UNIVERSITY OF WARSAW

Comparative Analysis of Parallel vs. Sequential SAT Solvers

Pierfrancesco Lijoi-K17413

UNIVERSITY OF WARSAW

Agenda

UNIVERSITY OF WARSAW

⚠ Motivation & Problem SAT

Given a **Boolean formula**:

$$(A \lor B) \land (\neg A \lor C)$$

our goal is to choose **True/False values** for each variable so that the **entire expression** evaluates to **True**:

- If **at least one** such assignment **exists**, the formula is “**satisfiable**”.
- If **no assignment** makes it True, the formula is “**unsatisfiable**”.

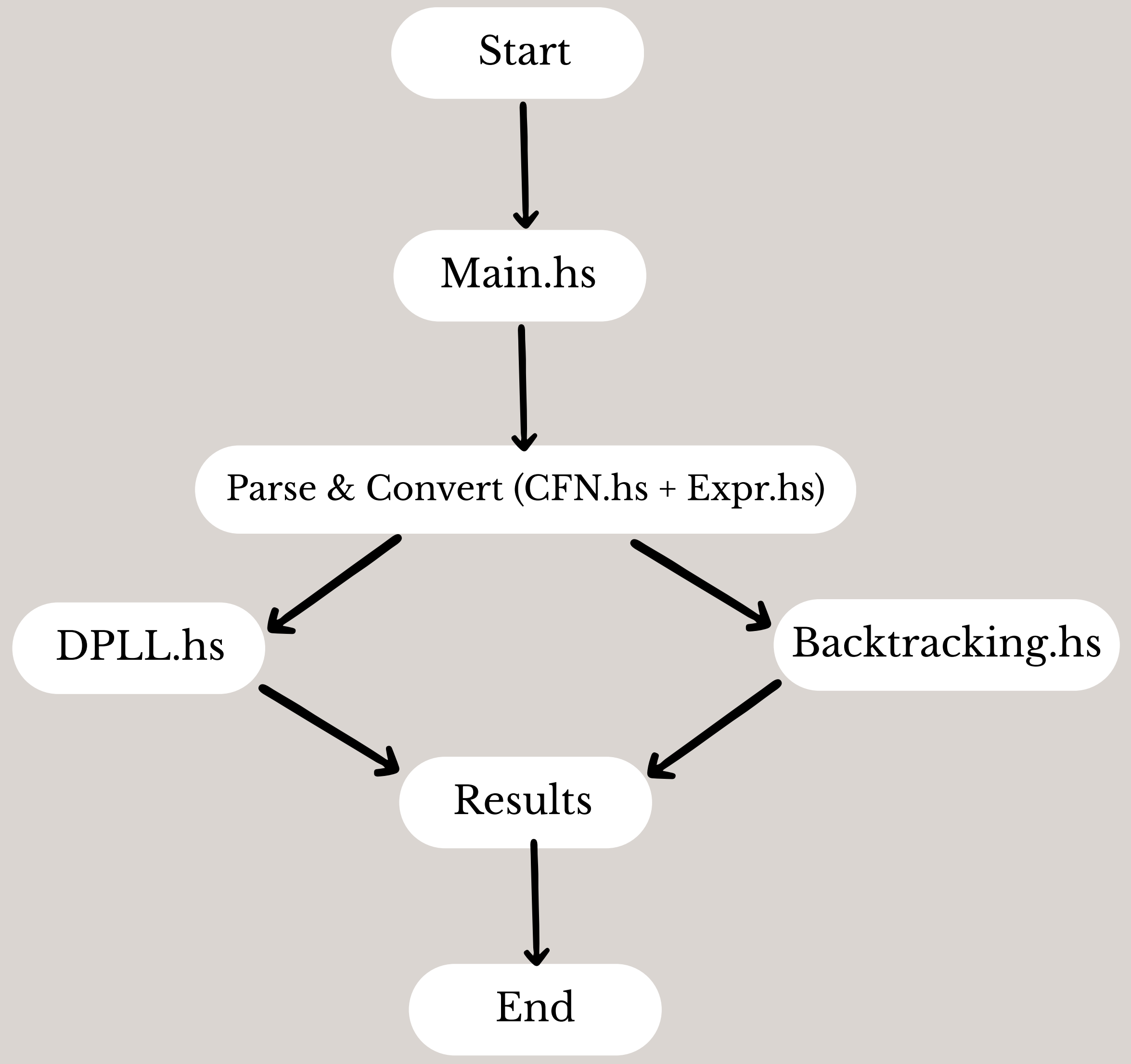
SAT is NP-complete, no known algorithm runs efficiently in every case.

Project Architecture 1

Because **SAT is NP-complete** and no single method solves all instances efficiently, my **project investigates** the difference **between a parallel solver** and a **purely sequential one**.
The code is organized into five modules:

- **Expr.hs**: Defines the Expr type and implements substitution, simplification, and CNF conversion so solvers can manipulate Boolean formulas uniformly.
- **CNF.hs:** Parses DIMACS CNF files into Expr, guaranteeing that both solvers start from the same input format
- **Backtracking.hs:** Implements a pure, sequential backtracking solver as our baseline
- **DPLL.hs:** Builds a parallel solver by adding unit-clause propagation, pure-literal elimination
- **Main.hs:** Coordinates the workflow and take measures time/memory/recursions/decisions

Project Architecture 2



Pipeline Scheme

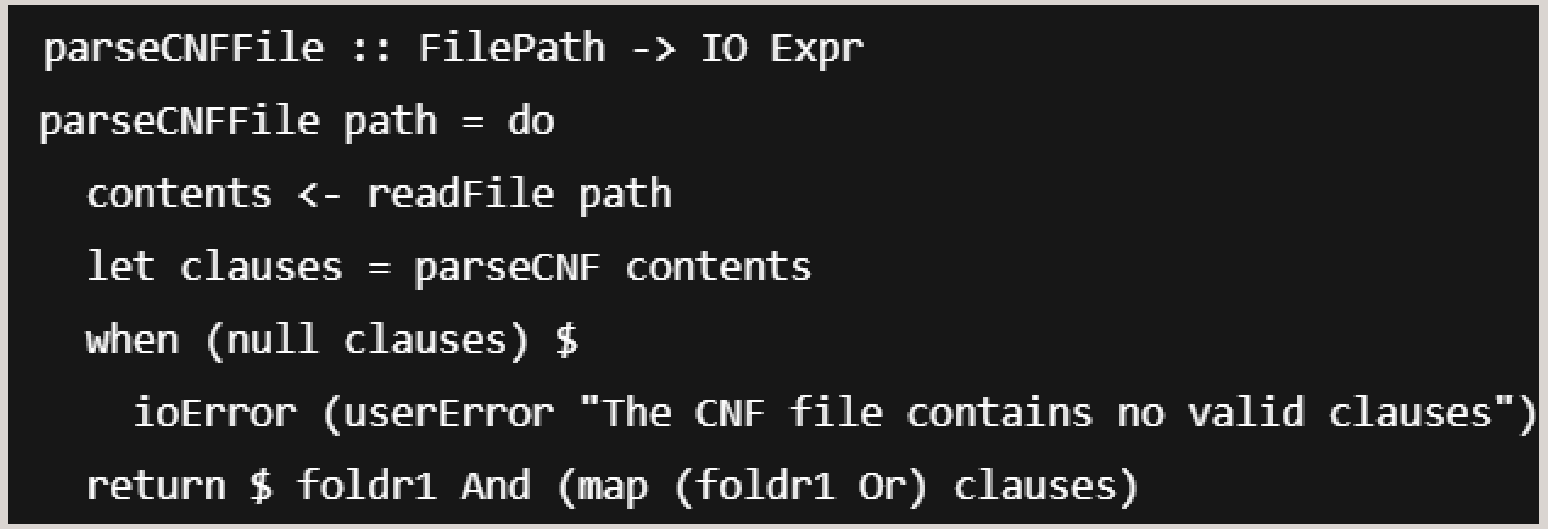
Project Architecture 3

CNF.hs – Parsing and Preparing Input

The first crucial step in our SAT solver pipeline is reading the problem definition. The CNF.hs module handles parsing standard DIMACS CNF files, which are widely used benchmarks in SAT research.

1. It **reads the input file**, **filters out** comments and irrelevant lines, **and extracts a list of clauses**.
2. **Each clause consists of literals**, which are converted **into our internal Expr format**.
3. Then **clauses are combined into a single expression** representing a **Conjunctive Normal Form (CNF)**

This preprocessing **ensures that the solver receives a clean, uniform, and validated formula**, simplifying its task immensely.



```haskell
parseCNFFile :: FilePath -> IO Expr
parseCNFFile path = do
  contents <- readFile path
  let clauses = parseCNF contents
  when (null clauses) $
    ioError (userError "The CNF file contains no valid clauses")
  return $ foldr1 And (map (foldr1 Or) clauses)
```

Project Architecture 4

Expr.hs – Representing and Converting Formulas to CNF

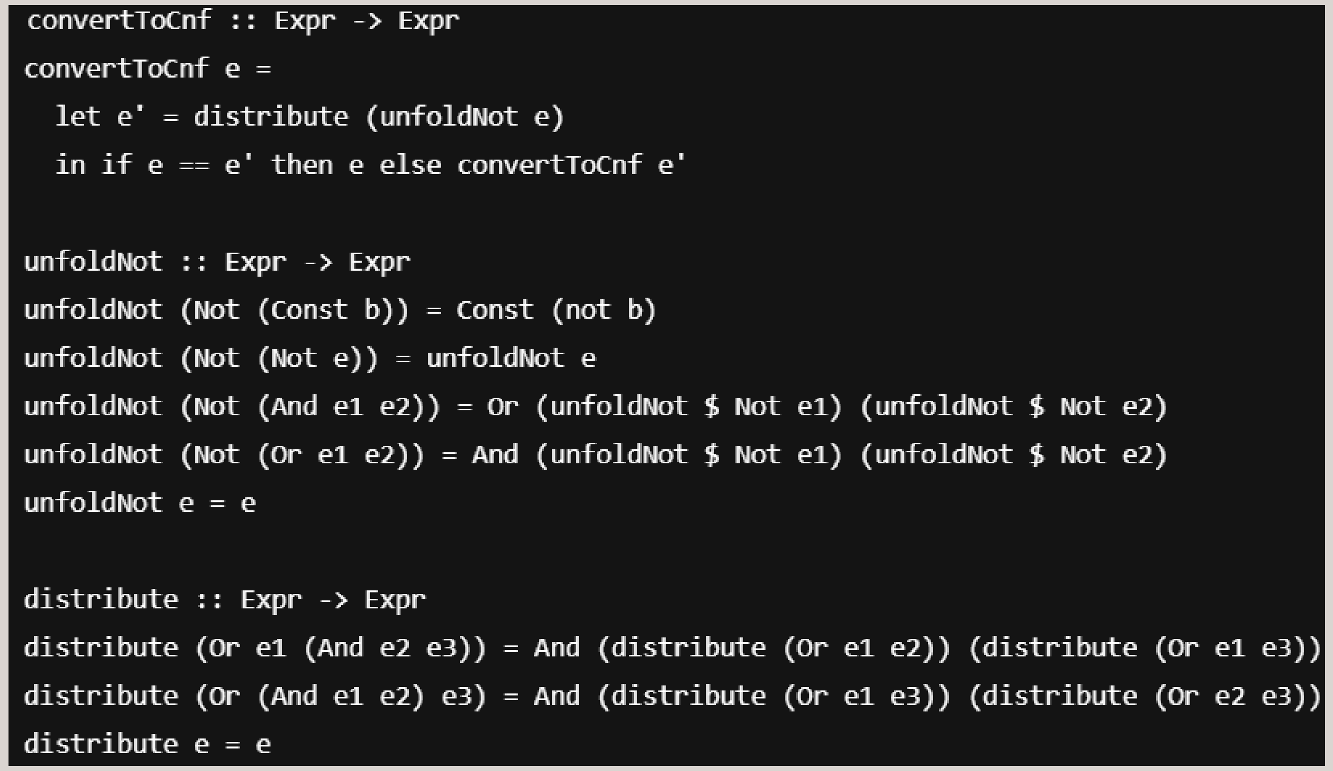
Once the raw clauses are parsed, Expr.hs takes charge of defining and manipulating the logical formulas.

- **Formulas are represented by** the algebraic da**ta type Expr**, which includes variables, Boolean constants, and logical operators (Not, And, Or).
- The **convertToCnf function recursively** rewrites expressions into CNF through **two key steps**:
  - **unfoldNot** applies **De Morgan's laws** and **eliminates double negations**.

$$A \wedge \neg(B \vee C) \quad \rightarrow \quad A \wedge (\neg B \wedge \neg C)$$

  - **distribute** applies **distributive laws** to ensure **OR operators only apply to literals**, never to AND expressions.

$$X \vee (Y \wedge Z) \quad \rightarrow \quad (X \vee Y) \wedge (X \vee Z)$$



```
convertToCnf :: Expr -> Expr
convertToCnf e =
  let e' = distribute (unfoldNot e)
  in if e == e' then e else convertToCnf e'

unfoldNot :: Expr -> Expr
unfoldNot (Not (Const b)) = Const (not b)
unfoldNot (Not (Not e)) = unfoldNot e
unfoldNot (Not (And e1 e2)) = Or (unfoldNot $ Not e1) (unfoldNot $ Not e2)
unfoldNot (Not (Or e1 e2)) = And (unfoldNot $ Not e1) (unfoldNot $ Not e2)
unfoldNot e = e

distribute :: Expr -> Expr
distribute (Or e1 (And e2 e3)) = And (distribute (Or e1 e2)) (distribute (Or e1 e3))
distribute (Or (And e1 e2) e3) = And (distribute (Or e1 e3)) (distribute (Or e2 e3))
distribute e = e
```

The process continues until the formula becomes a proper CNF.
CNF.hs parses input correctly; Expr.hs converts formulas to this form for efficient solving

Backtracking 1

IDEA : Imagine you're solving a puzzle by trial and error:
you make a guess, see if it works, and if not, you undo it and try something else.

That's exactly how our SAT solver operates . It picks a variable, guesses True or False, then solves the smaller formula. If that leads to a contradiction, it backtracks and tries the opposite guess.

1) Initial Simplification
Remove already satisfied clauses and drop false literals (e.g., A ∧ True → A)

2) Variable Selection
- If no unassigned variables remain, evaluate and return (True/False).
- Otherwise, pick an unassigned **variable** v.

3) True Branch
Assign v=True and recurse: go back to step 1 (simplify, select, ...).

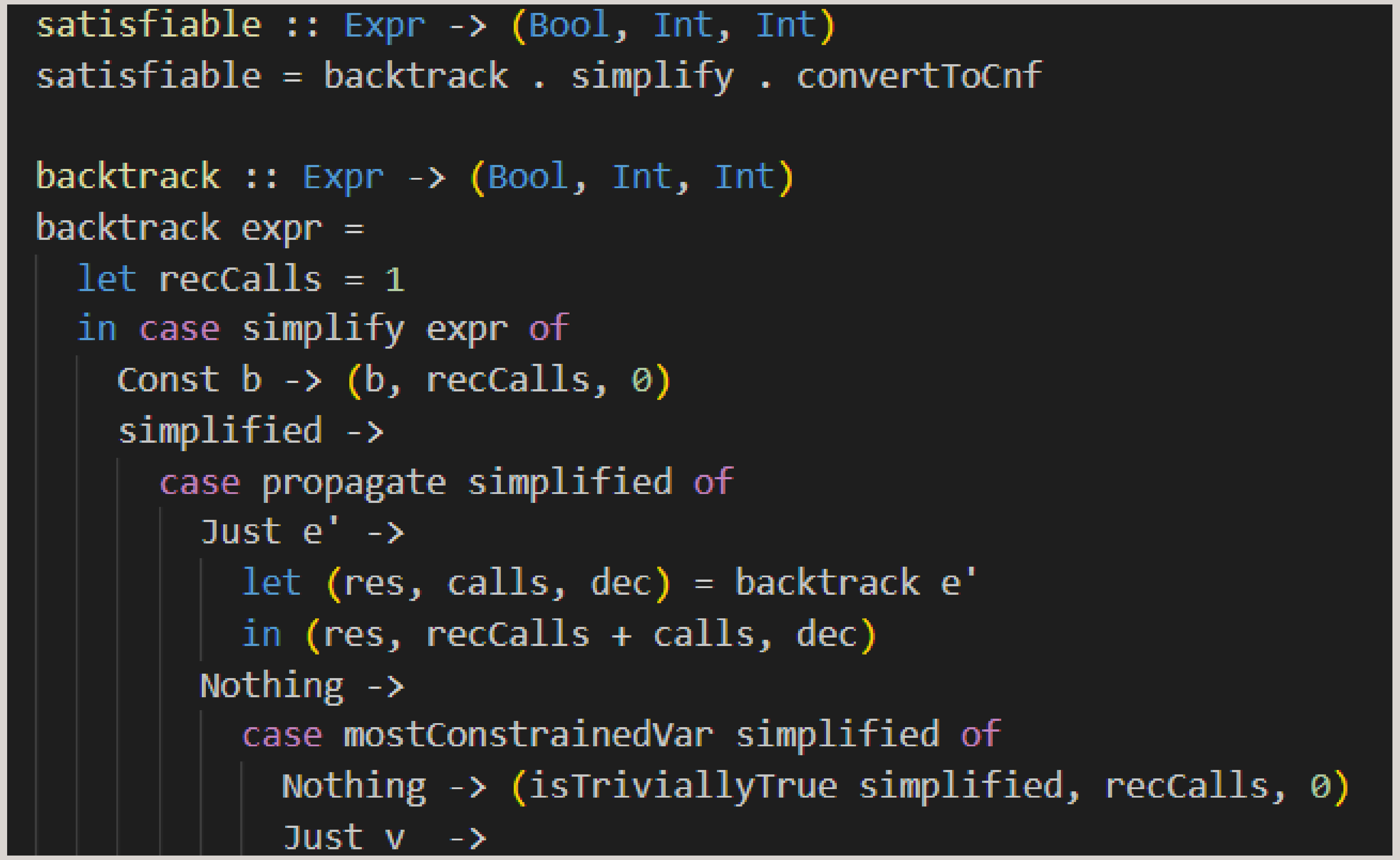
4) Backtrack → False Branch
If the True branch fails, undo its assignments, **set v=False**, and recurse again.

5) Combine Results
- If either branch succeeds, declare the formula satisfiable and stop.
- If both branches fail, declare unsatisfiable and backtrack up one level

Backtracking 2

Backtracking Search (Part 1)

- **recCalls** = 1: Each time backtrack is called, we increment a counter. This allows us to measure the intensity of the search.
- **simplify**: Applies basic simplification rules (e.g., X AND True becomes X). **(Stage 1)**
- **backtrack**: Calls the main recursive function
- **case simplify expr of Const b -> ...:** This is the base case. If the formula reduces to a constant (True or False), we've found the solution (or non-solution) for that branch. **(Stage 2.1)**
- **case propagate simplified of Just e' -> ...:** This is a small heuristic "trick." propagate looks for unit clauses or pure literals. If found, it further simplifies the formula and recurses on the already reduced problem (backtrack e'). **(Stage 2.2)**



```haskell
satisfiable :: Expr -> (Bool, Int, Int)
satisfiable = backtrack . simplify . convertToCnf

backtrack :: Expr -> (Bool, Int, Int)
backtrack expr =
  let recCalls = 1
  in case simplify expr of
    Const b -> (b, recCalls, 0)
    simplified ->
      case propagate simplified of
        Just e' ->
          let (res, calls, dec) = backtrack e'
          in (res, recCalls + calls, dec)
        Nothing ->
          case mostConstrainedVar simplified of
            Nothing -> (isTriviallyTrue simplified, recCalls, 0)
            Just v  ->
```

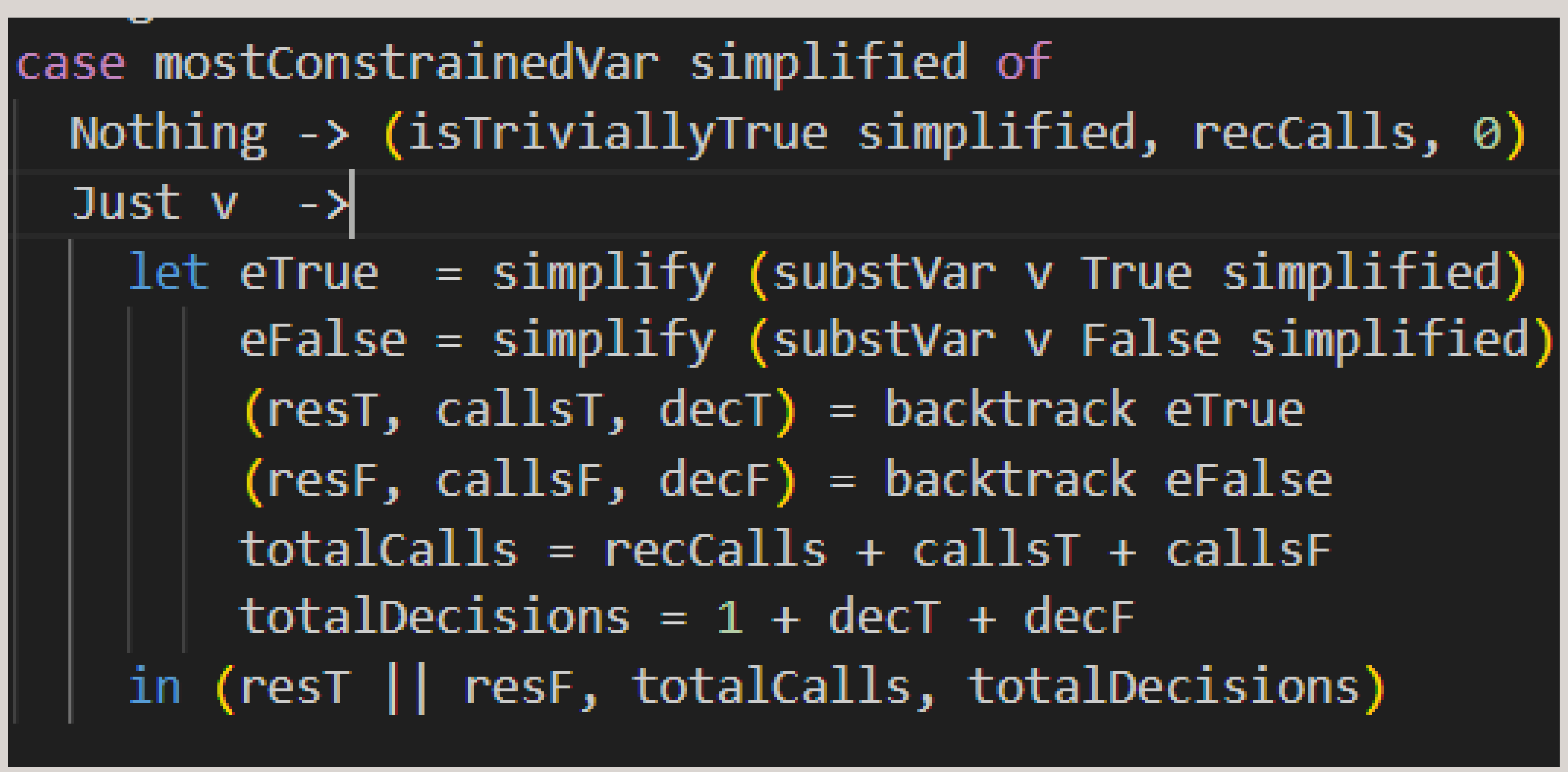
- **Why?** Reducing the problem before branching is always beneficial.

UNIVERSITY OF WARSAW

Backtracking 2.1

Backtracking Search (Part 2)

- **Nothing -> case mostConstrainedVar simplified of Nothing -> ...:** If propagate finds nothing to do, we must make a "decision." mostConstrainedVar selects the most frequently occurring variable in the formula, a simple heuristic for choosing a variable to assign.
- **eTrue = simplify (substVar v True simplified):** Creates a new formula where variable v has been substituted with True, then simplifies it.
- **eFalse = simplify (substVar v False simplified):** Does the same, but with False.
- **(resT, callsT, decT) = backtrack eTrue:** Recursively calls backtrack for the "True" branch. **(Stage 3)**
- **(resF, callsF, decF) = backtrack eFalse:** Recursively calls backtrack for the "False" branch. **(Stage 4)**
- **totalCalls = recCalls + callsT + callsF:** Aggregates recursive call counters from all branches.
- **totalDecisions = 1 + decT + decF:** Aggregates decision step counters (current step + those from child branches).
- **resT || resF:** The original problem is satisfiable if at least one of the two branches is satisfiable. **(Stage 5)**


```
case mostConstrainedVar simplified of
  Nothing -> (isTriviallyTrue simplified, recCalls, 0)
  Just v  ->
    let eTrue  = simplify (substVar v True simplified)
        eFalse = simplify (substVar v False simplified)
        (resT, callsT, decT) = backtrack eTrue
        (resF, callsF, decF) = backtrack eFalse
        totalCalls = recCalls + callsT + callsF
        totalDecisions = 1 + decT + decF
    in (resT || resF, totalCalls, totalDecisions)
```

Why it Works This Way: This part implements the core of backtracking: if it can't be solved directly, the problem is divided into sub-problems based on the possible assignments of a variable, and the results are combined. The additional metrics allow us to quantify the "cost" of this exploration

UNIVERSITY OF WARSAW

DPLL 1

IDEA : Imagine you're solving a puzzle. Instead of trying random pieces, DPLL first tidies up the obvious parts, like finding all the corners and edge pieces.
Only then does the "real" game begin.

It can do it throught two heuristics:

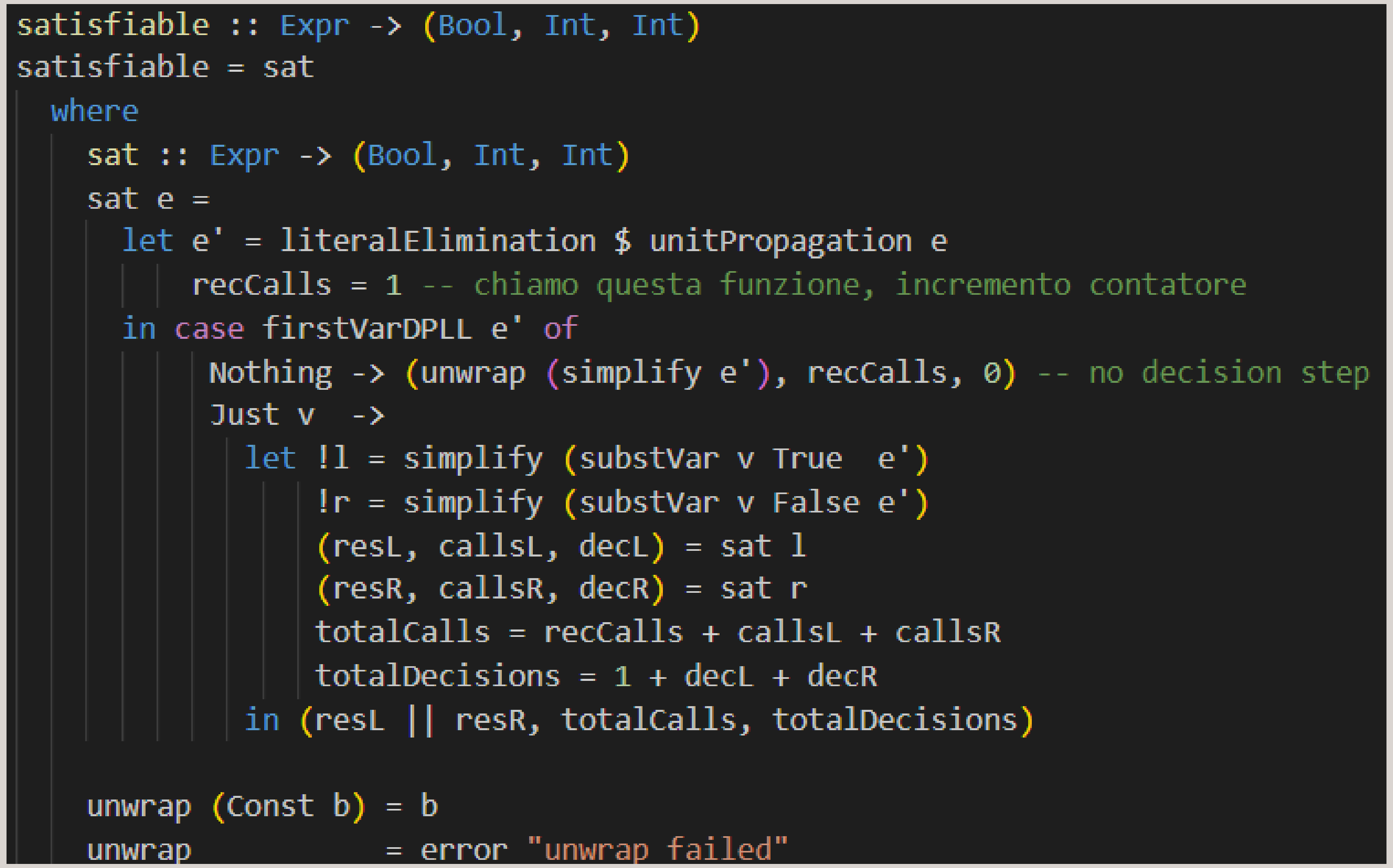
1. **Unit Clause Propagation (UCP)**:
   - If a clause contains only one unassigned literal (e.g., (A)), then A must be True for the clause to be satisfied.
   - This value is immediately assigned and propagated throughout the entire formula, simplifying other clauses containing A or (NOT A).
2. **Pure Literal Elimination (PLE)**:
   - A literal is "pure" if it appears only with one polarity (e.g., A but never NOT A).
   - If A is pure, we can assign it True (or False if it were NOT A pure) without affecting the satisfiability of other clauses.

DPLL 2

Main DPLL Structure

- **let e' = literalElimination $ unitPropagation e** : Unit-Clause Propagation then Pure-Literal Elimination reduce e to e' by applying all forced assignments and removing trivial clauses DPLL.
- **Core Backtracking on a Smaller Problem**
  - **From e'**, pick a variable, branch on True/False, and recurse exactly as in naïve backtracking.
  - **Because e'** is already pruned, the search tree is dramatically smaller

Result: By front-loading UCP and PLE before any branching, DPLL avoids exploring vast dead-end branches, achieving far more efficient SAT solving than pure backtracking.

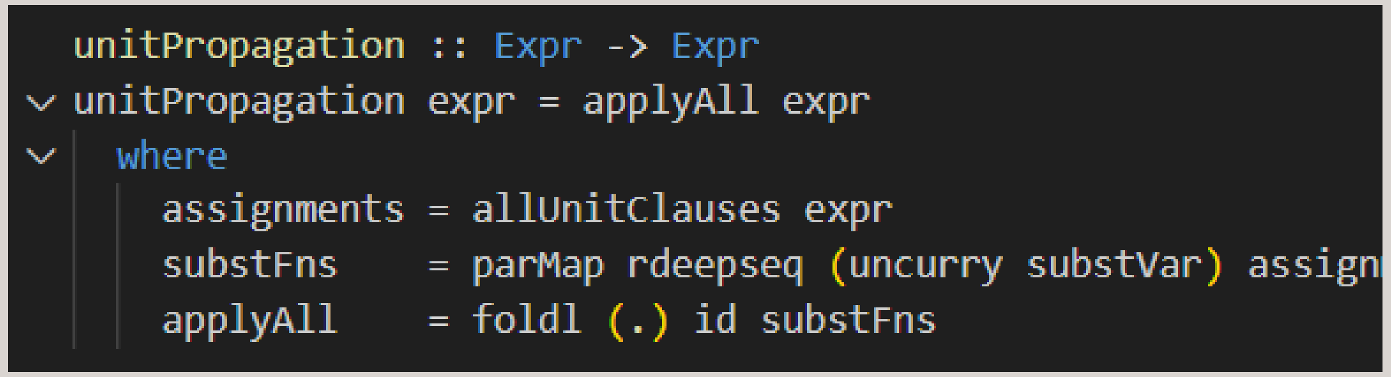


```haskell
satisfiable :: Expr -> (Bool, Int, Int)
satisfiable = sat
  where
    sat :: Expr -> (Bool, Int, Int)
    sat e =
      let e' = literalElimination $ unitPropagation e
          recCalls = 1 -- chiamo questa funzione, incremento contatore
      in case firstVarDPLL e' of
           Nothing -> (unwrap (simplify e'), recCalls, 0) -- no decision step
           Just v  ->
             let !l = simplify (substVar v True  e')
                 !r = simplify (substVar v False e')
                 (resL, callsL, decL) = sat l
                 (resR, callsR, decR) = sat r
                 totalCalls = recCalls + callsL + callsR
                 totalDecisions = 1 + decL + decR
             in (resL || resR, totalCalls, totalDecisions)

    unwrap (Const b) = b
    unwrap           = error "unwrap failed"
```

Unit Propagation with Parallelism 1

In DPLL, we first “tidy up” all forced assignments before any guessing. **By collecting every unit-clause at once and applying their substitutions in parallel, we collapse much of the formula in one swoop—**setting the stage for a far leaner search.

- **allUnitClauses expr**
  - Scans every clause for single-literal cases (A) or (¬B), returning a list like [(A,True), (B,False),...].
- **parMap rdeepseq (uncurry substVar) assignments**
  - **parMap**: Dispatches each substitution function to run on available cores.
  - **uncurry substVar**: Turns each (Var, Bool) pair into a function that replaces that variable in the expression.
  - **rdeepseq**: Forces each function to be fully evaluated (“**deep normal form**”) before considering it done, **avoiding lazy thunks** and **ensuring real parallel work**.
- **applyAll** = **foldl (.) id substFns**
  - Sequentially composes and applies all the generated substitution functions, yielding a formula with every unit clause satisfied.



```haskell
unitPropagation :: Expr -> Expr
unitPropagation expr = applyAll expr
  where
    assignments = allUnitClauses expr
    substFns    = parMap rdeepseq (uncurry substVar) assign
    applyAll    = foldl (.) id substFns
```

By generating and **evaluating these substitutions in parallel, we shrink the formula faster—**so subsequent DPLL steps operate on a much simpler problem.

Unit Propagation with Parallelism 2

Parallelism vs. Concurrency in Haskell

- **Relation to Slide 11par.md:**
  - **Parallelism:** Concerns s**imultaneous execution to improve speed** (e.g., using 4 cores to finish a calculation 4 times faster).
    - **Determinism:** In Haskell, pure parallel programs are deterministic. **The same input always produces the same output, regardless of the number of cores**. This is a huge advantage for debugging and predictability.
    - **No inherent risk of deadlock or race condition (in the pure context).**
- **Relation to Slide 12concur.md:**
  - **Concurrency:** Concerns structuring the program with **multiple threads of control**, which can execute interleaved or in parallel.
    - **Non-determinism:** The exact order of operations can vary, leading to different results.
    - **Requires synchronization:** Needs mechanisms like MVar or STM to manage shared state and prevent data access issues.
- **My Usage**: My project focuses on **deterministic parallelism (parMap)** to speed up the application of heuristics, **not on concurrency** for managing complex shared state

Unit Propagation with Parallelism 3

My Design Choice:

- **Focus:** The main **goal** of this project was to **compare** the **algorithmic performance** of Backtracking and DPLL, **not the complexity of concurrent state management.**
- **Added Complexity:** Introducing **MVar** or **STM** would have added the **need to manage potential deadlocks, race conditions, and the complexity of synchronizing state between threads**. This would have made it harder to isolate and analyze the efficiency of the SAT algorithms themselves.
- **Example:** In an industrial-grade SAT solver (like MiniSat), a "learned clause database" (clauses derived from contradictions) might be a shared data structure between threads. There, STM or MVar would be indispensable.

I prioritized the clarity and determinism of the functional implementation, choosing not to introduce the complexity of explicit concurrency.

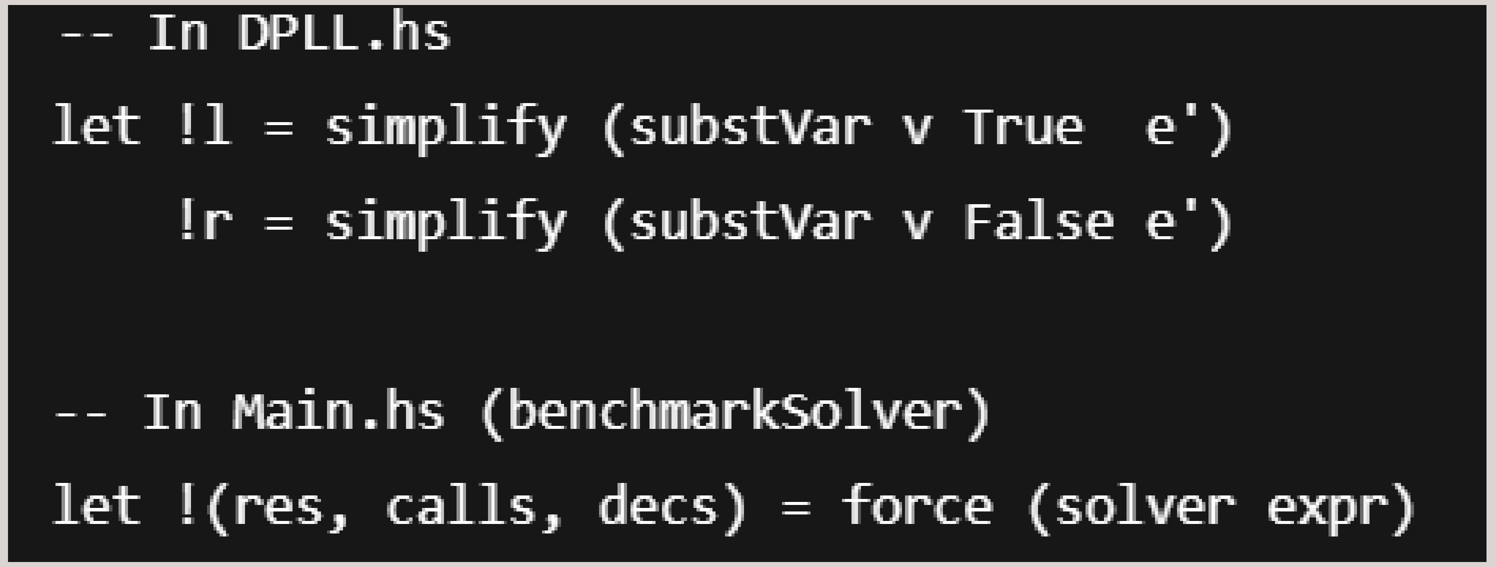
Benchmarking Methodology 1

Haskell's default laziness can hide the true cost of our solver steps. To get precise timings and metrics, we need to force evaluation.

- **The Laziness Problem in Benchmarking:**
  - If **we don't force evaluation**, a **time measurement might only record the time to create a "computation promise**" (called a "thunk"), not the time to execute the actual computation. The real work would then be performed at a later, unpredictable time.

- **BangPatterns (!) (As said previously):**
  - The **! in front of variables** (l and r in let !l = ...) **forces the evaluation** of the expression to its right **before execution proceeds.**
  - **Why**? It ensures that simplifications (which are part of the solver's work) happen immediately and are not postponed, **guaranteeing that the measured time for each branch is accurate.**

- **force (As said previously):**
  - Used in Main.hs to ensure that the final result of the solver is fully computed (in "normal form") before measuring the total execution time. Essential for precise benchmarking.



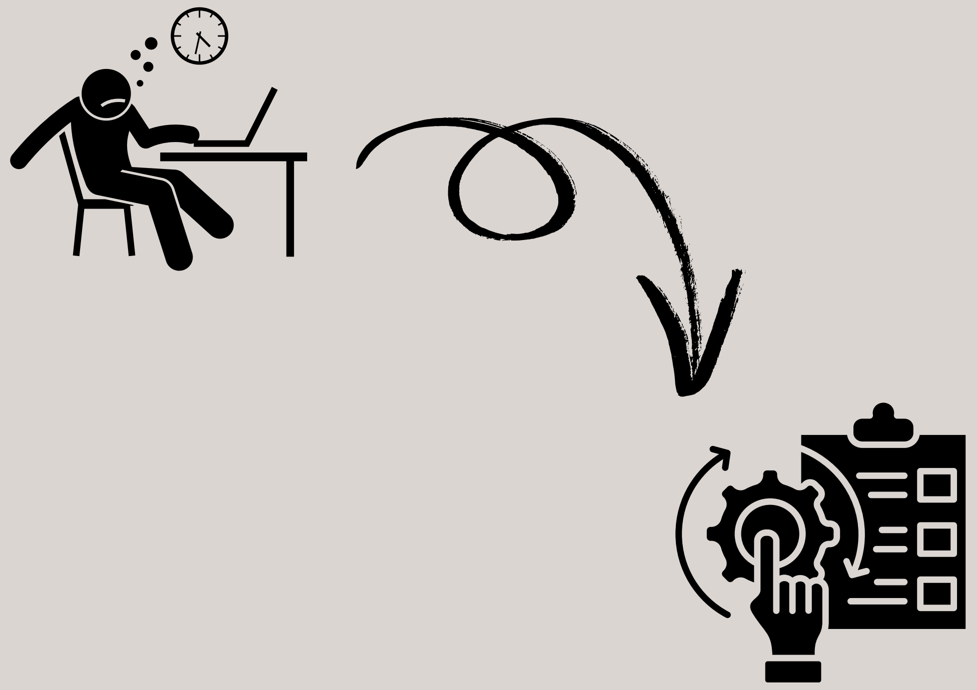
```haskell
-- In DPLL.hs
let !l = simplify (substVar v True  e')
    !r = simplify (substVar v False e')

-- In Main.hs (benchmarkSolver)
let !(res, calls, decs) = force (solver expr)
```

UNIVERSITY
OF WARSAW

Benchmarking Methodology 2

The Laziness Problem: As discussed in the previous slide, lazy evaluation can make it difficult to predict when and where work is actually performed, complicating benchmarking.

- **The Solution**:
  - **NFData**: A type class (an interface) **that evaluate all its components until there are no more "thunks" within it**. This ensures an expression is "ready."
  - **rdeepseq**: It is a function **that implements NFData for recursive structures** (like lists or expression trees), **ensuring all elements are evaluated. I have already used in parMap** ,as saw previously.



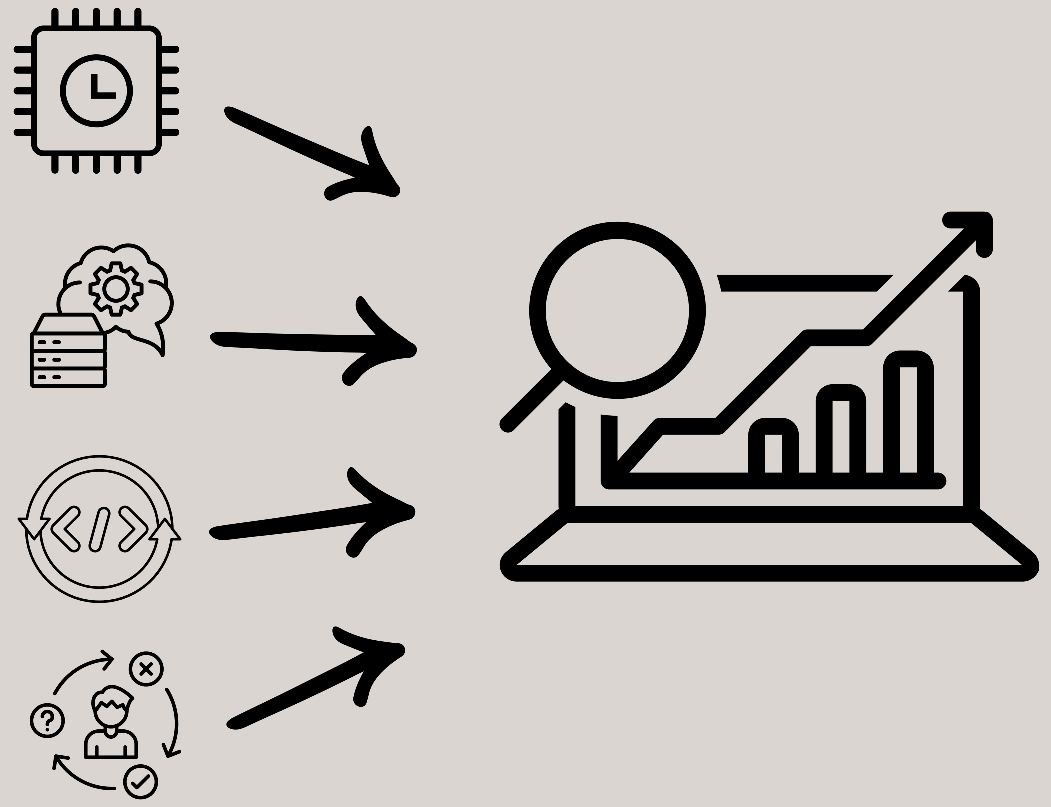
Why are they important? These techniques allow us to have precise control over when and where work is executed, which is essential for obtaining accurate performance measurements and ensuring that parallelism is effectively utilized.

Benchmarking Methodology 3

How we measure ?

Metrics Collected:

- **CPU Time:** The actual time the CPU spends solving the problem.
- **Memory Allocated:** How much memory is used during the solver's execution.
- **Number of Recursive Calls:** An indicator of how "deep" or "wide" the algorithm's exploration is.
- **Number of Decision Steps:** How many times the solver has to make a choice about a variable.

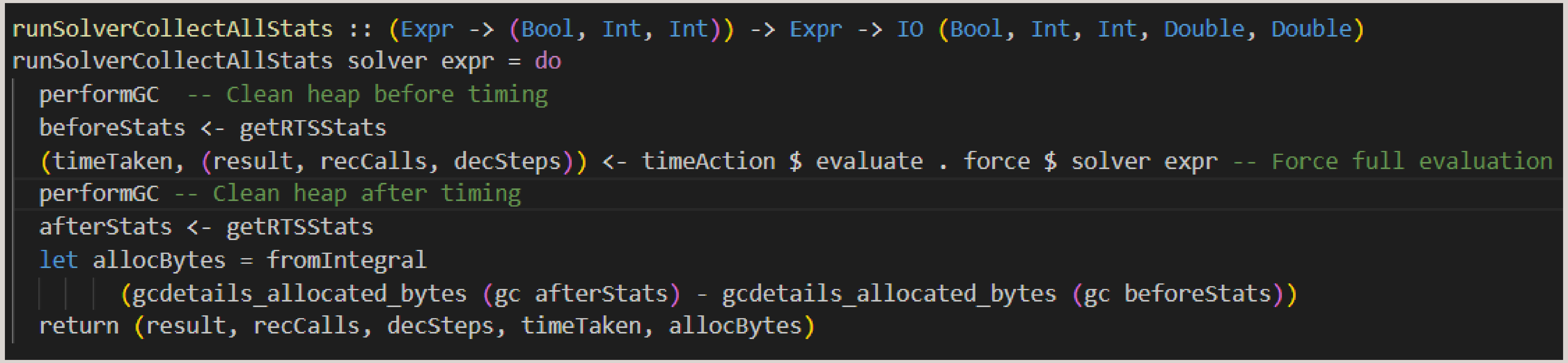


The "Why" is Crucial in Haskell: Haskell's "lazy" nature requires special attention to obtain accurate measurements. Without these precautions, times and memory usage could be misleading.

UNIVERSITY OF WARSAW

Benchmarking Methodology 4

How I have taken measure?



```haskell
runSolverCollectAllStats :: (Expr -> (Bool, Int, Int)) -> Expr -> IO (Bool, Int, Int, Double, Double)
runSolverCollectAllStats solver expr = do
  performGC  -- Clean heap before timing
  beforeStats <- getRTSStats
  (timeTaken, (result, recCalls, decSteps)) <- timeAction $ evaluate . force $ solver expr -- Force full evaluation
  performGC -- Clean heap after timing
  afterStats <- getRTSStats
  let allocBytes = fromIntegral
        (gcdetails_allocated_bytes (gc afterStats) - gcdetails_allocated_bytes (gc beforeStats))
  return (result, recCalls, decSteps, timeTaken, allocBytes)
```

- **performGC:** Forces a garbage collection before each run so that previous “thunks” or leftover allocations don’t skew our measurements.
- **force:** Ensures the solver’s entire computation is evaluated to normal form before we stop the clock—otherwise we’d only measure the time to create thunks, not to execute them.
- **getCPUTime & getRTSStats:** GHC primitives that report precise CPU usage and memory allocation, giving us reliable metrics for comparison.
- **Cross-Validation:** After both solvers run, we automatically compare their Boolean results to catch any implementation bugs on the spot.

UNIVERSITY OF WARSAW

Results & Analysis 1

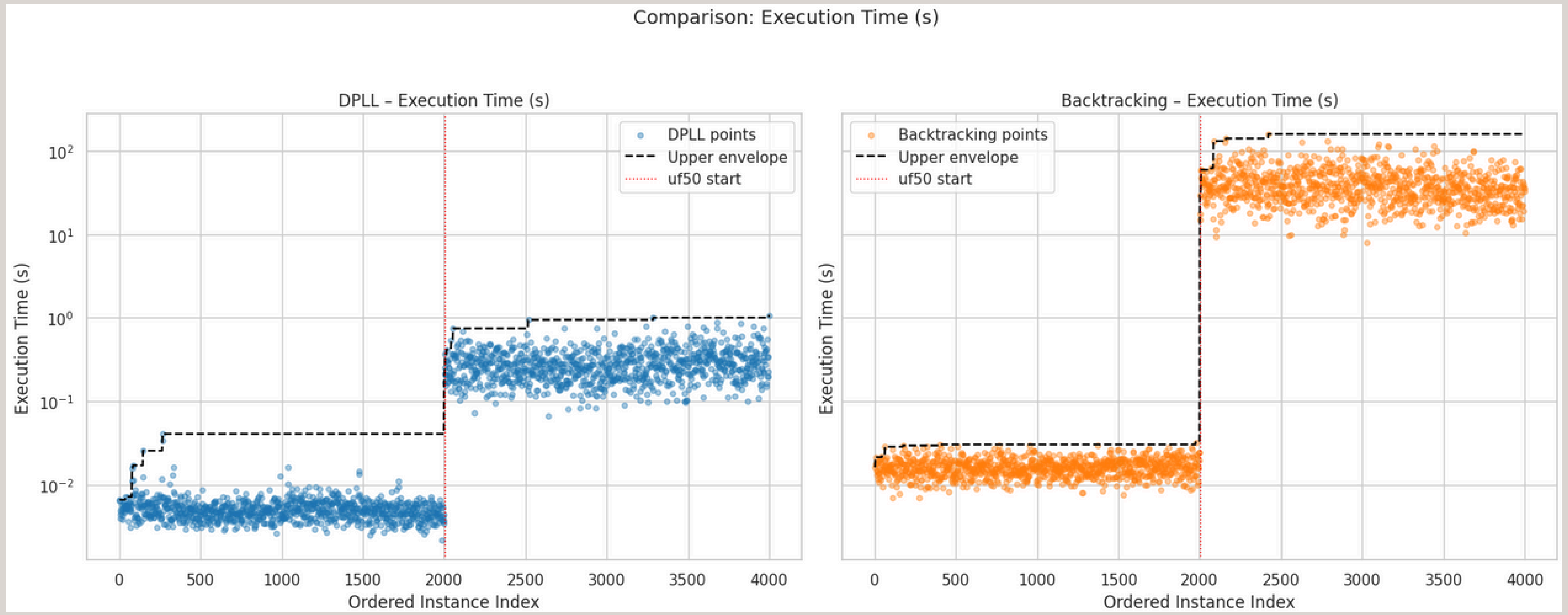
What you're seeing: execution times on a logarithmic scale for two problem sizes, uf20 (20 variables) and uf50 (50 variables). The x-axis orders instances by increasing difficulty; the y-axis shows solve time.

uf20 results:

- **DPLL** (parallel) **averages 5–15 ms**, with very few outliers above 20 ms.
- **Backtracking** (sequential) averages **20–30 ms**, occasionally spiking to 60 ms.

uf50 results:

- **DPLL** still resolves almost every instance in 0.2–0.8 s (**rarely exceeding 1 s**).
- **Backtracking** frequently runs into tens of seconds and often times out at our **300 s cutoff**.



Why the gap?

- **Unit-Clause Propagation and Pure-Literal Elimination in DPLL automatically satisfy or remove many clauses up front—this** "forced assignment" phase **slashes the size of the search tree before any guessing.**
- **Parallel substitution (via parMap rdeepseq) speeds up these propagations further.**
- **The backtracker**, lacking these clean-up steps and parallelism, must blindly **try almost every possible assignment**, causing **exponential** blow-up.

UNIVERSITY OF WARSAW

Results & Analysis 2

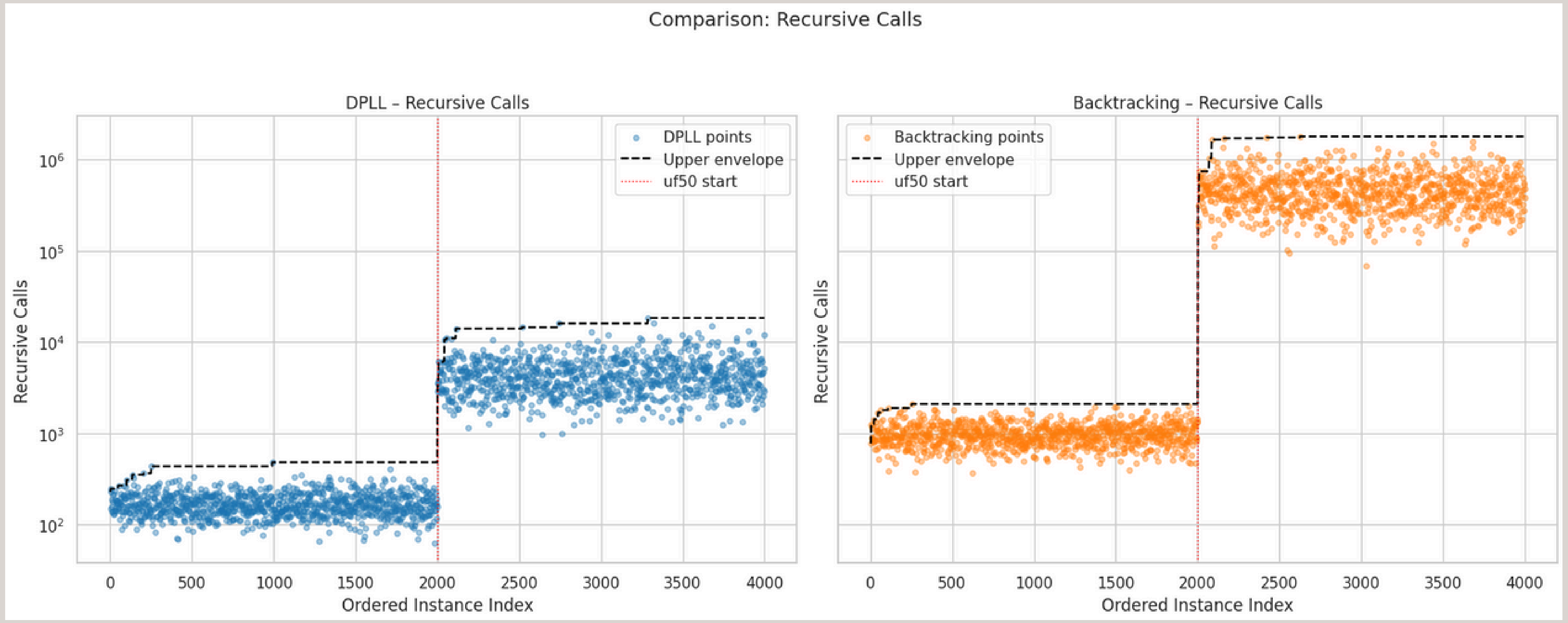
What you're seeing: the total number of recursive calls each solver makes, on a log scale, for two problem families, uf20 (first half of points) and uf50 (second half).

uf20 results:

- **DPLL** (parallel) typically uses **10^2–10^3** calls per instance.
- **Backtracking** (sequential) uses 10^3–10^4 calls **— an order of magnitude more**

uf50 results:

- **DPLL** climbs to 10^3–10^4 calls, reflecting larger formulas but **still manageable.**
- **Backtracking explodes into 10^5–10^6 calls**, often hitting recursion or time limits.



Why it matters:

- Every time **DPLL applies unit-clause or pure-literal rules, it prunes entire subtrees of the search space, skipping thousands of needless recursive explorations.**
- **The backtracker, lacking these deductions,** revisits dead-end branches over and over, causing its recursion count to skyrocket.

Results & Analysis 3

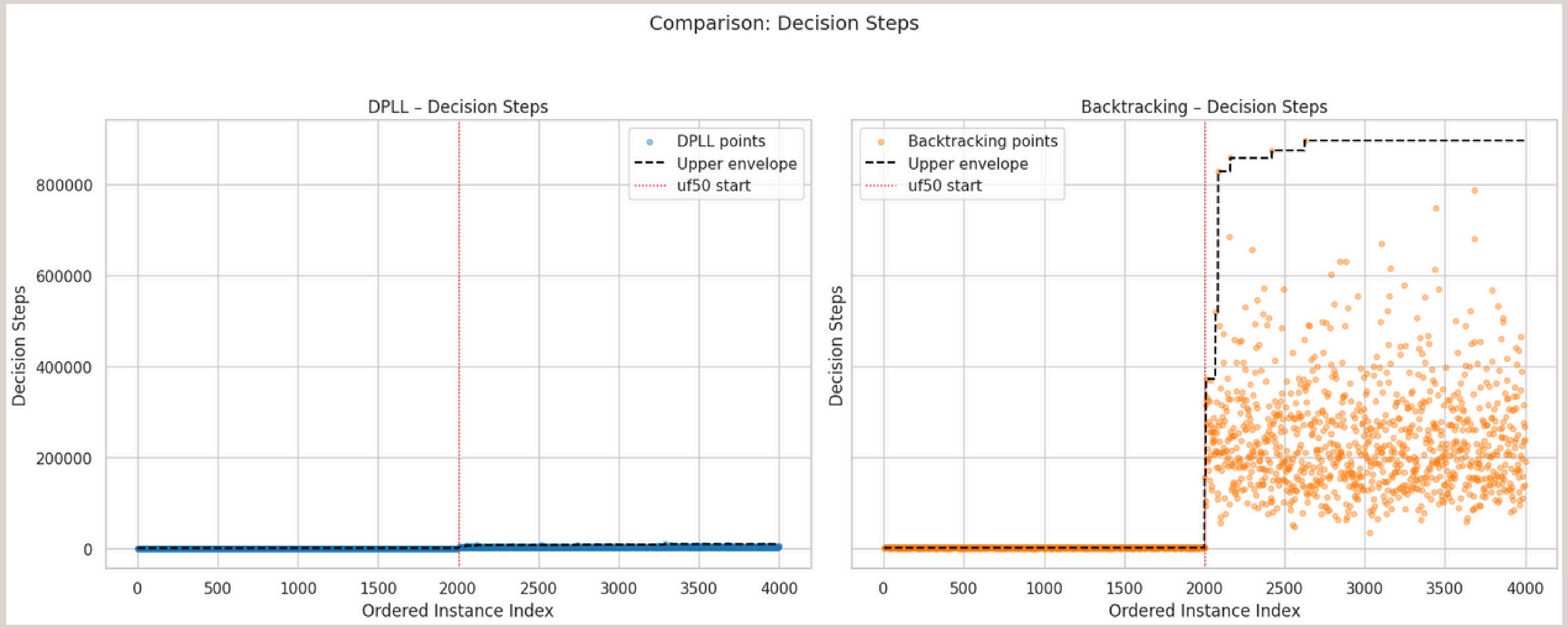
What "decision steps" are: each time the solver must explicitly pick a variable and try True vs. False.

uf20 results:

- **DPLL** averages just a few **×10 decisions per instance.**
- **Backtracking** sits in the low hundreds of guesses **—10× more work.**

uf50 results:

- **DPLL barely** climbs into the **low hundreds of decisions**
- **Backtracking explodes into tens of thousands** of individual True/False guesses



Key insight:

- Every propagation or pure-literal elimination in DPLL automates a decision you'd otherwise have to make by hand.
- Fewer manual branches means a dramatically smaller search tree and much faster solving.

UNIVERSITY OF WARSAW

Results & Analysis 4

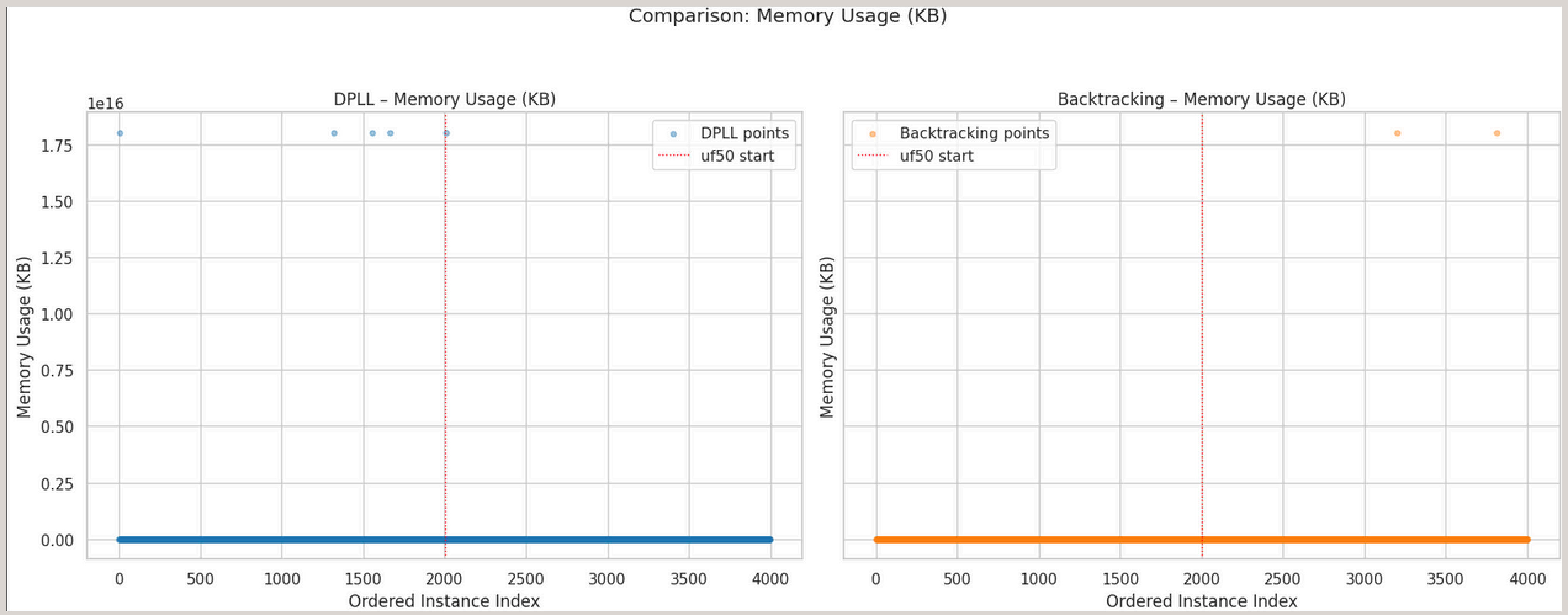
What "Memory Usage" are: memory usage in the whole execution time per each tests.

Flat, Identical Baseline:

- Across both uf20 and uf50 instance sets, DPLL and Backtracking trace virtually the same flat memory curve—just a few dozen kilobytes beyond GHC's own overhead.

No Major Spikes:

- Neither solver shows significant memory spikes in our tests. Even on harder uf50 formulas, the usage remains at that baseline.



Why It Matters:

Memory usage isn't the differentiator here—both approaches run in constant, minimal space. The real gains of DPLL show up in time and search-depth metrics, not in heap footprint.

Conclusion

What These Numbers Teach Us ?

1. **Heuristics Matter**. Unit propagation and pure-literal elimination shrink search by 1–3 orders of magnitude in calls, decisions, and runtime.
2. **Lightweight Parallelism Helps**. Using **parMap,rdeepseq** during unit propagation on multicore hardware yields **an additional 10–20% speedup**, all while keeping code deterministic and simple.
3. **Haskell Benchmarking Demands Discipline**. Without **BangPatterns, force, and performGC,** lazy evaluation **would hide** most of our work in thunks, skewing results.
4. **Practical Viability**. In our benchmarks, the Haskell DPLL solver effortlessly finishes 100-150 variable instances in under two seconds—and often in a few hundred milliseconds. Extrapolating these trends suggests that, with its pruning power intact, the same solver could handle SAT problems with 500+ variables in a similarly reasonable timeframe.
   In contrast, the backtracker bogs down beyond small toy examples, making it practical only for very limited—tens-of-variables—instances.

UNIVERSITY OF WARSAW

References

- **Cited Slides**: Benke, M. (2022). ZPF2022 Slides.
  Available at https: *//github.com/mbenke/zpf2022/tree/master/Slides*
- **Dataset di Test**: The tests with 20 and 50 clauses were taken from SATLIB.
  Hoos, H. (n.d.). SATLIB - Benchmarks. Disponibili su https: *//www.cs.ubc.ca/~hoos/SATLIB/benchm.html*

UNIVERSITY OF WARSAW

Thank you for your attention

Pierfrancesco Lijoi-K17413