Sealing Pointer-Based Optimizations behind Pure Functions

Type Theory Presentation

Buster Bosma
(s1061831)

Michiel Philipse
(s1016359)

iCIS | Digital Security
Radboud University

Sealing Pointer-Based Optimizations behind Pure Functions

DANIEL SELSAM, Microsoft Research, USA
SIMON HUDON*, Carnegie Mellon University, USA
LEONARDO DE MOURA, Microsoft Research, USA

Functional programming languages are particularly well-suited for building automated reasoning systems, since (among other reasons) a logical term is well modeled by an inductive type, traversing a term can be implemented generically as a higher-order combinator, and backtracking search is dramatically simplified by persistent datastructures. However, existing pure functional programming languages all suffer a major

Traversing a term requires time proportional to the *tree size* of the term as opposed to its *graph size*

15

perform simple operations on the memory addresses of terms, and yet allowing these operations to be used freely would clearly violate the basic premise of referential transparency. We show how to use dependent types to seal the necessary pointer-address manipulations behind pure functional interfaces while requiring only a negligible amount of additional trust. We have implemented our approach for the upcoming version (v4) of Lean, and our approach could be adopted by other languages based on dependent type theory as well.

CCS Concepts: • **Software and its engineering → Language features**.

Additional Key Words and Phrases: functional programming, interactive theorem proving, Lean

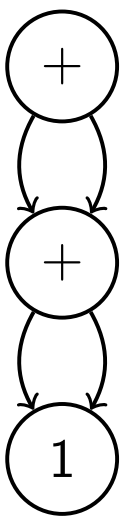
ACM Reference Format:
Daniel Selsam, Simon Hudon, and Leonardo de Moura. 2020. Sealing Pointer-Based Optimizations behind Pure Functions. *Proc. ACM Program. Lang.* 4, ICFP, Article 115 (August 2020), 20 pages. https://doi.org/10.1145/3408997

Example: Tower Terms

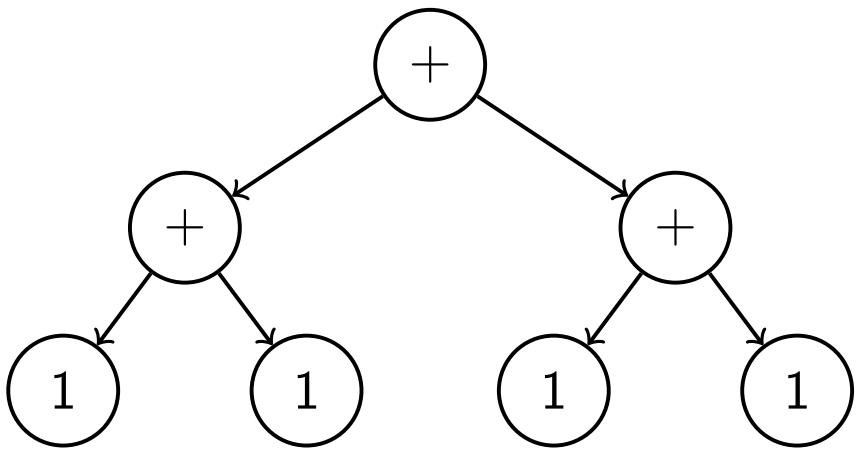
```
inductive Term : Type
| one : Term
| add : Term → Term → Term
```

```
def tower : Nat → Term
| 0   => one
| n+1 => let t := tower n; add t t
```

Graph of `tower 2`:



Tree of `tower 2`:



Overview

Pointer Equality Optimization

Traversing Terms in Linear Time with Hash-maps

Extensions

Evaluation

Overview

Pointer Equality Optimization

Traversing Terms in Linear Time with Hash-maps

Extensions

Evaluation

Pointer equality optimizations

- Reflexive binary relation: `r : α → α → Bool`
- Optimization: `if &x == &y then true else r x y`
- *Compute reflexive binary relation with pointer equality:*

```
def withPtrEq (x y : α) (k : Unit → Bool) (h : x = y → k () = true)
  : Bool := k ()
```

- This reference implementation will be replaced in low-level IR with faster but equivalent implementation:

`withPtrEq x y (λ _ => r x y) h` compiles to pseudo-code above

iCIS | Digital Security
Radboud University

Reflexive binary relation optimization

- Define *optimized* reflexive binary relation:

```
def withPtrRel (r : α → α → Bool) (h :    (x : α), r x x = true)
  : α → α → Bool := λ (x y : α) => withPtrEq x y (λ _ => r x y) h

where h := (λ (p : x = y) => p B h x)
```

Term Equality

- Standard term equality:

```
def termEqPure : Term → Term → Bool
| one, one => true
| add x1 y1, add x2 y2 => termEqPure x1 x2 && termEqPure y1 y2
| _, _ => false
```

- Optimized term equality:

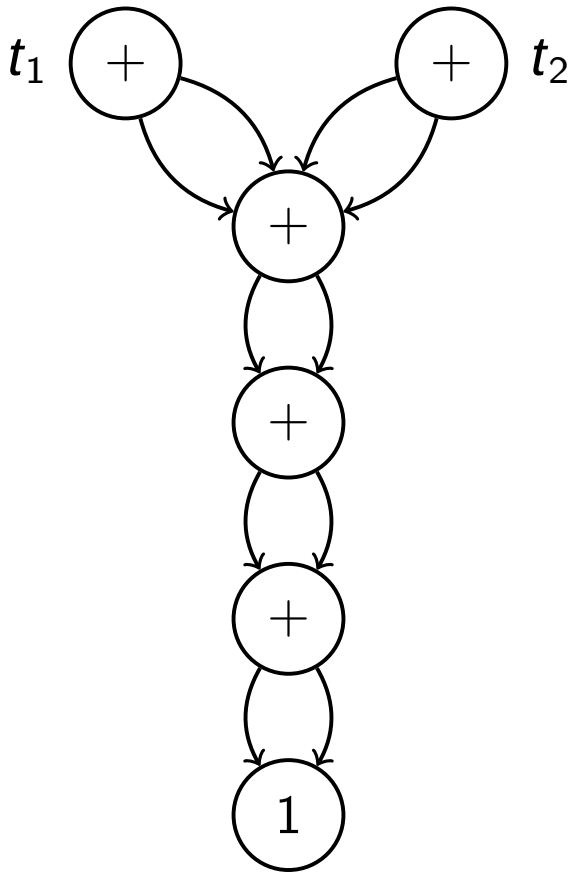
```
def termEqOneOff : Term → Term → Bool :=
      withPtrRel termEqPure termEqPureRefl
```

with a proof of reflexivity `termEqPureRefl`:

```
theorem termEqPureRefl :   (t : Term), termEqPure t t = true
```


One-o Pointer Equality Tests

Example of two terms where equality is still exponential with `termEqOneOff`:



Recursive Pointer Equality

- A `Prop` p is decidable if you have a proof (h : p) or a proof (h : ¬p):

```
inductive Decidable (p : Prop) : Type
| isTrue (h : p) : Decidable
| isFalse (h : ¬p) : Decidable
```

- Wrapper for `withPtrEq` with `Decidable`:

```
def withPtrEqDecEq (x y : α) (k : Unit → Decidable (x = y))
    : Decidable (x = y) :=
    let kb : Unit → Bool := λ _ => toBool (k ());
    let kbRfl : x = y → kb () = true := toBoolEqTrue (k ());
    let b : Bool := withPtrEq x y kb kbRfl;
    condEq b
      (λ (h : b = true) => isTrue (ofToBoolEqTrue (k ()) h))
      (λ (h : b = false) => isFalse (ofToBoolEqFalse (k ()) h))
```


Recursive Term Equality

- Recursive term equality:

```
def termDecEqAux :  (t1 t2 : Term), Decidable (t1 = t2)
| one, one => isTrue rfl
| add x1 y1, add x2 y2 =>
   match withPtrEqDecEq x1 x2 (λ _ => termDecEqAux x1 x2) with
   | isTrue h1 =>
      match withPtrEqDecEq y1 y2 (λ _ => termDecEqAux y1 y2)
      with
      | isTrue h2 => isTrue (h1 B h2 B rfl)
      | isFalse h2 => isFalse #
   | isFalse h1 => isFalse #
| one, add x y => isFalse #
| add x y, one => isFalse #
```

Recursive Term Equality (cont.)

- Extract boolean equality:

```
def termEqRec (t1 t2 : Term) : Bool := toBool (termDecEq t1 t2)
```

with:

```
def termDecEq :  (t1 t2 : Term), Decidable (t1 = t2 ) :=
λ t1 t2 => withPtrEqDecEq t1 t2 (λ _ => termDecEqAux t1 t2)
```


Overview

Pointer Equality Optimization

Traversing Terms in Linear Time with Hash-maps

Extensions

Evaluation

Intrusive Hash Codes

- Provide hash address in constructor:

```
inductive Term : Type
| one : Term
| add : Term → Term → Addr → Term
```

- Helper functions for hash addresses:

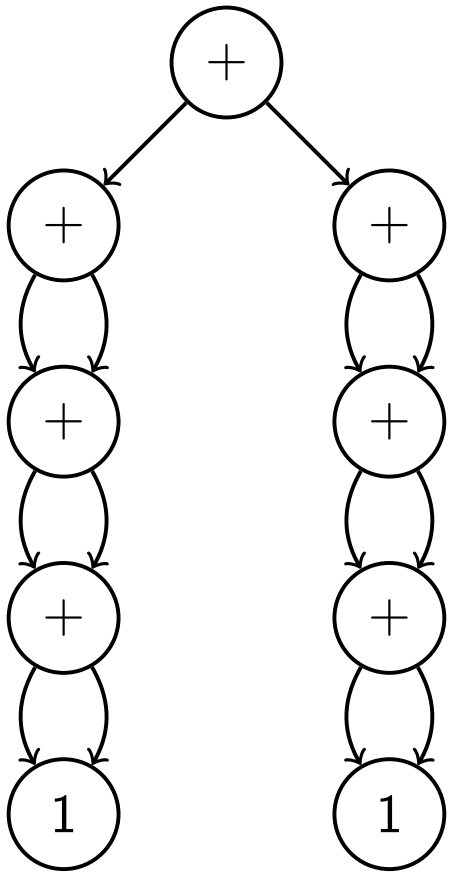
```
def add (t1 t2 : Term) : Term :=
   Term.add t1 t2 (mixHash (fastHash t1) (fastHash t2))

def fastHash : Term → Addr
| one => 7
| add t1 t2 h => h
```

Traversing Near-Perfect Towers

```
def evalNat : Term → StateM (HashMap Term Nat) Nat
  | t => do
  map ← get;
  match map.find? t with
    | some n => pure n
    | none =>
    match t with
      | one => pure 1
      | add t1 t2 hash => do
      n1 ← evalNat t1;
      n2 ← evalNat t2;
      let n := n1 + n2;
      modify (λ map => map.insert t n);
      pure n
```

Pointer-disjoint subterms



`evalNat` will still take exponential time!

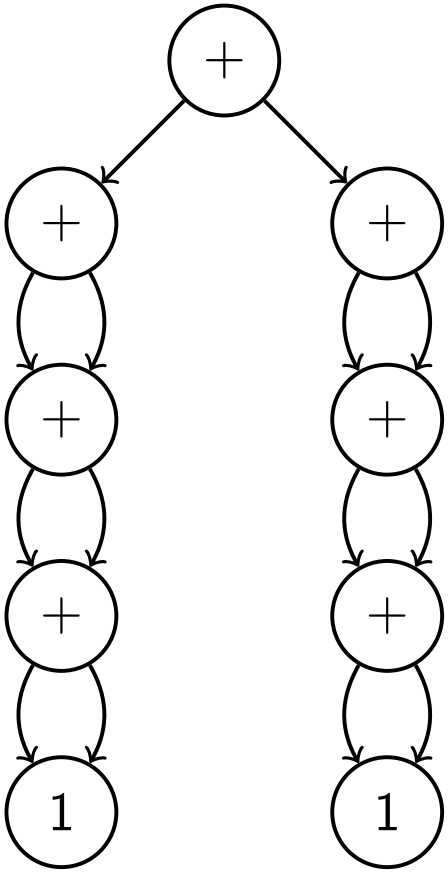
Traversing arbitrary terms

- Hash-maps still require equality checks due to *hash collisions*
- Equality checks can be optimized with recursive pointer equality
- This is only possible in linear time when the terms are *maximally shared*
- `shareCommon` can be seen as the identity function:
  `def shareCommon (x : α) : α := x`
- The compiler will return *maximally shared* versions of the inputs
- ```
  def evalNatRobust (t : Term) : Nat :=
                (evalNat (shareCommon t) HashMap.empty).1
  ```

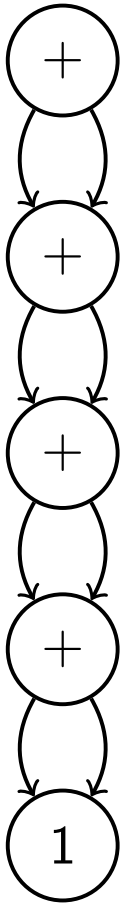


  ```

Fixing pointer-disjoint subterms



`t: Term`



`(shareCommon t): Term`

Downsides of intrusive hash codes

- Might impose an undesirable space overhead
- Require additional bookkeeping
- It can be difficult to design a good structural hash function
- It may be necessary to efficiently traverse existing terms of a type that lacks an intrusive hash

Overview

Pointer Equality Optimization

Traversing Terms in Linear Time with Hash-maps

Extensions

Evaluation

Extensions

- *Imprecise equality tests*: `withPtrEqResult`
  - Giving up rather than recursing in the absence of pointer equality
- *Pointer address hashing*: `withPtrAddr`
  - Using memory addresses directly as hash codes
- Traversing terms with *pointer address hashing*: `evalNatPtrCache`
  - Integrate the previous two extensions to create an alternative to `evalNatRobust`, without requiring intrusive hashes nor a call to the `shareCommon` primitive

Imprecise equality tests

- *Pointer equality tests*:
  - Optimization: `if &x == &y then true else r x y`

```
def withPtrEq (x y : α) (k : Unit → Bool)
          (h : x = y → k () = true) : Bool := k ()
```

- *Imprecise pointer equality tests*:
  - Optimization: `if &x == &y then k yesEqual else k unknown`

```
inductive PtrEqResult (x y : α) : Type
| unknown : PtrEqResult
| yesEqual : x = y → PtrEqResult

def withPtrEqResult [Subsingleton β] (x y : α)
        (k : PtrEqResult x y → β) : β := k unknown
```


iCIS | Digital Security
Radboud University

Subsingleton types

- A subsingleton can only have 0 or 1 inhabitants:

```
class Subsingleton (α : Type) : Prop := (h :  (x y : α), x = y)
```

- Needed to be able to trust functions that depend on pointer addresses
- Can still provide interesting results:

```
structure Result (f : α → β) (x : α) : Type :=
  (output : β) (h : output = f x)

structure Entry (f : α → β) : Type :=
  (input : α) (result : Result f input)
```


iCIS | Digital Security
Radboud University

Imprecise association list caches

```
def evalImpreciseBucket [Subsingleton γ] (x0 : α)
    (k : StateM γ (Result f x0))
    (update : Entry f → StateM γ Unit)
    : List (Entry f) → StateM γ (Result f x0)
  | [ ] => do
    r ← k;
    update (Entry.mk x0 r);
    pure r
  | (Entry.mk x r)::es =>
    withPtrEqResult x x0 (λ (pr : PtrEqResult (x = x0)) =>
      match x, pr, r with
      | _, yesEqual rfl, r => pure r
      | _, unknown _, r => evalImpreciseBucket es)
```

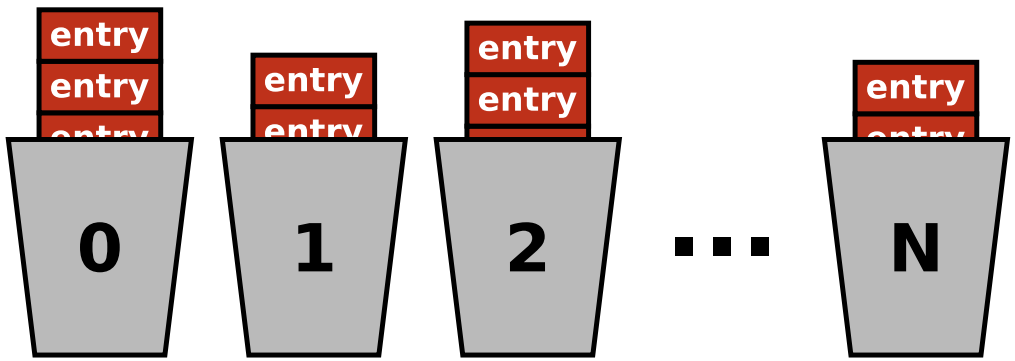
Pointer address hashing

- List lookup is slow, so we want to use hash buckets
- Subsingleton cache containing an array of buckets:

```
def PtrCache (f : α → β) : Type := Array (List (Entry f))
```

- Support direct pointer address manipulations:

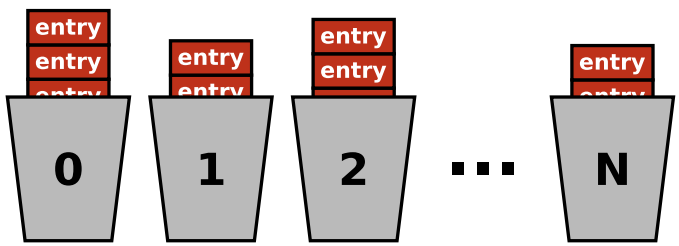
```
def withPtrAddr [Subsingleton β] (x : α) (k : Addr → β) : β := k 0
```



iCIS | Digital Security
Radboud University

Pointer address hashing (cont.)

```
def evalPtrCache (x : α) (k : StateM (PtrCache f) (Result f x))
   : StateM (PtrCache f) (Result f x) := do
  s ← get;
  withPtrAddr x (λ u =>
    Squash.lift s (λ buckets =>
      if buckets.size = 0 then k else do
        let i := u.toNat % buckets.size;
        let update (e : Entry f) : StateM (PtrCache f) Unit :=
          modifySquash (λ buckets =>
            Array.modify buckets i (λ es => e :: es));
        let es := Array.get! buckets i;
          evalImpreciseBucket x k update es))
```



iCIS | Digital Security
Radboud University

Traversing terms with pointer address hashing

```
def PtrCacheM (f : α → β) (x : α) := StateM (PtrCache f) (Result f x)

def evalNatPtrCache :   (t : Term), PtrCacheM evalNatNaive t
| one => pure (Result.mk 1 rfl)
| add t1 t2 => do
  Result.mk r1 hr1 ← evalPtrCache t1 (evalNatPtrCache t1);
  Result.mk r2 hr2 ← evalPtrCache t2 (evalNatPtrCache t2);
  let output : Nat := r1 + r2 ;
  let h : output = evalNatNaive t1 + evalNatNaive t2
                 := hr1 B hr2 B rfl;
  pure (Result.mk output h)
```

Overview

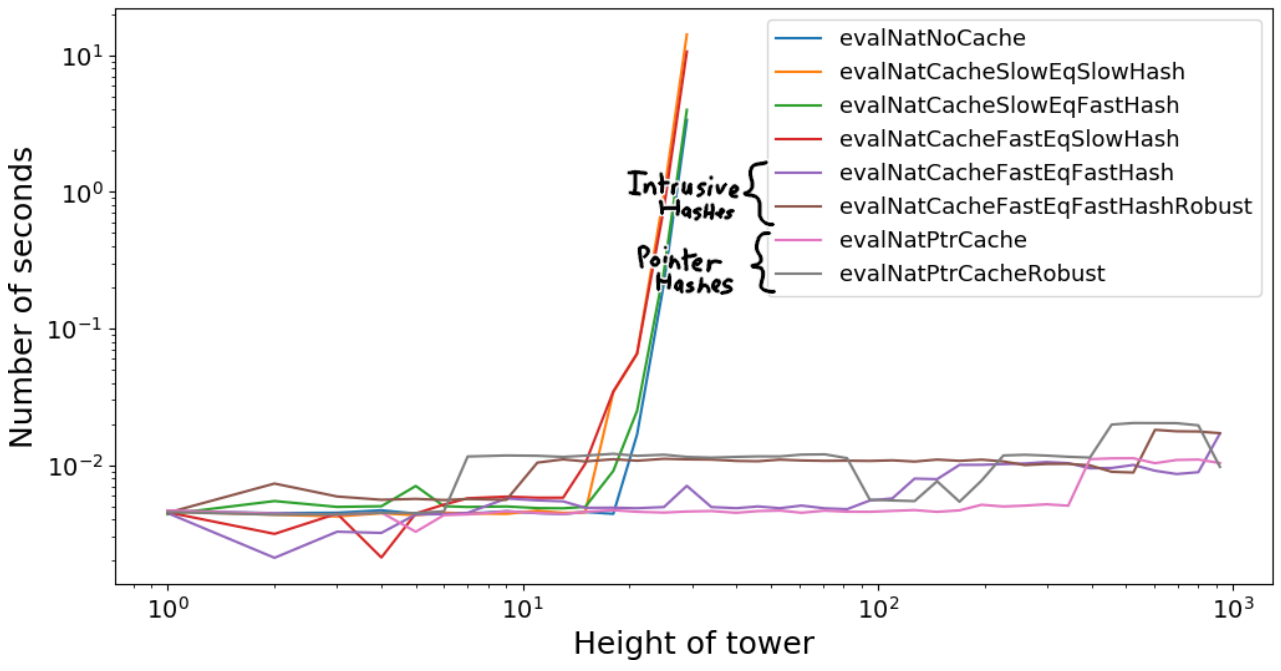
Pointer Equality Optimization

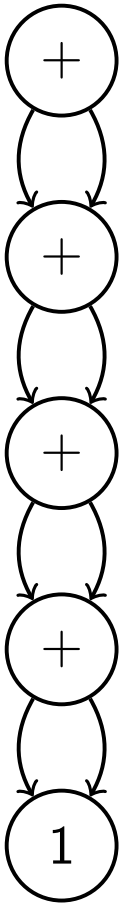
Traversing Terms in Linear Time with Hash-maps

Extensions

Evaluation

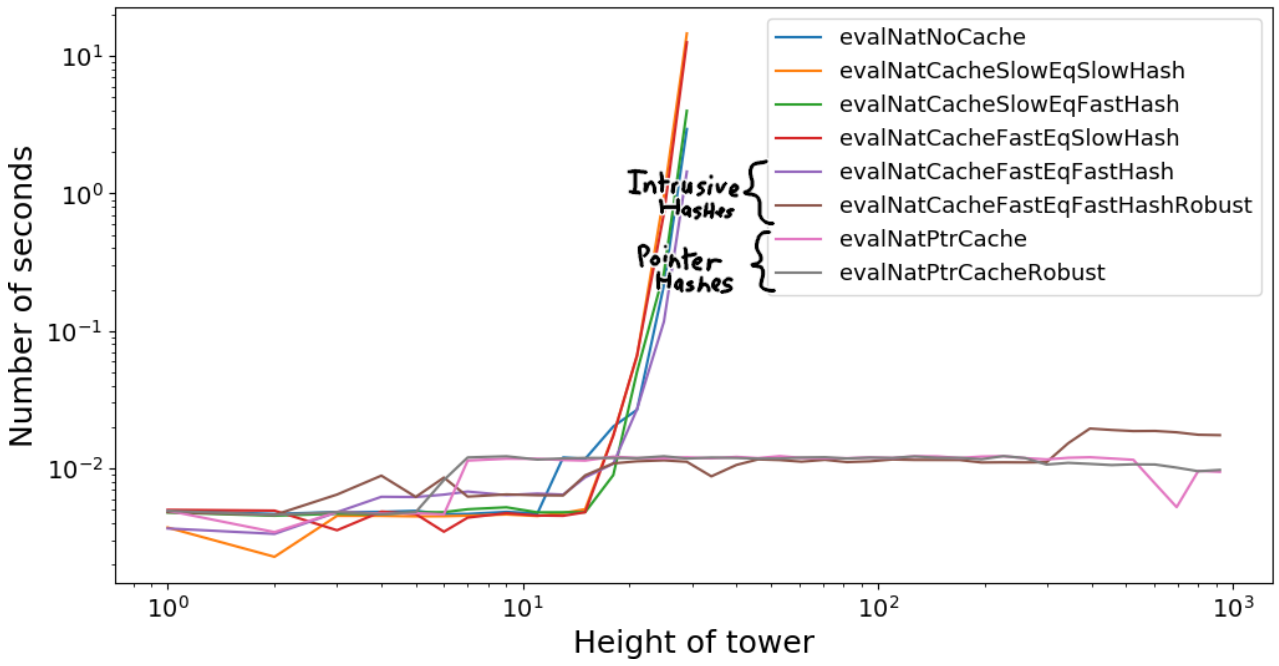
Maximally-shared term evaluation

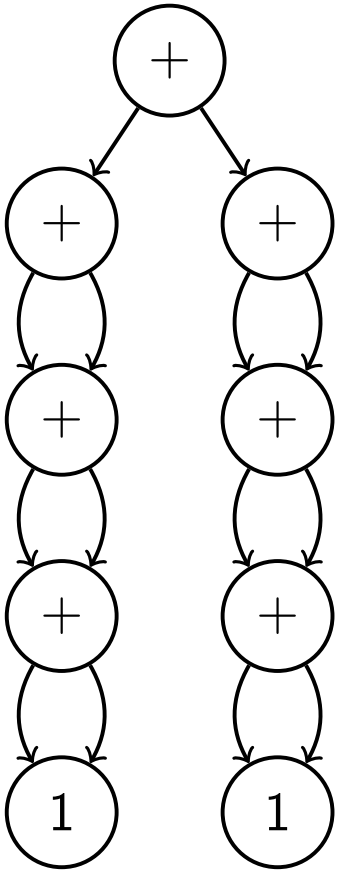




iCIS | Digital Security
Radboud University

Pointer-disjoint term evaluation





Conclusions

- Pointer equality can be used to optimize functional languages
- Reference implementations get replaced by faster equivalent implementations by compiler
- Use intrusive hashes and hash maps for efficient traversal
- Use `shareCommon` to make terms maximally shared
- Pointer addresses can only be trusted if you return subsingletons