Project 1 Part II: Metaproduct Primes

- What you know
  - Basic BDD data structure and JAVA implementation
  - A little bit about these things called "metaproducts"
- What you don't know
  - All the tricks with metaproducts
  - Using these to do Prime Implicants

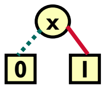
Copyright Notice

About Metaproducts

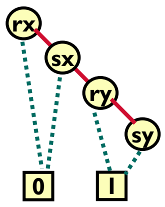
- Notation was created to support applications where we need to preserve the structure of things like SOP expressions
  - ...ie, if you really WANT to write x + x'
  - ...and you want to represent and manipulate it as a BDD, what to do?
- Metaproduct notation
  - Replace each variable "x" with a pair (rx, sx)
  - If you see x in a product, then you get (rx)(sx) in metaproduct
  - If you see x' in a product, then you get (rx)(sx') in metaproduct
  - If you don't see any x or x' at all, then you get (rx') in metaproduct
- In English
  - rx is the occurrence variable -> rx==1 says "x is here", rx==0 says "no x"
  - sx is the sign variable -> sx==1 says "x is positive", sx==0 "negative"

Metaproduct Example

- Suppose F(x,y) = x + xy'
  - This is really just == x, of course
  - Its BDD would be simply

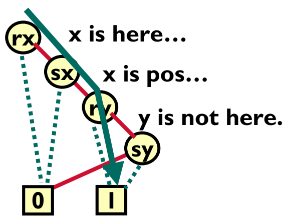


- As a metaproduct
  - Assume var ordering was x < y
  - Then new ordering is x < rx < sx < y < rx < sy
  - x becomes (rx)(sx)(ry')
  - xy' becomes (rx)(sx)(ry)(sy')

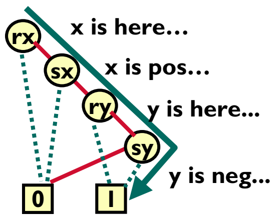


Metaproduct Example

- You interpret this by looking at "satisfying paths" to "1" node
  - There are 2 paths from root to "1", each makes a product term



Result: x



Result: xy'

  - Put the final products together for final answer: x + x'y

Metaproduct Example

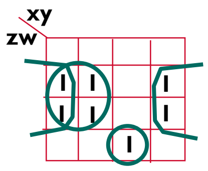
- What happens if a variable is not present?
  - We already saw this, but its worth noting
  - In F(x,y) = x + xy', consider "x" term
  - There's no "y" in there, but we still have to deal with "y"
  - Rule is: if there's no variable in there, you still have to include the occurrence variables for the missing original vars, but you include them in the negative polarity to note their absence
- So, term "x" becomes (rx)(sx)(ry')

Means "no y vars in here"

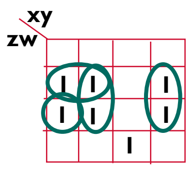
- If this was F(x,y,z,w) =x, what would happen?
  - We'd get (rx)(sx)(ry')(rz')(rw')

Metaproduct Primes

- Turns out you can represent Prime Implicants with metaproduct notation



These are all Primes – biggest product terms you can circle in a Kmap for your function



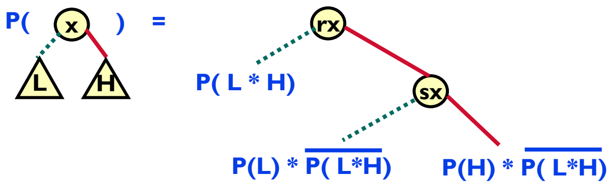
These are NOT all Primes – not all biggest product terms you can Circle in this Kmap

Metaproduct Primes

- Like with Boolean functions, problem is SIZE
  - **A function can have many many primes – way too many enumerate one by one**
  - **This is why representing them with something like a BDD is attractive, since its good at "compressing" Boolean functions**
  - **But this is also why we need a special notation, since we DON'T want the BDD to CHANGE our function to its canonical form**
  - **We need to represent it in some SOP form**
- Biq question: how do we find Primes using metaproducts
  - **Of course, it's gotta be something recursive, right…?**

Metaproduct Primes

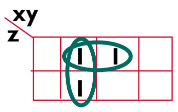
- There's another Shannon-style recursive decomposition
  - You start with a BDD in your original variables
  - You end up with a BDD in the (occurrence, sign) metaproduct variables
  - Final BDD represents, as SOP form, ALL the primes
- Basic decomposition
  - Let P(BDD root of F) = metaproduct BDD for all PRIMES in function F



P(x) =
L
H
rx
P(L * H)
sx
P(L) * $\overline{\text{P(L*H)}}$
P(H) * $\overline{\text{P(L*H)}}$

Metaproduct Primes

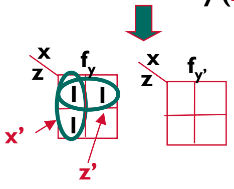
- …why does that work?
  - Roughly speaking –this is just the messy check for how to "reassemble" primes when they get split up during a Shannon decomposition



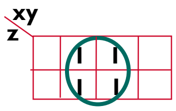
f(x,y,z) = x'y + yz' = 2 primes

= y f_y + y' $f_{y'}$

= y (x'+z') + y' (0)



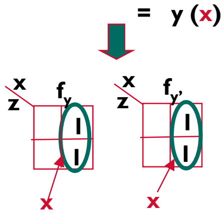
If we cofactor on y, the Kmaps for f_y , $f_{y'}$ are shown at left. This is what happens to our 2 primes; new question is how to **interpret** these new pieces.



f(x,y,z) = x = 1 prime

= y f_y + y' $f_{y'}$

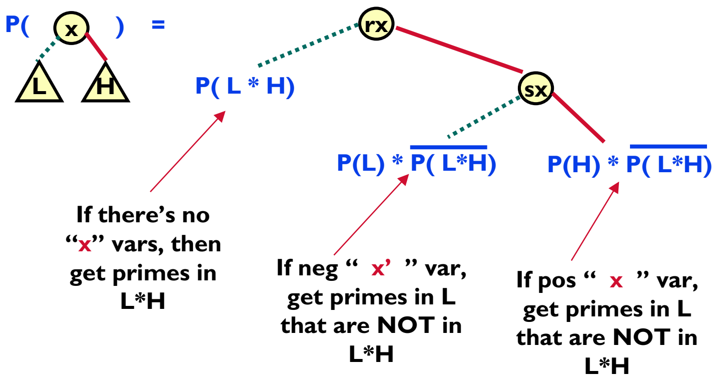
= y (x) + y' (x)



Cofactor on y, Kmaps for f_y , $f_{y'}$ shown at left. Issue is to interpret to reassemble primes

Metaproduct Primes

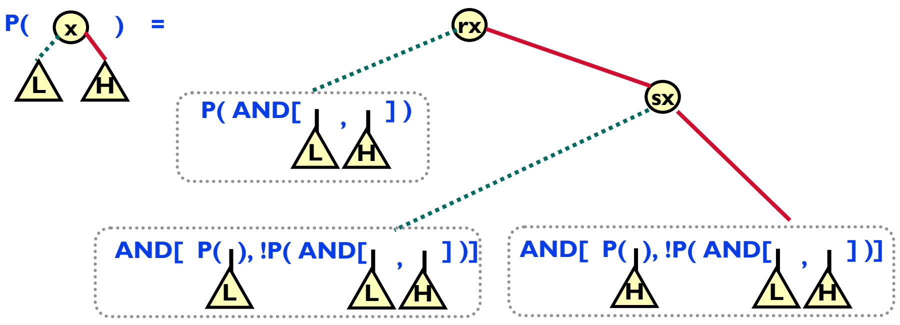
- “Reading” the decomposition



P(x) =
L
H
rx
sx
P(L * H)
P(L) * $\overline{\text{P(L*H)}}$
P(H) * $\overline{\text{P(L*H)}}$
If there’s no “x” vars, then get primes in L*H
If neg “ x’ ” var, get primes in L that are NOT in L*H
If pos “ x ” var, get primes in L that are NOT in L*H

Metaproduct Primes

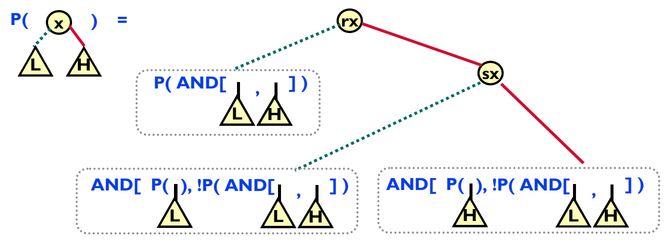
- Let’s be a little more careful on the details of BDDs and ops
  - Assumes you have AND and NOT (ie, “!”) on BDDs
  - Assumes P( ) calculated just like *ITE,* as a top-down recursion
  - Assume var order is fixed: for varys x,y,..., its: x < rx < sx < y < ry < sy ....



P(x) =
L
H
rx
sx
P(AND[L , H])
AND[P(L), !P(AND[L , H])]
AND[P(H), !P(AND[L , H])]

Metaproduct Primes: Termination

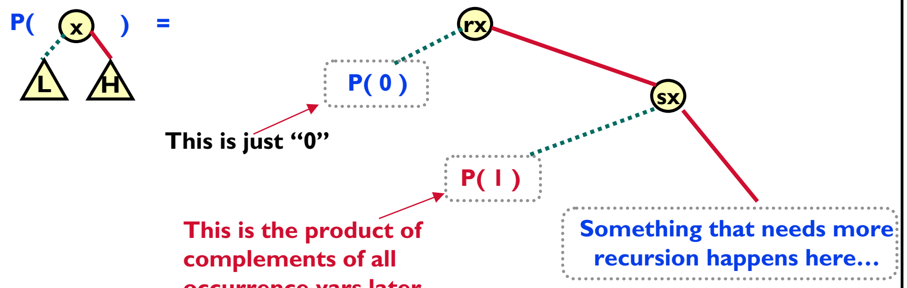
- So we know the recursion is:



- …next question: what are the termination conditions for P( )?
  - So, when can we quit, and return a known BDD node answer?
  - Easy case: P(0) = 0
  - Harder case: P(1) = a little messy…

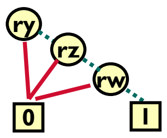
Metaproduct Primes: Termination

- Suppose vars are: x,y,z,w, and we have this recursion



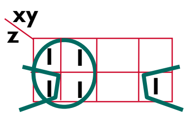
This is the product of complements of all occurrence vars later in the order, ie, after sx

In this case: (ry')(rz')(rw') =

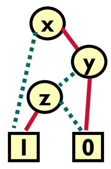


- Intuition
  - P(0) means "you're done – nothing more at all this prime term"
  - P(1) means "you're done – but remember that these vars are absent"

Primes Example

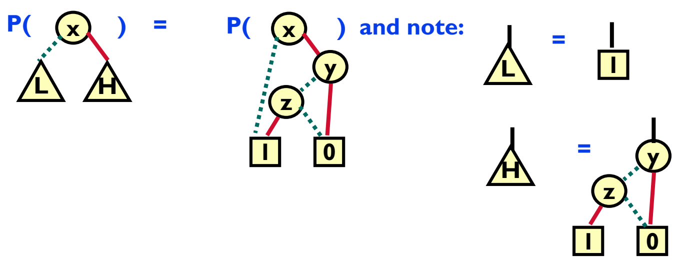


f(x,y,z) = x' + y'z = 2 primes



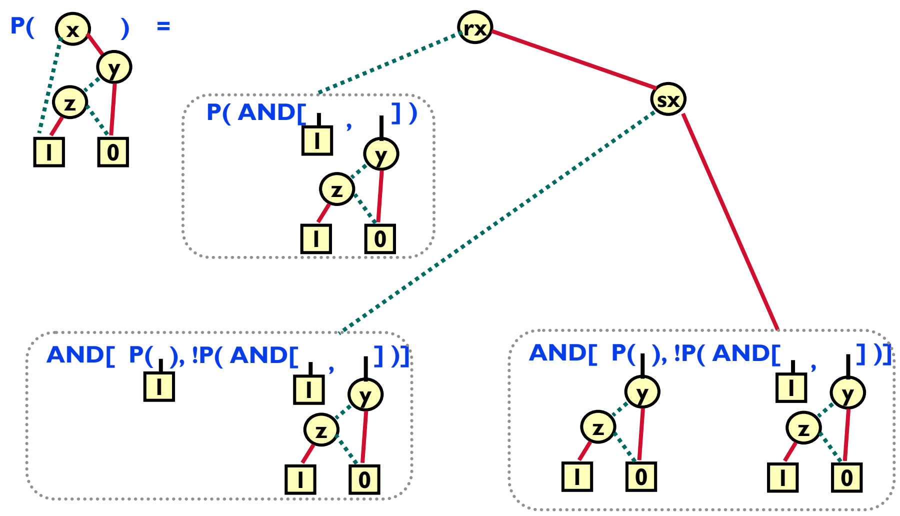
Ordinary BDD with var order: x < y < z

Apply recursion at root of f()



Primes Example

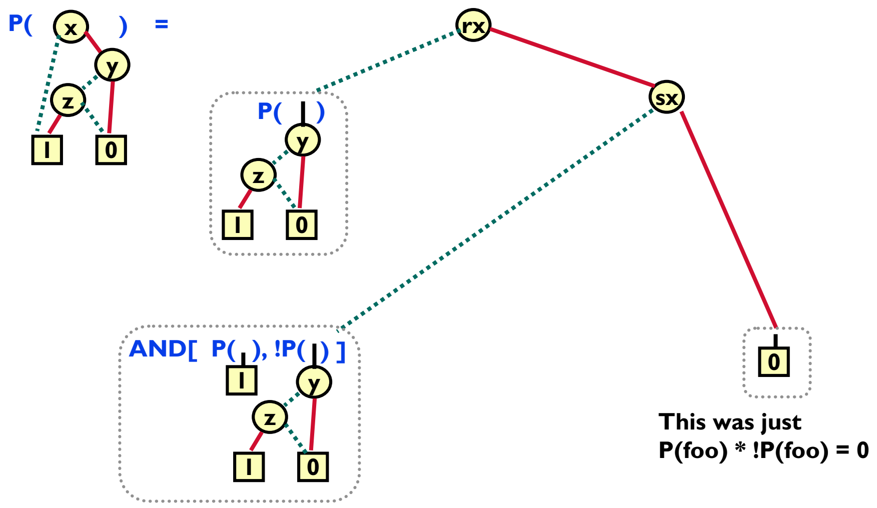
Applying recursion at root of f()



P(x y z 1 0) =
rx
sx
P(AND[1 , y z 1 0])
AND[P(1), !P(AND[1 , y z 1 0])]
AND[P(y z 1 0), !P(AND[1 , y z 1 0])]

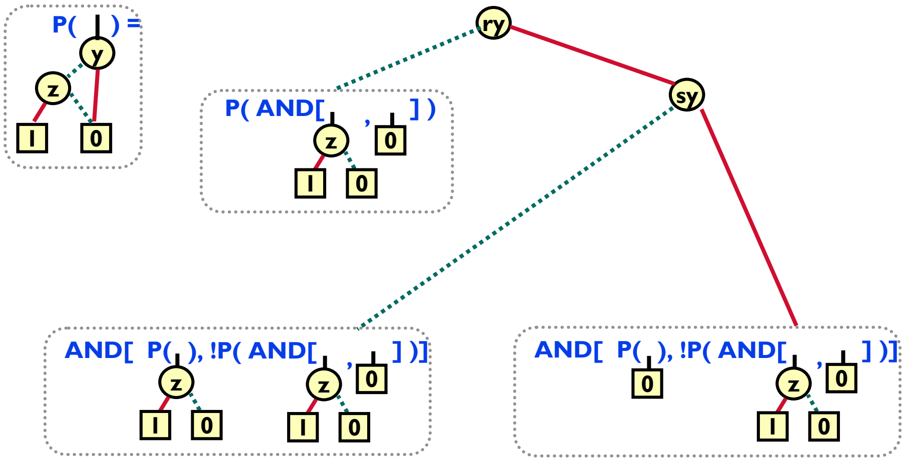
Primes Example

Do the obvious simplifications now (just to simplify for this manual example)



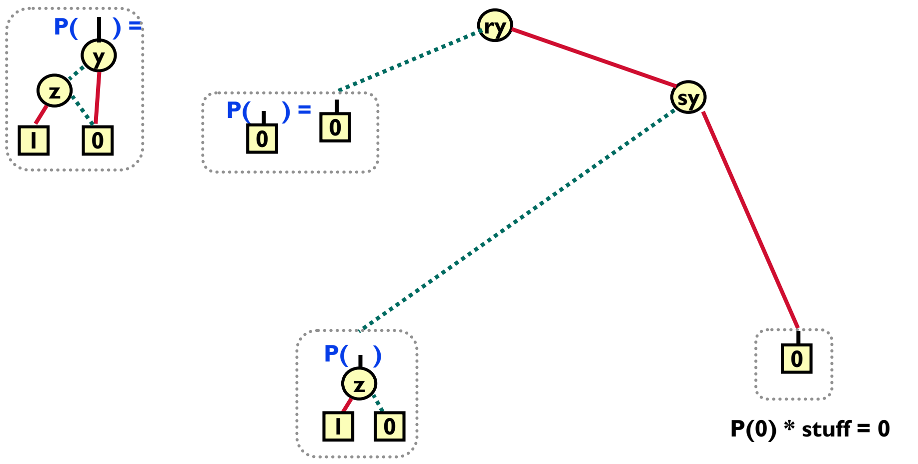
Prime Example

- OK, we need to do this one next



Prime Example

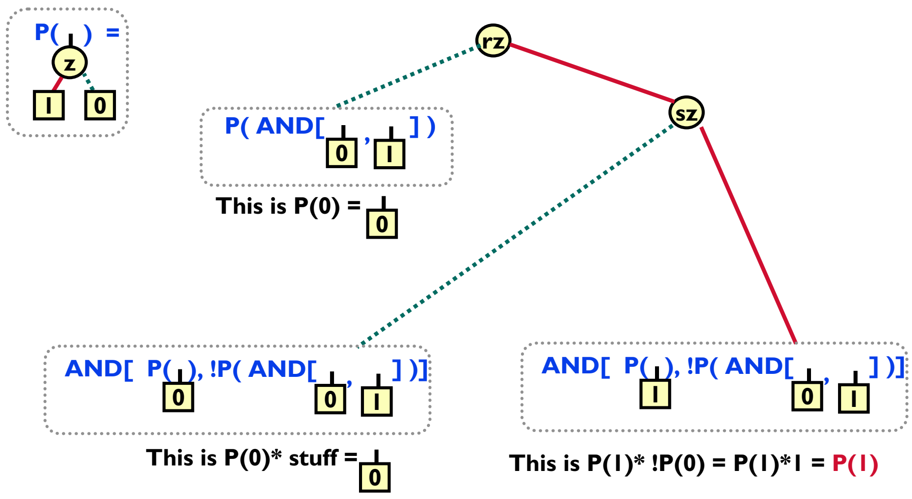
- Again, do obvious simplifications (just for this manual ex)



P(0) * stuff = 0

Prime Example

- We have to do this one next – and its easy…



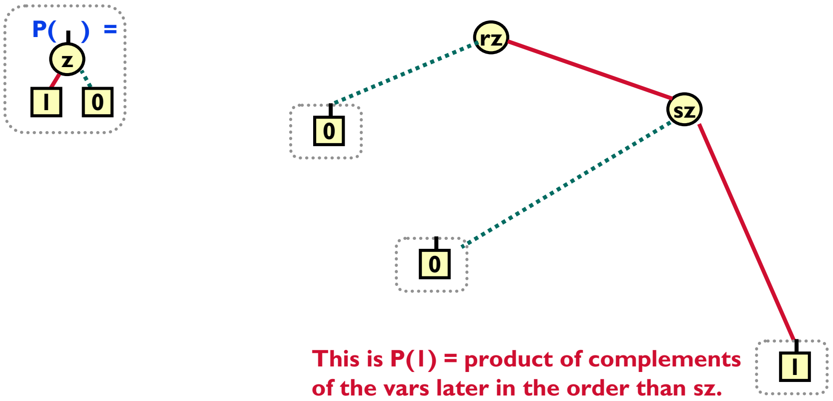
This is P(0) = 0

This is P(0)* stuff = 0

This is P(1)* !P(0) = P(1)*1 = P(1)

Prime Example

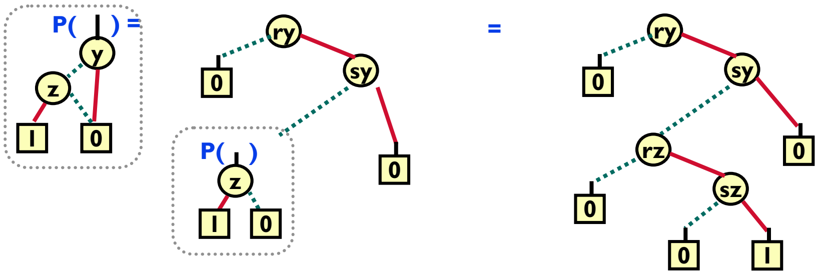
- We have to do this one next – and its easy…



This is P(1) = product of complements of the vars later in the order than sz. Since sz is the LAST var in the order, the rule is: this is just "1"

Prime Example

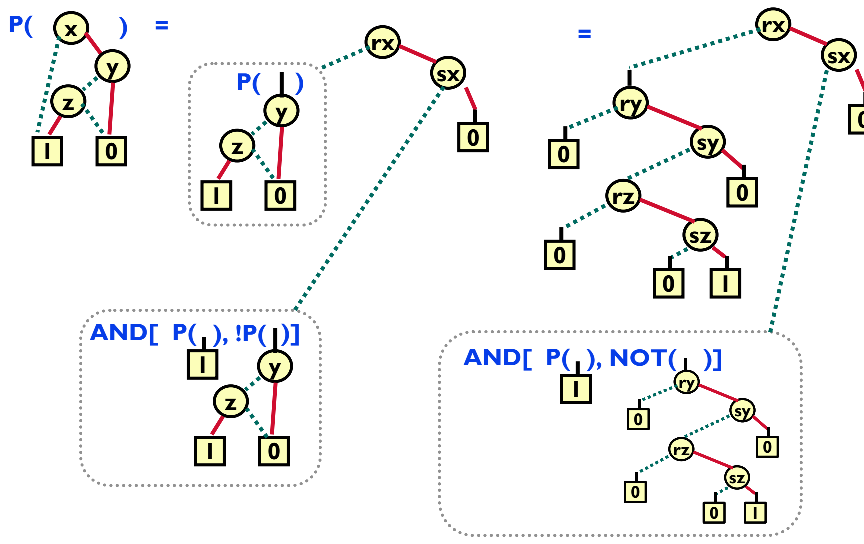
- Return results up recursive call tree…



Note – I'm leaving in all the separate "0" and "1" nodes just to simplify the drawing – it's a REAL BDD, there's only a single "1" and a single "0"…

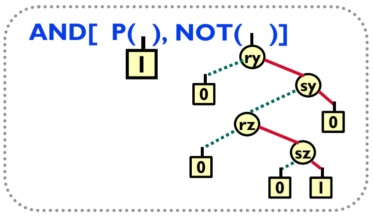
Prime Example

- Return results up recursive call tree…

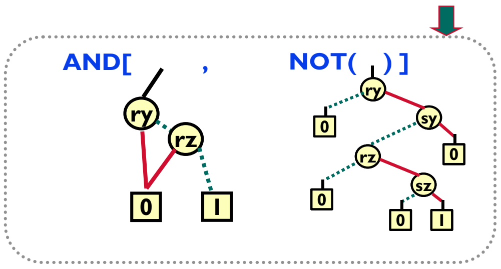


Prime Example

- This one is next to recurse on



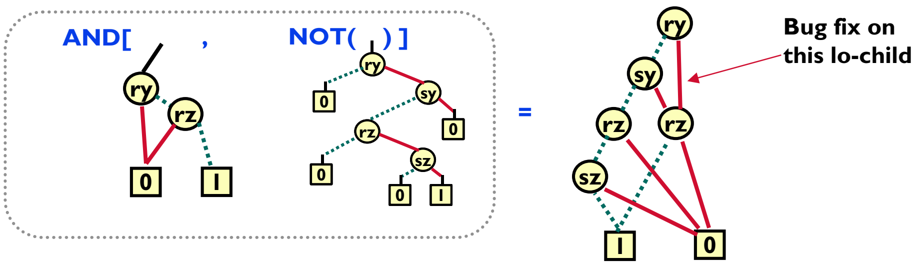
Since we know P(1) = product of complements of vars below sx in the order, this is supposed to be: (ry’)(rz’), so we get…



…which is just ordinary BDD ops

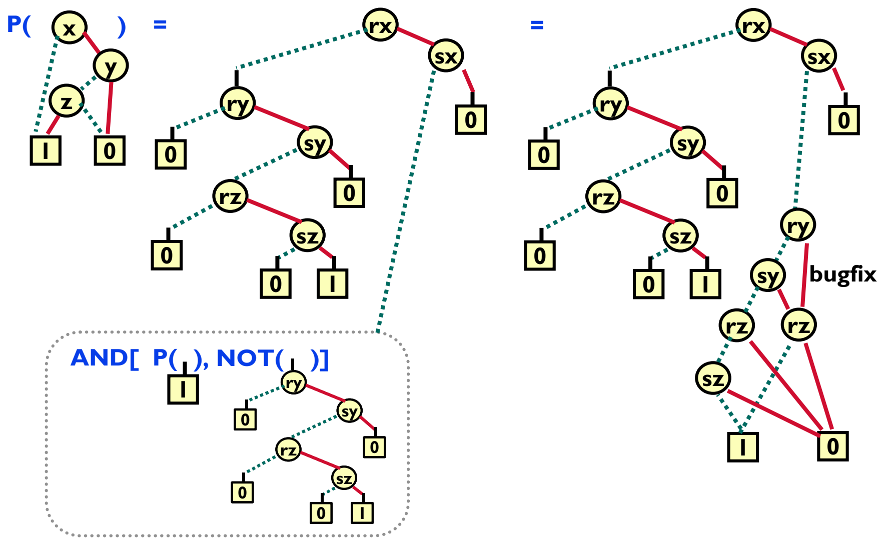
Prime Example

- BDD ops give this



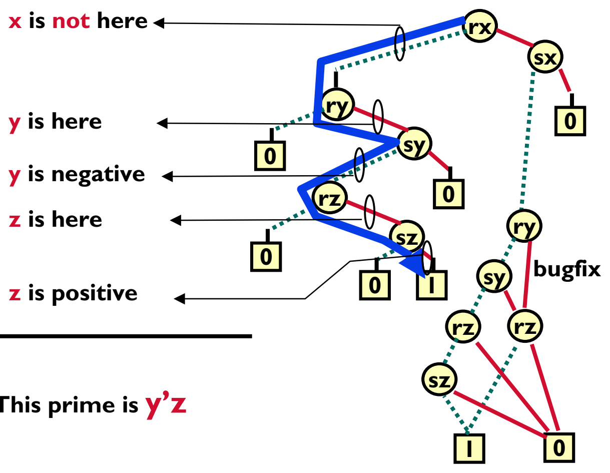
Prime Example

- Return results up recursive call tree…



..and, that's the Final Metaproduct for Prime( )

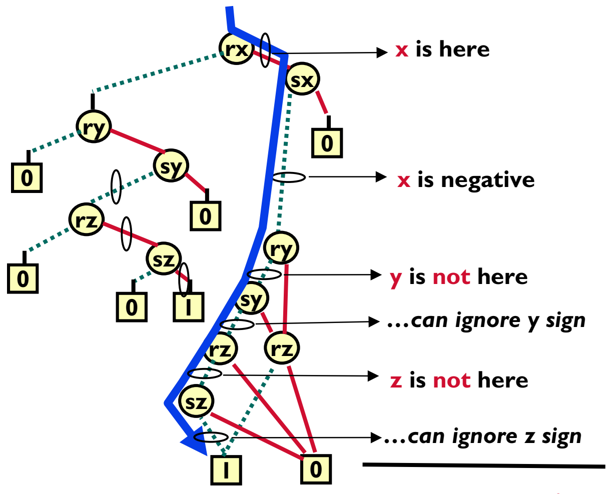
- Look for paths from root to "1"



This prime is y'z

Final Primes

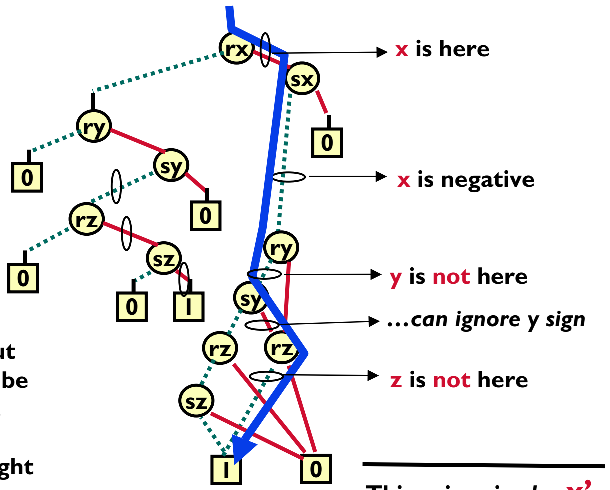
- More paths



This prime is x'

Final Primes

- …hey, there's another one…?

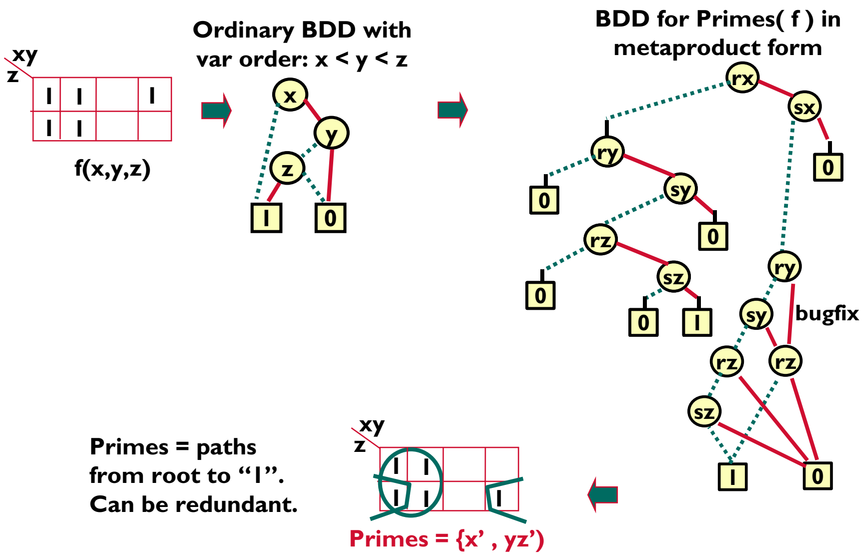


An unfortunate fact about metaproducts: they can be redundant about primes. You don't get the wrong ones – but you can get right ones several times.

This prime is *also* x'

Metaproduct Primes

- So what did we do?



Primes = paths from root to "I". Can be redundant.

…back to 18-760 Project 1 Part 2

- What do we want?
  - We want you to add P(BDD for function f) as an operator to your JAVA BDD package
  - Do it exactly like we showed here
    - Just like ITE: you descend the starting BDD for f, and you recursively "trace out" the BDD for P(f)
    - Assume you have all the vars defined in the right initial order. This means if the real vars are x, y, YOU have x, rx, sx, y, ry ,sy in order
  - You have 2 basic goals
    - To be able to transform a BDD for function f into Prime(f)
    - To print out some interesting "info" about these primes

Prime( ) Details

- Things to be careful about
  - Before doing anything, you probably want to build the function: (rx')(ry')(rz')…(rlast') for ALL your vars. And make an array of pointers to the nodes, so that when you need P(1) = product of complemented occurrence nodes below me – you can just look it up
  - You still need to call FindOrCreateNode()on the 2 new nodes you make. You want to build rx and its children first, call FindOrCreatNode(rx), then finish the recursion on rx, then call FindOrCreatNode(rx).
  - Do you want to do something like a different OPS table for the Prime computation? (It's not required…but think about it)
  - You will want to write a "printprime" routine that walks the paths to the "1" node, and prints out sensible product terms. DO NOT worry about the redundancy issue – not your problem.
  - You also want to build a "numprimes" routine that just prints out the number of paths to the "1" node. Think about it – you don't have to walk them all to do this, it's a very simple recursion if you know numprimes(hichild) and numprimes(lochild), and numprimes(1)=1 and numprimes(0)=0

Metaproduct Primes: Summary

- Interesting, sort of funky BDD application
  - Twists the usual interpretation of “canonical BDD form” around a lot
  - Works fine, a bit arcane
    - (This is a simplification of how people really do it. There are a bunch of other optimizations to get rid of those redundancies that make it a lot faster. Not worth the grief to go thru them all…they violate a lot of BDD rules.)
- For Project 1
  - Implement Prime(f)
  - Look on the /afs/ece/class/ee760/proj1 directory for more details, and for some info about benchmarks to run
  - Ask TA and Prof questions if there are any issues at all on this one