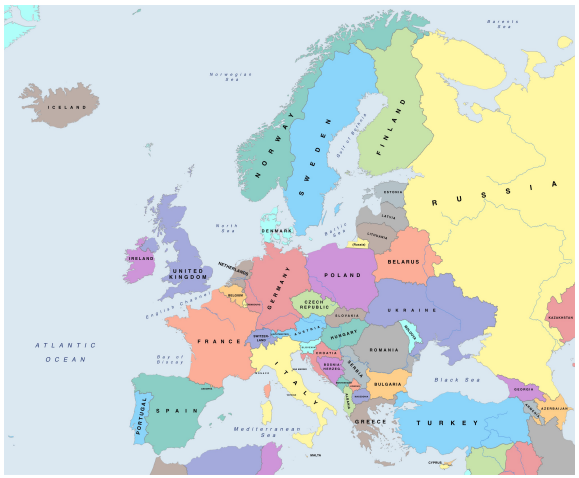
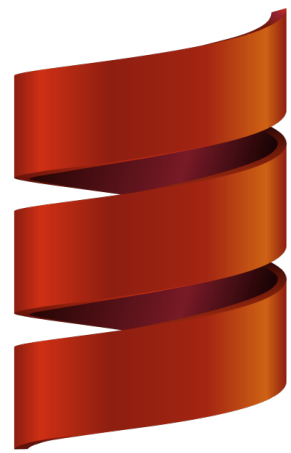
Coloring Maps with Binary Decision Diagrams in Scala

Jim Newton

EPITA Research Laboratory

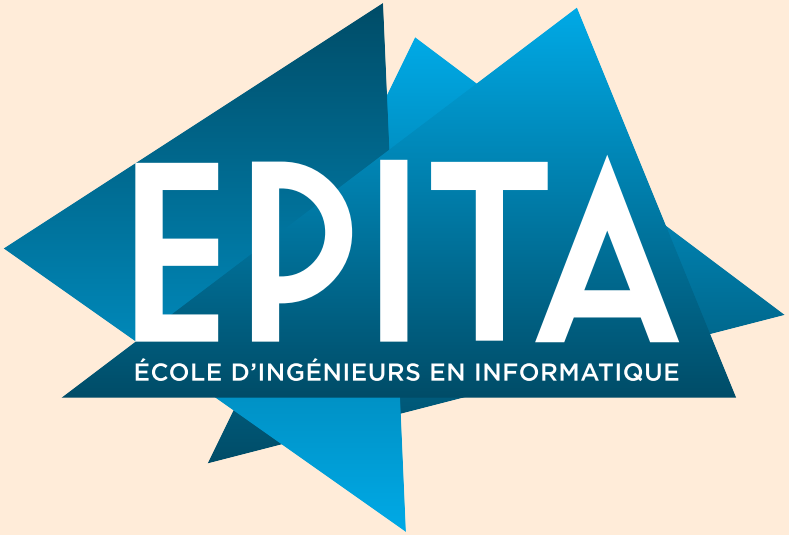
September 22, 2026





Who is Jim?

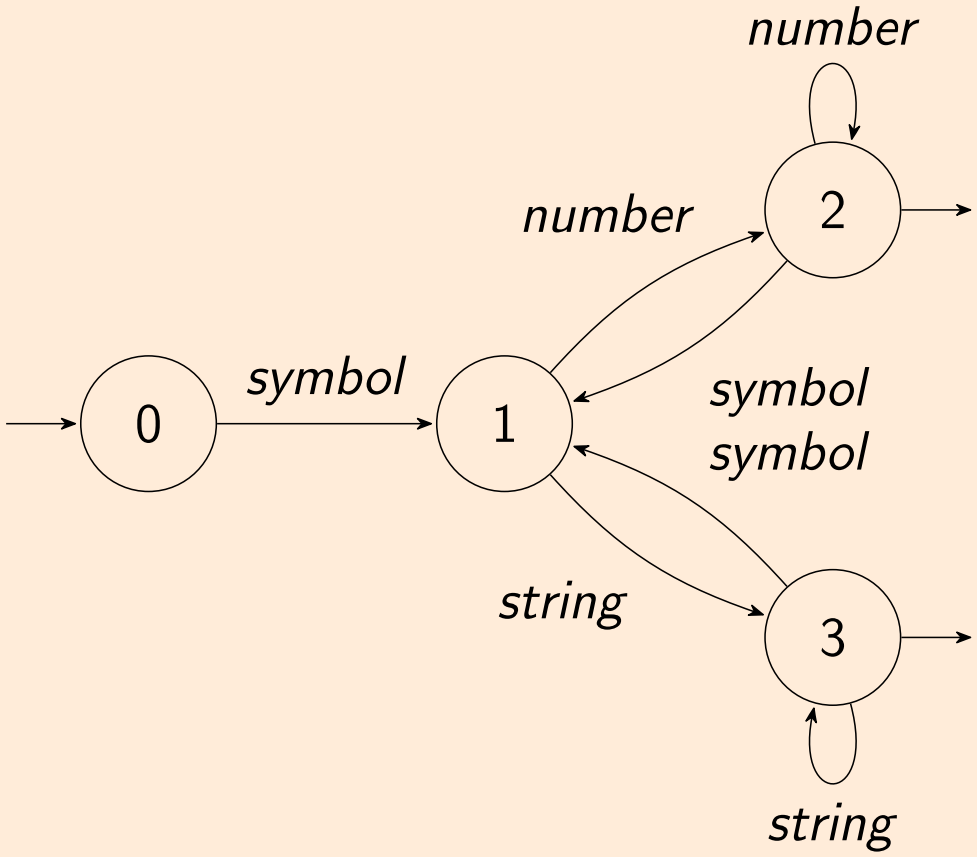
I am an *Ensignant Chercher* at EPITA in Paris.



Who is Jim?

I am an *Ensignant Chercher* at EPITA in Paris.

Recent research: *Representing and Computing with Types in Dynamically Typed Languages*.

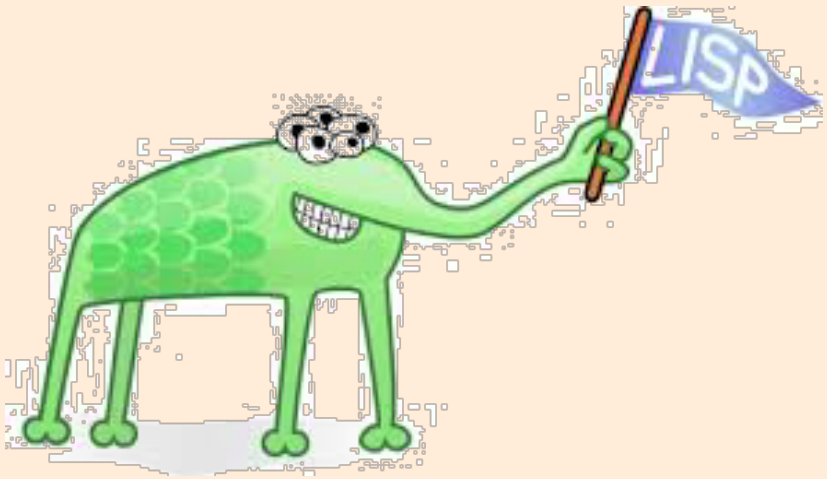


Who is Jim?

I am an *Ensignant Chercher* at EPITA in Paris.

Recent research: *Representing and Computing with Types in Dynamically Typed Languages*.

I am a career Lisper — relatively new to Scala.



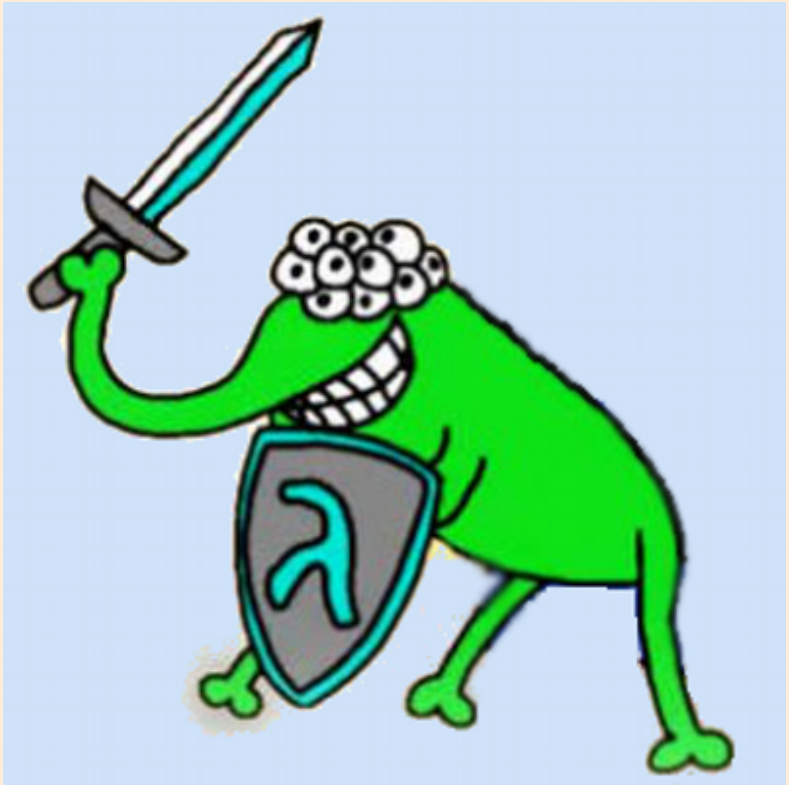
Who is Jim?

I am an *Ensignant Chercher* at EPITA in Paris.

Recent research: *Representing and Computing with Types in Dynamically Typed Languages*.

I am a career Lisper — relatively new to Scala.

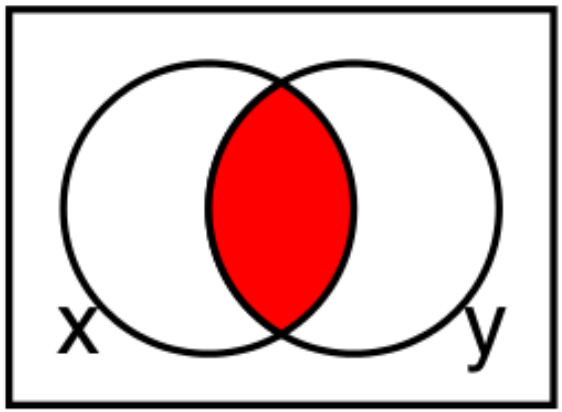
Scala and Lisp are weird. I sort of like weird things.



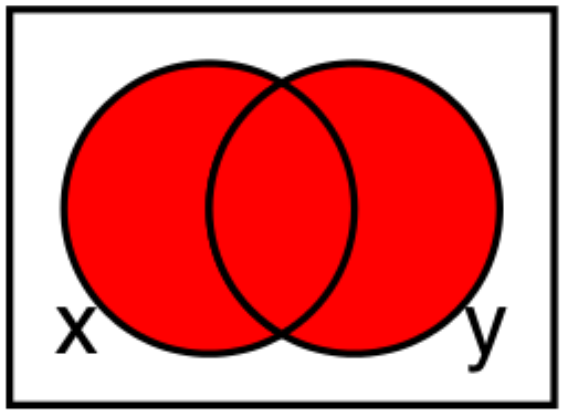
Overview

What are Boolean functions and Boolean algebra?

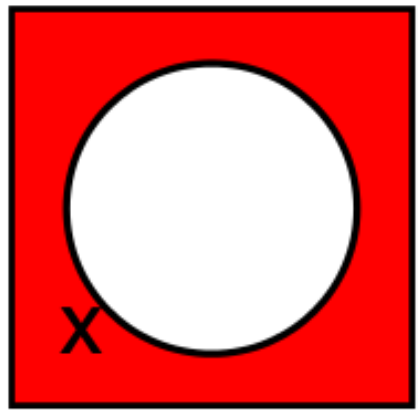
Applications of Boolean Algebra



$x \wedge y$



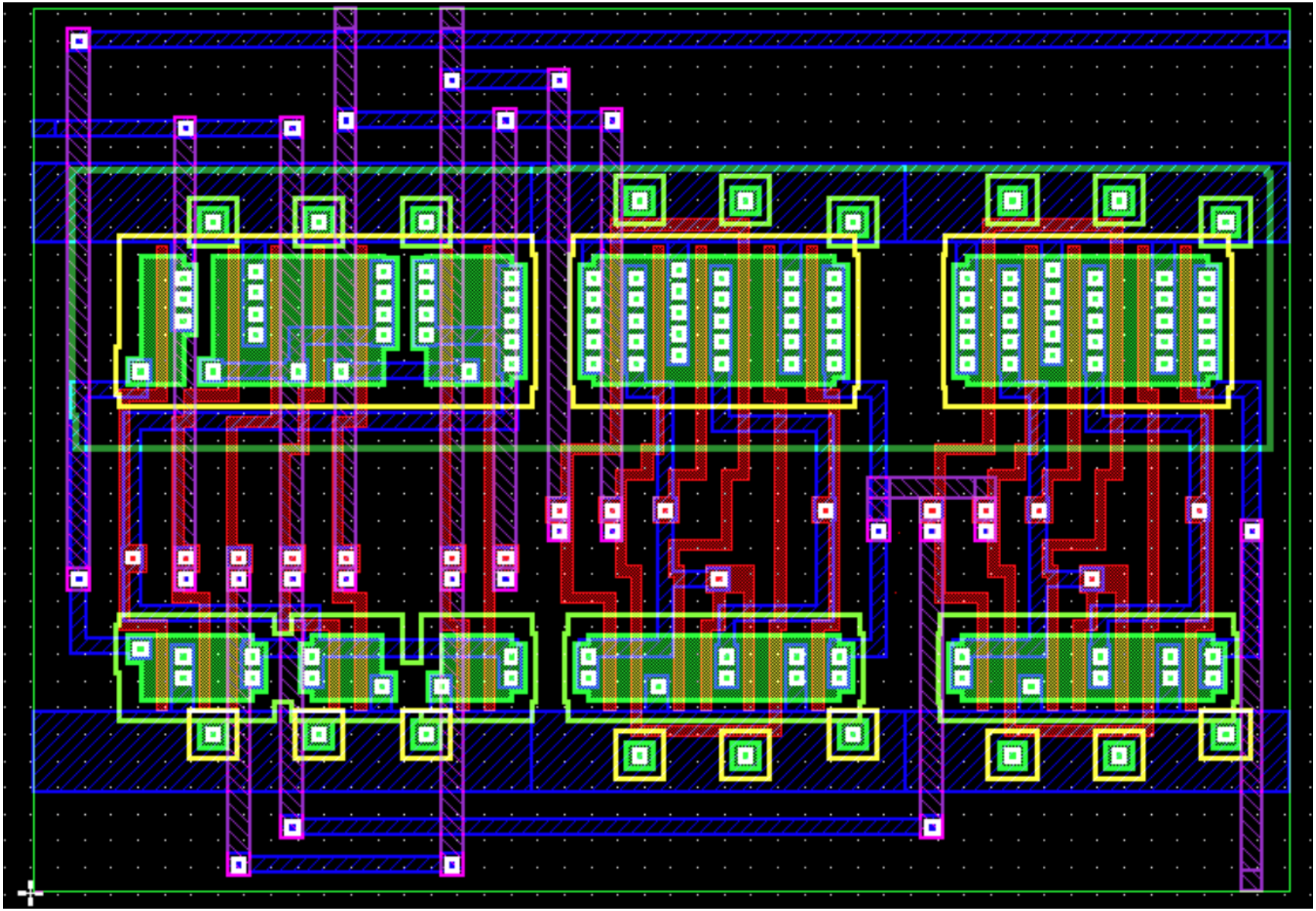
$x \vee y$



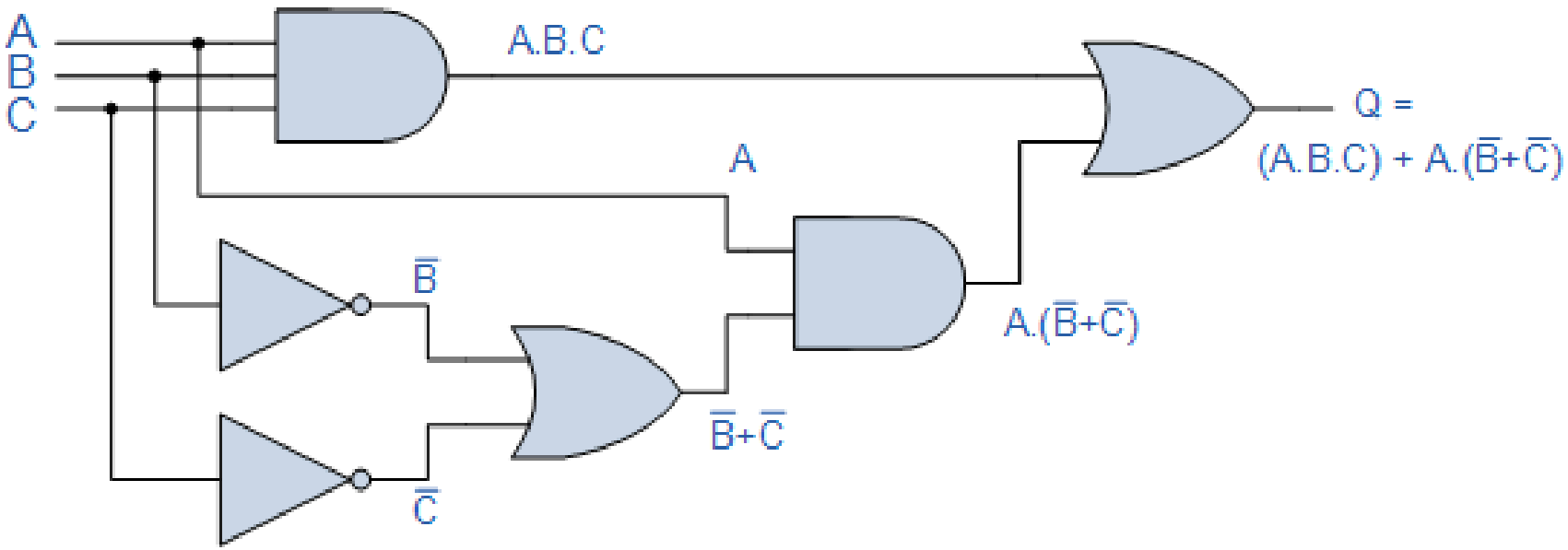
$\neg x$

Many problems in Computer Science can be modeled with set operations.

Applications of Boolean Algebra

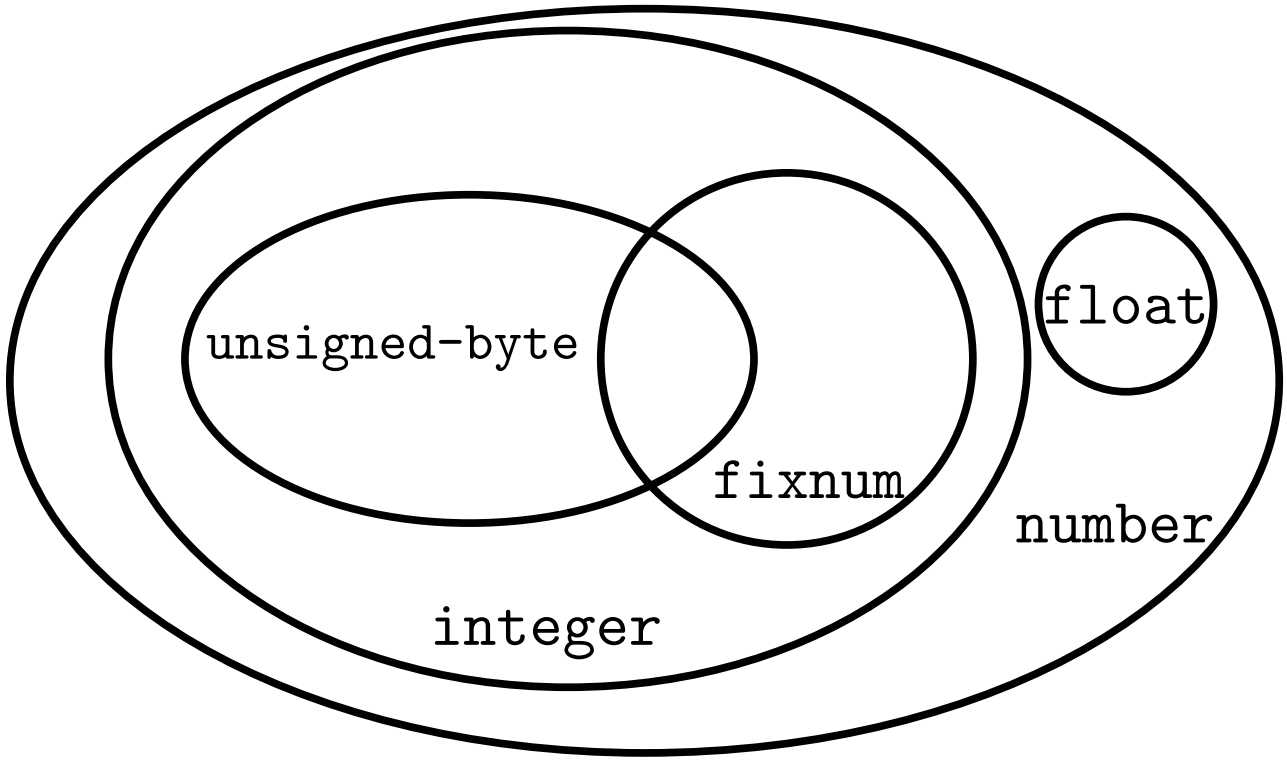


Applications of Boolean Algebra



Boolean algebra heavily used in IC design.

Applications of Boolean Algebra



Some type inference engines model types as sets, and type operations as set operations.

Boolean expression of variables

Many common mathematical notations for Boolean expressions.

- $\neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$

Boolean expression of variables

Many common mathematical notations for Boolean expressions.

- $\neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$
- $U \setminus ((A \cap C) \cup (B \cap C) \cup (B \cap D))$

Boolean expression of variables

Many common mathematical notations for Boolean expressions.

- $\neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$
- $U \setminus ((A \cap C) \cup (B \cap C) \cup (B \cap D))$
- $\overline{A \cdot C + B \cdot C + B \cdot D}$

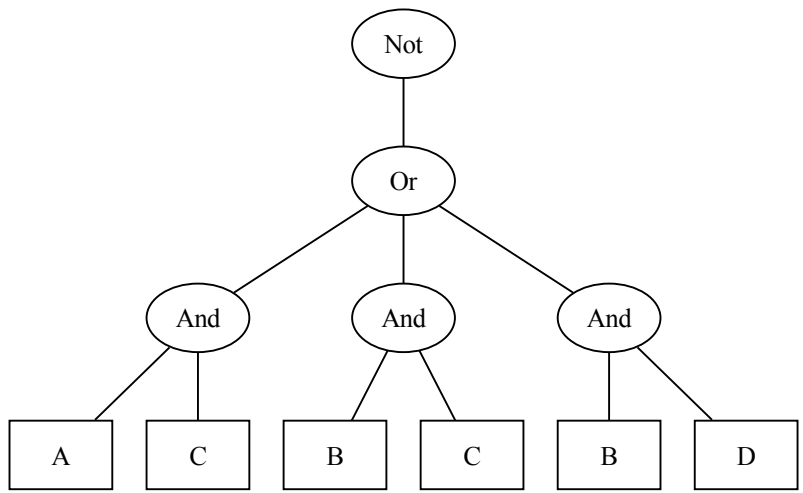
Boolean expression of variables

Many common mathematical notations for Boolean expressions.

- $\neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$
- $U \setminus ((A \cap C) \cup (B \cap C) \cup (B \cap D))$
- $\overline{A \cdot C + B \cdot C + B \cdot D}$
- $\overline{AC + BC + BD}$

Possible Representations of a Boolean expression

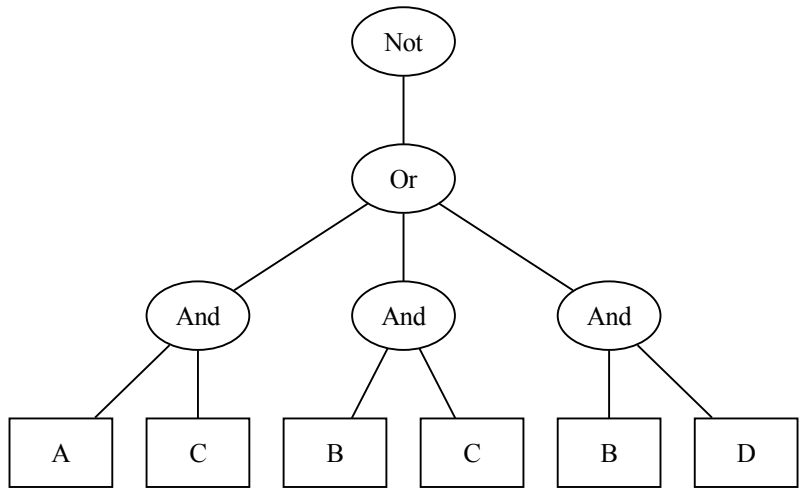
$$\neg\ ((A \wedge C) \vee (B \wedge C) \vee (B \wedge D))$$



Memory:

Possible Representations of a Boolean expression

$$\neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$$

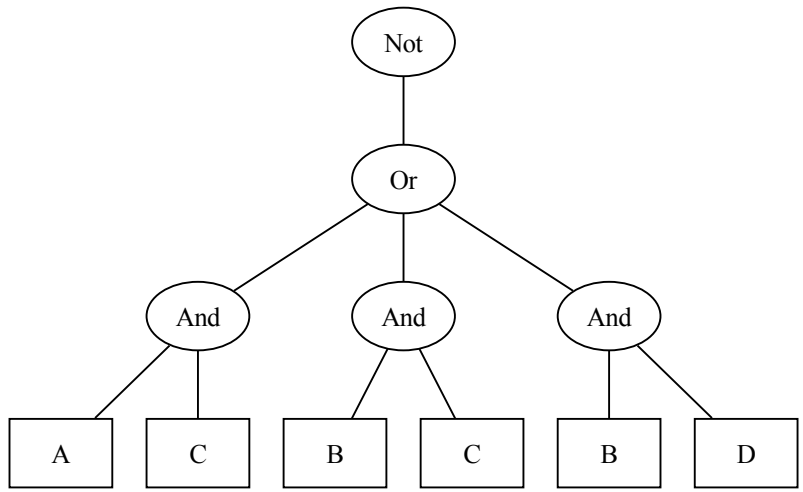


Memory:

Homoiconic: `'(not (or (and A C) (and B C) (and B D)))`

Possible Representations of a Boolean expression

$$\neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$$



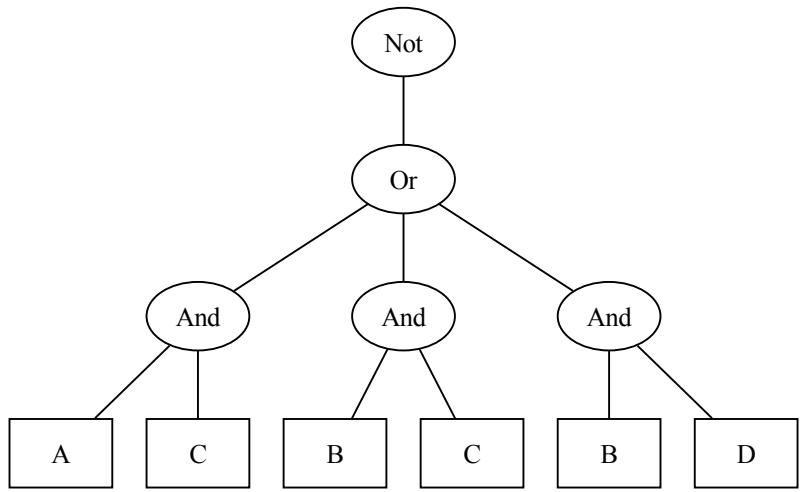
Memory:

Homoiconic: `'(not (or (and A C) (and B C) (and B D)))`

Programmatic: `Not(Or(And('A, 'C), And('B, 'C), And('B, 'D)))`

Possible Representations of a Boolean expression

$$\neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$$



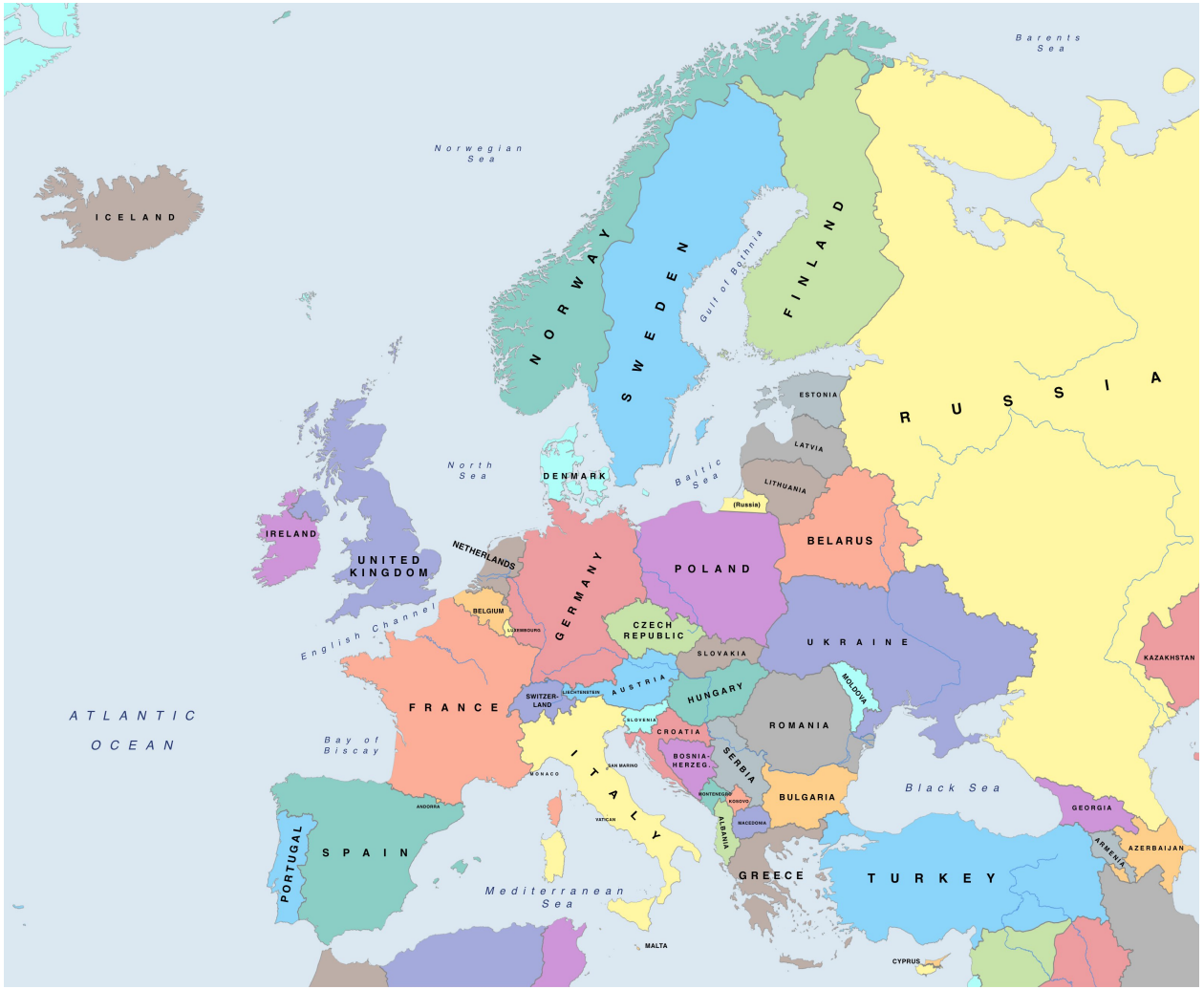
Memory:

Homoiconic: `'(not (or (and A C) (and B C) (and B D)))`

Programmatic: `Not(Or(And('A, 'C), And('B, 'C), And('B, 'D)))`

WARNING, quote is unfortunately deprecated in Scala 3.

The 4-color problem.



This map uses 11 colors, including the ocean. We can do better.

The 4-color problem.

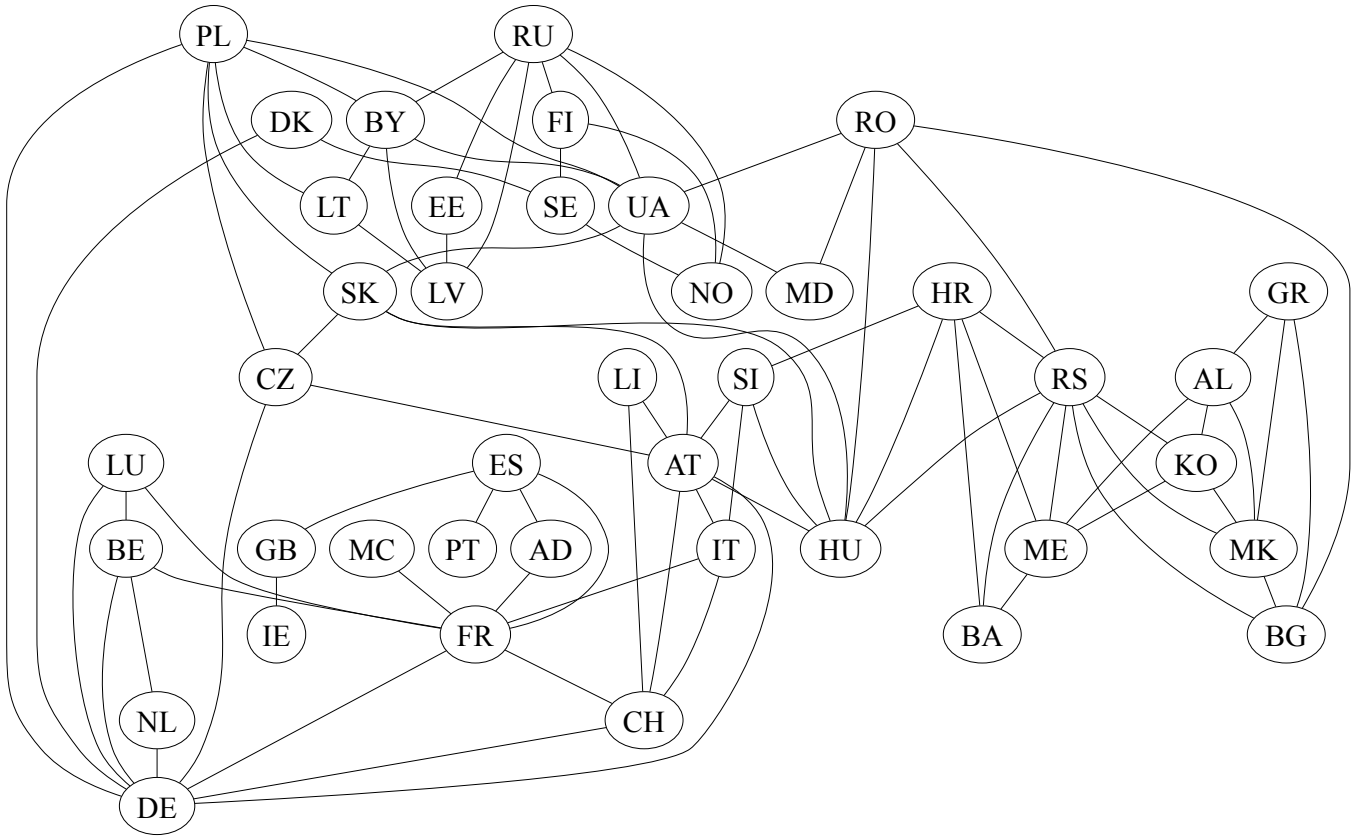
Given a map of states/countries, paint each region so that no adjacent regions share the same color.

This can always be done with 4 crayons.

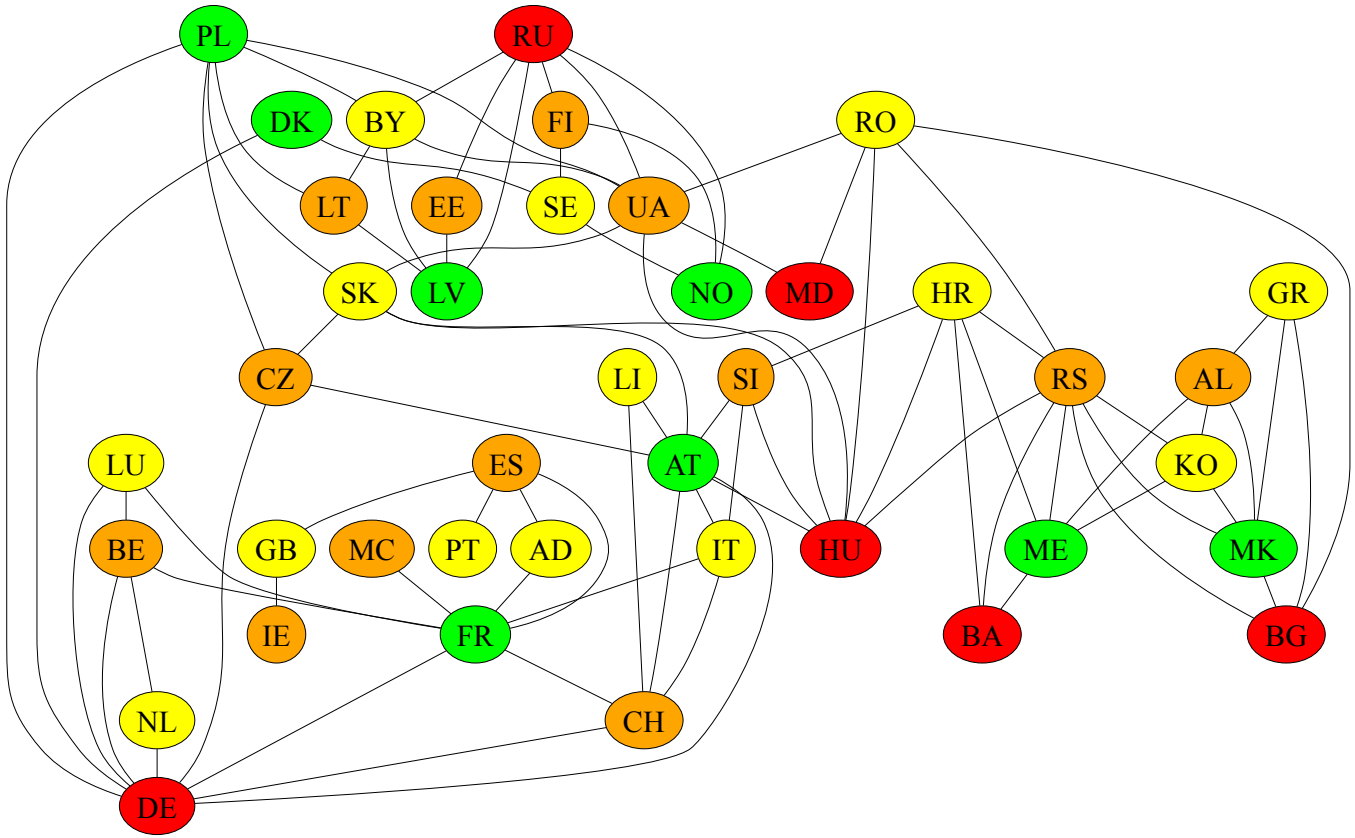
The 4-color problem.

The map can be converted to a graph where countries represent nodes, and common borders represent vertices.

The 4-color problem.



The 4-color problem.



The 4-color problem.

How to determine which colors to paint each node?

Solved by satisfying a particular Boolean function.

We can construct this Boolean function incrementally using `fold`.

The 4-color problem.

Since we know the map can be colored with 4 colors, we assign two Boolean variables to each region. The two Boolean values is sufficient to denote 4 colors.

- $0, 0$ — RED
- $0, 1$ — ORANGE
- $1, 0$ — YELLOW
- $1, 1$ — GREEN

The 4-color problem.

Each shared border introduces one constraint.

E.g., denote the color of France by $A_{\text{fr}}, B_{\text{fr}}$,
and Spain by $A_{\text{es}}, B_{\text{es}}$,
then we can construct a constraint

$$\begin{aligned}
\text{fr} \to \text{es} &= (A_{\text{fr}} \neq A_{\text{es}}) \vee (B_{\text{fr}} \neq B_{\text{es}}) \\
&= (A_{\text{fr}} \oplus A_{\text{es}}) \vee (B_{\text{fr}} \oplus B_{\text{es}}) \\
&= (A_{\text{fr}} \wedge \neg A_{\text{es}} \vee \neg A_{\text{fr}} \wedge A_{\text{es}}) \vee (B_{\text{fr}} \wedge \neg B_{\text{es}} \vee \neg B_{\text{fr}} \wedge B_{\text{es}})
\end{aligned}$$

The 4-color problem.

To color the map:

1. Generate a Boolean function which incorporates all these border constraints, and
2. find a solution which *satisfies* all the constraints.

How to satisfy a Boolean function.

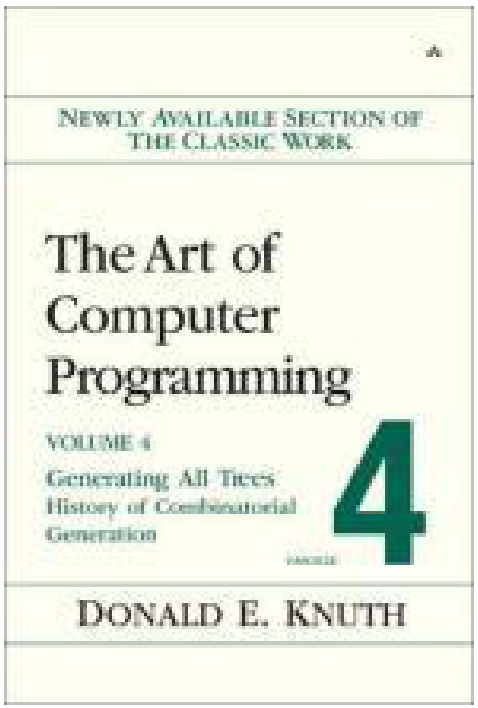
This is a well known problem in CS called *SAT*.

We might represent the Boolean function as an AST, and perform symbolic manipulation to find a solution. This is HARD.

We will represent the Boolean function as a special data structure called a BDD. This is elegant.

What are BDDs?

Donald Knuth's new toy.



Binary decision diagrams (BDDs) are wonderful, and the more I play with them the more I love them. ... I've been like a child with a new toy, being able now to solve problems that I never imagined would be tractable... I suspect that many readers will have the same experience ... there will always be more to learn about such a fertile subject.
[Donald Knuth, *Art of Computer Science, Volume 4*]

What are BDDs?

A BDD (Binary Decision Diagram) is a data structure used to represent a Boolean function.

Represents the Boolean function in a particular form:

- Canonical
- Compressed

Boolean expression of variables

Truth table for $F = \neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$

A	B	C	D	F
1	1	1	1	0
1	1	1	0	0
1	1	0	1	0
1	1	0	0	1
1	0	1	1	0
1	0	1	0	0
1	0	0	1	1
1	0	0	0	1
0	1	1	1	0
0	1	1	0	0
0	1	0	1	0
0	1	0	0	1
0	0	1	1	1
0	0	1	0	1
0	0	0	1	1
0	0	0	0	1

Boolean expression of variables

Truth table for $F = \neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$

A	B	C	D	F
T	T	T	T	F
T	T	T	F	F
T	T	F	T	F
T	T	F	F	T
T	F	T	T	F
T	F	T	F	F
T	F	F	T	T
T	F	F	F	T
F	T	T	T	F
F	T	T	F	F
F	T	F	T	F
F	T	F	F	T
F	F	T	T	T
F	F	T	F	T
F	F	F	T	T
F	F	F	F	T

Boolean expression of variables

Truth table for $F = \neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$

A	B	C	D	F
$\top$	$\top$	$\top$	$\top$	$\bot$
$\top$	$\top$	$\top$	$\bot$	$\bot$
$\top$	$\top$	$\bot$	$\top$	$\bot$
$\top$	$\top$	$\bot$	$\bot$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\bot$
$\top$	$\bot$	$\top$	$\bot$	$\bot$
$\top$	$\bot$	$\bot$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\top$	$\top$	$\bot$
$\bot$	$\top$	$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$	$\top$	$\bot$
$\bot$	$\top$	$\bot$	$\bot$	$\top$
$\bot$	$\bot$	$\top$	$\top$	$\top$
$\bot$	$\bot$	$\top$	$\bot$	$\top$
$\bot$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$

Boolean expression of variables

Truth table for $F = \neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$

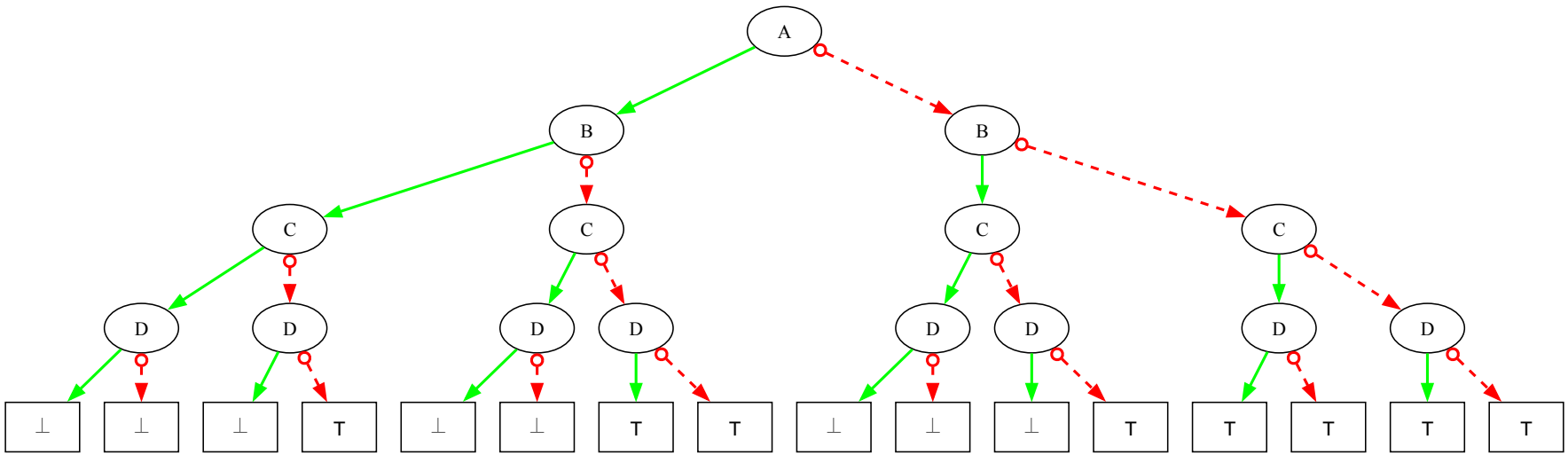
A	B	C	D	F
$\top$	$\top$	$\top$	$\top$	$\bot$
$\top$	$\top$	$\top$	$\bot$	$\bot$
$\top$	$\top$	$\bot$	$\top$	$\bot$
$\top$	$\top$	$\bot$	$\bot$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\bot$
$\top$	$\bot$	$\top$	$\bot$	$\bot$
$\top$	$\bot$	$\bot$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\top$	$\top$	$\bot$
$\bot$	$\top$	$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$	$\top$	$\bot$
$\bot$	$\top$	$\bot$	$\bot$	$\top$
$\bot$	$\bot$	$\top$	$\top$	$\top$
$\bot$	$\bot$	$\top$	$\bot$	$\top$
$\bot$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$

Once the correct row is found, evaluating the Boolean function is $O(1)$.

However, the truth table is $O(2^n)$ in memory.

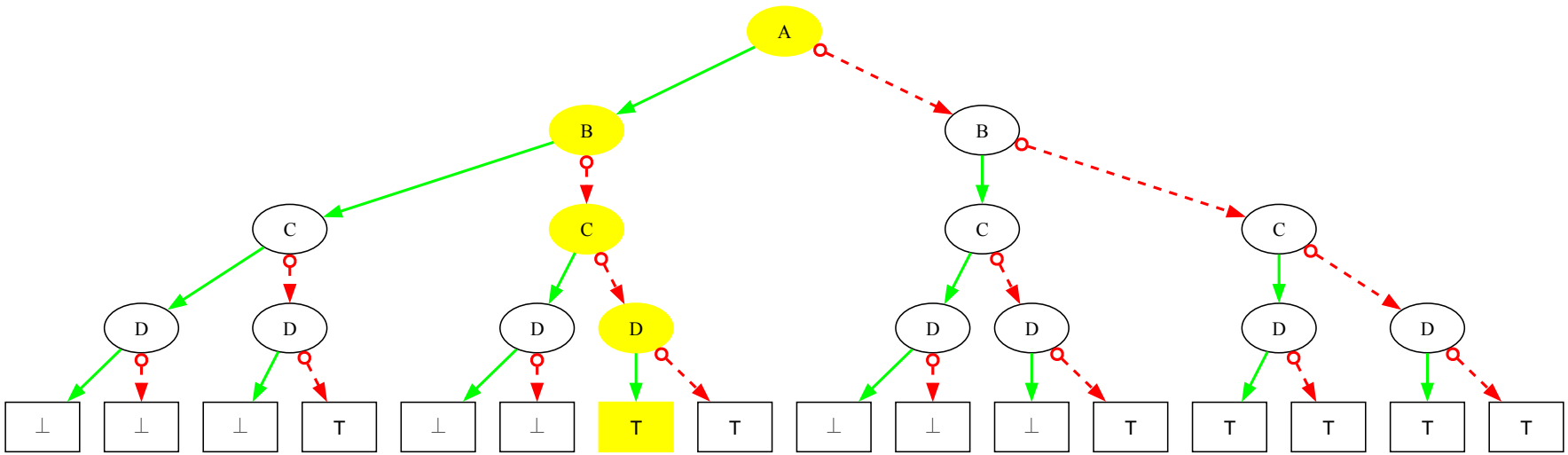
Boolean expression of variables

Boolean Expression: $\neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$



Boolean expression of variables

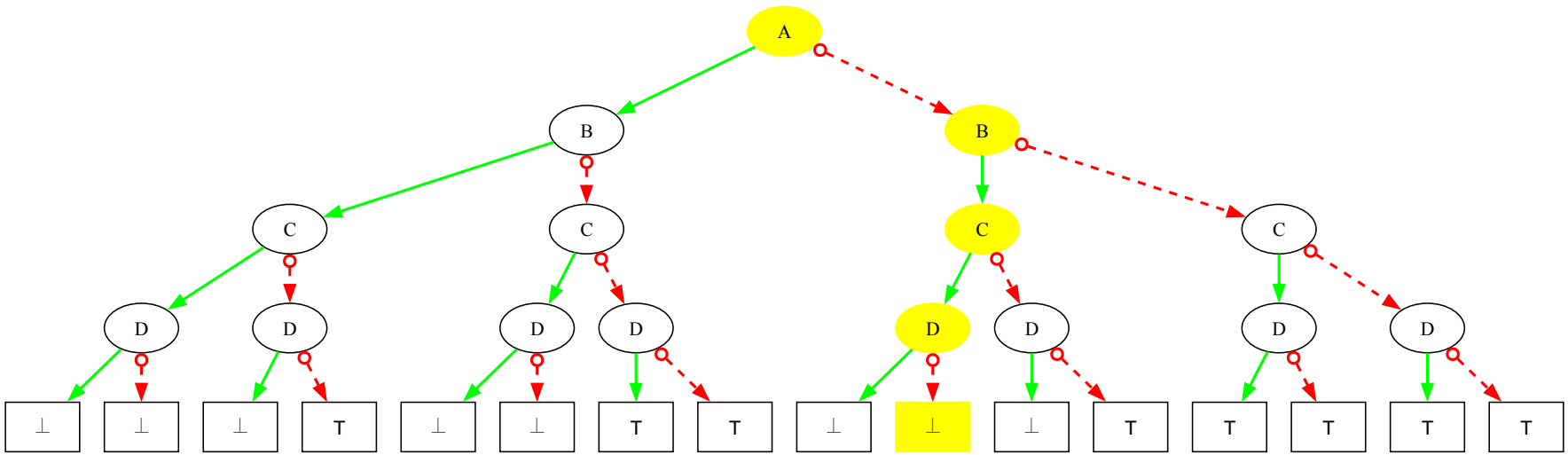
Every path from root to leaf corresponds to one row of the truth table.



A	B	C	D	$\neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$
$\top$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\top$	$\bot$	$\bot$

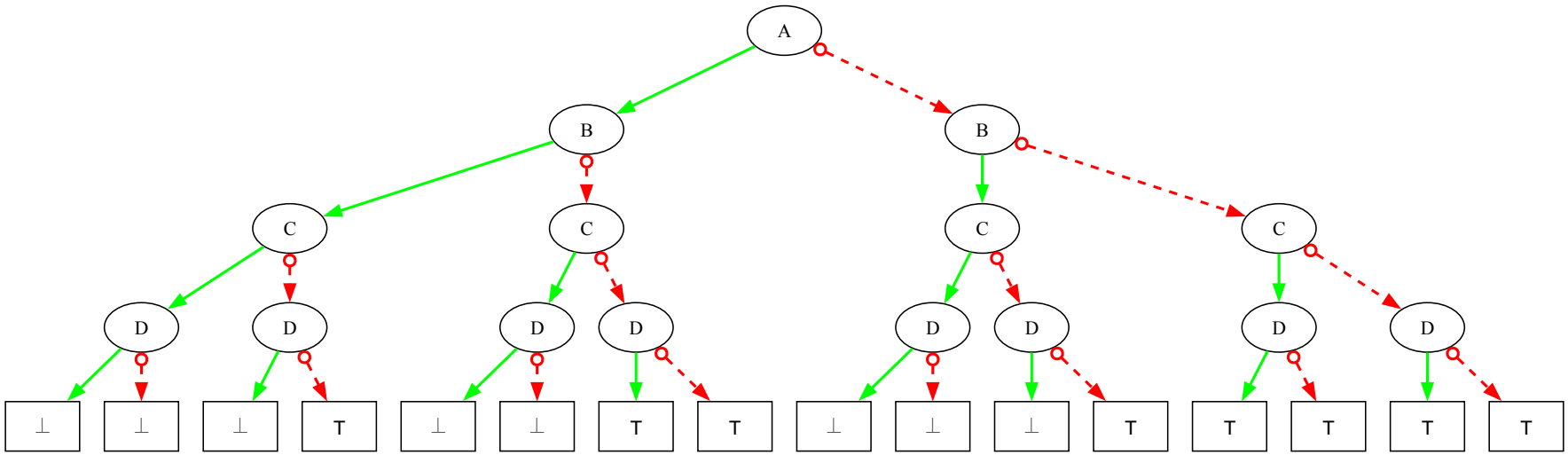
Boolean expression of variables

Every path from root to leaf corresponds to one row of the truth table.



A	B	C	D	$\neg((A \wedge C) \vee (B \wedge C) \vee (B \wedge D))$
$\top$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\top$	$\bot$	$\bot$

Boolean expression of variables



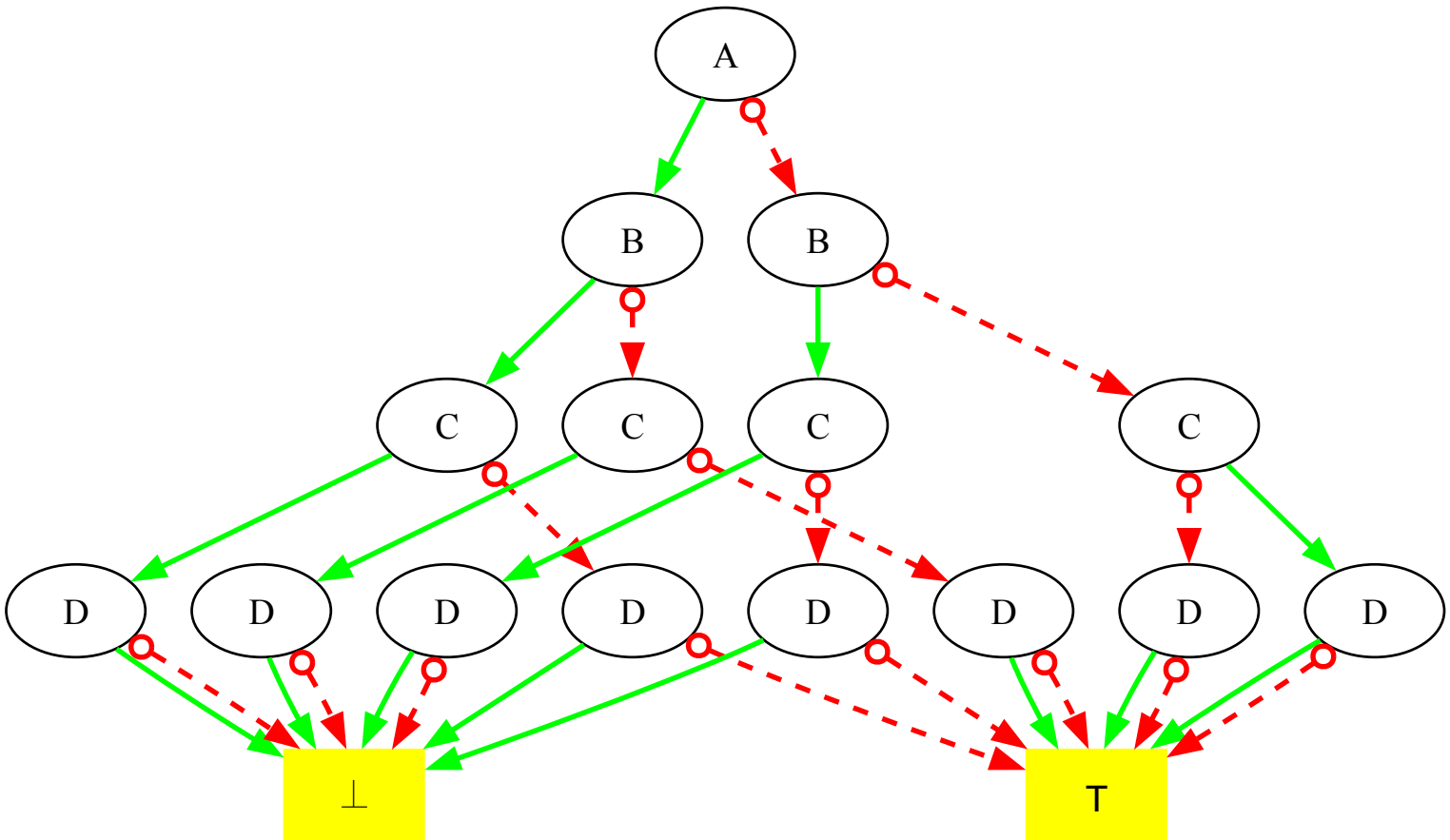
4 variables $\implies 2^{4+1} - 1 = 31$ nodes

The graph size grows exponentially with number of variables.

We can do better.

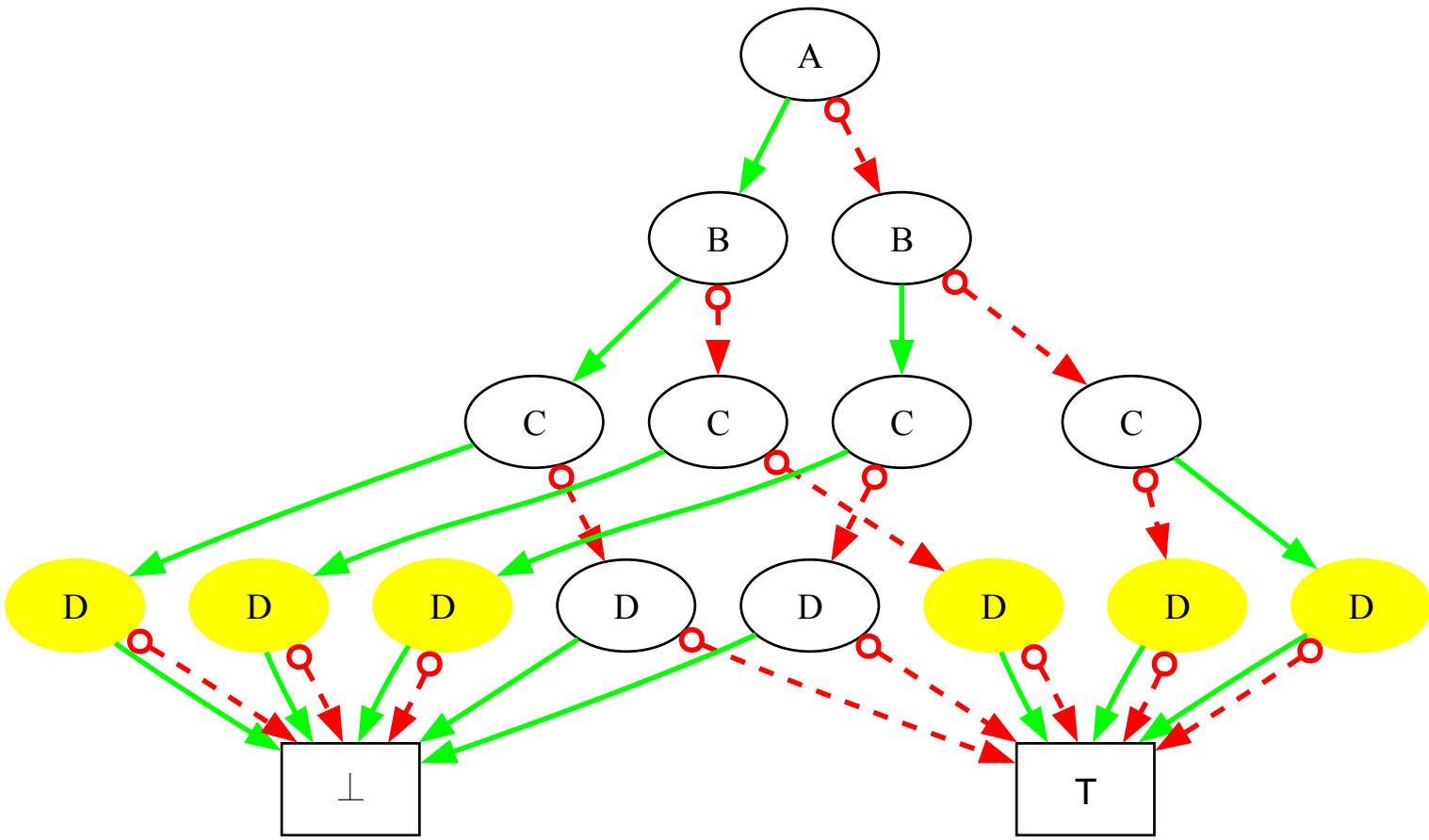
Standard Rule 1: the *terminal* rule

There are 3 standard reduction rules. The terminal rule allows us to replace leaf nodes with singleton objects, $\bot$ and $\top$. Divides size by 2.

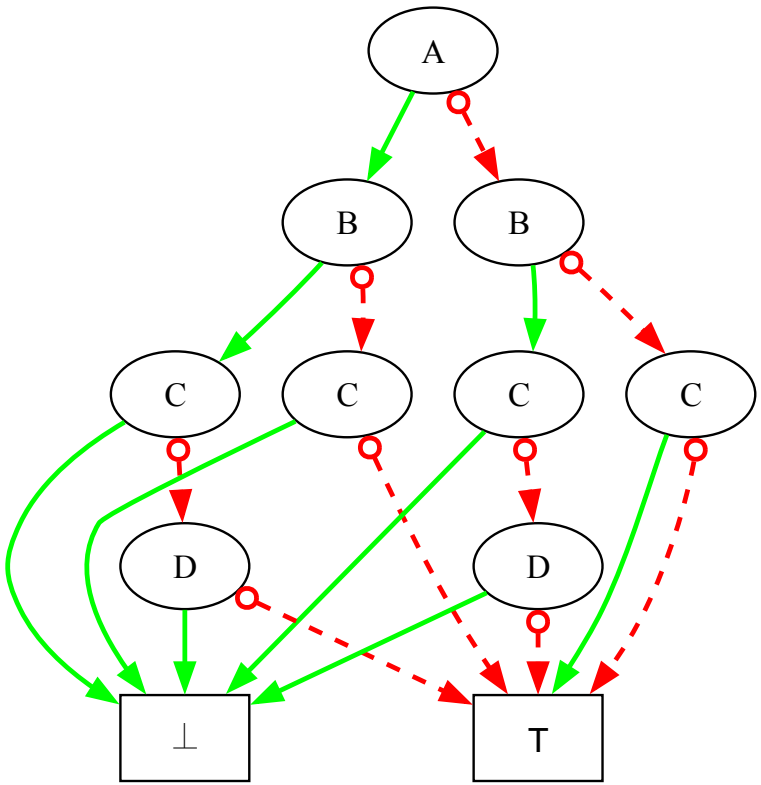


Standard Rule 2: the *deletion* rule

The deletion rule allows us to remove nodes which have the same red (false) and green (true) pointer.

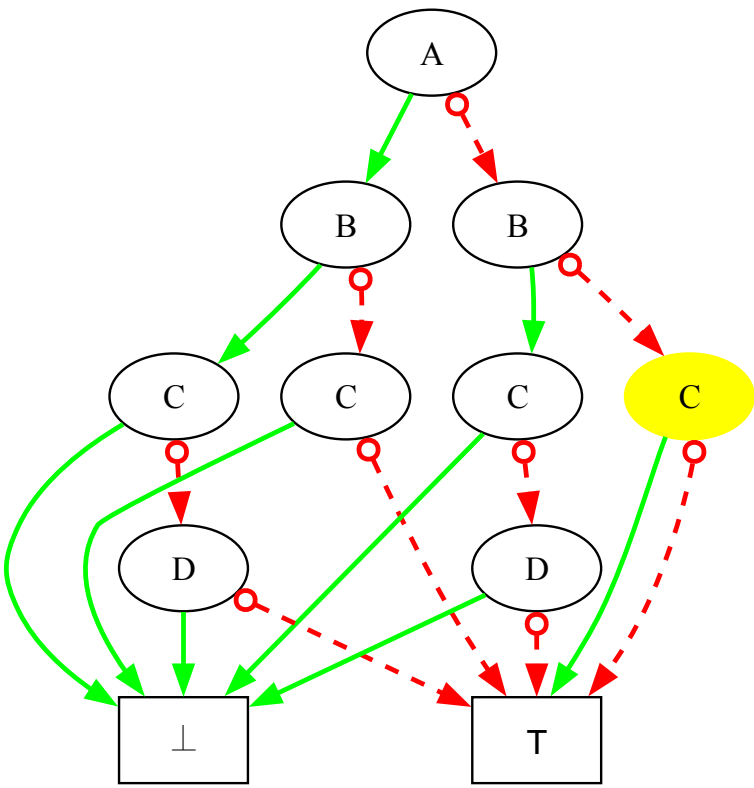


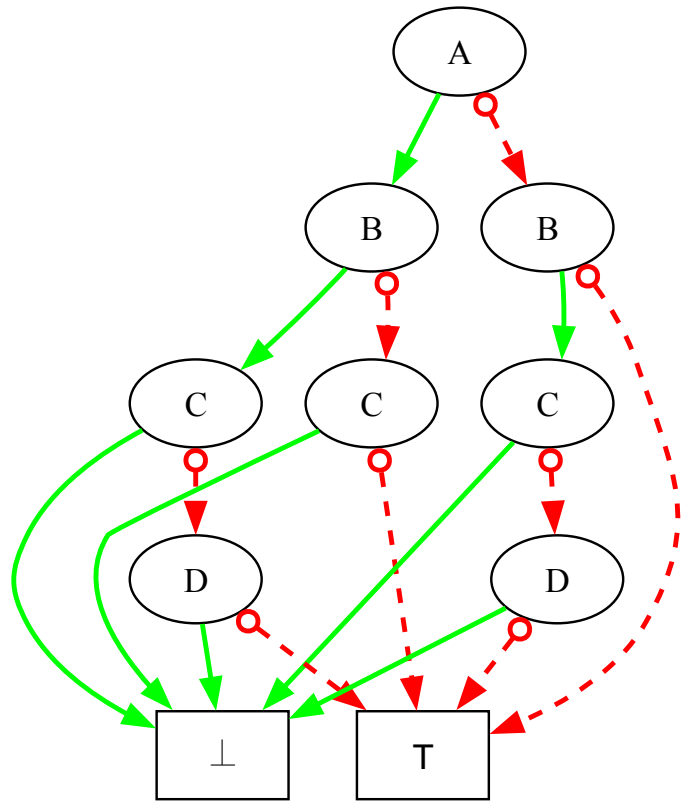
Reducing to 11 nodes



More reduction

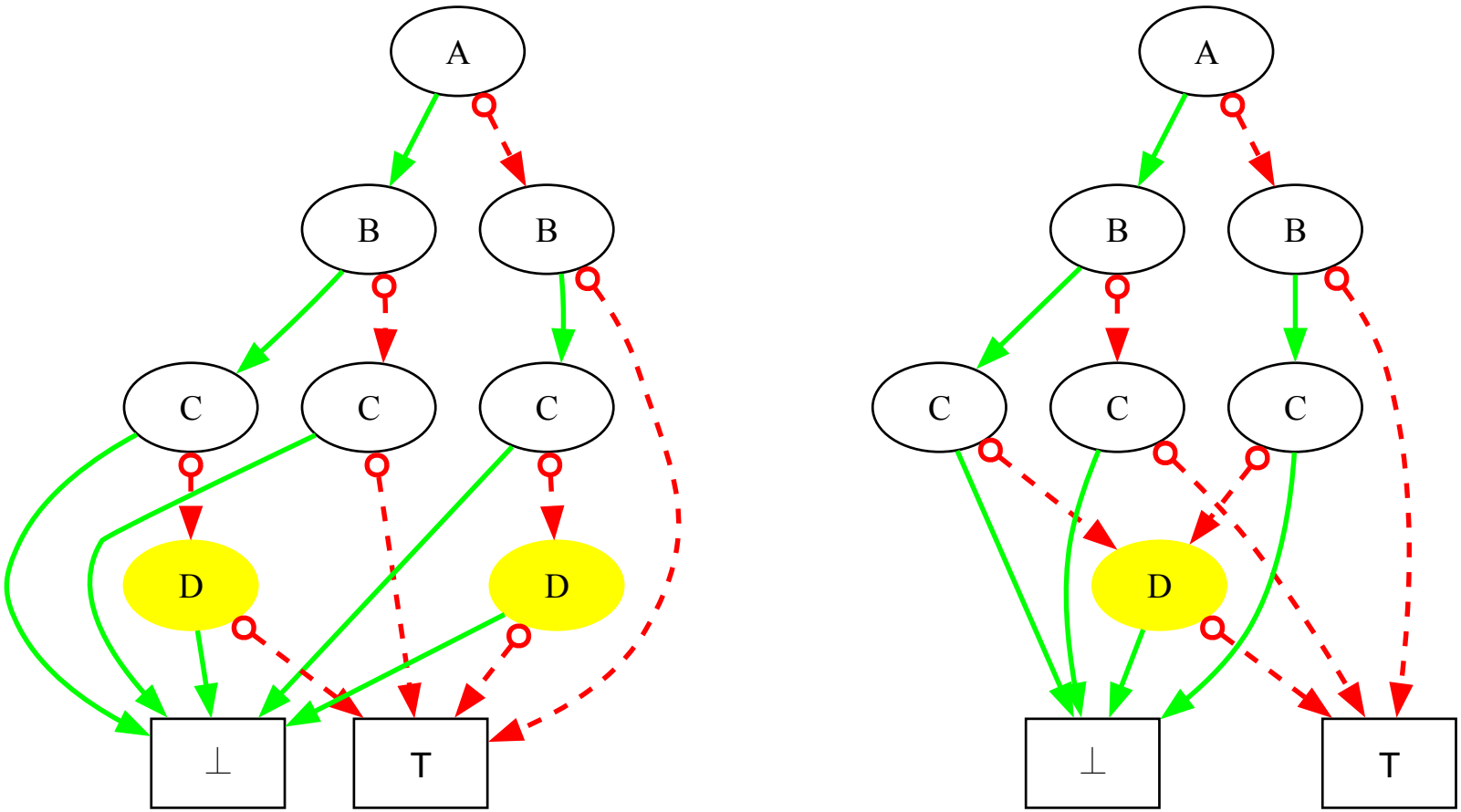
The *deletion* rule can be applied multiple times.





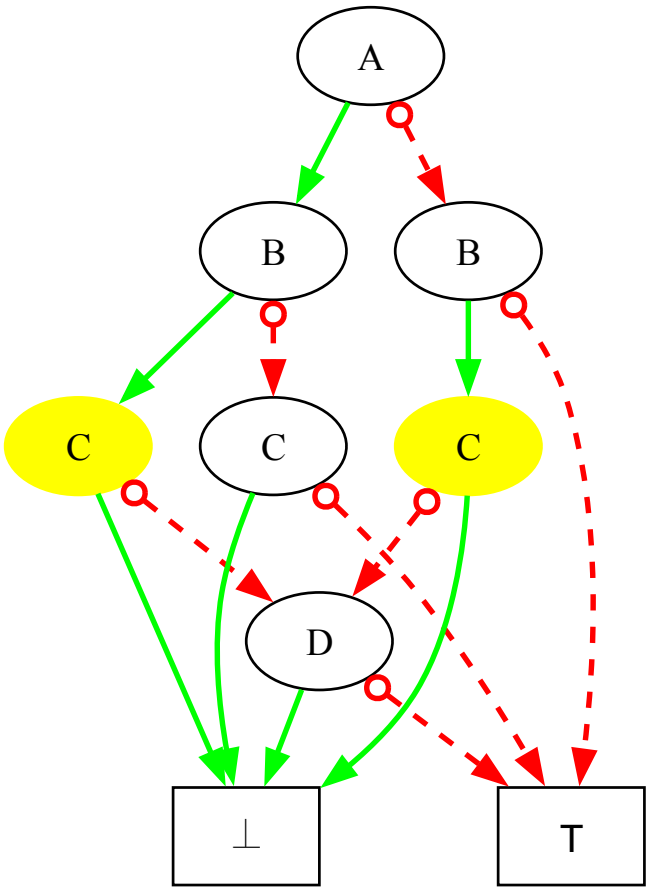
Standard Rule 3: the *merging* rule

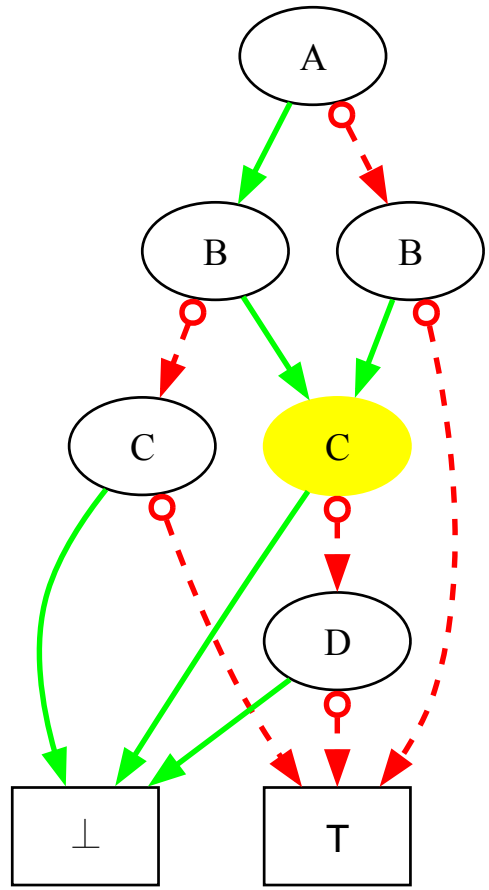
The merging rule allows us to merge structurally congruent nodes, *i.e.*, with same children, and same label.



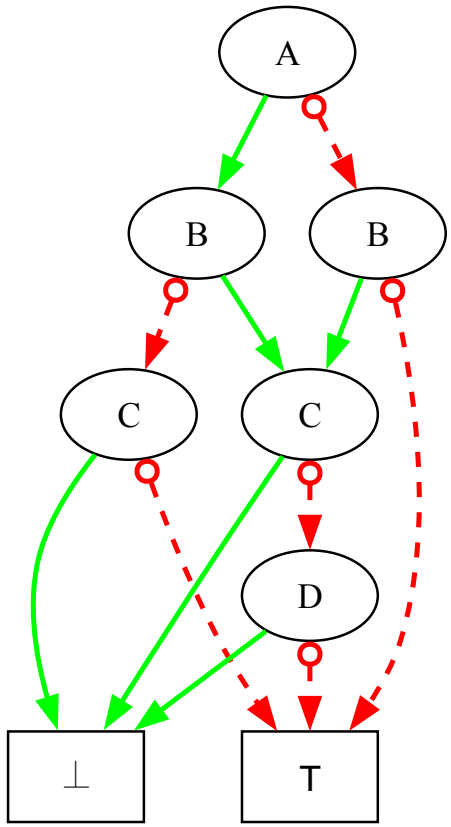
More congruent nodes

The *merging* rule can be applied multiple times.





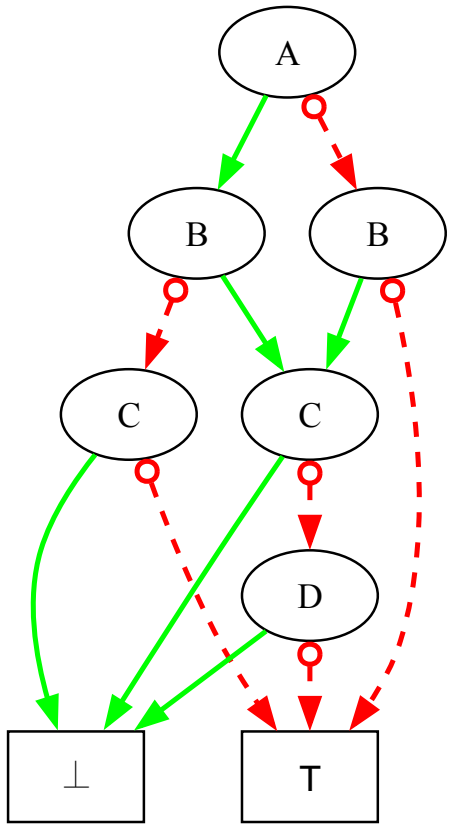
ROBDD: Reduced ordered binary decision diagram



$$\neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$$

Started with 31 nodes. We need only 8 nodes to represent this function.

ROBDD: Reduced Ordered binary decision diagram



Nodes are

- ordered
- from top to bottom
- by variable name.

If $v_i \to v_j$ then $i < j$.

In the figure: $A < B < C < D$

Creative Commons Attribution ShareAlike, Author: Georg Mittenecker

The BDD is the *Eierlegende Wollmilchsau* of Boolean algebra.

BDDs have many (many many..) surprising features and uses.

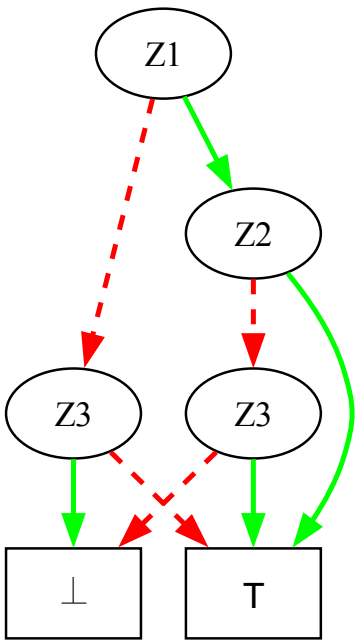
Some features of BDDs

We will look at just a few such features.

- Canonical form for Boolean function
- Compressed form of truth table
- Test equivalence of Boolean functions with `eq`, $\Omega(1)$.
- Test vacuity and habitation, $\Omega(1)$.
- Solve SAT in linear time, $\Omega(n)$.
- Serialize to a disjunctive form
- Evaluate n-variable Boolean function, $\Omega(n)$.
- Supports Boolean algebra, `And`, `Or`, `Not`, `AndNot`, `Xor`.

BDD is a canonical representation

Recall: the final form of the BDD depends only on the truth table—not in any way on the syntactically representation of the Boolean function.



$$\neg(\neg z_1 \wedge z_3) \vee (z_1 \wedge \neg z_2 \wedge \neg z_3)$$

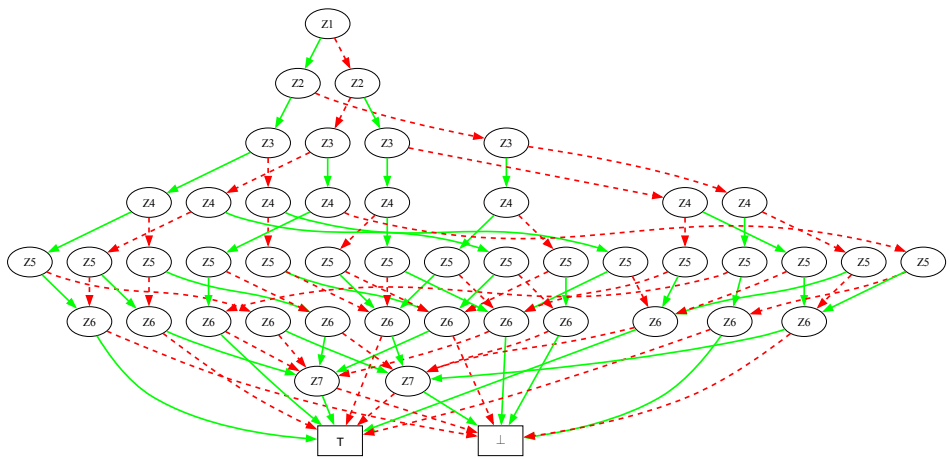
$$= (z_1 \wedge z_2) \vee (z_1 \wedge \neg z_2 \wedge z_3) \vee (\neg z_1 \wedge \neg z_3)$$

$$= \big((z_1 \vee \neg z_2) \wedge (z_1 \vee z_3) \wedge (\neg z_1 \vee z_2) \wedge (z_2 \vee z_3)\big) \vee (\neg z_1 \wedge \neg z_3)$$

BDD is a compressed representation

n	BDD size	TT size	ρ_n
1	3	3	100.000%
2	5	7	71.429%
3	7	15	46.667%
4	11	31	35.484%
5	19	63	30.159%
6	31	127	24.409%
7	47	255	18.431%
8	79	511	15.460%
9	143	1023	13.978%
10	271	2047	13.239%
11	511	4095	12.479%
12	767	8191	9.364%
13	1279	16K	7.807%
14	2303	32K	7.028%
15	4351	65K	6.639%
16	8447	131K	6.445%
17	16639	262K	6.347%
18	33023	524K	6.299%
19	65791	1Meg	6.274%
20	131071	2Meg	6.250%
21	196607	4Meg	4.687%

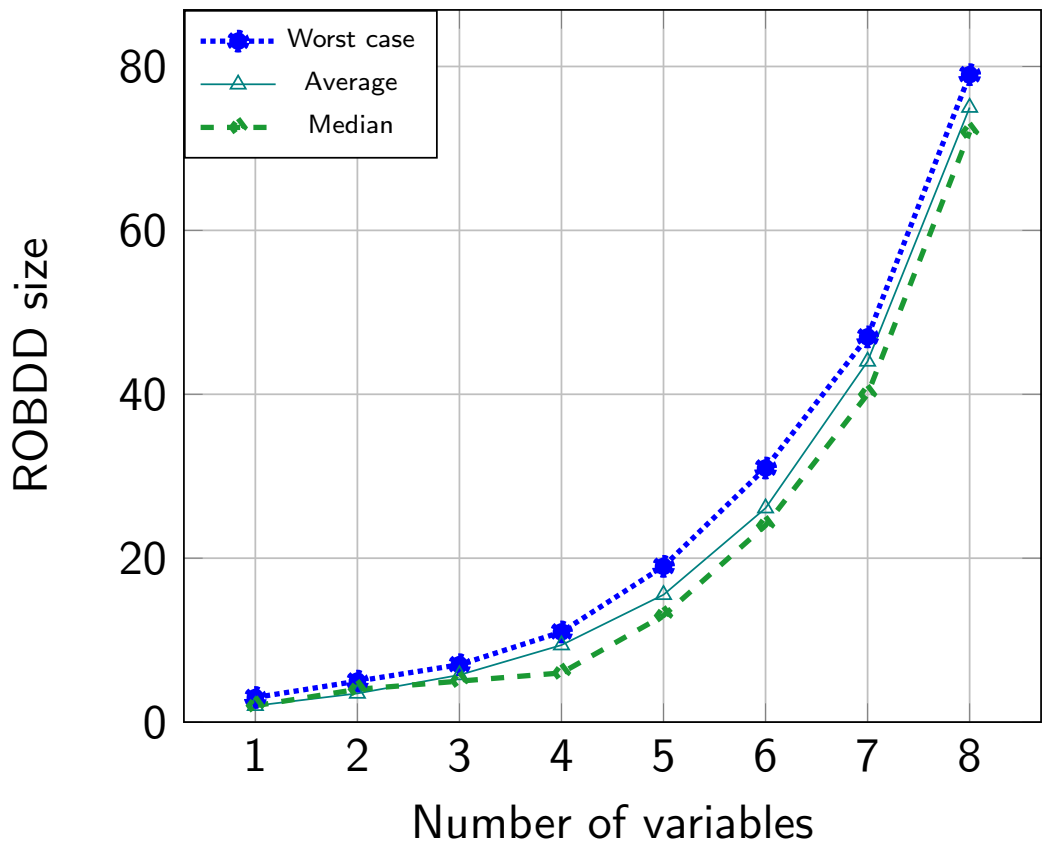
A BDD of 7 Boolean variables



BDD is a compressed representation

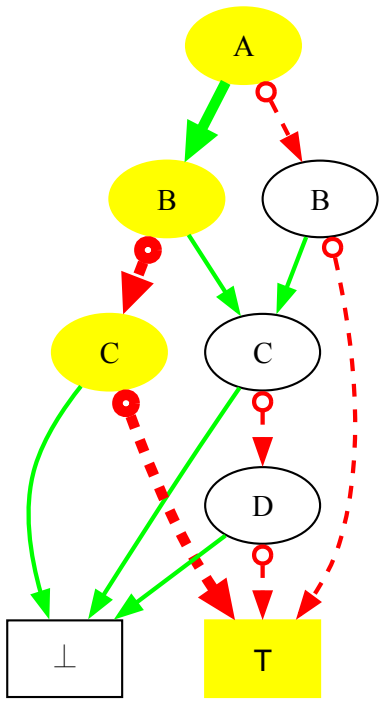
n	BDD size	TT size	ρ_n
1	3	3	100.000%
2	5	7	71.429%
3	7	15	46.667%
4	11	31	35.484%
5	19	63	30.159%
6	31	127	24.409%
7	47	255	18.431%
8	79	511	15.460%
9	143	1023	13.978%
10	271	2047	13.239%
11	511	4095	12.479%
12	767	8191	9.364%
13	1279	16K	7.807%
14	2303	32K	7.028%
15	4351	65K	6.639%
16	8447	131K	6.445%
17	16639	262K	6.347%
18	33023	524K	6.299%
19	65791	1Meg	6.274%
20	131071	2Meg	6.250%
21	196607	4Meg	4.687%

max $\approx$ *average*



Satisfying a Boolean Function

A Boolean function of n variables can be *satisfied* in n steps.



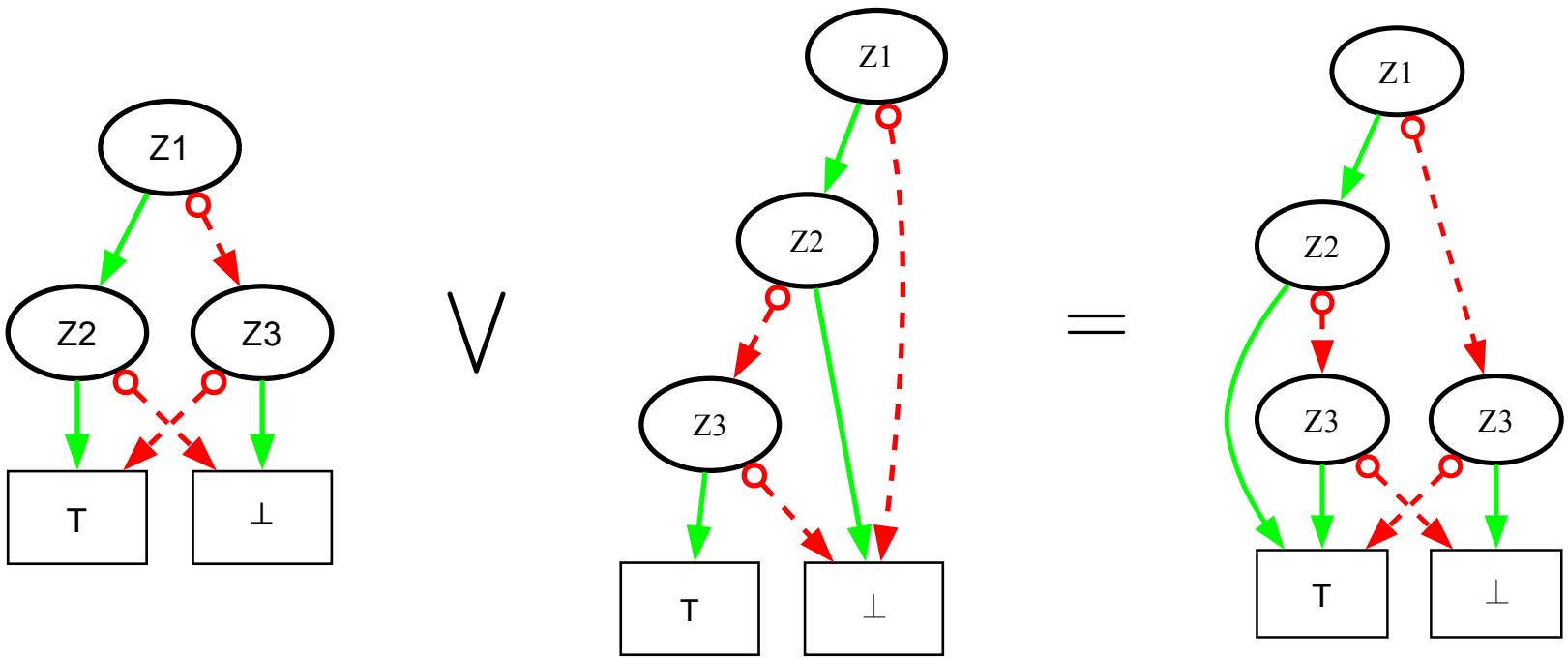
Is suffices to trace any path to $\top$.
This can be done in *one pass*.
Trace any path to the penultimate level.
Either the green or red arrow leads to $\top$.

A	B	C	D	$\neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$
$\top$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\top$	$\bot$	$\bot$

Boolean operations defined on BDDs

Union / OR / $\vee$

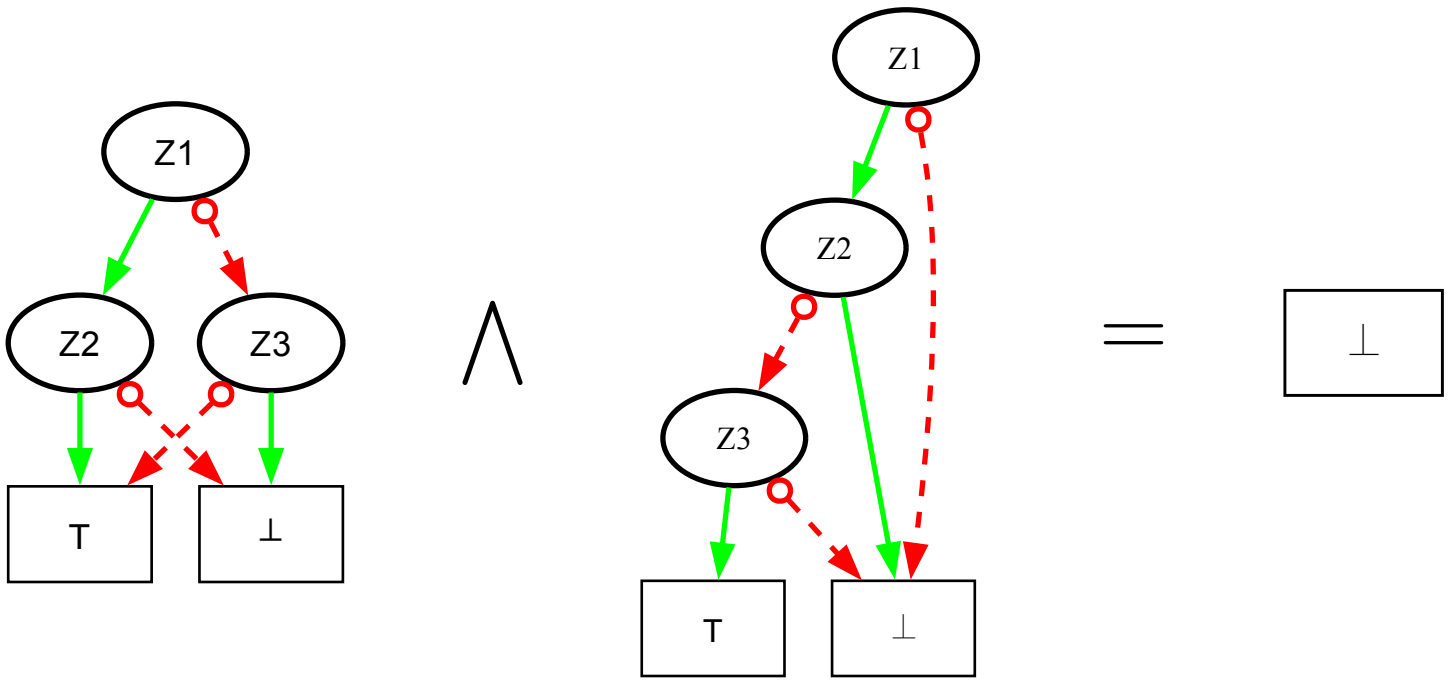
$$(Z_1 \wedge Z_2 \vee \neg Z_1 \wedge \neg Z_2) \vee (Z_1 \wedge \neg Z_2 \wedge Z_3) = Z_1 \wedge Z_2 \vee \neg Z_1 \wedge \neg Z_2 \vee Z_1 \wedge \neg Z_2 \wedge Z_3$$



Boolean operations defined on BDDs

Intersection / AND / $\wedge$

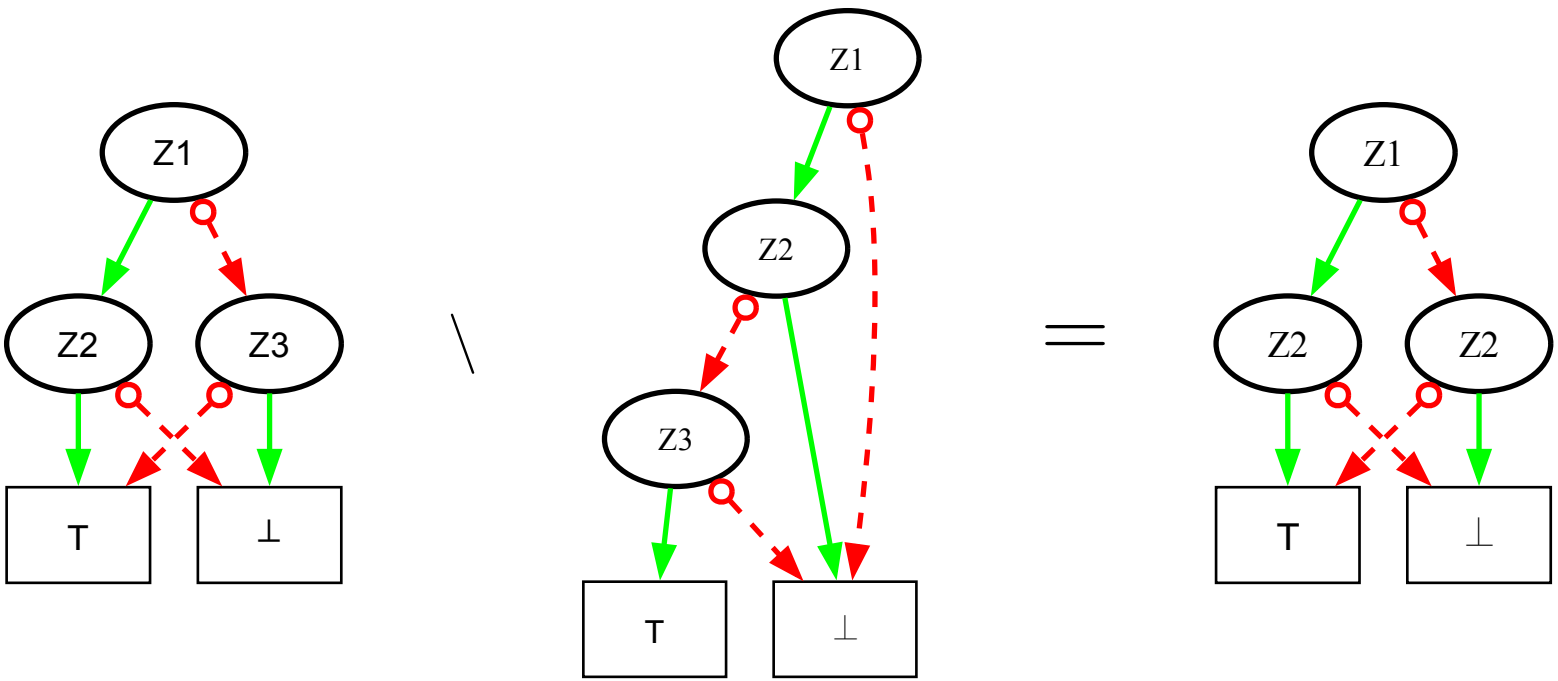
$$(Z_1 \wedge Z_2 \ \vee \ \neg Z_1 \wedge \neg Z_2) \wedge (Z_1 \wedge \neg Z_2 \wedge Z_3) \ = \ \bot$$



Boolean operations defined on BDDs

Relative complement / AND-NOT / \

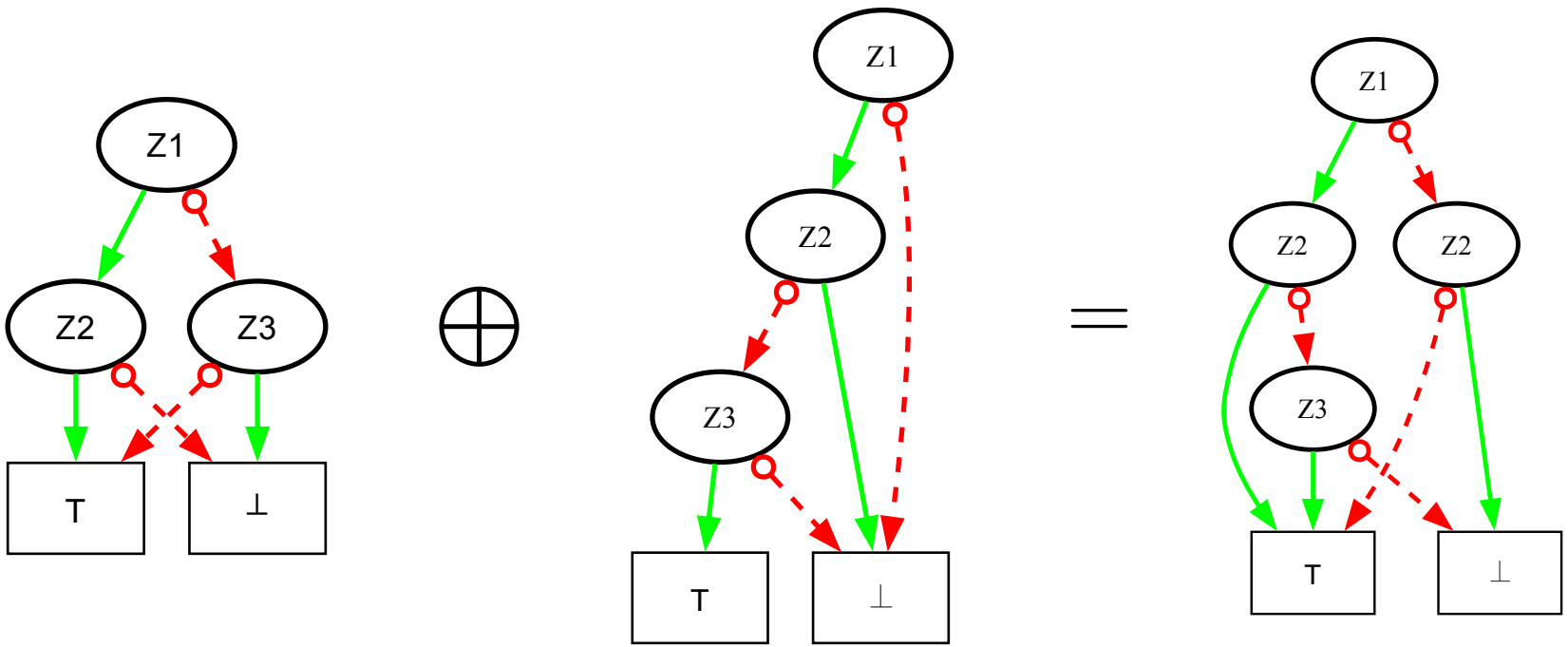
$$(Z_1 \wedge Z_2 \ \vee \ \neg Z_1 \wedge \neg Z_2) \wedge \neg (Z_1 \wedge \neg Z_2 \wedge Z_3) \ = \ Z_1 \wedge Z_2 \ \vee \ \neg Z_1 \wedge \neg Z_2$$



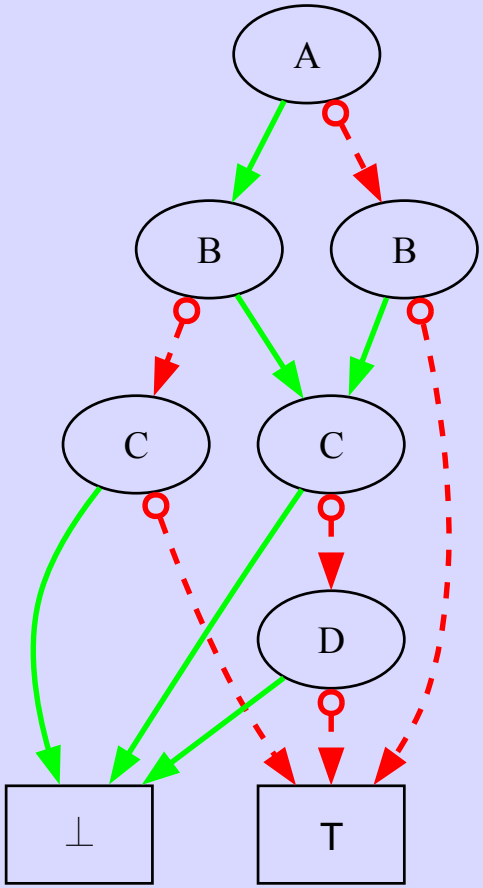
Boolean operations defined on BDDs

Exclusive Or / Symmetric difference / XOR / $\oplus$

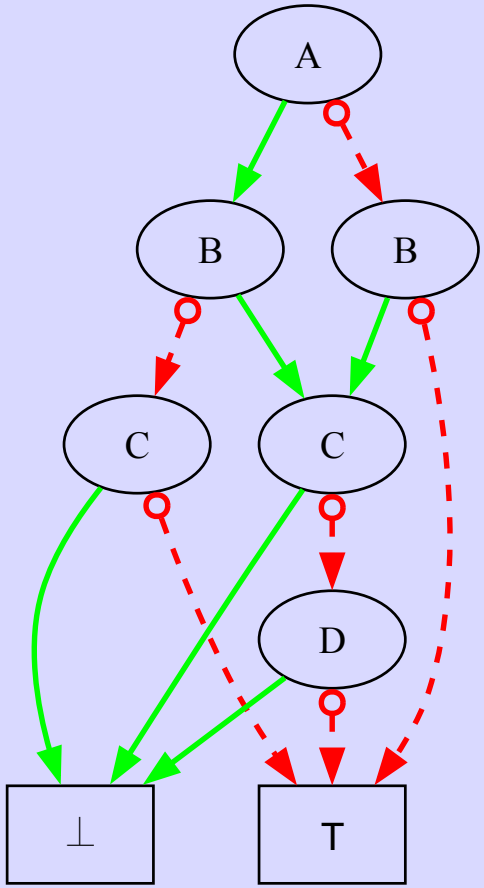
$$(Z_1 \wedge Z_2 \vee \neg Z_1 \wedge \neg Z_2) \oplus (Z_1 \wedge \neg Z_2 \wedge Z_3) = \neg Z_1 \wedge \neg Z_2 \vee Z_1 \wedge Z_2 \vee Z_1 \wedge \neg Z_2 \wedge Z_3$$



How can we implement BDDs in Scala?



How can we implement BDDs in Scala?



- ADT
- Factory functions
- Dynamic variable
- Weak hash table
- Implicit conversion

Scala code for Bdd constructors

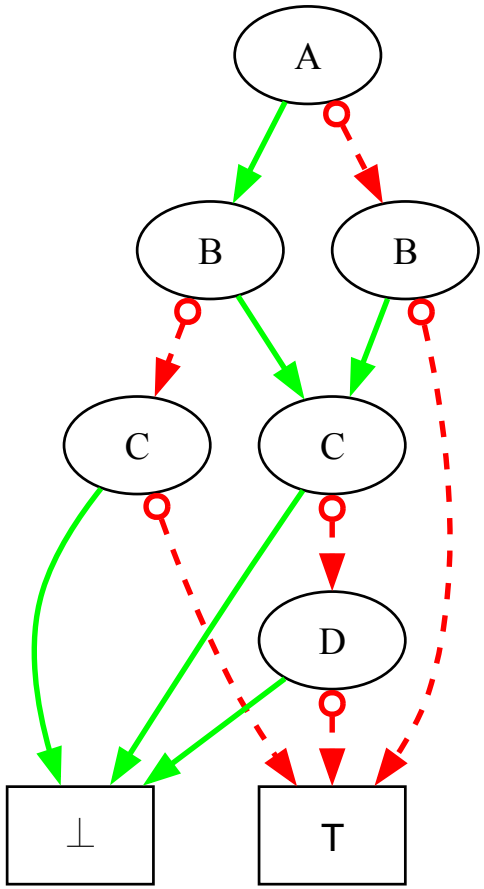
What we want: `And(1,Or(-2,3))` for

$$x_1 \wedge (\neg x_2 \vee x_3)$$

Strategy:

- First: we implement low level constructor: `BddNode(...)`
- Then: a smarter constructor: `Bdd(...)`
- Finally: Boolean algebra.

Reminder of structure of BDD



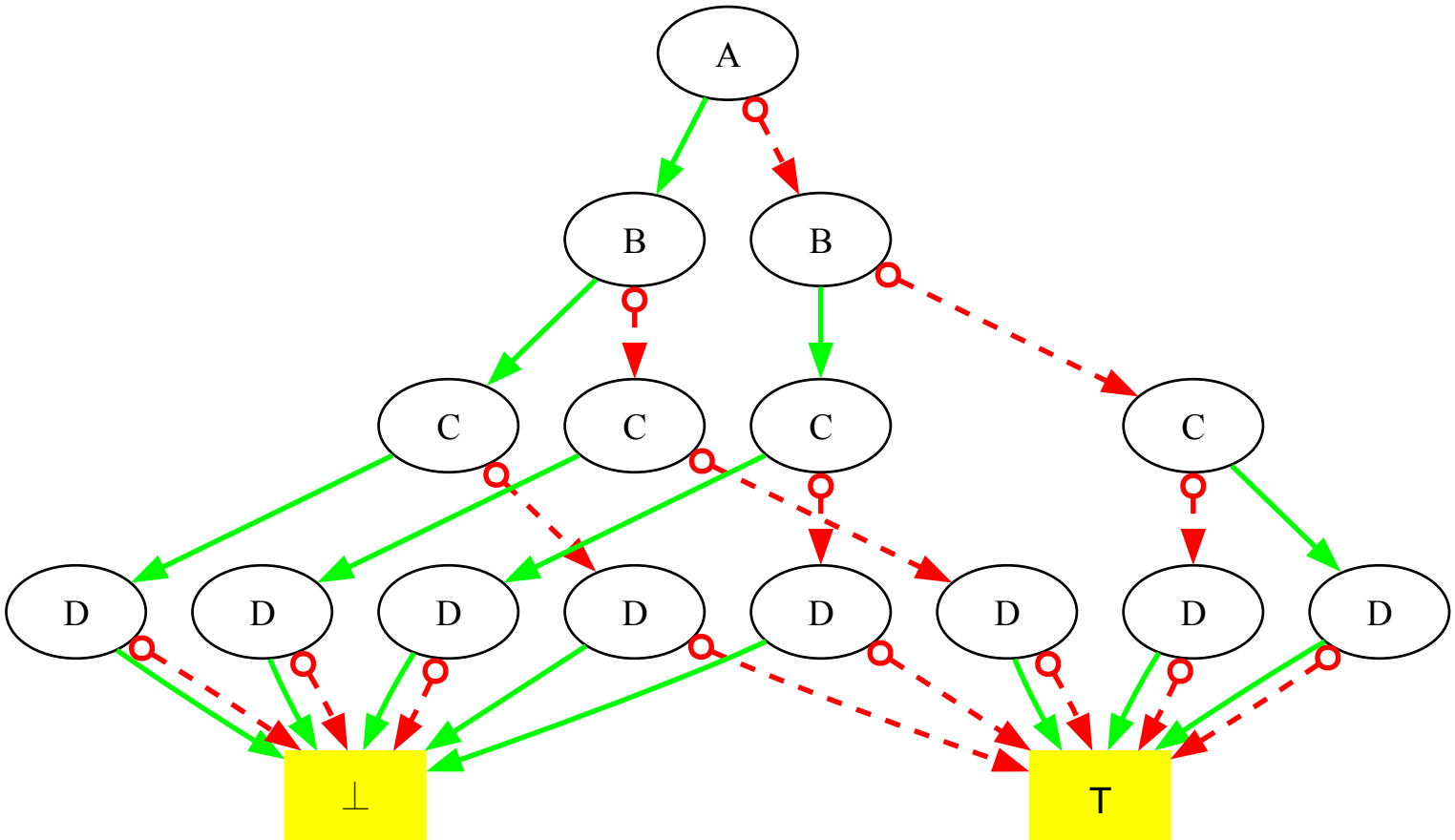
- Two node types: *internal* and *leaf*
- Some nodes have multiple parents
- Internal nodes have a label
- Internal nodes have exactly two children
- Two *leaf* types: *true* and *false*

Bdd class definition as Algebraic Data Type

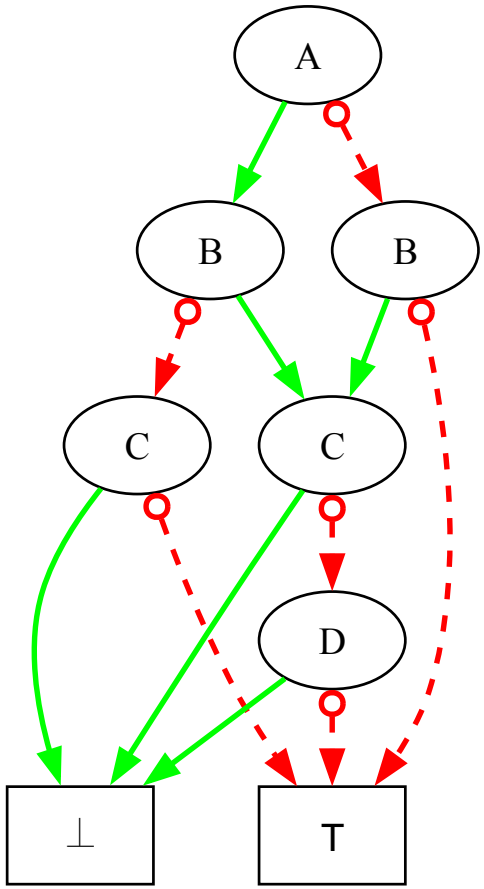
```
sealed abstract class Bdd {}

case class BddNode(label:Int, positive:Bdd, negative:Bdd)
    extends Bdd {}

sealed abstract class BddTerm extends Bdd {}

object BddTrue extends BddTerm {
  override def toString = "T"
}

object BddFalse extends BddTerm {
  override def toString = "F"
}
```

Reminder of *terminal* rule

The two ADT objects `BddTrue` and `BddFalse` automatically enforce the *terminal* rule.



Reminder of *variable ordering*



$$A < B < C < D$$

$$X_1 < X_2 < X_3 < X_4 < \ldots < X_n$$

Enforce variable ordering

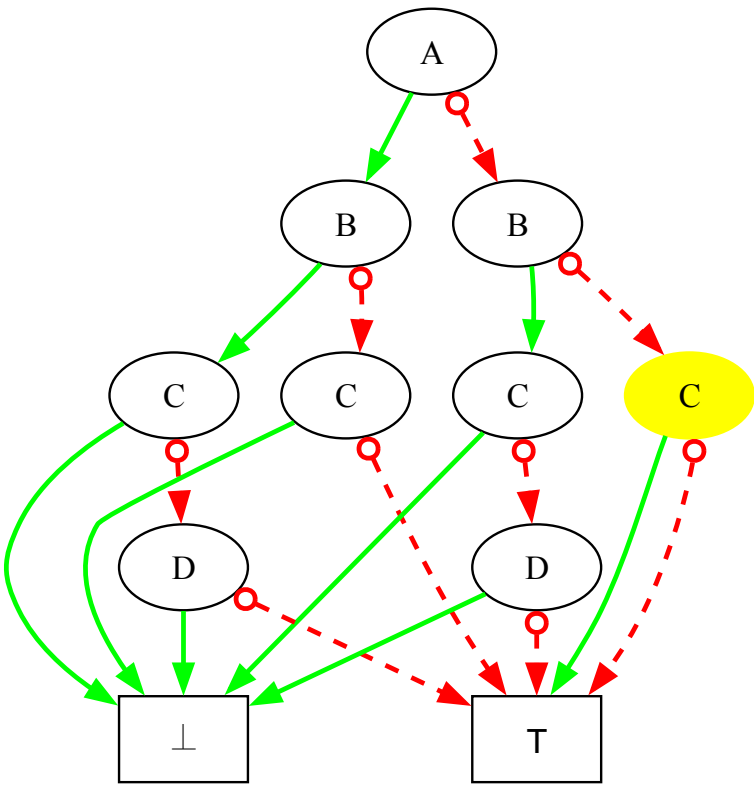
If a `BddNode` is a child of a `BddNode`, then we must enforce:

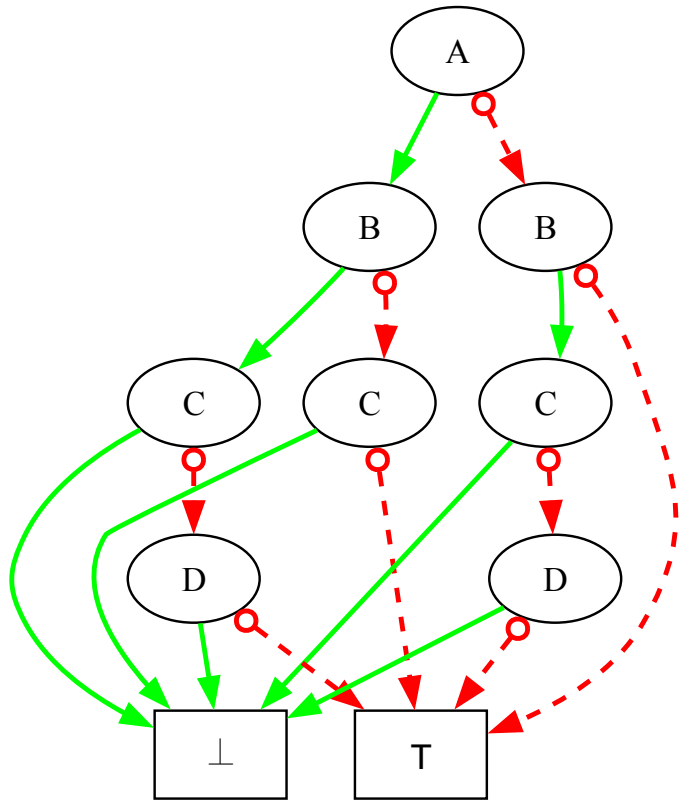
$$child.label > parent.label\ .$$

```
case class BddNode(label:Int, positive:Bdd, negative:Bdd)
    extends Bdd {
  ...
  def validateChild(child: Bdd): Unit = {
    child match {
      case child: BddNode => assert(child.label > label, ...)
      case _: BddTerm => ()
    }
  }
  validateChild(positive)
  validateChild(negative)
}
```

Reminder of *deletion* rule

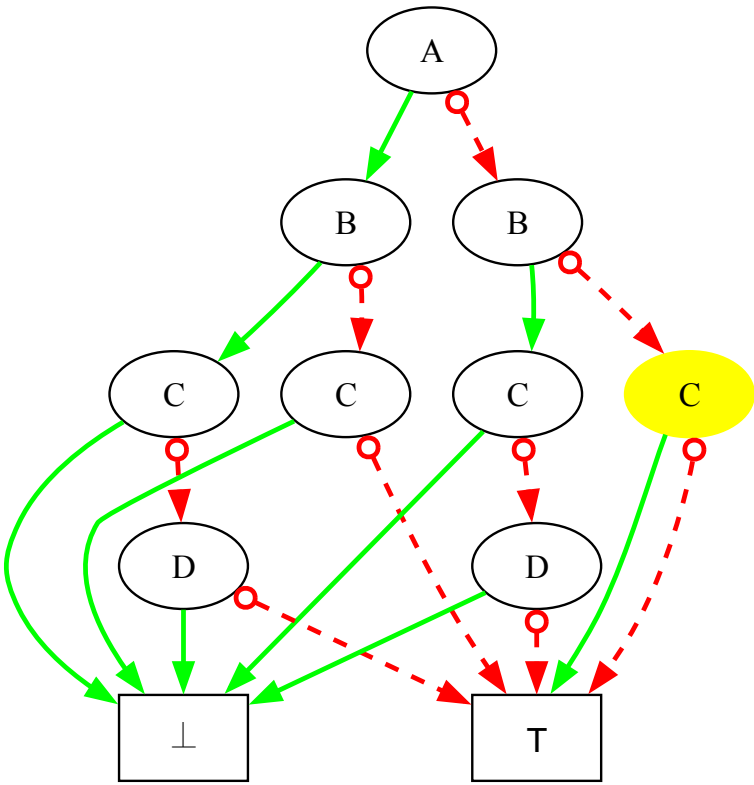
If both child nodes are same, then point to common child.





Reminder of *deletion* rule

If both child nodes are same, then point to common child.



Two possible implementation strategies:

1. Search for the node and delete it: $O(2^n)$ Bad idea
2. Prevent it by construction: $O(1)$ Good idea

Enforcing *deletion* rule

The *deletion* rule is enforced in the constructor `Bdd.apply` method, which tests the `positive` and `negative` children for equivalence.

```
object Bdd { // companion object

  def apply(label: Int, positive: Bdd, negative: Bdd): Bdd = {
    if (positive == negative)
      positive
    else {
      // This call to BddNode constructor is wrong!
      // It VIOLATES the merging rule.
      BddNode(label,positive,negative)
    }
  }
}
```

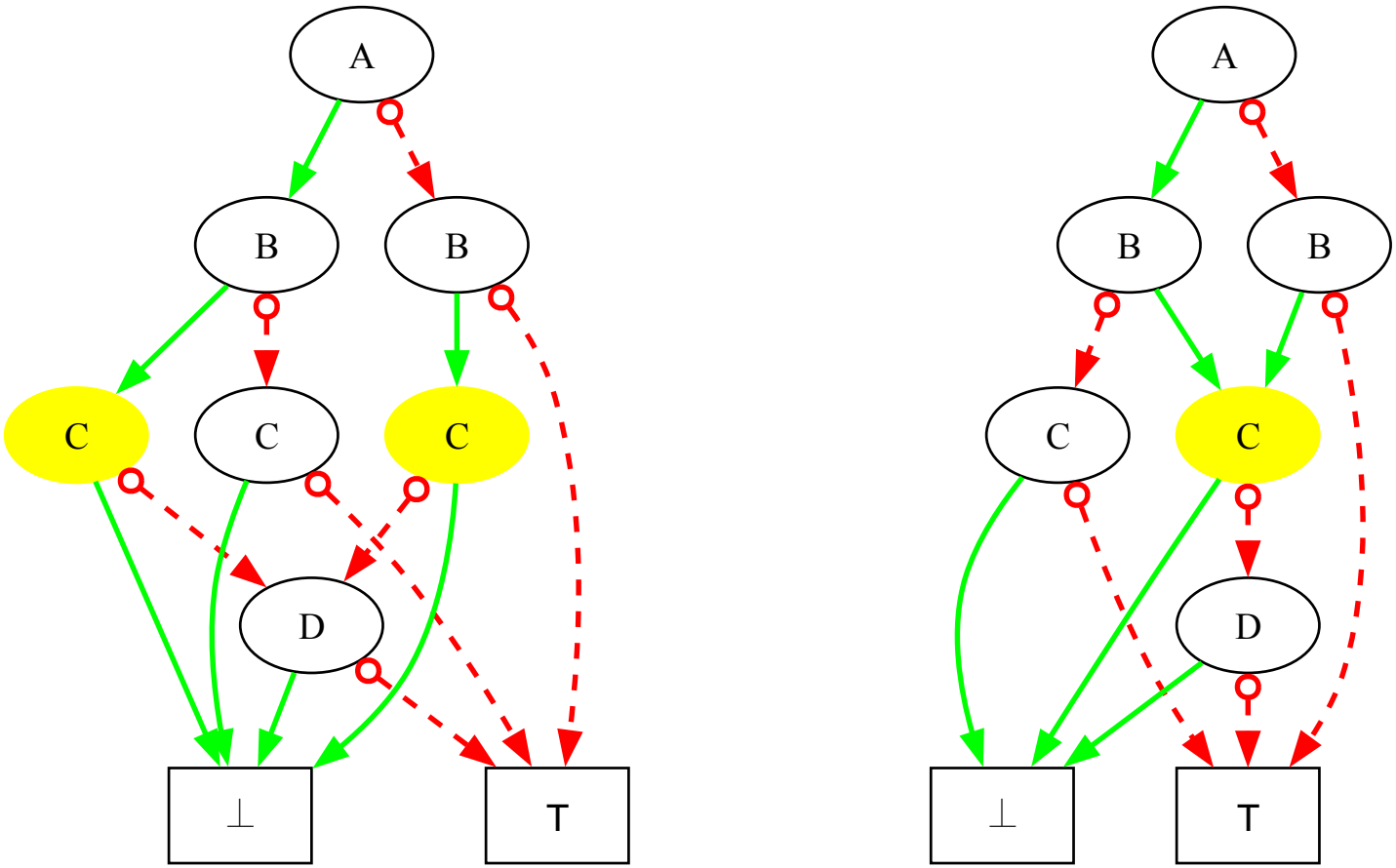
Override equivalence

BDDs are `equal` *only if* they are the exact same object, `eq`.
Avoid allocating the same object multiple times.

```
case class BddNode(label:Int, positive:Bdd, negative:Bdd)
    extends Bdd {

  override def equals(that: Any): Boolean = {
    that match {
      case b: Bdd => this eq b
      case _ => false
    }
  }

  override def hashCode(): Int =
    System.identityHashCode(this)
}
```

Reminder of *merging* rule

Congruent nodes are merged.



Searching for congruent nodes is slow. Therefore, we implement the *merging* rule, by ensuring that congruent nodes never get allocated.

Enforcing *merging* rule

We test whether the same `BddNode` has already been allocated and return that, else allocate a new one.

```
object Bdd {

  def apply(label: Int, positive: Bdd, negative: Bdd): Bdd = {
    if (positive == negative) positive
    else {
      maybeHash.value match {
        case None => ??? // later
        case Some(hash) =>
          hash.get((label, positive, negative)) match {
            case Some(bdd: BddNode) => bdd
            case None | Some(null) =>
              val bdd = BddNode(label, positive, negative)
              hash((label, positive, negative)) = bdd
              bdd
  }}}}
```

Enforcing *merging* rule

We test whether the same `BddNode` has already been allocated and return that, else allocate a new one.

```
object Bdd {

  def apply(label: Int, positive: Bdd, negative: Bdd): Bdd = {
    if (positive == negative) positive
    else {
      maybeHash.value match {
        case None => ??? // later
        case Some(hash) =>
          hash.get((label, positive, negative)) match {
            case Some(bdd: BddNode) => bdd
            case None | Some(null) =>
              val bdd = BddNode(label, positive, negative)
              hash((label, positive, negative)) = bdd
              bdd
  }}}}
```

Enforcing *merging* rule

We test whether the same `BddNode` has already been allocated and return that, else allocate a new one.

```
object Bdd {

  def apply(label: Int, positive: Bdd, negative: Bdd): Bdd = {
    if (positive == negative) positive
    else {
      maybeHash.value match {
        case None => ??? // later
        case Some(hash) =>
          hash.get((label, positive, negative)) match {
            case Some(bdd: BddNode) => bdd
            case None | Some(null) =>
              val bdd = BddNode(label, positive, negative)
              hash((label, positive, negative)) = bdd
              bdd
  }}}}
```

Enforcing *merging* rule

We test whether the same `BddNode` has already been allocated and return that, else allocate a new one.

```
object Bdd {

  def apply(label: Int, positive: Bdd, negative: Bdd): Bdd = {
    if (positive == negative) positive
    else {
      maybeHash.value match {
        case None => ??? // later
        case Some(hash) =>
          hash.get((label, positive, negative)) match {
            case Some(bdd: BddNode) => bdd
            case None | Some(null) =>
              val bdd = BddNode(label, positive, negative)
              hash((label, positive, negative)) = bdd
              bdd
  }}}}
```

Enforcing *merging* rule

We test whether the same `BddNode` has already been allocated and return that, else allocate a new one.

```
object Bdd {

  def apply(label: Int, positive: Bdd, negative: Bdd): Bdd = {
    if (positive == negative) positive
    else {
      maybeHash.value match {
        case None => ??? // later
        case Some(hash) =>
          hash.get((label, positive, negative)) match {
            case Some(bdd: BddNode) => bdd
            case None | Some(null) =>
              val bdd = BddNode(label, positive, negative)
              hash((label, positive, negative)) = bdd
              bdd
  }}}}
```

Enforcing *merging* rule with Dynamic Variables

Provide a *dynamic extent* for `BddNode` construction and computation via `scala.util.DynamicVariable` and *call-by-name*.

```
object Bdd {
  ...
  import scala.util.DynamicVariable

  type BDD_HASH = mutable.Map[(Int, Bdd, Bdd), BddNode]
  val maybeHash = new DynamicVariable[Option[BDD_HASH]](None)

  def withNewBddHash[A](delayedCode: => A): A = {
    maybeHash.withValue(Some(newHash())) { delayedCode }
  }
  def newHash(): BDD_HASH = ???
}
```

Enforcing *merging* rule with Dynamic Variables

Provide a *dynamic extent* for `BddNode` construction and computation via `scala.util.DynamicVariable` and *call-by-name*.

```
object Bdd {
  ...
  import scala.util.DynamicVariable

  type BDD_HASH = mutable.Map[(Int, Bdd, Bdd), BddNode]
  val maybeHash = new DynamicVariable[Option[BDD_HASH]](None)

  def withNewBddHash[A](delayedCode: => A): A = {
    maybeHash.withValue(Some(newHash())) { delayedCode }
  }
  def newHash(): BDD_HASH = ???
}
```

Enforcing *merging* rule with Dynamic Variables

Provide a *dynamic extent* for `BddNode` construction and computation via `scala.util.DynamicVariable` and *call-by-name*.

```
object Bdd {
  ...
  import scala.util.DynamicVariable

  type BDD_HASH = mutable.Map[(Int, Bdd, Bdd), BddNode]
  val maybeHash = new DynamicVariable[Option[BDD_HASH]](None)

  def withNewBddHash[A](delayedCode: => A): A = {
    maybeHash.withValue(Some(newHash())) { delayedCode }
  }
  def newHash(): BDD_HASH = ???
}
```

Enforcing *merging* rule

Trigger a `sys.error` if attempt to allocate BDD outside dynamic extent of `Bdd.withNewHash`.

```
object Bdd {

  def apply(label: Int, positive: Bdd, negative: Bdd): Bdd = {
    if (positive == negative) positive
    else {
      maybeHash.value match {
        case None => sys.error("called outside dynamic extent of
            withNewBddHash(...)")
        case Some(hash) => ...
      }
    }
  }
  ...
}
```

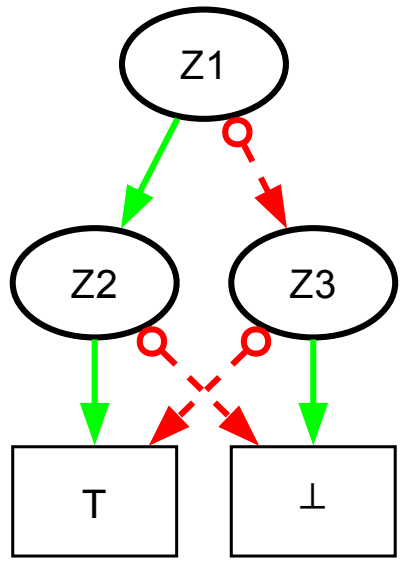
Weak Values Hash Map

Use `org.jboss.util.collection.WeakValueHashMap` to allow entries to be removed from hash and GC'ed once no longer referenced *elsewhere*.

```
object Bdd {
  ...
  def newHash(): BDD_HASH = {
    import scala.collection.JavaConverters._
    import org.jboss.util.collection._

    new WeakValueHashMap[(Int, Bdd, Bdd), BddNode].asScala
  }
}
```

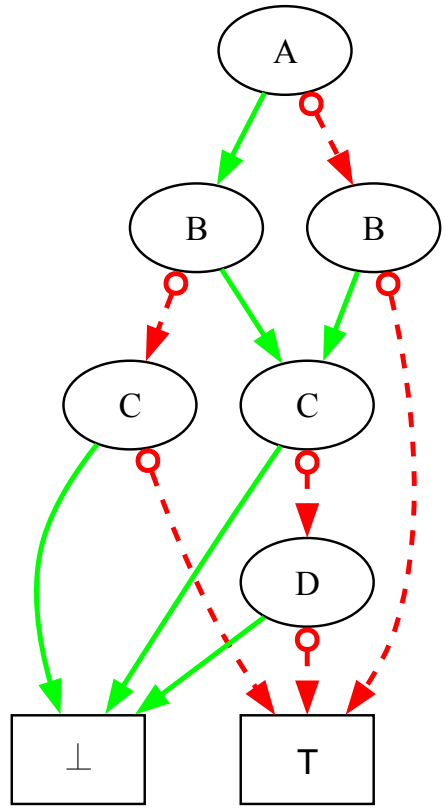
Bdd construction examples

```
withNewBddHash {
  val z3 = Bdd(3, BddFalse, BddTrue)
  val z2 = Bdd(2, BddTrue, BddFalse)

  Bdd(1, z2, z3)
}
```



Bdd construction examples

```
withNewBddHash{
  val d = Bdd(4, BddFalse, BddTrue)
  val c1 = Bdd(3, BddFalse, BddTrue)
  val c2 = Bdd(3, BddFalse, d)
  val b1 = Bdd(2, c2, c1)
  val b2 = Bdd(2, c2, BddTrue)

  Bdd(1, b2, b1) // ** note
  fslow(Bdd(1, b1, b2))
}
```



** While `fslow` is running, `Bdd(1,b2,b1)` is no longer referenced, except within the `WeakValueHashMap`. Since the reference is weak, the object *can be* GC'ed, and auto-deleted from the `WeakValueHashMap`. Afterward `hash.get(1,b2,b1)` may return `Some(null)`.

Now the *meat* of the Bdd class is finished.
We need the syntax sugar.

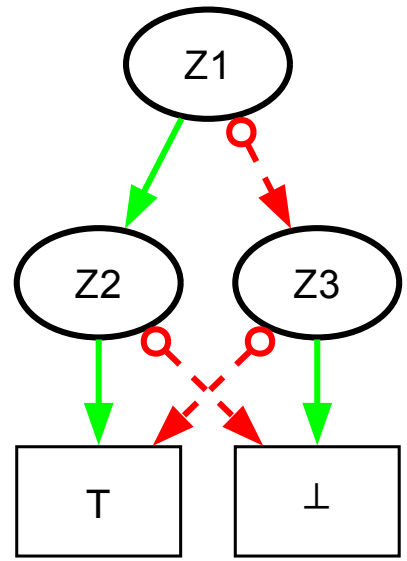
Easier construction for common cases: unary `Bdd.apply`

Provide *easier* unary constructor for two common cases.
`Bdd(1)` for `Bdd(1,BddTrue,BddFalse)`, and
`Bdd(-2)` for `Bdd(2,BddFalse,BddTrue)`.

```
object Bdd {

  def apply(label: Int): Bdd = {
    require(label != 0)
    if (label > 0)
      Bdd(label, BddTrue, BddFalse)
    else // ( label < 0)
      Bdd(-label, BddFalse, BddTrue)
  }
  // we saw the 3-ary constructor already
  def apply(label: Int, positive: Bdd, negative: Bdd): Bdd = ...
  ...
}
```

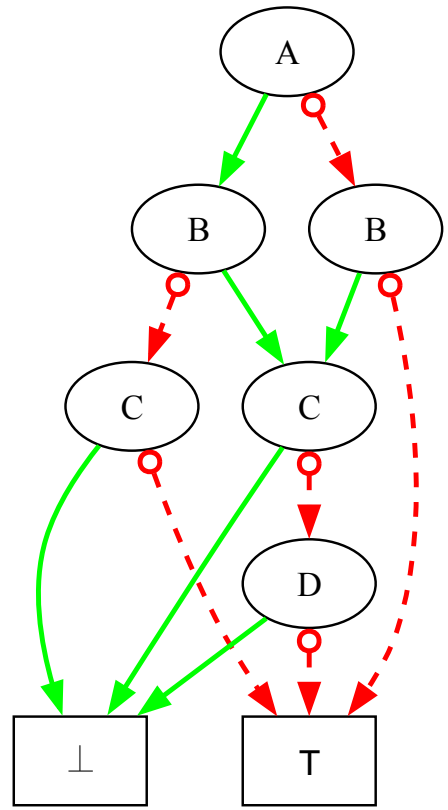
Same Bdd construction examples, simplified

```
import bdd._
import bdd.Bdd._

withNewBddHash {
  val z2 = Bdd(2)
  val z3 = Bdd(-3)

  Bdd(1, z2, z3)
}
```



Same Bdd construction examples, simplified

```
import bdd._
import bdd.Bdd._

withNewBddHash{
  val d = Bdd(-4)
  val c1 = Bdd(-3)
  val c2 = Bdd(3, BddFalse, d)
  val b1 = Bdd(2, c2, c1)
  val b2 = Bdd(2, c2, BddTrue)

  Bdd(1, b1, b2)
}
```

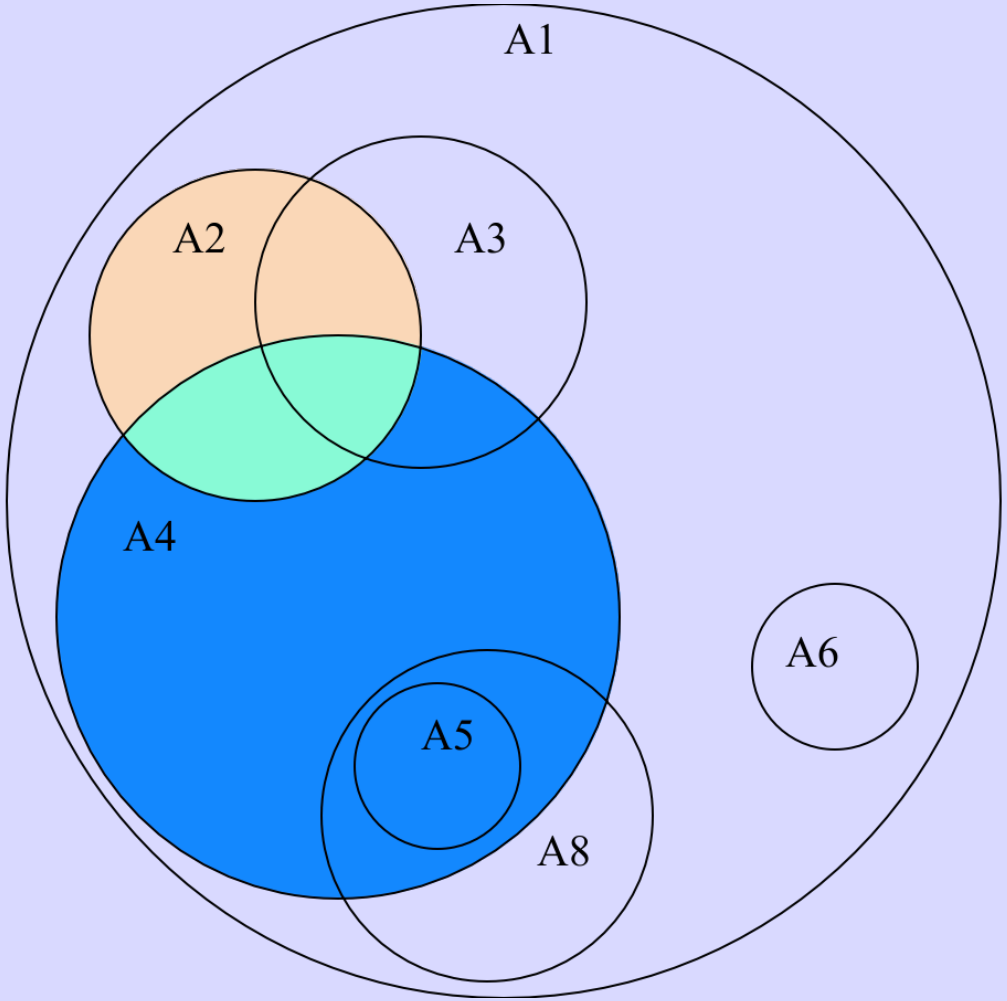


Construction structurally vs via Boolean formula

BDDs are rarely constructed from their structure (except for test cases); rather, in practice they are constructed from Boolean formulas.

We would like to extend our Scala implementation to accommodate constructing BDDs given the Boolean formulas they represent.

How to implement Boolean operations between BDDs?



Representations of Boolean Functions

- We would like to write something isomorphic to the following.

$$x_1 \vee x_2 \vee \neg x_3 \vee (\neg x_1 \wedge x_4) \vee (x_2 \wedge \neg(x_1 \vee x_3))$$

Representations of Boolean Functions

- We would like to write something isomorphic to the following.

$$x_1 \vee x_2 \vee \neg x_3 \vee (\neg x_1 \wedge x_4) \vee (x_2 \wedge \neg(x_1 \vee x_3))$$

- In prefix notation it looks like this:

$$Or(x_1, x_2, Not(x_3), And(Not(x_1), x_4), And(x_2, Not(Or(x_1, x_3))))$$

Representations of Boolean Functions

- We would like to write something isomorphic to the following.

$$x_1 \vee x_2 \vee \neg x_3 \vee (\neg x_1 \wedge x_4) \vee (x_2 \wedge \neg(x_1 \vee x_3))$$

- In prefix notation it looks like this:

$$Or(x_1, x_2, Not(x_3), And(Not(x_1), x_4), And(x_2, Not(Or(x_1, x_3))))$$

- Common convention: positive and negative integers represent *positive* and *negative* Boolean variable literals:
  - $1 \rightarrow x_1$
  - $-2 \rightarrow \neg x_2$

Representations of Boolean Functions

- We would like to write something isomorphic to the following.

$$x_1 \vee x_2 \vee \neg x_3 \vee (\neg x_1 \wedge x_4) \vee (x_2 \wedge \neg(x_1 \vee x_3))$$

- In prefix notation it looks like this:

$$Or(x_1, x_2, Not(x_3), And(Not(x_1), x_4), And(x_2, Not(Or(x_1, x_3))))$$

- Common convention: positive and negative integers represent *positive* and *negative* Boolean variable literals:
  - $1 \rightarrow x_1$
  - $-2 \rightarrow \neg x_2$
- This makes for a concise Scala DSL notation.

```
Or(1, 2, -3, And(-1, 4), And(2, Not(Or(1, 3))))
```

Heterogeneous typed arguments of var-args functions

The functions `And`, `Or`, `Not` return an instance of `Bdd`.

```
Or(1, 2, -3, And(-1, 4), And(2, Not(Or(1, 3))))
```

We see from expression that `And` and `Or` must be implemented to accept multiple arguments whose types are `Int` or `Bdd`.

```
def Or(args:List[Bdd | Int]):Bdd = {...} // wrong
```

Heterogeneous typed arguments of var-args functions

The functions `And`, `Or`, `Not` return an instance of `Bdd`.

```
Or(1, 2, -3, And(-1, 4), And(2, Not(Or(1, 3))))
```

We see from expression that `And` and `Or` must be implemented to accept multiple arguments whose types are `Int` or `Bdd`.

```
def Or(args:List[Bdd | Int]):Bdd = {...} // wrong
```

However; *union types* sadly do not exist in Scala 2.x.

Heterogeneous typed arguments of var-args functions

The functions `And`, `Or`, `Not` return an instance of `Bdd`.

```
Or(1, 2, -3, And(-1, 4), And(2, Not(Or(1, 3))))
```

We see from expression that `And` and `Or` must be implemented to accept multiple arguments whose types are `Int` or `Bdd`.

```
def Or(args:List[Bdd | Int]):Bdd = {...} // wrong
```

However; *union types* sadly do not exist in Scala 2.x.

But wait! Isn't `Either[]` a union type?

`Either` is not Or

Despite the English language pun, `Either[]` is not a union type. It is evidence of a type system that does not have a union type. `Either` is an attempt to fix it in the user space, and you can't.[Rich Hickey]

$$Int \subset Int \cup Bdd$$
$$Int \cup Bdd = Bdd \cup Int$$
$$Int = Int \cup Int$$

$$Int \not\subset Either[Int, Bdd]$$
$$Either[Int, Bdd] \not\subset Either[Bdd, Int]$$
$$Int \neq Either[Int, Int]$$

`Either` does not have any mathematical properties of Or. It is not associative, not commutative...

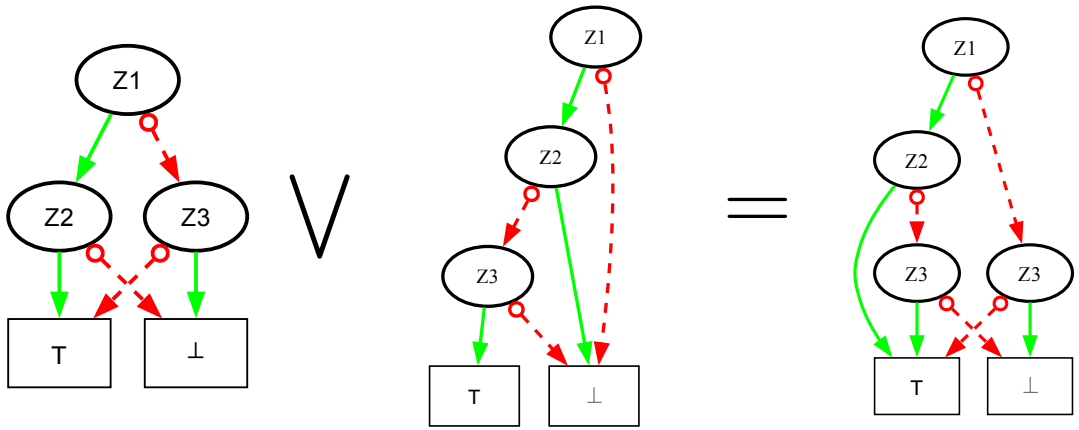
Problem: We need: heterogeneously typed var-args functions.

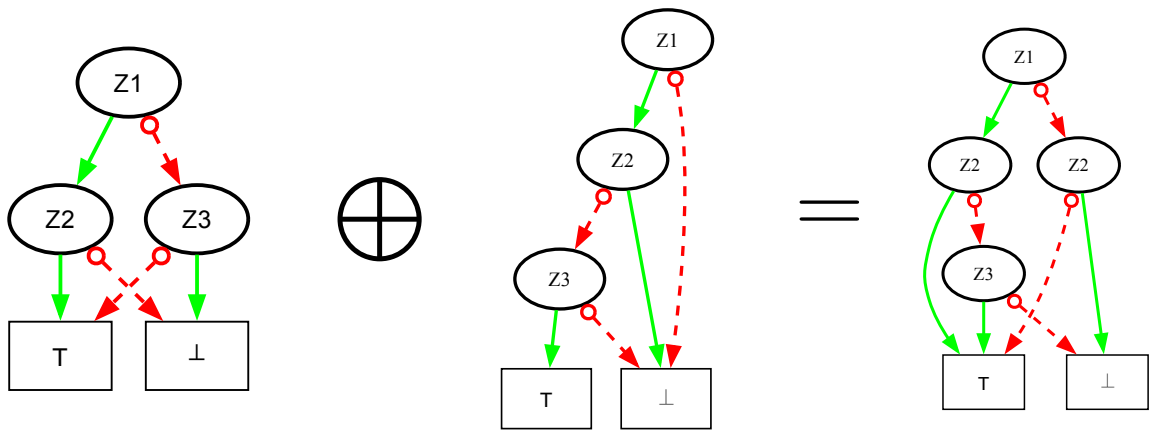
How can we implement them?

First we look at the binary function case.

We'll return later to the problematic case.

Example. Boolean operations `Or` and `Xor`

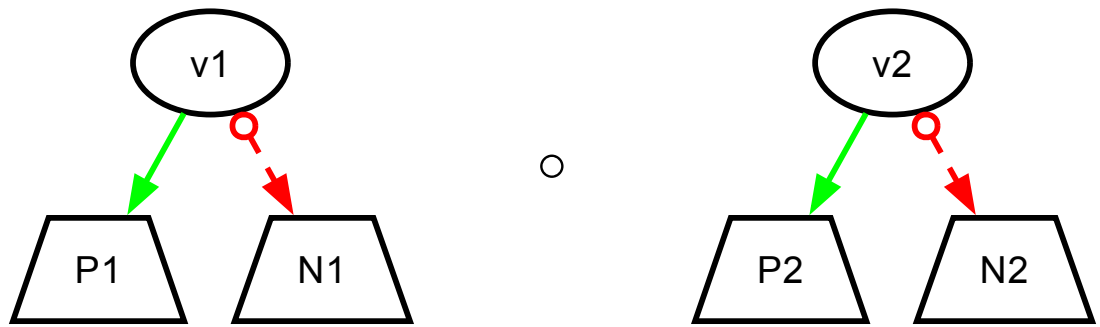




All binary operations follow the same recursive pattern.

$$B_1 = BddNode(v_1, P_1, N_1)$$
$$B_2 = BddNode(v_2, P_2, N_2)$$

$B_1 \circ B_2 =$



$$B_1 \circ B_2 = \begin{cases} BddNode(v_1,\ (P_1 \circ P_2),\ (N_1 \circ N_2)) & \text{for } v_1 = v_2 \\ BddNode(v_1,\ (P_1 \circ B_2),\ (N_1 \circ B_2)) & \text{for } v_1 < v_2 \\ BddNode(v_2,\ (B_1 \circ P_2),\ (B_1 \circ N_2)) & \text{for } v_1 > v_2 \end{cases}$$

Define Binary operations as ADT

`BinaryOperation` class implements 0-ary, unary, and binary cases as `apply` methods.

```
sealed abstract class BinaryOperation() {
  def apply():Bdd
  def apply(bdd:Bdd):Bdd
  def apply(b1: Bdd, b2: Bdd): Bdd
  ...
}
object Or extends BinaryOperation {
  def apply():Bdd = BddFalse
  def apply(bdd:Bdd):Bdd = bdd
  def apply(b1:Bdd,b2:Bdd):Bdd = ???
}
object And extends BinaryOperation { ... }
object Xor extends BinaryOperation { ... }
object AndNot extends BinaryOperation { ... }
```

Fill in the apply methods for And, Or, etc

Notice that Bdd.bddOp is a recursive definition with not termination clauses. Recursion termination is provided by objects which extend BinaryOperation.

```
object Or extends BinaryOperation {
  ...
  def apply(b1:Bdd,b2:Bdd):Bdd = {
    (b1,b2) match {
      case (_,_) if b1 eq b2 => b1
      case (BddTrue,_) => BddTrue
      case (BddFalse,_) => b2
      case (_,BddTrue) => BddTrue
      case (_,BddFalse) => b1
      case (b1:BddNode,b2:BddNode) => bddOp(b1,b2)
    }
  }
}
```

Generic operation as `BinaryOperation.bddOp` method

$$B_1 = BddNode(v_1, P_1, N_1) \quad B_2 = BddNode(v_2, P_2, N_2)$$

$$B_1 \circ B_2 = \begin{cases} Bdd(v_1,\ (P_1 \circ P_2),\ (N_1 \circ N_2)) & \text{for } v_1 = v_2 \\ Bdd(v_1,\ (P_1 \circ B_2),\ (N_1 \circ B_2)) & \text{for } v_1 < v_2 \\ Bdd(v_2,\ (B_1 \circ P_2),\ (B_1 \circ N_2)) & \text{for } v_1 > v_2 \end{cases}$$

```
object BinaryOperation { ...
  def bddOp(b1: BddNode, b2: BddNode): Bdd =
    val ( BddNode(l1,p1,n1), BddNode(l2,p2,n2) ) = (b1,b2)
    if (l1 == l2)
        Bdd(l1, apply(p1, p2), apply(n1, n2))
    else if (l2 < l2)
        Bdd(l1, apply(p1, b2), apply(n1, b2))
    else
        Bdd(l2, apply(b1, p2), apply(b1, n2))
}
```

Examples of binary operations, limitations

We may use the following *binary* notation:

```
withNewBddHash {
  Or( Bdd(1),
      Or( Bdd(-2), Bdd(-3))
}
```

But the notation is awkward. We'd like to use a much simpler form

```
withNewBddHash {
  Or(1,-2,-3) // fails
}
```

Examples of binary operations, limitations

Furthermore, we can't pass integers directly to `And`, `Or`, nor pass multiple arguments.

```
withNewBddHash {
  Or(1,-2) // fails
  And(1,Or(3,4)) // fails
  Xor(1,Or(3,4),And(2,-1)) // fails
}
```

Remember, we promised to return later to this problem.

Later is now.

Implicit conversions

We could define $2^2 = 4$ `Bdd.bddOp` methods for all combinations of `Bdd` and `Int`;
likewise $2^3 = 8$ 3-ary methods; and
likewise $2^4 = 16$ 4-ary methods;
up to $2^{22} > 4 \times 10^6$.

Instead, we define an `implicit` conversion.

```
object Bdd {
  ...
  implicit def int2bdd(raw: Int): Bdd = Bdd(raw)
}
```

Some Scala experts warn against this kind of `implicit` conversion, but I don't see any other option as Scala 2.x does not support union types.

Implicit conversions

The `implicit` allows the following syntax:

```
withNewBddHash {
  Not(Or(Xor(Bdd(1),
             And(Bdd(-2), Bdd(-3))),
         AndNot(Bdd(2), Bdd(3))))
}
```

to be expressed much simpler as

```
withNewBddHash {
  Not(Or(Xor(1, And(-2,-3)),
         AndNot(2,3)))
}
```

But it still only works for *binary function* application.

Var-args Bdd.bddOp method

For var-args Boolean operators, we declare a var-args constructor, for BinaryOperation.apply method which delegates to reduceLeft[1] of binary operations.

```
sealed abstract class BinaryOperation() {
  ...
  def apply(args: Bdd*): Bdd = apply(args.toList)

  def apply(args:List[Bdd]):Bdd = {
    args match {
      case Nil => apply()
      case bdd::Nil => apply(bdd)
      case _ => args.reduceLeft{(acc,bdd) => apply(acc,bdd)}
    }
  }
}
```

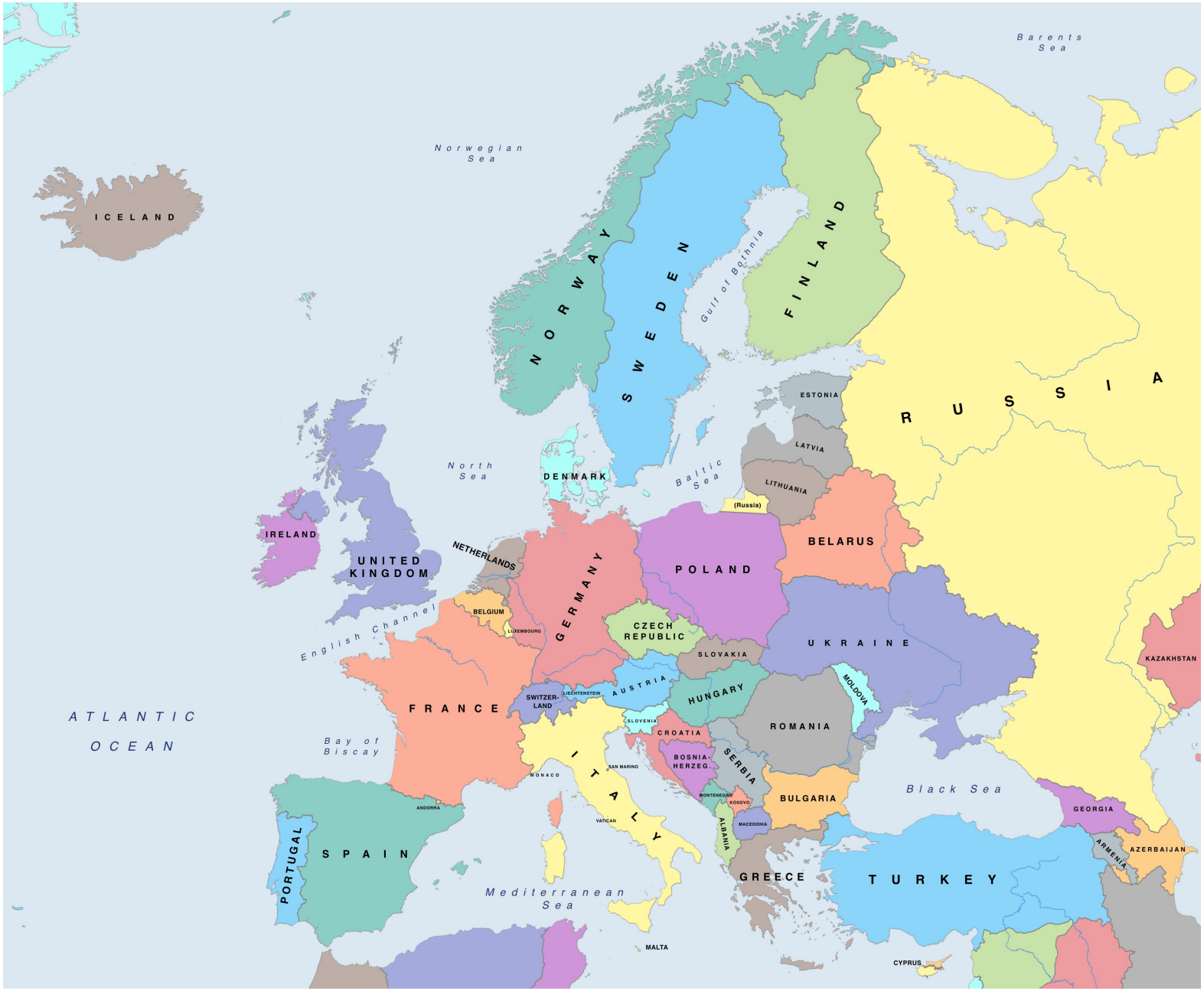
[1] reduceLeft has poor performance in some cases, reduceTree would be better.

Example of var-args Boolean operations

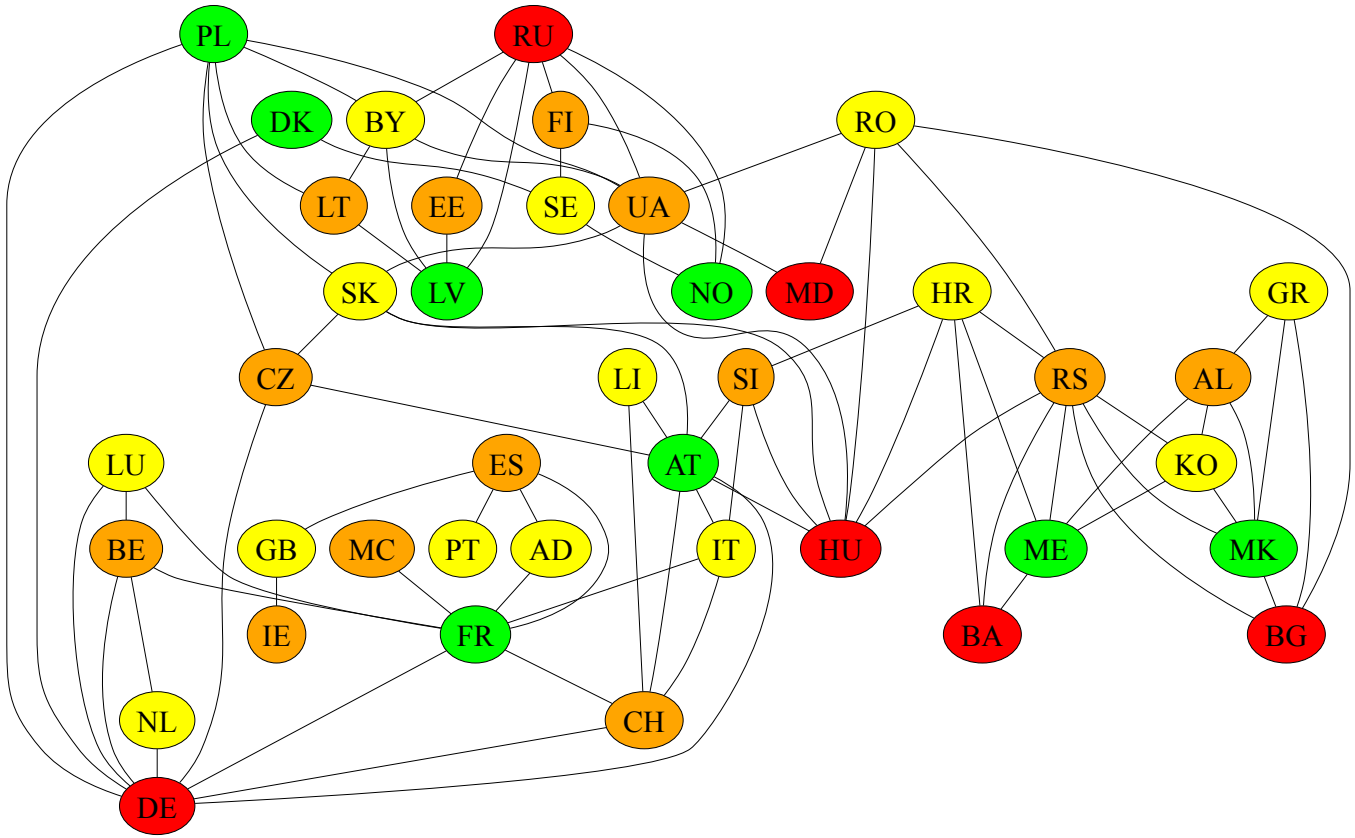
Mix and match `Int` with `Bdd` and arbitrarily many arguments.

```
withNewBddHash {
  Not(Or(-2,
         3,
         Xor(1, 2, And(-2,-3,4)),
         AndNot(2,3)))
}
```

4-color Map Problem



Back to the four color problem



Back to the four color problem

Construct a Boolean function, by conjoining all the *border constraints*.

$$(\mathsf{pt} \to \mathsf{es}) \wedge (\mathsf{es} \to \mathsf{fr}) \wedge (\mathsf{fr} \to \mathsf{de}) \wedge (\mathsf{fr} \to \mathsf{be}) \ldots$$

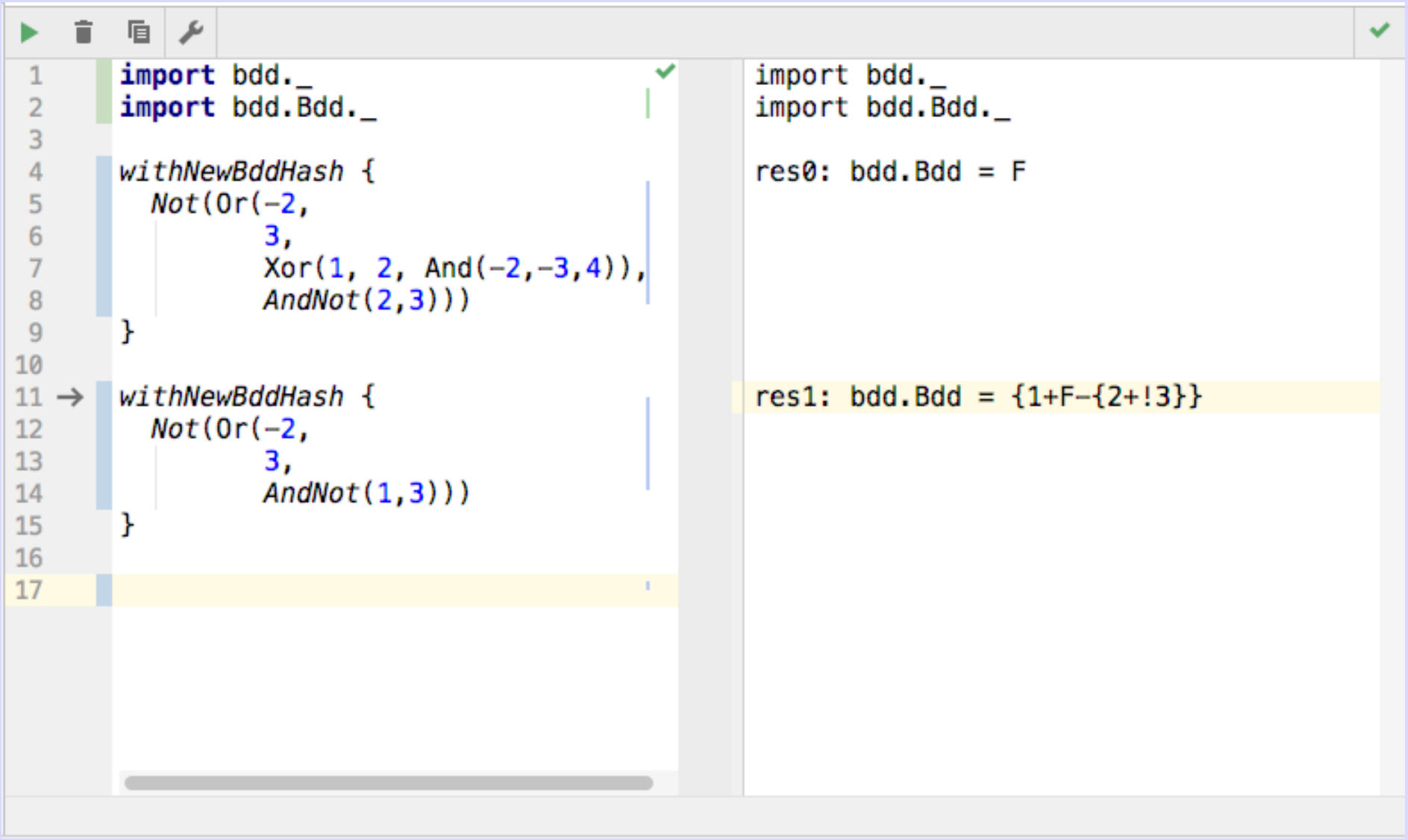
The entire function is generated via `fold`:

$$\bigwedge_{(\alpha,\beta)\in \mathit{borders}} (A_\alpha \oplus A_\beta) \vee (B_\alpha \oplus B_\beta)$$

For 40 European countries, this is a Boolean function of 80 variables.

Find assignments for 80 Boolean variables to *satisfy* a Boolean function.

Demo Playing with BDDs



Perspectives and Open Questions

Fully functional implementation

It is not clear how to best implement BDDs fully functionally.

Current area of research.

Decrease memory consumption

Current usage of `org.jboss.util.collection.WeakValueHashMap` consume HUGE amount of memory.

Consider refactoring to use `WeakSet`.

Serialization

How to serialize disjunctive form. In the Common Lisp implementation we serialized to s-expression such as:

```
(or (and a b) (and b (not c)))
```

We don't currently use the `symbol` data type.
It is being deprecated in Scala 3.

Inefficient reduction from Scala standard library

```
sealed abstract class BinaryOperation() {
  ...
  def apply(args:List[Bdd]):Bdd = {
    args match {
      ...
      case _ => args.reduceLeft{(acc,bdd) => apply(acc,bdd)}
    }
  }
}
```

To call `reduceLeft` is sometimes inefficient. We need `reduceTree`.

- $((((((((x_1 + x_2) + x_3) + x_4) + x_5) + x_6) + x_7) + x_8)...)$
- $\Big(\big((x_1 + x_2) + (x_3 + x_4)\big) + \big((x_5 + x_6) + (x_7 + x_8)\big) + ...\Big)$

Representing variable ordering

BDD structure heavily depends on variable ordering. We'd like to support different orderings, but disallow invalid Boolean combinations of `Bdd` instances which assume different ordering.

This means at run-time different `Bdd` instances need to support different orderings $v_1 < v_2$, and the `HashMap` must understand or at least validate consistent orderings.

Conclusion

- BDDs express Boolean Functions canonically and succinctly
- Scala implementation of Binary Decision Diagrams
- ADT representation
- Implementation of Boolean algebra operations
- Code available at `https://gitlab.lrde.epita.fr/jnewton/regular-type-expression.git`

Q/A

Questions?

Example of `foldLeft` adding `Rational` numbers

$$\frac{1}{23}+\frac{1}{29}+\frac{1}{31}+\frac{1}{37}+\frac{1}{41}+\frac{1}{43}+\frac{1}{47}+\frac{1}{53}+\frac{1}{57}+\frac{1}{67}$$

$$\begin{aligned}
\frac{1}{23}+\frac{1}{29} &= \frac{52}{667}\\
\frac{52}{667}+\frac{1}{31} &= \frac{2279}{20677}\\
\frac{2279}{20677}+\frac{1}{37} &= \frac{105000}{765049}\\
\frac{105000}{765049}+\frac{1}{41} &= \frac{5070049}{31367009}\\
\frac{5070049}{31367009}+\frac{1}{43} &= \frac{249379116}{1348781387}\\
\frac{249379116}{1348781387}+\frac{1}{47} &= \frac{13069599839}{63392725189}\\
\frac{13069599839}{63392725189}+\frac{1}{53} &= \frac{756081516656}{3359814435017}\\
\frac{756081516656}{3359814435017}+\frac{1}{57} &= \frac{46456460884409}{191509422795969}\\
\frac{46456460884409}{191509422795969}+\frac{1}{67} &= \frac{3304092302051372}{12831131327329923}
\end{aligned}$$

Example of `foldLeft` adding `Rational` numbers

$$(\frac{1}{23} + \frac{1}{29}) + \frac{1}{31} + \frac{1}{37} + \frac{1}{41} + \frac{1}{43} + \frac{1}{47} + \frac{1}{53} + \frac{1}{57} + \frac{1}{67}$$

$$\begin{aligned}
\frac{1}{23} + \frac{1}{29} &= \frac{52}{667} \\
\frac{52}{667} + \frac{1}{31} &= \frac{2279}{20677} \\
\frac{2279}{20677} + \frac{1}{37} &= \frac{105000}{765049} \\
\frac{105000}{765049} + \frac{1}{41} &= \frac{5070049}{31367009} \\
\frac{5070049}{31367009} + \frac{1}{43} &= \frac{249379116}{1348781387} \\
\frac{249379116}{1348781387} + \frac{1}{47} &= \frac{13069599839}{63392725189} \\
\frac{13069599839}{63392725189} + \frac{1}{53} &= \frac{756081516656}{3359814435017} \\
\frac{756081516656}{3359814435017} + \frac{1}{57} &= \frac{46456460884409}{191509422795969} \\
\frac{46456460884409}{191509422795969} + \frac{1}{67} &= \frac{3304092302051372}{12831131327329923}
\end{aligned}$$

Example of `foldLeft` adding `Rational` numbers

$$(\frac{52}{667} + \frac{1}{31}) + \frac{1}{37} + \frac{1}{41} + \frac{1}{43} + \frac{1}{47} + \frac{1}{53} + \frac{1}{57} + \frac{1}{67}$$

$$\begin{aligned}
\frac{1}{23} + \frac{1}{29} &= \frac{52}{667} \\
\frac{52}{667} + \frac{1}{31} &= \frac{2279}{20677} \\
\frac{2279}{20677} + \frac{1}{37} &= \frac{105000}{765049} \\
\frac{105000}{765049} + \frac{1}{41} &= \frac{5070049}{31367009} \\
\frac{5070049}{31367009} + \frac{1}{43} &= \frac{249379116}{1348781387} \\
\frac{249379116}{1348781387} + \frac{1}{47} &= \frac{13069599839}{63392725189} \\
\frac{13069599839}{63392725189} + \frac{1}{53} &= \frac{756081516656}{3359814435017} \\
\frac{756081516656}{3359814435017} + \frac{1}{57} &= \frac{46456460884409}{191509422795969} \\
\frac{46456460884409}{191509422795969} + \frac{1}{67} &= \frac{3304092302051372}{12831131327329923}
\end{aligned}$$

Example of `foldLeft` adding `Rational` numbers

$$(\frac{2279}{20677} + \frac{1}{37}) + \frac{1}{41} + \frac{1}{43} + \frac{1}{47} + \frac{1}{53} + \frac{1}{57} + \frac{1}{67}$$

$$\begin{aligned}
\frac{1}{23} + \frac{1}{29} &= \frac{52}{667} \\
\frac{52}{667} + \frac{1}{31} &= \frac{2279}{20677} \\
\frac{2279}{20677} + \frac{1}{37} &= \frac{105000}{765049} \\
\frac{105000}{765049} + \frac{1}{41} &= \frac{5070049}{31367009} \\
\frac{5070049}{31367009} + \frac{1}{43} &= \frac{249379116}{1348781387} \\
\frac{249379116}{1348781387} + \frac{1}{47} &= \frac{13069599839}{63392725189} \\
\frac{13069599839}{63392725189} + \frac{1}{53} &= \frac{756081516656}{3359814435017} \\
\frac{756081516656}{3359814435017} + \frac{1}{57} &= \frac{46456460884409}{191509422795969} \\
\frac{46456460884409}{191509422795969} + \frac{1}{67} &= \frac{3304092302051372}{12831131327329923}
\end{aligned}$$

Example of `foldLeft` adding `Rational` numbers

$$(\frac{105000}{765049} + \frac{1}{41}) + \frac{1}{43} + \frac{1}{47} + \frac{1}{53} + \frac{1}{57} + \frac{1}{67}$$

$$\begin{aligned}
\frac{1}{23} + \frac{1}{29} &= \frac{52}{667} \\
\frac{52}{667} + \frac{1}{31} &= \frac{2279}{20677} \\
\frac{2279}{20677} + \frac{1}{37} &= \frac{105000}{765049} \\
\frac{105000}{765049} + \frac{1}{41} &= \frac{5070049}{31367009} \\
\frac{5070049}{31367009} + \frac{1}{43} &= \frac{249379116}{1348781387} \\
\frac{249379116}{1348781387} + \frac{1}{47} &= \frac{13069599839}{63392725189} \\
\frac{13069599839}{63392725189} + \frac{1}{53} &= \frac{756081516656}{3359814435017} \\
\frac{756081516656}{3359814435017} + \frac{1}{57} &= \frac{46456460884409}{191509422795969} \\
\frac{46456460884409}{191509422795969} + \frac{1}{67} &= \frac{3304092302051372}{12831131327329923}
\end{aligned}$$

Example of `foldLeft` adding `Rational` numbers

$$\left(\frac{5070049}{31367009} + \frac{1}{43}\right) + \frac{1}{47} + \frac{1}{53} + \frac{1}{57} + \frac{1}{67}$$

$$\begin{aligned}
\frac{1}{23} + \frac{1}{29} &= \frac{52}{667} \\
\frac{52}{667} + \frac{1}{31} &= \frac{2279}{20677} \\
\frac{2279}{20677} + \frac{1}{37} &= \frac{105000}{765049} \\
\frac{105000}{765049} + \frac{1}{41} &= \frac{5070049}{31367009} \\
\frac{5070049}{31367009} + \frac{1}{43} &= \frac{249379116}{1348781387} \\
\frac{249379116}{1348781387} + \frac{1}{47} &= \frac{13069599839}{63392725189} \\
\frac{13069599839}{63392725189} + \frac{1}{53} &= \frac{756081516656}{3359814435017} \\
\frac{756081516656}{3359814435017} + \frac{1}{57} &= \frac{46456460884409}{191509422795969} \\
\frac{46456460884409}{191509422795969} + \frac{1}{67} &= \frac{3304092302051372}{12831131327329923}
\end{aligned}$$

Example of `foldLeft` adding `Rational` numbers

$$(\frac{249379116}{1348781387} + \frac{1}{47}) + \frac{1}{53} + \frac{1}{57} + \frac{1}{67}$$

$$\begin{aligned}
\frac{1}{23} + \frac{1}{29} &= \frac{52}{667} \\
\frac{52}{667} + \frac{1}{31} &= \frac{2279}{20677} \\
\frac{2279}{20677} + \frac{1}{37} &= \frac{105000}{765049} \\
\frac{105000}{765049} + \frac{1}{41} &= \frac{5070049}{31367009} \\
\frac{5070049}{31367009} + \frac{1}{43} &= \frac{249379116}{1348781387} \\
\frac{249379116}{1348781387} + \frac{1}{47} &= \frac{13069599839}{63392725189} \\
\frac{13069599839}{63392725189} + \frac{1}{53} &= \frac{756081516656}{3359814435017} \\
\frac{756081516656}{3359814435017} + \frac{1}{57} &= \frac{46456460884409}{191509422795969} \\
\frac{46456460884409}{191509422795969} + \frac{1}{67} &= \frac{3304092302051372}{1283113131327329923}
\end{aligned}$$

Example of `foldLeft` adding `Rational` numbers

$$(\frac{13069599839}{63392725189} + \frac{1}{53}) + \frac{1}{57} + \frac{1}{67}$$

$$\begin{aligned}
\frac{1}{23} + \frac{1}{29} &= \frac{52}{667} \\
\frac{52}{667} + \frac{1}{31} &= \frac{2279}{20677} \\
\frac{2279}{20677} + \frac{1}{37} &= \frac{105000}{765049} \\
\frac{105000}{765049} + \frac{1}{41} &= \frac{5070049}{31367009} \\
\frac{5070049}{31367009} + \frac{1}{43} &= \frac{249379116}{1348781387} \\
\frac{249379116}{1348781387} + \frac{1}{47} &= \frac{13069599839}{63392725189} \\
\frac{13069599839}{63392725189} + \frac{1}{53} &= \frac{756081516656}{3359814435017} \\
\frac{756081516656}{3359814435017} + \frac{1}{57} &= \frac{46456460884409}{191509422795969} \\
\frac{46456460884409}{191509422795969} + \frac{1}{67} &= \frac{3304092302051372}{1283113132732923}
\end{aligned}$$

Example of `foldLeft` adding `Rational` numbers

$$(\frac{756081516656}{3359814435017} + \frac{1}{57}) + \frac{1}{67}$$

$$\begin{aligned}
\frac{1}{23} + \frac{1}{29} &= \frac{52}{667} \\
\frac{52}{667} + \frac{1}{31} &= \frac{2279}{20677} \\
\frac{2279}{20677} + \frac{1}{37} &= \frac{105000}{765049} \\
\frac{105000}{765049} + \frac{1}{41} &= \frac{5070049}{31367009} \\
\frac{5070049}{31367009} + \frac{1}{43} &= \frac{249379116}{1348781387} \\
\frac{249379116}{1348781387} + \frac{1}{47} &= \frac{13069599839}{63392725189} \\
\frac{13069599839}{63392725189} + \frac{1}{53} &= \frac{756081516656}{3359814435017} \\
\frac{756081516656}{3359814435017} + \frac{1}{57} &= \frac{46456460884409}{191509422795969} \\
\frac{46456460884409}{191509422795969} + \frac{1}{67} &= \frac{3304092302051372}{1283113131327329923}
\end{aligned}$$

Example of `foldLeft` adding `Rational` numbers

$$\left(\frac{46456460884409}{191509422795969} + \frac{1}{56}\right)$$

$$\begin{aligned}
\frac{1}{23} + \frac{1}{29} &= \frac{52}{667} \\
\frac{52}{667} + \frac{1}{31} &= \frac{2279}{20677} \\
\frac{2279}{20677} + \frac{1}{37} &= \frac{105000}{765049} \\
\frac{105000}{765049} + \frac{1}{41} &= \frac{5070049}{31367009} \\
\frac{5070049}{31367009} + \frac{1}{43} &= \frac{249379116}{1348781387} \\
\frac{249379116}{1348781387} + \frac{1}{47} &= \frac{13069599839}{63392725189} \\
\frac{13069599839}{63392725189} + \frac{1}{53} &= \frac{756081516656}{3359814435017} \\
\frac{756081516656}{3359814435017} + \frac{1}{57} &= \frac{46456460884409}{191509422795969} \\
\frac{46456460884409}{191509422795969} + \frac{1}{67} &= \frac{330409230205137 2}{12831131327329923}
\end{aligned}$$

Example of `foldLeft` adding `Rational` numbers

$$\frac{330409230205137 2}{1283113132732992 3}$$

$$\begin{aligned}
\frac{1}{23} + \frac{1}{29} &= \frac{52}{667} \\
\frac{52}{667} + \frac{1}{31} &= \frac{2279}{20677} \\
\frac{2279}{20677} + \frac{1}{37} &= \frac{105000}{765049} \\
\frac{105000}{765049} + \frac{1}{41} &= \frac{5070049}{31367009} \\
\frac{5070049}{31367009} + \frac{1}{43} &= \frac{249379116}{1348781387} \\
\frac{249379116}{1348781387} + \frac{1}{47} &= \frac{13069599839}{63392725189} \\
\frac{13069599839}{63392725189} + \frac{1}{53} &= \frac{756081516656}{3359814435017} \\
\frac{756081516656}{3359814435017} + \frac{1}{57} &= \frac{46456460884409}{191509422795969} \\
\frac{46456460884409}{191509422795969} + \frac{1}{67} &= \frac{330409230205137 2}{1283113132732992 3}
\end{aligned}$$

Example of `reduceTree` adding `Rational` numbers

$$\left(\left(\left(\frac{1}{23}+\frac{1}{29}\right)+\left(\frac{1}{31}+\frac{1}{37}\right)\right)+\left(\left(\frac{1}{41}+\frac{1}{43}\right)+\left(\frac{1}{47}+\frac{1}{53}\right)\right)\right)+\left(\frac{1}{57}+\frac{1}{67}\right)$$

$$\frac{1}{23}+\frac{1}{29}=\frac{52}{667}$$
$$\frac{1}{31}+\frac{1}{37}=\frac{68}{1147}$$
$$\frac{52}{667}+\frac{68}{1147}=\frac{105000}{765049}$$
$$\frac{1}{41}+\frac{1}{43}=\frac{84}{1763}$$
$$\frac{1}{47}+\frac{1}{53}=\frac{100}{2491}$$
$$\frac{84}{1763}+\frac{100}{2491}=\frac{385544}{4391633}$$
$$\frac{105000}{765049}+\frac{385544}{4391633}=\frac{756081516656}{3359814435017}$$
$$\frac{1}{57}+\frac{1}{67}=\frac{124}{3819}$$
$$\frac{756081516656}{3359814435017}+\frac{124}{3819}=\frac{3304092302051372}{12831131327329923}$$

Example of `reduceTree` adding `Rational` numbers

$$\left(\left(\left(\frac{1}{23}+\frac{1}{29}\right)+\left(\frac{1}{31}+\frac{1}{37}\right)\right)+\left(\left(\frac{1}{41}+\frac{1}{43}\right)+\left(\frac{1}{47}+\frac{1}{53}\right)\right)\right)+\left(\frac{1}{57}+\frac{1}{67}\right)$$

$$\frac{1}{23}+\frac{1}{29}=\frac{52}{667}$$

$$\frac{1}{31}+\frac{1}{37}=\frac{68}{1147}$$

$$\frac{52}{667}+\frac{68}{1147}=\frac{105000}{765049}$$

$$\frac{1}{41}+\frac{1}{43}=\frac{84}{1763}$$

$$\frac{1}{47}+\frac{1}{53}=\frac{100}{2491}$$

$$\frac{84}{1763}+\frac{100}{2491}=\frac{385544}{4391633}$$

$$\frac{105000}{765049}+\frac{385544}{4391633}=\frac{756081516656}{3359814435017}$$

$$\frac{1}{57}+\frac{1}{67}=\frac{124}{3819}$$

$$\frac{756081516656}{3359814435017}+\frac{124}{3819}=\frac{3304092302051372}{12831131327329923}$$

Example of `reduceTree` adding `Rational` numbers

$$\left(\left(\frac{52}{667} + \left(\frac{1}{31} + \frac{1}{37}\right)\right) + \left(\left(\frac{1}{41} + \frac{1}{43}\right) + \left(\frac{1}{47} + \frac{1}{53}\right)\right)\right) + \left(\frac{1}{57} + \frac{1}{67}\right)$$

$$\frac{1}{23} + \frac{1}{29} = \frac{52}{667}$$

$$\frac{1}{31} + \frac{1}{37} = \frac{68}{1147}$$

$$\frac{52}{667} + \frac{68}{1147} = \frac{105000}{765049}$$

$$\frac{1}{41} + \frac{1}{43} = \frac{84}{1763}$$

$$\frac{1}{47} + \frac{1}{53} = \frac{100}{2491}$$

$$\frac{84}{1763} + \frac{100}{2491} = \frac{385544}{4391633}$$

$$\frac{105000}{765049} + \frac{385544}{4391633} = \frac{756081516656}{3359814435017}$$

$$\frac{1}{57} + \frac{1}{67} = \frac{124}{3819}$$

$$\frac{756081516656}{3359814435017} + \frac{124}{3819} = \frac{3304092302051372}{12831131327329923}$$

Example of `reduceTree` adding `Rational` numbers

$$\left(\left(\frac{52}{667}+\frac{68}{1147}\right)+\left(\left(\frac{1}{41}+\frac{1}{43}\right)+\left(\frac{1}{47}+\frac{1}{53}\right)\right)\right)+\left(\frac{1}{57}+\frac{1}{67}\right)$$

$$\frac{1}{23}+\frac{1}{29}=\frac{52}{667}$$
$$\frac{1}{31}+\frac{1}{37}=\frac{68}{1147}$$
$$\frac{52}{667}+\frac{68}{1147}=\frac{105000}{765049}$$
$$\frac{1}{41}+\frac{1}{43}=\frac{84}{1763}$$
$$\frac{1}{47}+\frac{1}{53}=\frac{100}{2491}$$
$$\frac{84}{1763}+\frac{100}{2491}=\frac{385544}{4391633}$$
$$\frac{105000}{765049}+\frac{385544}{4391633}=\frac{756081516656}{3359814435017}$$
$$\frac{1}{57}+\frac{1}{67}=\frac{124}{3819}$$
$$\frac{756081516656}{3359814435017}+\frac{124}{3819}=\frac{3304092302051372}{12831131327329923}$$

Example of `reduceTree` adding `Rational` numbers

$$\left(\frac{105000}{765049} + \left(\left(\frac{1}{41} + \frac{1}{43}\right) + \left(\frac{1}{47} + \frac{1}{53}\right)\right)\right) + \left(\frac{1}{57} + \frac{1}{67}\right)$$

$$\frac{1}{23} + \frac{1}{29} = \frac{52}{667}$$
$$\frac{1}{31} + \frac{1}{37} = \frac{68}{1147}$$
$$\frac{52}{667} + \frac{68}{1147} = \frac{105000}{765049}$$
$$\frac{1}{41} + \frac{1}{43} = \frac{84}{1763}$$
$$\frac{1}{47} + \frac{1}{53} = \frac{100}{2491}$$
$$\frac{84}{1763} + \frac{100}{2491} = \frac{385544}{4391633}$$
$$\frac{105000}{765049} + \frac{385544}{4391633} = \frac{756081516656}{3359814435017}$$
$$\frac{1}{57} + \frac{1}{67} = \frac{124}{3819}$$
$$\frac{756081516656}{3359814435017} + \frac{124}{3819} = \frac{3304092302051372}{12831131327329923}$$

Example of `reduceTree` adding `Rational` numbers

$$\left(\frac{105000}{765049} + \left(\frac{84}{1763} + \left(\frac{1}{47} + \frac{1}{53}\right)\right)\right) + \left(\frac{1}{57} + \frac{1}{67}\right)$$

$$\frac{1}{23} + \frac{1}{29} = \frac{52}{667}$$

$$\frac{1}{31} + \frac{1}{37} = \frac{68}{1147}$$

$$\frac{52}{667} + \frac{68}{1147} = \frac{105000}{765049}$$

$$\frac{1}{41} + \frac{1}{43} = \frac{84}{1763}$$

$$\frac{1}{47} + \frac{1}{53} = \frac{100}{2491}$$

$$\frac{84}{1763} + \frac{100}{2491} = \frac{385544}{4391633}$$

$$\frac{105000}{765049} + \frac{385544}{4391633} = \frac{756081516656}{3359814435017}$$

$$\frac{1}{57} + \frac{1}{67} = \frac{124}{3819}$$

$$\frac{756081516656}{3359814435017} + \frac{124}{3819} = \frac{3304092302051372}{1283113132732923}$$

Example of `reduceTree` adding `Rational` numbers

$$\left(\frac{105000}{765049} + \left(\frac{84}{1763} + \frac{100}{2491}\right)\right) + \left(\frac{1}{57} + \frac{1}{67}\right)$$

$$\frac{1}{23} + \frac{1}{29} = \frac{52}{667}$$

$$\frac{1}{31} + \frac{1}{37} = \frac{68}{1147}$$

$$\frac{52}{667} + \frac{68}{1147} = \frac{105000}{765049}$$

$$\frac{1}{41} + \frac{1}{43} = \frac{84}{1763}$$

$$\frac{1}{47} + \frac{1}{53} = \frac{100}{2491}$$

$$\frac{84}{1763} + \frac{100}{2491} = \frac{385544}{4391633}$$

$$\frac{105000}{765049} + \frac{385544}{4391633} = \frac{756081516656}{3359814435017}$$

$$\frac{1}{57} + \frac{1}{67} = \frac{124}{3819}$$

$$\frac{756081516656}{3359814435017} + \frac{124}{3819} = \frac{3304092302051372}{12831131327329923}$$

Example of `reduceTree` adding `Rational` numbers

$$\left(\frac{105000}{765049} + \frac{385544}{4391633}\right) + \left(\frac{1}{57} + \frac{1}{67}\right)$$

$$\frac{1}{23} + \frac{1}{29} = \frac{52}{667}$$

$$\frac{1}{31} + \frac{1}{37} = \frac{68}{1147}$$

$$\frac{52}{667} + \frac{68}{1147} = \frac{105000}{765049}$$

$$\frac{1}{41} + \frac{1}{43} = \frac{84}{1763}$$

$$\frac{1}{47} + \frac{1}{53} = \frac{100}{2491}$$

$$\frac{84}{1763} + \frac{100}{2491} = \frac{385544}{4391633}$$

$$\frac{105000}{765049} + \frac{385544}{4391633} = \frac{756081516656}{3359814435017}$$

$$\frac{1}{57} + \frac{1}{67} = \frac{124}{3819}$$

$$\frac{756081516656}{3359814435017} + \frac{124}{3819} = \frac{3304092302051372}{12831131327329923}$$

Example of `reduceTree` adding `Rational` numbers

$$\frac{756081516656}{3359814435017} + \left(\frac{1}{57} + \frac{1}{67}\right)$$

$$\frac{1}{23} + \frac{1}{29} = \frac{52}{667}$$

$$\frac{1}{31} + \frac{1}{37} = \frac{68}{1147}$$

$$\frac{52}{667} + \frac{68}{1147} = \frac{105000}{765049}$$

$$\frac{1}{41} + \frac{1}{43} = \frac{84}{1763}$$

$$\frac{1}{47} + \frac{1}{53} = \frac{100}{2491}$$

$$\frac{84}{1763} + \frac{100}{2491} = \frac{385544}{4391633}$$

$$\frac{105000}{765049} + \frac{385544}{4391633} = \frac{756081516656}{3359814435017}$$

$$\frac{1}{57} + \frac{1}{67} = \frac{124}{3819}$$

$$\frac{756081516656}{3359814435017} + \frac{124}{3819} = \frac{3304092302051372}{12831131327329923}$$

Example of `reduceTree` adding `Rational` numbers

$$\frac{756081516656}{3359814435017} + \frac{124}{3819}$$

$$\frac{1}{23} + \frac{1}{29} = \frac{52}{667}$$
$$\frac{1}{31} + \frac{1}{37} = \frac{68}{1147}$$
$$\frac{52}{667} + \frac{68}{1147} = \frac{105000}{765049}$$
$$\frac{1}{41} + \frac{1}{43} = \frac{84}{1763}$$
$$\frac{1}{47} + \frac{1}{53} = \frac{100}{2491}$$
$$\frac{84}{1763} + \frac{100}{2491} = \frac{385544}{4391633}$$
$$\frac{105000}{765049} + \frac{385544}{4391633} = \frac{756081516656}{3359814435017}$$
$$\frac{1}{57} + \frac{1}{67} = \frac{124}{3819}$$
$$\frac{756081516656}{3359814435017} + \frac{124}{3819} = \frac{3304092302051372}{1283113131327329923}$$

Example of `reduceTree` adding `Rational` numbers

$$\frac{3304092302051372}{1283113131327329923}$$

$$\frac{1}{23} + \frac{1}{29} = \frac{52}{667}$$

$$\frac{1}{31} + \frac{1}{37} = \frac{68}{1147}$$

$$\frac{52}{667} + \frac{68}{1147} = \frac{105000}{765049}$$

$$\frac{1}{41} + \frac{1}{43} = \frac{84}{1763}$$

$$\frac{1}{47} + \frac{1}{53} = \frac{100}{2491}$$

$$\frac{84}{1763} + \frac{100}{2491} = \frac{385544}{4391633}$$

$$\frac{105000}{765049} + \frac{385544}{4391633} = \frac{756081516656}{3359814435017}$$

$$\frac{1}{57} + \frac{1}{67} = \frac{124}{3819}$$

$$\frac{756081516656}{3359814435017} + \frac{124}{3819} = \frac{3304092302051372}{1283113131327329923}$$

Example of `pair-wise` adding `Rational` numbers

$$(\frac{1}{23}+\frac{1}{29})+(\frac{1}{31}+\frac{1}{37})+(\frac{1}{41}+\frac{1}{43})+(\frac{1}{47}+\frac{1}{53})+(\frac{1}{57}+\frac{1}{67})$$

$$\begin{aligned}
\frac{1}{23}+\frac{1}{29} &= \frac{52}{667} \\
\frac{1}{31}+\frac{1}{37} &= \frac{68}{1147} \\
\frac{1}{41}+\frac{1}{43} &= \frac{84}{1763} \\
\frac{1}{47}+\frac{1}{53} &= \frac{100}{2491} \\
\frac{1}{57}+\frac{1}{67} &= \frac{124}{3819}
\end{aligned}$$

Example of pair-wise adding Rational numbers

$$(\frac{52}{667} + \frac{68}{1147}) + (\frac{84}{1763} + \frac{100}{2491}) + \frac{124}{3819}$$

$$\frac{1}{23} + \frac{1}{29} = \frac{52}{667}$$
$$\frac{1}{31} + \frac{1}{37} = \frac{68}{1147}$$
$$\frac{1}{41} + \frac{1}{43} = \frac{84}{1763}$$
$$\frac{1}{47} + \frac{1}{53} = \frac{100}{2491}$$
$$\frac{1}{57} + \frac{1}{67} = \frac{124}{3819}$$
$$\frac{52}{667} + \frac{68}{1147} = \frac{105000}{765049}$$
$$\frac{84}{1763} + \frac{100}{2491} = \frac{385544}{4391633}$$

Example of pair-wise adding Rational numbers

$$(\frac{105000}{765049} + \frac{385544}{4391633}) + \frac{124}{3819}$$

$$\frac{1}{23} + \frac{1}{29} = \frac{52}{667}$$
$$\frac{1}{31} + \frac{1}{37} = \frac{68}{1147}$$
$$\frac{1}{41} + \frac{1}{43} = \frac{84}{1763}$$
$$\frac{1}{47} + \frac{1}{53} = \frac{100}{2491}$$
$$\frac{1}{57} + \frac{1}{67} = \frac{124}{3819}$$
$$\frac{52}{667} + \frac{68}{1147} = \frac{105000}{765049}$$
$$\frac{84}{1763} + \frac{100}{2491} = \frac{385544}{4391633}$$
$$\frac{105000}{765049} + \frac{385544}{4391633} = \frac{756081516656}{3359814435017}$$

Example of `pair-wise` adding `Rational` numbers

$$\frac{756081516656}{3359814435017} + \frac{124}{3819}$$

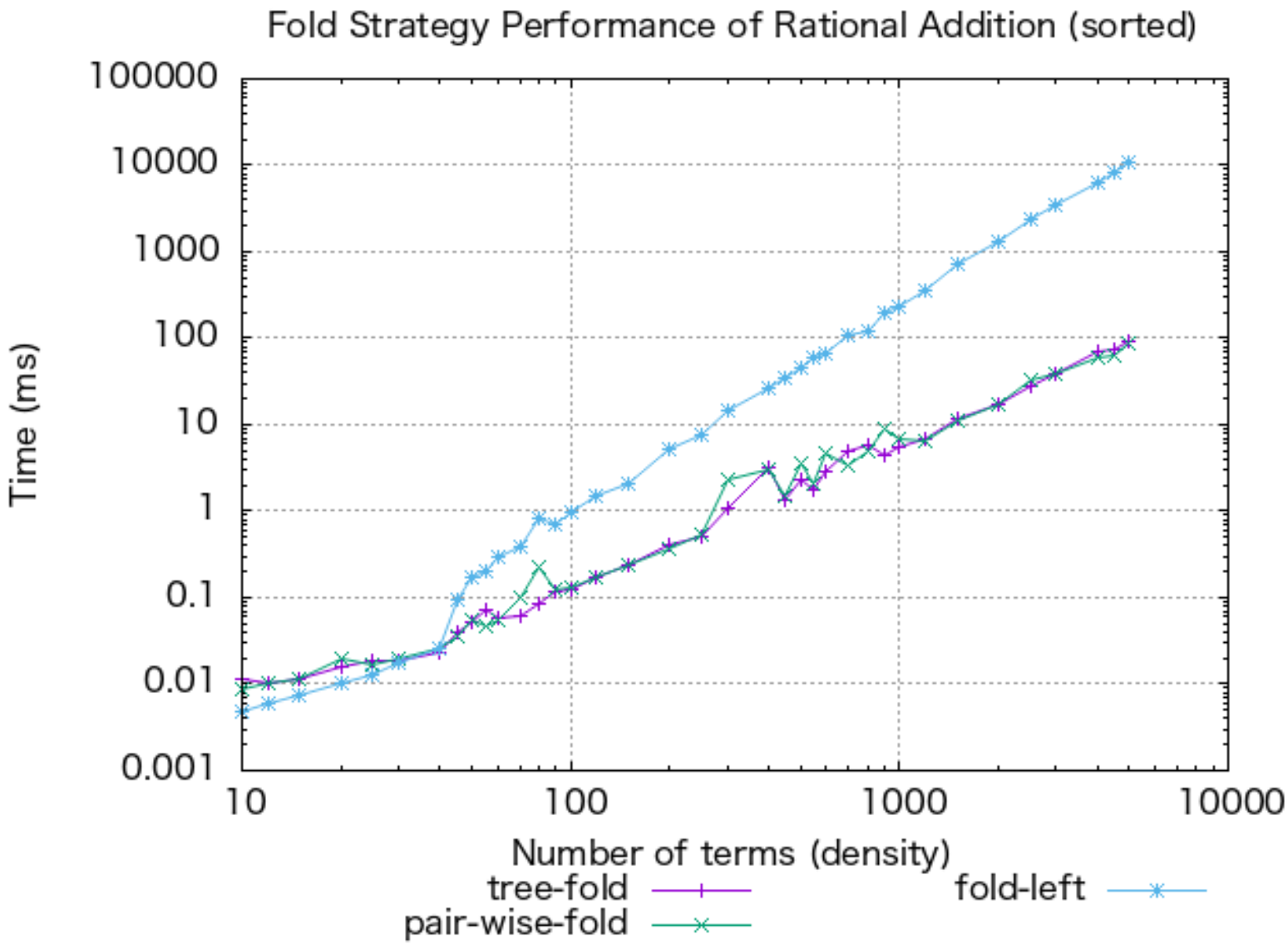
$$\frac{1}{23} + \frac{1}{29} = \frac{52}{667}$$
$$\frac{1}{31} + \frac{1}{37} = \frac{68}{1147}$$
$$\frac{1}{41} + \frac{1}{43} = \frac{84}{1763}$$
$$\frac{1}{47} + \frac{1}{53} = \frac{100}{2491}$$
$$\frac{1}{57} + \frac{1}{67} = \frac{124}{3819}$$
$$\frac{52}{667} + \frac{68}{1147} = \frac{105000}{765049}$$
$$\frac{84}{1763} + \frac{100}{2491} = \frac{385544}{4391633}$$
$$\frac{105000}{765049} + \frac{385544}{4391633} = \frac{756081516656}{3359814435017}$$
$$\frac{756081516656}{3359814435017} + \frac{124}{3819} = \frac{3304092302051372}{12831131327329923}$$

Example of `pair-wise` adding `Rational` numbers

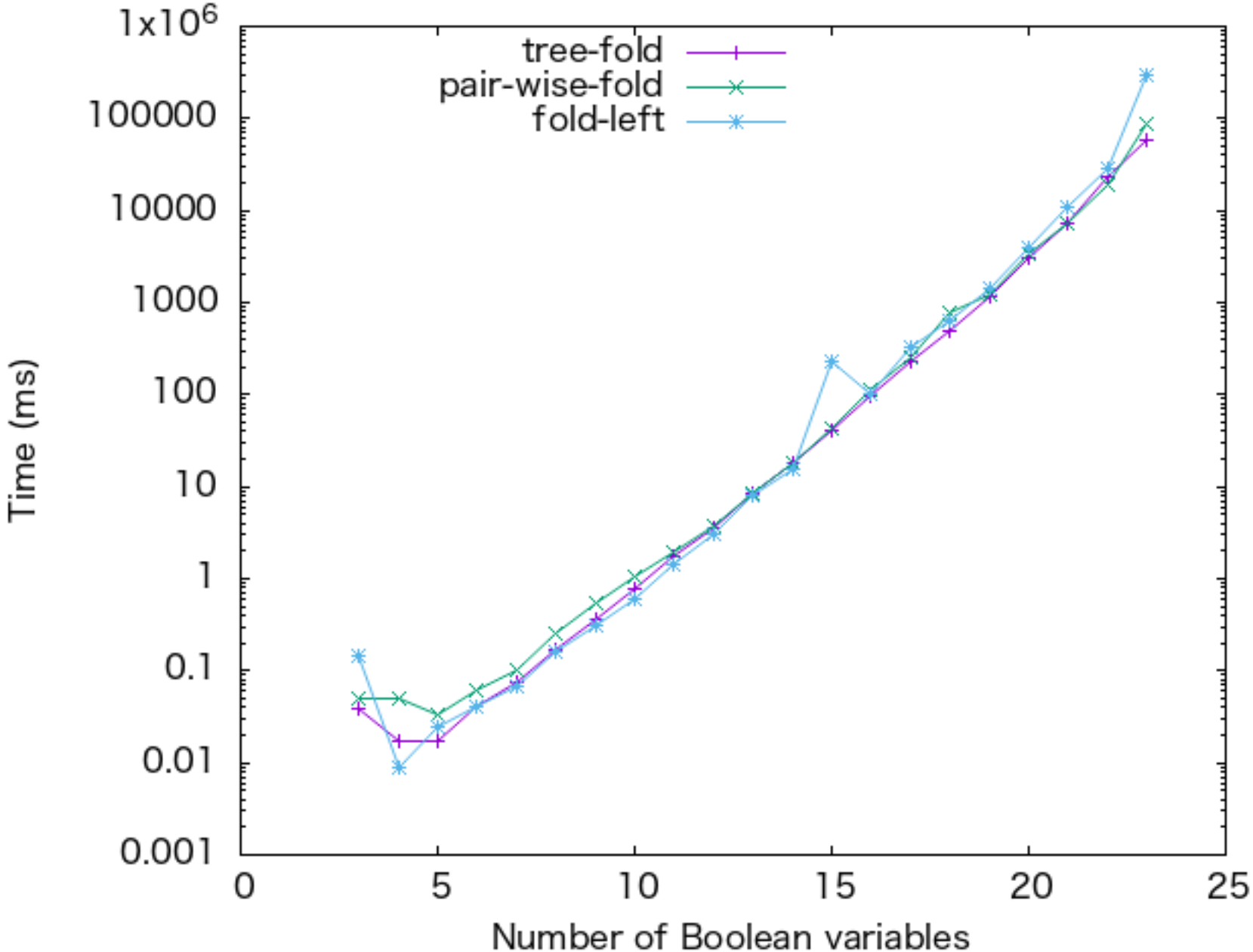
$$\frac{3304092302051372}{1283113131327329923}$$

$$\frac{1}{23} + \frac{1}{29} = \frac{52}{667}$$
$$\frac{1}{31} + \frac{1}{37} = \frac{68}{1147}$$
$$\frac{1}{41} + \frac{1}{43} = \frac{84}{1763}$$
$$\frac{1}{47} + \frac{1}{53} = \frac{100}{2491}$$
$$\frac{1}{57} + \frac{1}{67} = \frac{124}{3819}$$
$$\frac{52}{667} + \frac{68}{1147} = \frac{105000}{765049}$$
$$\frac{84}{1763} + \frac{100}{2491} = \frac{385544}{4391633}$$
$$\frac{105000}{765049} + \frac{385544}{4391633} = \frac{756081516656}{3359814435017}$$
$$\frac{756081516656}{3359814435017} + \frac{124}{3819} = \frac{3304092302051372}{1283113131327329923}$$

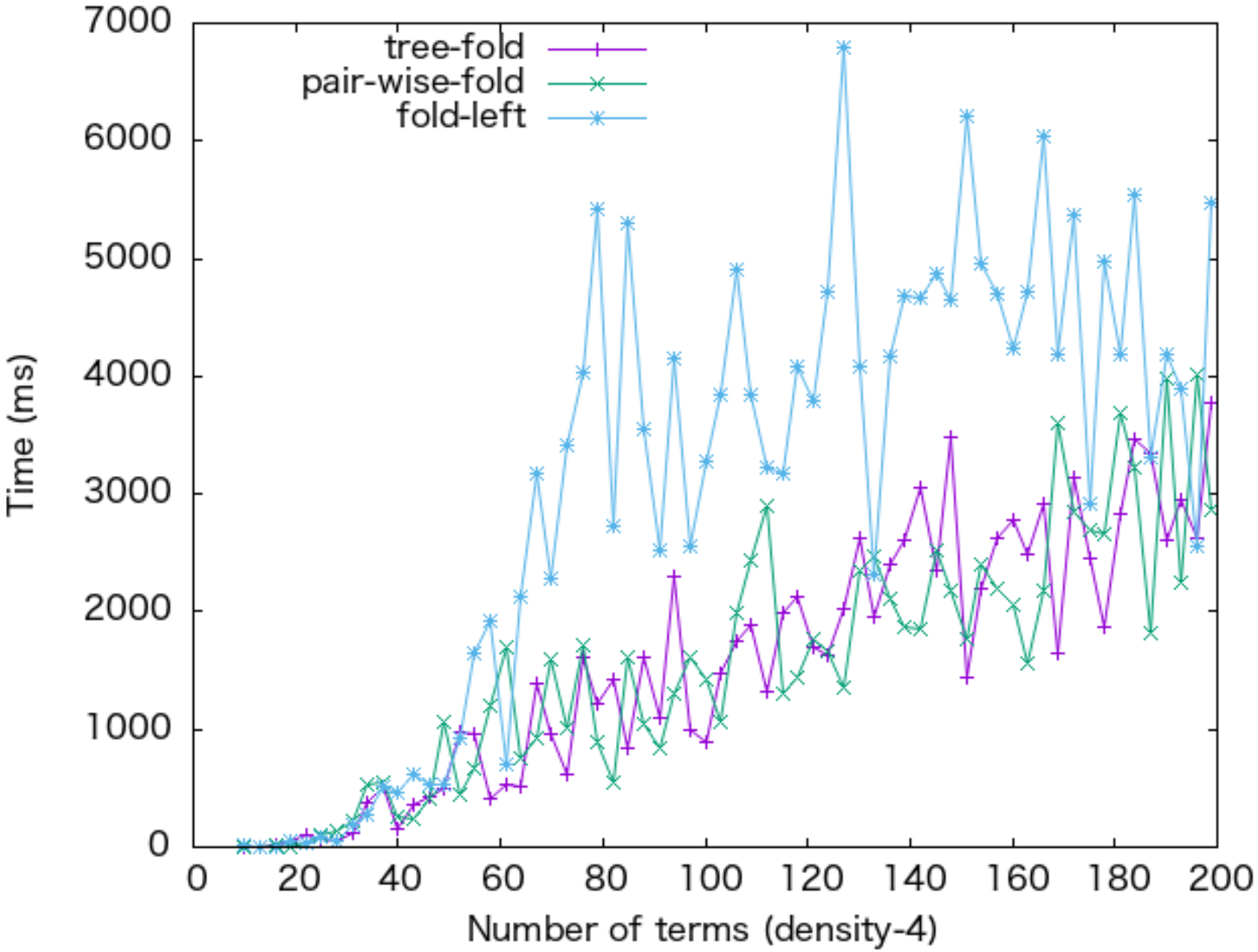
Fold Strategy Performance of Rational Addition



Fold Strategy Performance for Bdd Construction



Fold Strategy Performance of $n = 30$ $bits = 4$



Ratio tree-fold / default-fold for $n = 30$