   , Y , []   ).
mul(X    , [], []   ) :- pos(X).
mul([1]  , Y , Y    ) :- pos(Y).
mul([o|X], Y , [o|Z]) :- pos(Y), pos(X), pos(Z),
                         mul(X,Y,Z).
mul([1|X], Y , Z    ) :- pos(Y), pos(X), gtl(Z),
                         mul(X,Y,Z1), add([o|Z1],Y,Z).
```

Bound intermediate shape by surrounding arguments

```
mul([]   , Y , []   ).
mul(X    , [], []   ) :- pos(X).
mul([1]  , Y , Y    ) :- pos(Y).
mul([o|X], Y , [o|Z]) :- pos(Y), pos(X), pos(Z),
                         mul(X,Y,Z).
mul([1|X], Y , Z    ) :- pos(Y), pos(X), gt1(Z),
                         less13(Z1,Z,[1|X],Y),
                         mul(X,Y,Z1), add([o|Z1],Y,Z).
```

Bound intermediate shape by surrounding arguments

```
mul([]    , Y , []    ).
mul(X     , [], []    ) :- pos(X).
mul([1]   , Y , Y     ) :- pos(Y).
mul([o|X], Y , [o|Z]) :- pos(Y), pos(X), pos(Z),
                         mul(X,Y,Z).
mul([1|X], Y , Z     ) :- pos(Y), pos(X), gtl(Z),
                         less13(Z1,Z,[1|X],Y),
                         mul(X,Y,Z1), add([o|Z1],Y,Z).
```

Bound `Z1` if either `Z` is bounded or `X` and `Y` are bounded.

$$\texttt{less13}(Z_1, Z, X, Y) \iff |Z_1| < |Z| \ \wedge \ |Z_1| < |X| + |Y| + 1$$

```
less13([]    ,[_|_],_    ,_    ).
less13([_|Z1],[_|Z],[]   ,[_|Y]) :- less13(Z1,Z,[],Y).
less13([_|Z1],[_|Z],[_|X],Y    ) :- less13(Z1,Z,X ,Y).
```

Conclusion

"If patriotism doesn't move you,
consider the intellectual adventure involved."

Conclusion

> “If patriotism doesn’t move you,
> consider the intellectual adventure involved.”
> “What progress?
> What can you do beyond multiplication?”
>
> —*The Feeling of Power*

More predicates

- Division with remainder: long division bottom-up
- Exponentiation and logarithm with remainder

Related work

- Declaring numbers in Curry (Braßel, Fischer, & Huch)
- Haskell types for static guarantees (Hallgren; Kiselyov)

Relational predicates! Correctness proofs!