Formal Methods in Differential and Linear Trail Search

Benoît Viguier

October 7, 2016

Overview

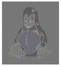
Introduction

Hashing vs Encryption

Hashing

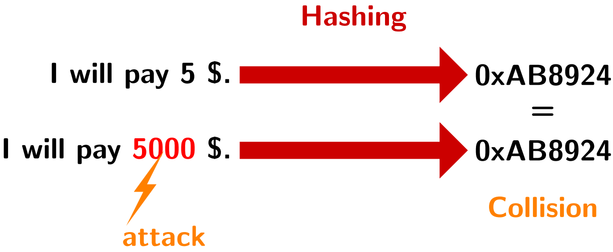
Hash 

 Plaintext

Ciphertext 

Encryption

Second pre-image attack



Keccak

Sponge construction

Sponge construction + invertible permutation f named $\textsc{Keccak}\text{-}f[b]$.

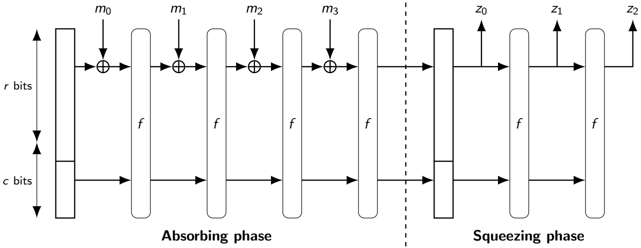


Figure 1: A sponge construction

bit rate (r) + capacity (c) = width (b)

$\textsc{Keccak}\text{-}f[1600] = (\iota \circ \chi \circ \pi \circ \rho \circ \theta)^{24}$ and $b = 1600$

State

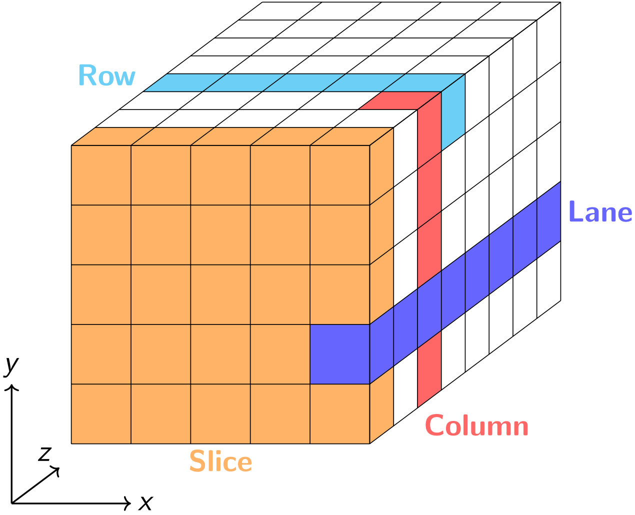


Figure 2: Keccak[200] state

Keccak-f: θ

Linear mixing layer on column parity.

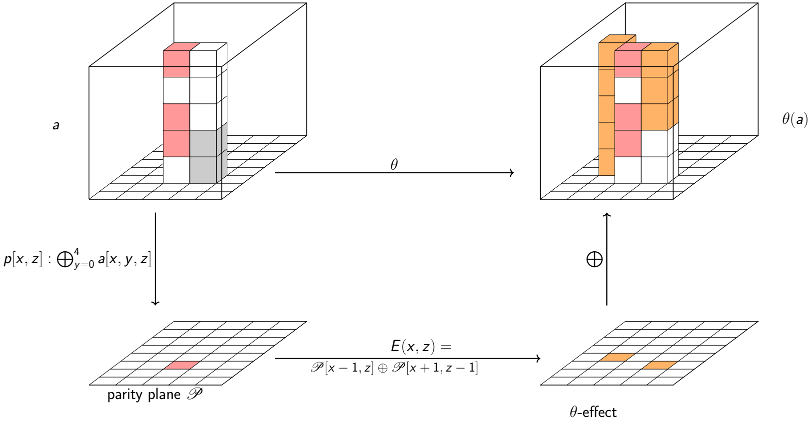


Figure 3: Application of θ to a state.

$\mathscr{P}$ is called the parity plane.

Keccak-f: ρ and π

Bit-wise cyclic shift rotation on lanes.

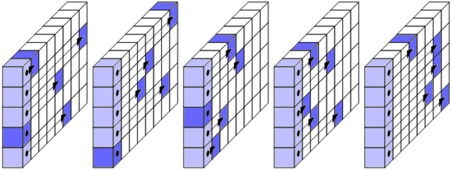


Figure 4: The ρ transformation

Lane transposition.

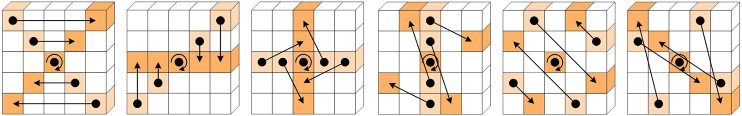


Figure 5: The π transposition

Non-linear mapping f algebraic degree of 2 which operates on rows.

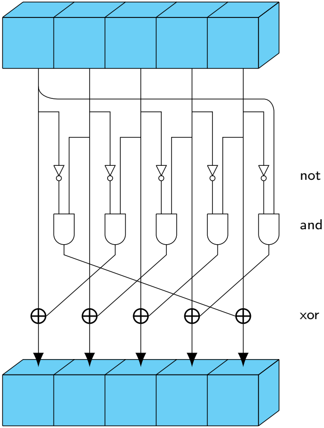


Figure 6: The χ transformation

Differential Cryptanalysis

Differentials

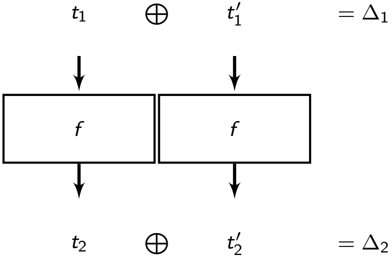


Figure 7: A differential $(\Delta_1 \overset{f}{\Rightarrow} \Delta_2)$

Given an input difference Δ_1, chances are that a difference Δ_2 will occur. It can be associated with a probability: $P[(\Delta_1 \Rightarrow \Delta_2)]$.

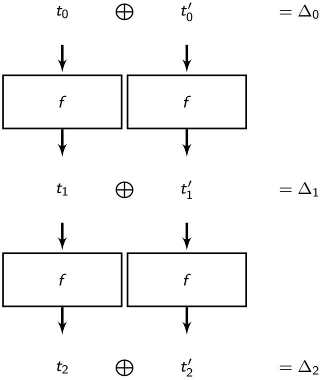


Figure 8: A trail ($\Delta_0 \overset{f}{\Rightarrow} \Delta_1 \overset{f}{\Rightarrow} \Delta_2$)

Goal: Find a trail ($\Delta_0 \overset{f}{\Rightarrow} \cdots \overset{f}{\Rightarrow} \Delta_n$) such as $\Delta_n = 0$ ($\Leftrightarrow$ collision).

There are $2^{1600} - 1$ input differences possible for Keccak-$f[1600]$.

Estimated number of hydrogen atoms in the Universe: $\approx 2^{265}$

Tree decomposition

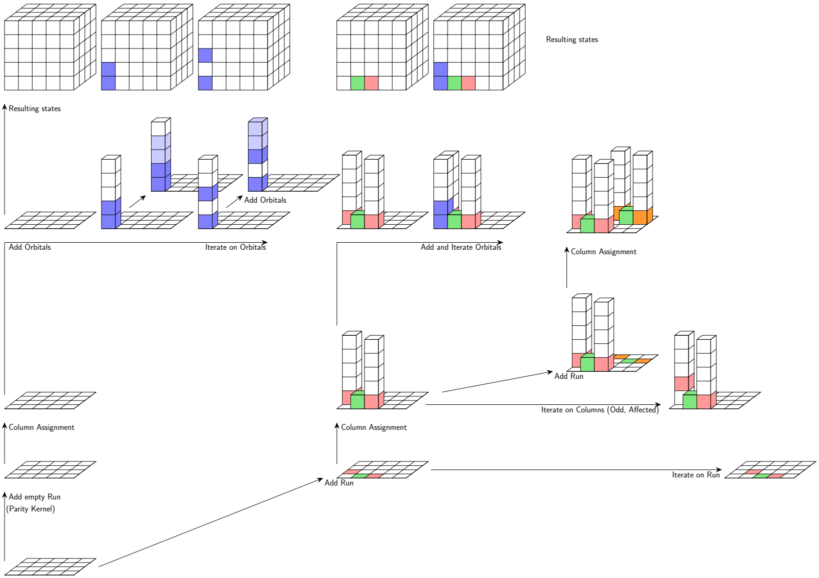


Figure 9: Tree decomposition of the search.

From Code to Proofs

What are the verifications needed?

- orbitals (involution, order)
- colunms assignement (order)
- runs (order, z-canonicity factorization)
- and more...

What are the difficulties?

- C++ $\Rightarrow$ no VST, no FRAMA-C, no Why3.
- Huge source code!

What have been done during this Internship?

- Using Hoare Logic.
- Orbitals: involution and order

The time I would have spent on more proofs would not have been compensated by the gain of the correction.

Tree decomposition

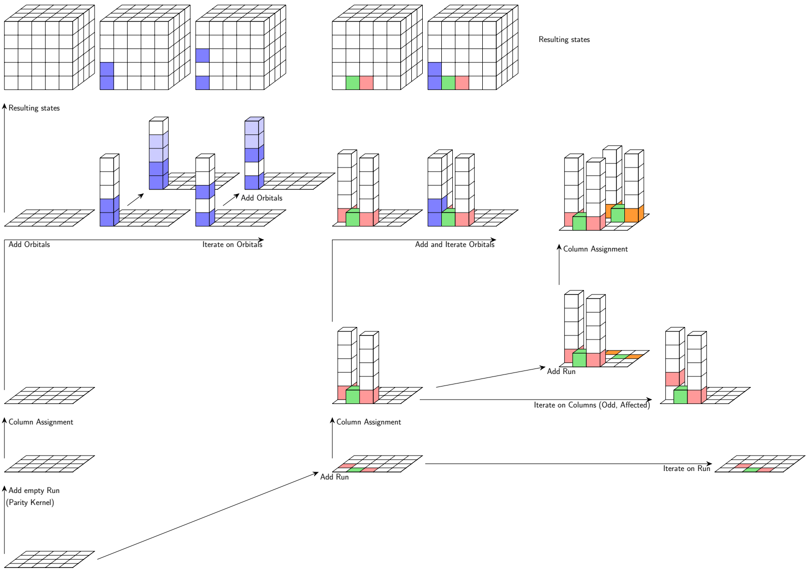


Figure 10: Tree decomposition of the search.

Tree decomposition

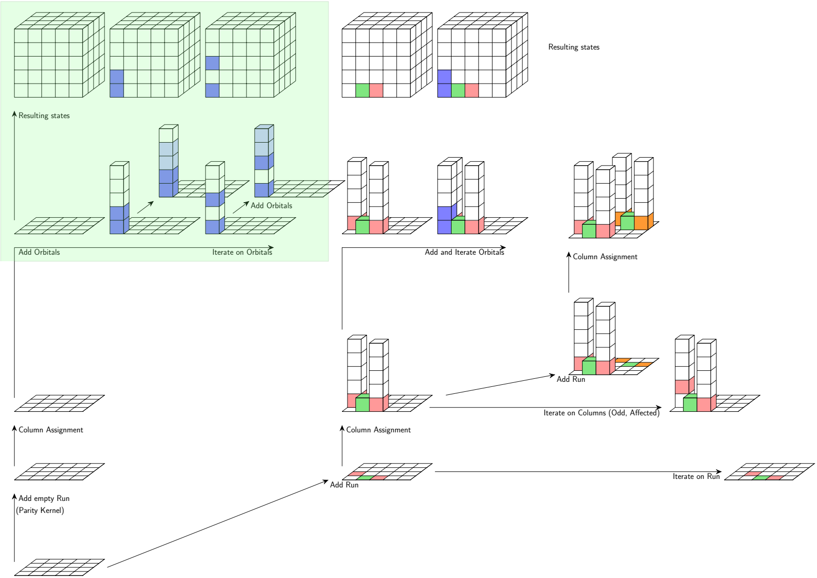


Figure 10: Tree decomposition of the search.

Tree decomposition

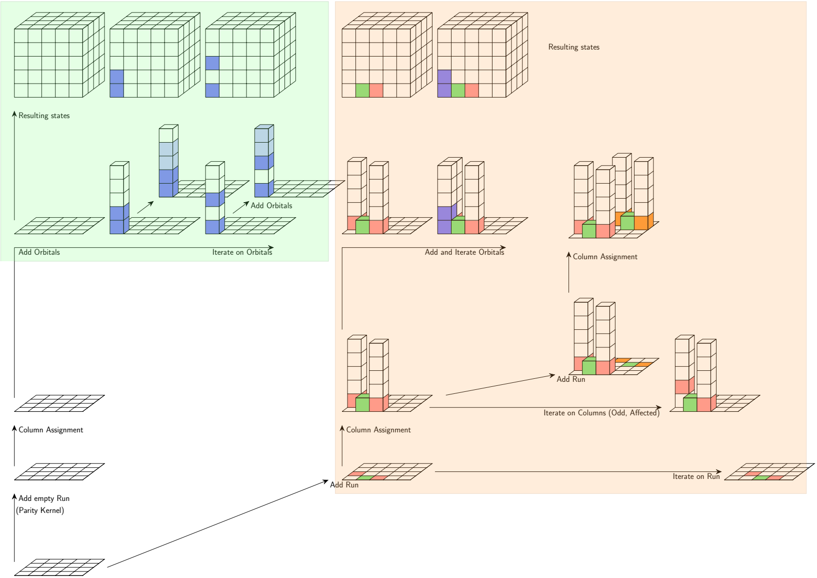


Figure 10: Tree decomposition of the search.

Semantics of Trees and Iterators

Tree traversal: Tree definition

```
Section trees.
  Variable (X : Type).

  (* we do not want the too weak Coq generated
    induction principles *)
  Unset Elimination Schemes.

  Inductive Tree : Type :=
    node : X → list Tree → Tree.
  Set Elimination Schemes.

  Section Tree_ind.
    Variable P : Tree → Prop.
    Hypothesis HP : ∀ a ll,
      (∀ x, In x ll → P x) →
        P (node a ll).

    Definition Tree_ind : ∀ t, P t.
  End Tree_ind.
End trees.
```

Code 1: Tree definition

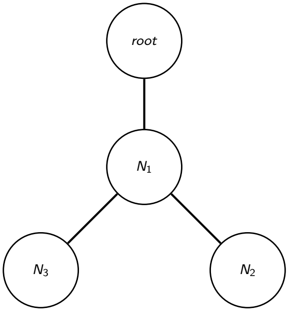


Figure 11: Tree

Induction principle:

1. prove the property for a tree with no children.
2. Assume that the property is `True` for all children, prove it for the parent.

Tree traversal: Path definition

```
Definition Path := list nat.

Definition getNode (p:Path) (t:Tree X) : option (Tree X) := ...
```

Code 2: Path definition

Each node from the tree can be accessed by a path specified as the list of the index of the child to consider.

- `[ ]` returns *root*.
- `[0]` returns N_1.
- `[0,0]` returns N_2.
- `[1,0]` returns N_3.

$getNode\,(p)$ returns `Some` (n, l) if a node n with childrens l exists or `None`.

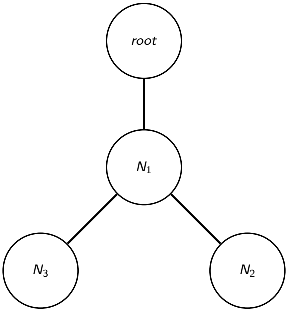


Figure 12: Tree

Tree traversal: Moves

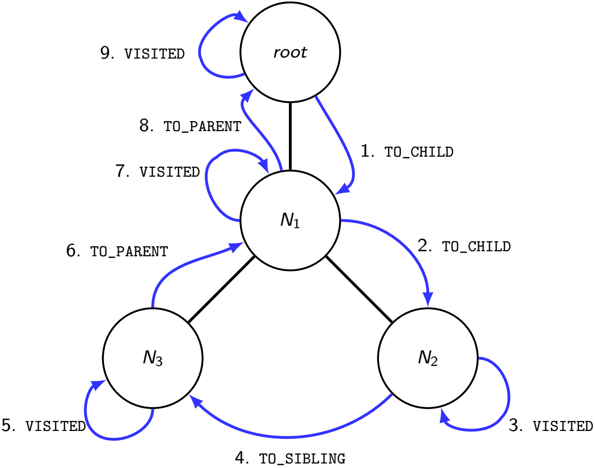


Figure 13: Iteration through a tree

```
Inductive MoveSS : Type := TO_PARENT | TO_CHILD | TO_SIBLING | VISITED.
```

Code 3: Definition of the movements

Tree traversal: Rules

We can use Small-step semantics to specify rules over moves.

$$it : (move, path, visited\ nodes) \rightarrow (move', path', visited\ nodes')$$

```
Inductive iterator_smallstep_v X :
  Tree X → MoveSS * Path * (Visited X)→ MoveSS * Path * (Visited X) → Prop :=
   ...
```

$$\frac{}{(\mathtt{TO\_PARENT}, p, v) \rightarrow (\mathtt{VISITED}, p, v)}\ (\text{visit\_up})$$

$$\frac{m \neq \mathtt{TO\_PARENT} \qquad m \neq \mathtt{VISITED} \qquad getNode\,(p) \mapsto \mathtt{Some}\ (n, [\,])}{(m, p, v) \rightarrow (\mathtt{VISITED}, p, n :: v)}\ (\text{visit\_no\_sons})$$

$$\frac{m \neq \mathtt{TO\_PARENT} \qquad m \neq \mathtt{VISITED} \qquad getNode\,(p) \mapsto \mathtt{Some}\ (n, l) \qquad l \neq [\,]}{(m, p, v) \rightarrow (\mathtt{TO\_CHILD}, 0 :: p, n :: v)}\ (\text{down})$$

$$\frac{getNode\,(h :: p) \mapsto \mathtt{Some}\ (n, l) \qquad getNode\,(h + 1 :: p) \mapsto \mathtt{None}}{(\mathtt{VISITED}, h :: p, v) \rightarrow (\mathtt{TO\_PARENT}, p, v)}\ (\text{up})$$

$$\frac{getNode\,(h + 1 :: p) \mapsto \mathtt{Some}\ (n, l)}{(\mathtt{VISITED}, h :: p, v) \rightarrow (\mathtt{TO\_SIBLING}, h + 1 :: p, v)}\ (\text{next})$$

1. `visit_up`
   If we just went back to the parent,
   the next move is `VISITED`.
2. `visit_no_sons`
   If the node does not have children,
   the next move is `VISITED`.
3. `down`
   If the node has a child
   (and the node is not `VISITED`),
   the next move is `TO_CHILD`.
4. `up`
   If the node is `VISITED`
   and has no siblings,
   the next move is `TO_PARENT`
5. `next`
   If the node is `VISITED` and has siblings,
   the next move is `TO_SIBLING`

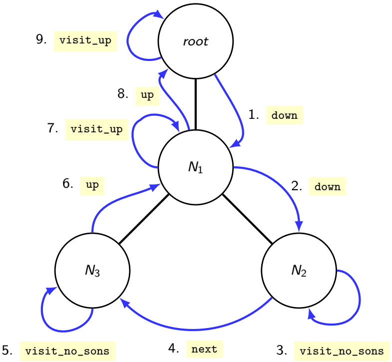


Figure 14: Iteration rules applied to tree traversal

Tree traversal: Theorems

Iterator is **deterministic**:

$$\forall\, move\ path\ visited,$$
$$(\,\forall\, move_1\ path_1\ visited_1,\ it : (move, path, visited) \rightarrow (move_1, path_1, visited_1) \wedge$$
$$\forall\, move_2\ path_2\ visited_2,\ it : (move, path, visited) \rightarrow (move_2, path_2, visited_2)\,) \Rightarrow$$
$$move_1 = move_2 \wedge path_1 = path_2 \wedge visited_1 = visited_2$$

Iterator's traversal is **complete**:

$$it : (\mathtt{TO\_CHILD}, [\,], [\,]) \rightarrow^* (\mathtt{VISITED}, [\,], visited)$$

where *visited* is the list of the values of all the nodes

Tree pruning

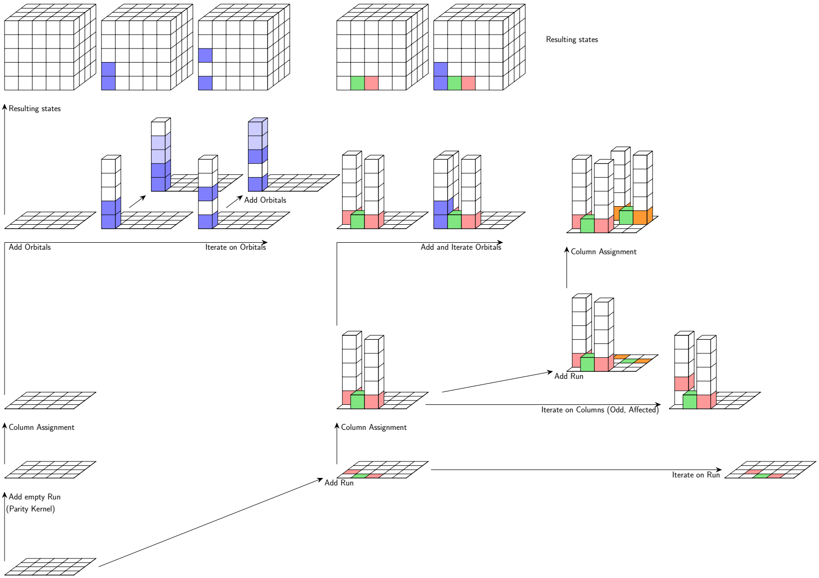


Figure 15: Tree pruning.

Tree pruning

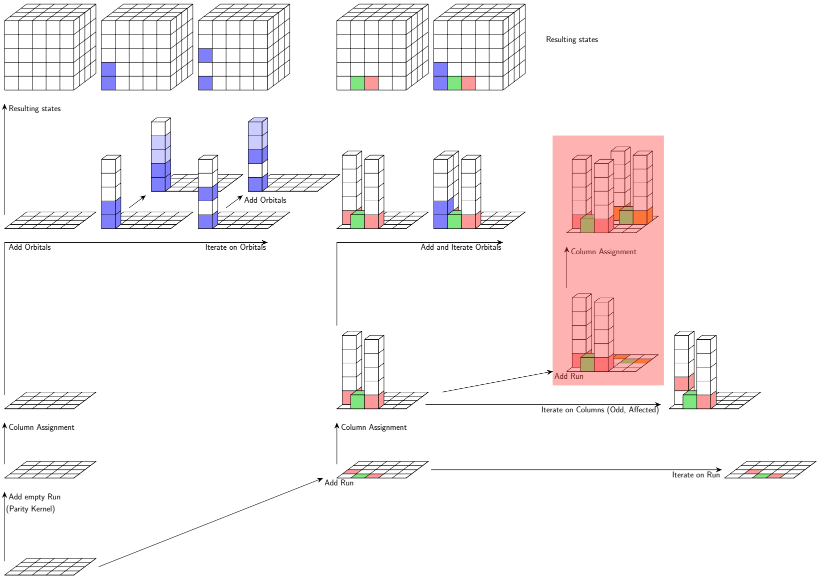


Figure 15: Tree pruning.

Tree traversal: Rules Augmented

The iterator should also cut branches of the tree when some conditions are met (simulated by the evaluation of a function $B : node \rightarrow Bool$)

$$\frac{}{(\texttt{TO\_PARENT}, p, v) \rightarrow (\texttt{VISITED}, p, v)} \text{(visit\_up)}$$

$$\frac{m \neq \texttt{TO\_PARENT} \qquad m \neq \texttt{VISITED} \qquad getNode\ p \mapsto \texttt{Some}\ (n, [\,]) \qquad \boxed{B\ n = \texttt{True}}}{(m, p, v) \rightarrow (\texttt{VISITED}, p, \boxed{n} :: v)} \text{(visit\_no\_sons\_true)}$$

$$\frac{m \neq \texttt{TO\_PARENT} \qquad m \neq \texttt{VISITED} \qquad getNode\ p \mapsto \texttt{Some}\ (n, l) \qquad l \neq [\,] \qquad \boxed{B\ n = \texttt{True}}}{(m, p, v) \rightarrow (\texttt{TO\_CHILD}, 0 :: p, \boxed{n} :: v)} \text{(down)}$$

$$\frac{m \neq \texttt{VISITED} \qquad getNode\ p \mapsto \texttt{Some}\ (n, l) \qquad \boxed{B\ n = \texttt{False}}}{(m, p, v) \rightarrow (\texttt{VISITED}, p, v)} \text{(down\_forbiden)}$$

$$\frac{getNode\ (h :: p) \mapsto \texttt{Some}\ (n, l) \qquad getNode\ (h + 1 :: p) \mapsto \texttt{None}}{(\texttt{VISITED}, h :: p, v) \rightarrow (\texttt{TO\_PARENT}, p, v)} \text{(up)}$$

$$\frac{getNode\ (h + 1 :: p) \mapsto \texttt{Some}\ (n, l)}{(\texttt{VISITED}, h :: p, v) \rightarrow (\texttt{TO\_SIBLING}, h + 1 :: p, v)} \text{(next)}$$

Proven Iterator

Toward an Iterator

The iterator (*manager*) should provide the next move with the minimum of required information.

- Path
- is the last move *toward the parent* ?
- move `VISITED` will be skipped.

Iterator in Gallina

```
(*
what are the assumptions before going in this function? Make no such assumption.
Only need to know only one thing: was the last move TO_PARENT (last_up = true)?
*)
Definition manager X (t:Tree X) (B:X → bool) (pl:option (Path*bool)) :
option (MoveSS) :=
match pl with
| None ⇒ None
| Some (p,last_up) ⇒ match getNode p t with
  | None ⇒ None
  | _ ⇒
    if andb (NodeValid p t B) (negb last_up) then  (* ∧                        *)
      if ChildExists p t then                      (* |                        *)
        Some TO_CHILD                              (* |                        *)
      else                                         (* |                        *)
        if SiblingExists p t then                  (* |  This part will be     *)
          Some TO_SIBLING                          (* |  directly translated   *)
        else                                       (* |  into C++.             *)
          Some TO_PARENT                           (* |                        *)
    else                                           (* |                        *)
        if SiblingExists p t then                  (* |                        *)
          Some TO_SIBLING                          (* |                        *)
        else                                       (* |                        *)
          Some TO_PARENT                           (* ∨                        *)
  end
end.
```

Code 4: Given a path we can select the next move

C++: Iterator

```
/**
 * The code is not optimized, it is written as defined in Coq
 */
Move Manager::next_move() {

  if (path->isNodeValid() && !is_last_move_to_parent) {
    if (path->hasChild()) {
      return TO_CHILD;
    }
    else {
      if (path->hasSiblings()) {
        return TO_SIBLING;
      }
      else {
        return TO_PARENT;
      }
    }
  }
  else {
    if (path->hasSiblings()) {
      return TO_SIBLING;
    }
    else {
      return TO_PARENT;
    }
  }
}
```

Code 5: Definition of the Manager in C++

The C++ Gallina Equivalence

```
Definition manager X (t:Tree X) (B:X → bo
option (MoveSS) :=
match pl with
| None ⇒  None
| Some (p,last_up) ⇒  match getNode p t w
  | None ⇒  None
  | _ ⇒
    if andb (NodeValid p t B) (negb last_u
      if ChildExists p t then
        Some TO_CHILD
      else
        if SiblingExists p t then
          Some TO_SIBLING
        else
          Some TO_PARENT
    else
        if SiblingExists p t then
          Some TO_SIBLING
        else
          Some TO_PARENT
  end
end.
```

Figure 16: Code Gallina

```
Move Manager::next_move() {

  if (path->isNodeValid() && !is_
    if (path->hasChild()) {
      return TO_CHILD;
    }
    else {
      if (path->hasSiblings()) {
        return TO_SIBLING;
      }
      else {
        return TO_PARENT;
      }
    }
  }
  else {
    if (path->hasSiblings()) {
      return TO_SIBLING;
    }
    else {
      return TO_PARENT;
    }
  }
}
```

Figure 17: Code C++

Equivalence

$$\forall\, tree\ path\ last\_up\ move\ path',$$
$$manager(tree, path, last\_up) \mapsto move \land apply(move, path) \mapsto path' \Rightarrow$$
$$it : (\ldots, path, \ldots) \to (move, path', \ldots)$$

```
Theorem managerEqSemantic :
  ∀ X (B:X → bool) (tree:Tree X) (m m':MoveSS) (p p':Path) last_up last_up',
(*
  Define the equivalence between the last movement and the last_up boolean
  value as hypotheses.
*)
  (last_up' = true ↔ (m' = TO_PARENT)) →
  (last_up = false ↔ (m = TO_CHILD ∨ m = TO_SIBLING)) →
  (last_up = true ↔ (m = TO_PARENT) ∧ NodeExists (0::p) tree = true) →

(* Apply the move to the path and return the boolean value to for the manager *)
  applyMove p m' = Some (p',last_up') →

  (* manager hypothesis *)
  manager tree B (Some (p,last_up)) = Some m'

  →

 (* Either we have an intermediate VISITED step *)
 (iterator_nv B tree (m,p) (VISITED, p) ∧ iterator_nv B tree (VISITED,p) (m', p')
 (* Or we are right *)
  ∨ iterator_nv B tree (m,p) (m', p')).
```

Code 6: Theorem of the implication between the manager and the semantic iterator

Tree decomposition

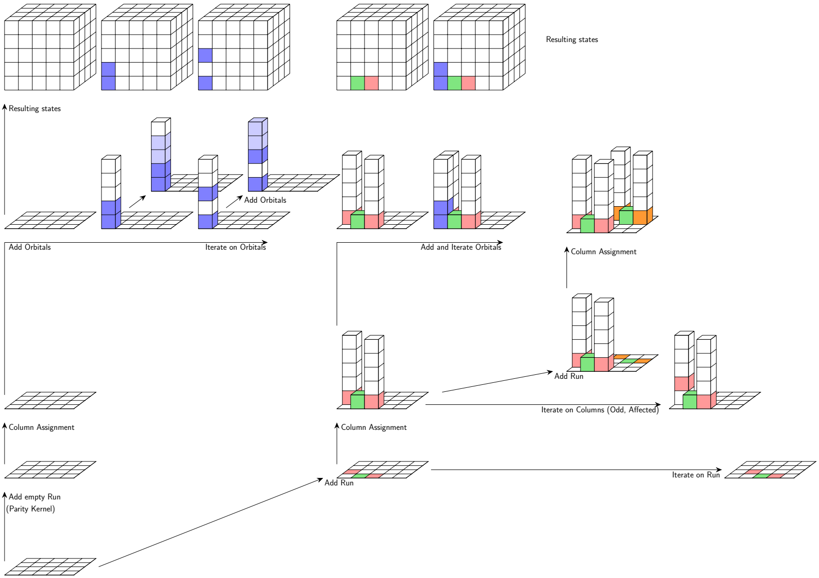


Figure 18: Tree decomposition of the search.

C++: Tree and moves to stack

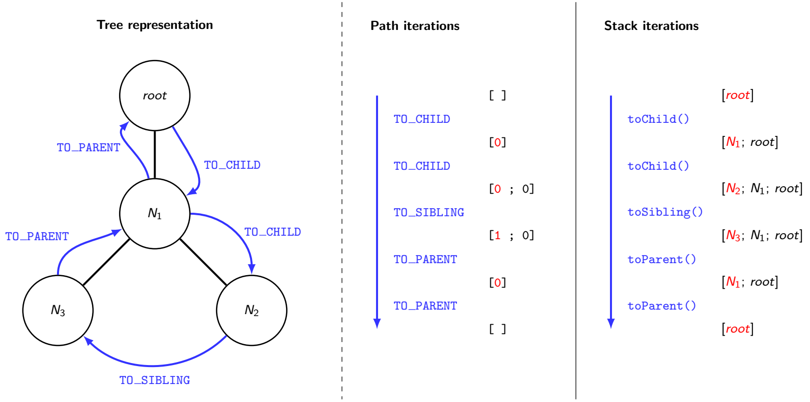


Figure 19: Tree/Path and Stack equivalence. The head of the list/stack is in red.

What do we have to trust ?

What do we trust?

- Calculus of Inductive Construction
- Specification and Small-step semantics
- Tree implementation and specification: **WEAK LINK**
- Translation from GALLINA to C++
- GCC
- Coq kernel, Ocaml compiler, Ocaml Runtime, CPU.

Conclusion

Conclusion

From Code to Proofs:

The orbitals iterator has been proven correct (involution + order) with Hoare logic.

From Proofs to Code:

- Specification of generic tree
- Specification of an iterator in Small-step semantics
- Definition of an abstract iterator (*manager*) which fully traverse any given tree.

By providing such iterator, we reduce the trust to the tree definition/construction.

Questions?

Thank you !

Differentials: Probabilities and Weight

We define the **weight (w)** of a differential as follow.

$$P[(\Delta_1 \Rightarrow \Delta_2)] = \frac{1}{2^w}$$

The weight of a trail $Q = (\Delta_0 \Rightarrow \cdots \Rightarrow \Delta_n)$ is the sum of the weight of its differentials.

$$w(Q) = \sum_{i=0}^{n-1} w(\Delta_i \Rightarrow \Delta_{i+1})$$

Remark: **Affine applications have no influence on the probabilities of differentials.**

Trails in Keccak

Let Q be a trail of differences $a_0, a_1, \dots a_n$:

$$Q = a_0 \xRightarrow{\chi\circ\pi\circ\rho\circ\theta} a_1 \xRightarrow{\chi\circ\pi\circ\rho\circ\theta} \dots \xRightarrow{\chi\circ\pi\circ\rho\circ\theta} a_n$$

$$Q = a_0 \xRightarrow{\pi\circ\rho\circ\theta} b_0 \overset{\chi}{\Rightarrow} a_1 \xRightarrow{\pi\circ\rho\circ\theta} \dots \overset{\chi}{\Rightarrow} a_n$$

Because $\pi \circ \rho \circ \theta$ is linear, we have $w(a_i \xRightarrow{\pi\circ\rho\circ\theta} b_i) = 0$. Therefore:

$$Q = \sum_{i=0}^{n-1} w(b_i \overset{\chi}{\Rightarrow} a_{i+1})$$

The weight depend only on the propagation of b_i through χ.

Trails and Affine Applications

Affine applications have no influence on the probabilities of differentials.

Proof:

- $K \in \mathrm{GF}(2)^n$ a constant;
- A a permutation matrix of $GF(2)^n$;
- $f : \mathrm{GF}(2)^n \to \mathrm{GF}(2)^n$ such as $f(x) = Ax + K$;
- Δ and Δ' two differences

$$\begin{aligned} P(\Delta \overset{f}{\Rightarrow} \Delta') > 0 &\Leftrightarrow \exists\, t, \Delta' = f(t) + f(t + \Delta) \\ &\Leftrightarrow \exists\, t, \Delta' = At + K + A(t + \Delta) + K \\ &\Leftrightarrow \Delta' = A\Delta \end{aligned}$$

Therefore the probability of a differential over an affine application is 1. $\square$

Propagation through χ

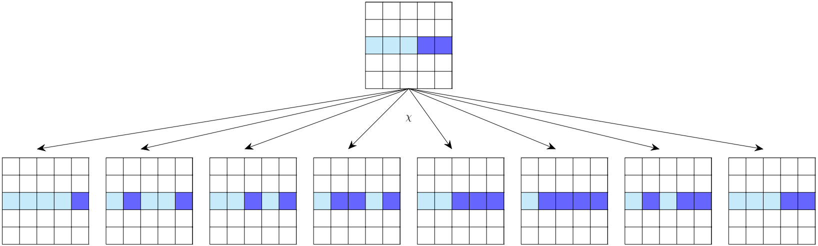


Figure 20: For a given input difference, list of possible differences after χ.

input	propagation through χ						
difference	offset	base elements				$w(.)$	$\|\| . \|\|$
00000	00000					0	0
00001	00001	00010	00100			2	1
00011	00001	00010	00100	01000		3	2
00101	00001	00010	01100	10000		3	2
10101	00001	00010	01100	10001		3	3
00111	00001	00010	00100	01000	10000	4	3
01111	00001	00011	00100	01000	10000	4	4
11111	00001	00011	00110	01100	11000	4	5

Table 1: Space of possible output differences, weight, and Hamming weight of all row differences.

Adding active bits to the state will never decrease the weight.

Orbitals

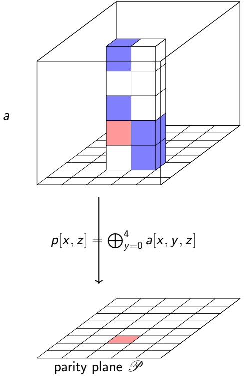


Figure 21: State and Parity.

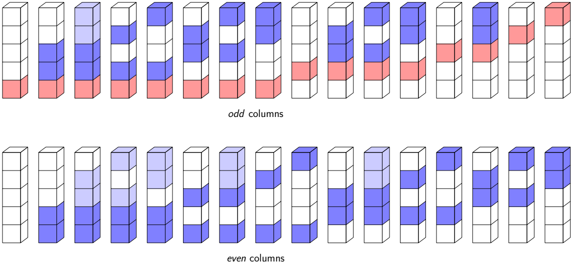


Figure 22: Orbitals, active bits are coloured.

Small-step semantics and big-step semantics

Small steps

Rules which specify from configuration c and state s, one can go to configuration c' and state s'.

$$< c, s > \longrightarrow < c', s' >$$

$$< c, s > \longrightarrow^* < \delta, \sigma >$$

Big steps

Rules which specify the entire transition from a configuration c and state s' to a final state σ

$$< c, s > \Downarrow \sigma$$

Equivalence big steps - small steps

$$< c, s > \longrightarrow^* < \delta, \sigma > \Leftrightarrow < c, s > \Downarrow \sigma$$

Tree traversal: Proof (1/6)

$$it : (\texttt{TO\_CHILD}, [\,], [\,]) \to^* (\texttt{VISITED}, [\,], \mathit{visited})$$

Provide genericity:

$$\forall\, \mathit{path}\ \mathit{pred}, it : (\texttt{TO\_CHILD}, \mathit{path}, \mathit{pred}) \to^* (\texttt{VISITED}, \mathit{path}, \mathit{visited} :: \mathit{pred})$$

By induction:

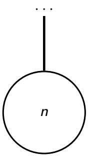


Figure 23: $getNode\ path = Some(n, [\,])$

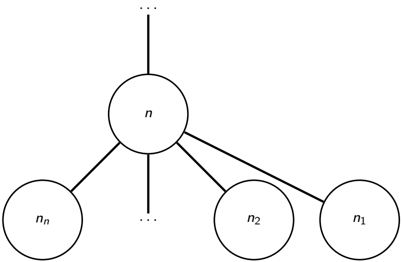


Figure 24: $getNode\ path = Some(n, l)$

Tree traversal: Proof (2/6)

Goal:

$$\forall\ path\ pred, it : (\texttt{TO\_CHILD}, path, pred) \rightarrow^* (\texttt{VISITED}, path, visited :: pred)$$

By induction:

$getNode\ path = Some(n, [\,])$ ✓

$getNode\ path = Some(n, l)$

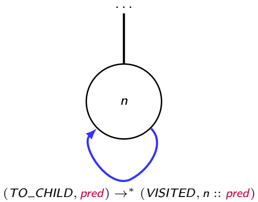


Figure 25: $getNode\ path = Some(n, [\,])$

Tree traversal: Proof (3/6)

Goal:

$$\forall\ path\ pred, it : (\texttt{TO\_CHILD}, path, pred) \rightarrow^* (\texttt{VISITED}, path, visited :: pred)$$

By induction:

$getNode\ path = Some(n, [\,])$ ✓

$getNode\ path = Some(n, l)$

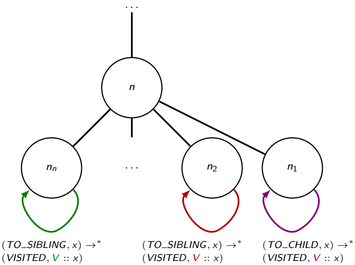


Figure 26: $getNode\ path = Some(n, l)$

Tree traversal: Proof (4/6)

Goal:

$$\forall\, path\ pred, it : (\texttt{TO\_CHILD}, path, pred) \rightarrow^* (\texttt{VISITED}, path, visited :: pred)$$

By induction:

$getNode\ path = Some(n, [\,])$ ✓

$getNode\ path = Some(n, l)$

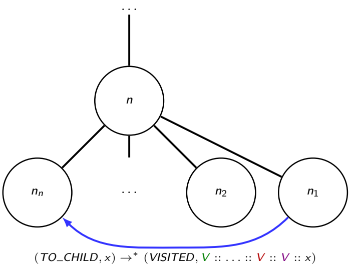


Figure 27: $getNode\ path = Some(n, l)$

Tree traversal: Proof (5/6)

Goal:

$$\forall\, path\ pred, it : (\texttt{TO\_CHILD}, path, pred) \rightarrow^* (\texttt{VISITED}, path, visited :: pred)$$

By induction:

$getNode\ path = Some(n, [\,])$ ✓

$getNode\ path = Some(n, l)$

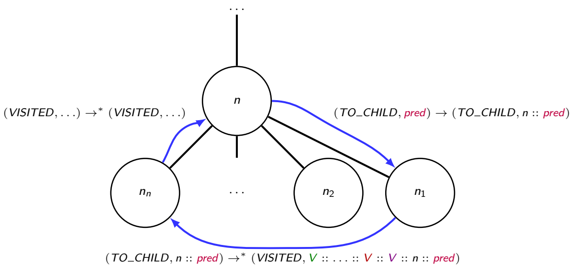


Figure 28: $getNode\ path = Some(n, l)$

Tree traversal: Proof (6/6)

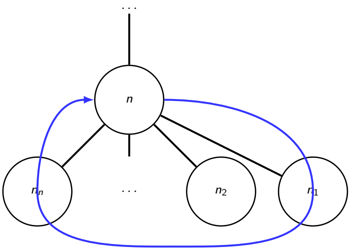
Goal:

$$\forall\, path\ pred, it : (\texttt{TO\_CHILD}, path, pred) \rightarrow^* (\texttt{VISITED}, path, visited :: pred)$$

By induction:

$getNode\ path = Some(n, [\,])$ ✓

$getNode\ path = Some(n, l)$ ✓



$(TO\_CHILD, pred) \rightarrow^* (VISITED, V :: \ldots :: V :: V :: n :: pred)$

Figure 29: $getNode\ path = Some(n, l)$