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 limitation in these domains: traversing a term requires time proportional to the tree size of the term as opposed to its graph size. This limitation would be particularly devastating when building automation for interactive theorem provers such as Lean and Coq, for which the exponential blowup of term-tree sizes has proved to be both common and difficult to prevent. All that is needed to recover the optimal scaling is the ability to 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

1 INTRODUCTION

Functional programming languages are particularly well-suited for building automated reasoning systems for several reasons. First, a logical term is naturally represented by an inductive type, whereas such terms are notoriously awkward to encode in object-oriented languages. Second, traversing a term can be implemented generically as a higher-order combinator, and so much boilerplate control-flow code can be avoided. Third, backtracking search is dramatically simplified by the use of persistent datastructures, since branches can be paused and resumed at will. Indeed most interactive theorem provers are written in functional programming languages: Isabelle/HOL [Nipkow et al. 2002] is written in Poly/ML [Matthews 1985], Coq is written in OCaml [Leroy et al. 2018], Agda [Bove et al. 2009] and Idris [Brady 2013] are both written in Haskell [Jones 2003], and

*This paper describes work performed while author Simon Hudon was at Microsoft Research.

Authors' addresses: Daniel Selsam, daselsam@microsoft.com, Microsoft Research, One Microsoft Way, Redmond, WA, 98052, USA; Simon Hudon, simon.hudon@gmail.com, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, 15213, USA; Leonardo de Moura, leonardo@microsoft.org, Microsoft Research, One Microsoft Way, Redmond, WA, 98052, USA.

2475-1421/2020/8-ART115
https://doi.org/10.1145/3408997

Lean [de Moura et al. 2015] was written in C++ [Ellis and Stroustrup 1990] but is being rewritten in Lean itself.

Functional programming languages shine in this domain, yet to the best of our knowledge the *pure* fragments of existing functional programming languages such as Haskell [Jones 2003], Gallina [Huet 1992] (*i.e.* the language of Coq), Idris [Brady 2013], Agda [Bove et al. 2009], Miranda [Turner 1986], PureScript [Freeman 2015] and Lean all suffer a critical limitation: traversing a term requires time proportional to the tree size of the term as opposed to its graph size. This limitation is particularly devastating in automated reasoning where the basic operations can and do produce terms whose tree representations are exponentially larger than their graph representations. Even a single first-order unification can produce such explosion in principle, with the canonical example unifying $f(x_1, \ldots, x_n)$ with $f(g(x_2, x_2), \ldots, g(x_{n+1}, x_{n+1}))$ [Goubault 1994]. The problem is exacerbated when writing automation for interactive theorem provers such as Lean and Coq since terms are often the result of long chains of user-written meta-programs (*i.e.* tactics). In Lean's mathematics library, mathlib [mathlib Community 2020], despite conscious effort to avoid idioms known to cause this kind of explosion (*e.g.* those pointed out by [Garillot 2011]), there are still proofs that contain only 20,000 nodes when viewed as graphs but 2.5 billion nodes when viewed as trees.

All that is needed to traverse terms in time proportional to their graph sizes rather than their tree sizes is the ability to perform simple operations on their memory addresses. However, allowing unrestricted use of these operations would clearly violate the basic premise of referential transparency. In this work, 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. Our work is particularly relevant for building high-performance systems for automated reasoning, but the pointer-based optimizations we consider are ubiquitous in real-world software projects and may provide performance improvements in diverse domains.

We assume a dependently typed language that gets compiled to a low-level imperative IR, and our approach is based on the following insights. First, if a function is treated as an opaque primitive throughout the compilation process all the way down to the IR, the body of the function can then be replaced with a low-level imperative version. Second, since the compiler and runtime of a language are already trusted, it requires very little additional trust to assume that simple properties that the runtime relies on do indeed hold, for example that two live objects with the same memory address must be equal. Third, by making use of these assumptions one can often formulate sufficient conditions for the replacement code for a given function to be faithful to the original pure definition. These conditions can then be encoded formally using dependent types and required as additional arguments to the functions in question. Then by design every full application of the functions can be safely replaced in the IR with their low-level imperative versions.

We stress that our accelerated implementations are more than just type-safe: they are functionally indistinguishable from the pure reference implementations. Thus any theorem one proves about one's pure functional implementations holds for the accelerated version as well. 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. Complete versions of all examples in the paper are available in the supplementary material.

2 PRELIMINARIES

For our present purposes, the distinguishing feature of dependently typed programming languages is that proofs are first-class in the language. In particular, a function can take a proof as an argument, thereby ensuring that it can never be fully applied unless the corresponding precondition is satisfied. We illustrate with the classic example of returning the head of a non-empty list:

```
1  def List.head : ∀ (xs : List α) (pf : xs ≠ []), α
2  | [], (pf : [] ≠ []) => absurd rfl pf
3  | x::_, _ => x
```

In addition to the list `(xs : List α)`, the function `List.head` takes an additional argument `(pf : xs ≠ [])` constituting a proof that the list `xs` is not empty. Note that the type `xs ≠ []` *depends on* the term `xs`, hence the name *dependent types*. The function body starts by jointly pattern-matching on `xs` and `pf`. In the `[]` branch (Line 2), the type of `pf` becomes `[] ≠ []`, which contradicts the reflexivity of equality `rfl : [] = []`. The `absurd` function takes two contradictory facts as inputs and lets us produce a term of any type we wish, in this case α. Finally, in the non-empty branch (Line 3), the function ignores the proof and returns the head of the list.

To simplify the presentation, we replace almost all proofs in the paper with the symbol #—no matter how trivial the proofs may be—and relegate their details to the supplementary material. Equality-substitution proofs are an exception, and we think it improves readability to include them. We use the notation ▹ as follows: if `(x y : α) (p : x = y) (h : r x)`, then `p ▹ h` is a proof of `r y`. Note that if there were multiple occurrences of `x` in the type of `h`, the subset of occurrences to substitute would be inferred from the context.

Our presentation makes use of the *squash* type former (also known as *e.g.* the proposition truncation and the (-1)-truncation) that turns any type into a subsingleton type, *i.e.* a type with at most one element [Univalent Foundations Program 2013]. More precisely, for any type α we can form the type $\|\alpha\|$ such that for any `(x : α)`, `|x|` has type $\|\alpha\|$, and `∀ (x y : α), |x| = |y|`. If `(β : Type)` is a subsingleton, then we can *lift* a function `f : α → β` to a function `Squash.lift f : ‖α‖ → β` such that `∀ (x : α), Squash.lift f |x| = f x`. Squashing can be defined in terms of *quotient types* (see *e.g.* Altenkirch and Kaposi [2016]; Bortin and Lüth [2010]; Cohen [2013]; Hofmann [1995]; Nogin [2002]; Univalent Foundations Program [2013]), as the special case of quotienting by the trivial relation that always returns true.

In several places, we use the standard `Unit` type with one trivial constructor:

```
inductive Unit : Type
| () : Unit
```

Note that in Haskell, the `()` notation is used for both the type `Unit` and the value `() : Unit`. Our presentation is simplified by the use of the *state monad* [Wadler 1990] as is common in Haskell to weave (functional) state through computations conveniently:

```
def StateM σ α := σ → α × σ

def get : StateM σ σ := λ s => (s, s)
def set (s : σ) : StateM σ Unit := λ _ => ((), s)
def modify (f : σ → σ) : StateM σ Unit := λ s => ((), f s)
def pure (x : α) : StateM σ α := λ s => (x, s)

def bind (c₁ : StateM σ α) (c₂ : α → StateM σ β) : StateM σ β :=
λ s => let (x, s) := c₁ s; c₂ x s

def modifySquash (f : α → α) : StateM ‖α‖ Unit :=
modify (Squash.lift (λ x => |f x|))
```

We also adopt Haskell's *do* notation, so that *e.g.* `do s ← get; set (f s); pure true` is sugar for `bind get (λ s => bind (set (f s)) (λ _ => pure true))`, which itself is equivalent to `λ s => (true, f s)`.

Our presentation is also simplified by the use of typeclasses [Wadler and Blott 1989], which are structures that can be synthesized automatically by backward chaining [Selsam et al. 2020; Sozeau and Oury 2008]. A simple example is the class of types possessing a default element:

```
class HasDefault (α : Type) : Type := (default : α)
```

with example instances:

```
instance : HasDefault Nat := { default := 0 }
instance : HasDefault (Option α) := { default := none }
```

We can define a function `default` that takes a `HasDefault α` instance as an *instance-implicit* argument, indicated by square brackets:

```
def default (α : Type) [HasDefault α] : α := HasDefault.default α
```

Instance-implicit arguments do not need to be passed explicitly, and are instead synthesized automatically by typeclass resolution based on the instances that have been registered. For example, `default Nat` will return `0`, whereas `default (Option String)` will return `none`. The class of subsingletons is particularly useful in our setting:

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

Recall from above that $\|\alpha\|$ is a subsingleton for all types α. Another subsingleton that we use is the result of applying a function to a given input argument:

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

We also use the fact that products of subsingletons are subsingletons, and that functions mapping to subsingletons are themselves subsingletons. Together these imply that if α and β are both subsingletons, then so is the state monad computation `StateM α β := α → β × α`.

We also need a type to represent decidable propositions:

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

Note that since the parameter `p` is necessarily a parameter of the types returned by the constructors, it is not necessary to make this dependence explicit and we write `Decidable` rather than `Decidable p`. Equality in dependent type theory is not in general decidable; whereas `Bool` is the standard two-element datatype from traditional programming languages, `Prop` is the type of all propositions, and not every proposition has a proof or a disproof (*e.g.* by the Halting Problem). The `Decidable` typeclass lets us blur the distinction between `Prop` and `Bool` in the common case by projecting decidable propositions to booleans automatically. We can make this conversion explicit with the function `toBool : Decidable p → Bool`, which satisfies the following basic properties:

```
def toBool : Decidable p → Bool
| isTrue  _ => true
| isFalse _ => false

theorem toBoolEqTrue (d : Decidable p) (h : p) : toBool d = true
theorem ofToBoolEqTrue (d : Decidable p) (h : toBool d = true) : p
theorem ofToBoolEqFalse (d : Decidable p) (h : toBool d = false) : ¬ p
```

Note that different values of type `Decidable p` may correspond to radically different algorithms for deciding `p`. Although `Decidable` is a typeclass in Lean, for our presentation it is more convenient to always pass the `Decidable` arguments explicitly.

Lastly, we need the following helper function for branching on a boolean with access to equality proofs in both branches:

```
def condEq (b : Bool) (h₁ : b = true → β) (h₂ : b = false → β) : β
```

3 POINTER EQUALITY OPTIMIZATIONS

3.1 withPtrEq

Imperative programmers routinely use pointer equality to accelerate reflexive binary relations such as structural equality. Suppose we are evaluating a reflexive binary relation `r : α → α → Bool` on two terms `(t₁ t₂ : α)`. If `t₁` and `t₂` have the same address in memory, then they must be the same object, and hence `r t₁ t₂` can safely return true without proceeding further. However, this optimization is unsound if `r` is not actually reflexive, and confirming that an arbitrary relation is reflexive falls well beyond the capabilities of existing functional programming languages based on simple type theory. Fortunately, languages based on dependent type theory can establish such properties at compile time, and so confirm that particular uses of this trick are sound.

To support this idiom and others, we introduce the following new primitive:

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

Viewed as a pure function, it simply evaluates the thunk `k` and returns the result. We refer to this pure implementation as the function's *reference implementation*, and our goal will be to replace the reference implementation in the low-level IR with a faster but still functionally equivalent implementation. The dependently-typed argument `(h : x = y → k () = true)` represents a proof that the thunk `k` will return `true` whenever `x = y`. Thus if `withPtrEq` is ever fully applied, and if we could somehow determine that its first two arguments were equal (*e.g.* by pointer equality), we could evaluate the thunk `k` correctly by simply returning `true`.

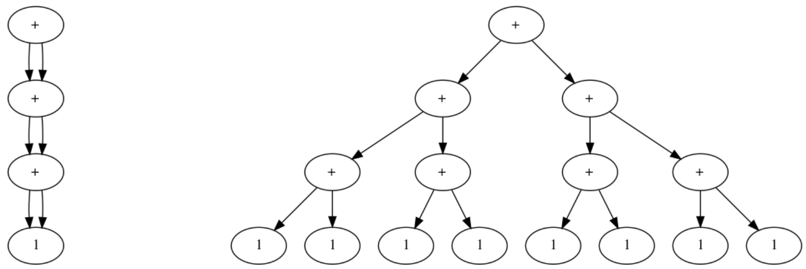
The pure reference implementation notwithstanding, the compiler can treat this definition as a new opaque primitive until reaching the low-level imperative IR, which has support for accessing the memory addresses of objects, and which already relies on the assumption that two live objects with the same memory address must be equal. Thus by chaining together this implicit assumption about the runtime with the proof `(h : x = y → k () = true)` provided as argument to `withPtrEq`, a simple meta-logical argument establishes the soundness of replacing the opaque `withPtrEq` in the IR with a version that immediately returns `true` if the addresses of `x` and `y` are equal, and evaluates the thunk if they are not. The Lean compiler ensures auxiliary closures are not allocated at runtime for the parameter `k`, and erases the proof `h`. More specifically, `withPtrEq x y (λ _ => f x y) h` will be compiled into the following low-level IR code (presented as pseudocode):

```
if ptrAddr x == ptrAddr y then true else f x y
```

The low-level IR is compiled to C in a straightforward manner, and the supplementary material shows how to inspect the exact C code generated for all examples in the paper.

The `withPtrEq` primitive can be used to accelerate the test of a reflexive binary relation. We can define a function `withPtrRel` in terms of `withPtrEq` that takes a binary relation `r` along with a proof that the relation is reflexive, and returns a pointer-equality-accelerated version whose reference implementation is identical to the reference implementation of the original 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) (λ (p : x = y) => p ▸ h x)
```



(a) Graph

(b) Tree

Fig. 1. Comparing the graph and tree representations of the term `tower 4`. In general, `tower n` has size $\Theta(n)$ as a graph but size $\Theta(2^n)$ as a tree. There is no way to traverse this term in sub-exponential time using the pure fragments of any existing languages. By sealing low-level pointer operations behind functional reference implementations, we recover the optimal $\Theta(n)$ scaling while preserving purity.

3.2 One-off Pointer Equality Tests

Even a single application of `withPtrEq` can provide exponential speedups in certain situations. Consider the following simple term language:

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

along with the following function to generate a term tower:

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

Figure 1 shows both the graph and the tree representations of `tower 4`. The relevant point is that the size of the graph is $\Theta(n)$ whereas the size of the unfolded tree is $\Theta(2^n)$. One of the main motivations of the present work is that *there is no way to traverse this term in sub-exponential time using existing pure functional languages*. There are ways to construct other kinds of entities from the bottom up that are isomorphic to this term in the presence of some additional state and that can be traversed efficiently, for example by either of the first two approaches described in [Braibant et al. 2014]. However, in existing pure languages, there is no way to efficiently traverse a term of a standard inductive type like the one above. For example, the following pure functional equality test will require $\Theta(2^n)$ time to even confirm that two pointer-identical towers of height `n` are equal:

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

Thus a single pointer equality test at the outset can provide exponential speedups on this problem (once its reflexivity has been established):

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

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

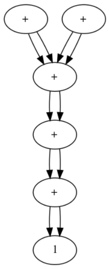


Fig. 2. Two towers that are each of the form add (tower n) (tower n), where all four towers are pointer equal but where the two outermost add operations are not. On this example, the simplistic termEqOneOff will only consider pointer equality at the respective roots, and so will take exponential despite the near-total sharing between the respective terms. We show in §3.3 how to degrade gracefully in the presence of non-pointer-identical constructors by using withPtrEq at each recursive call.

However, any deviation from perfect sharing would cause the speedups from termEqOneOff to evaporate. Figure 2 shows two towers that are each of the form add (tower n) (tower n), where all four towers are pointer equal but where the two outermost add operations are not. Then since termEqOneOff t_1 t_2 falls back on termEqPure t_1 t_2 once its two arguments are found not to be pointer equal, it will take exponential time to evaluate. In order to degrade gracefully in the presence of non-pointer-identical constructors, pointer equality must be checked at each recursive call.

3.3 Recursive Pointer Equality Tests

The primitive withPtrEq introduced above is sufficient to support recursive pointer equality tests as well, though the construction is more involved. We now show how to construct a recursively-accelerated equality test for the simple term language of §3.2. The main complication is that in order for the outer call to establish that it is reflexive, the recursive calls must return proofs that their results are reflexive. We could accomplish this by requiring a thunk that returns a subtype { b : Bool // x = y → b = true }, but since we are primarily interested in speeding up equality rather than an arbitrary reflexive relation, we accomplish the same goal using the standard Decidable type introduced in §2:

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

First, we need the following generic helper function withPtrDecEq that tries to decide x = y by passing a provided thunk to withPtrEq:

```
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))
```

Whereas withPtrEq takes a thunk returning Bool, withPtrEqDecEq takes a thunk k returning a term of type Decidable (x = y), which constitutes both a boolean value (whether x and y are equal) along with a proof that the boolean value is consistent with whether or not x and y are actually equal. First, `withPtrEqDecEq` creates a boolean thunk `kb` that can be passed to `withPtrEq`, that uses `toBool` to extract the boolean out of the `Decidable (x = y)` value returned by the thunk `k` (Line 2). It then establishes the proof obligation for `withPtrEq` using `toBoolEqTrue` (Line 3) and calls `withPtrEq` (Line 4). Finally, `condEq` is used to branch on the value of `b` (Line 5), and in each branch the proofs are lifted to terms of type `Decidable (x = y)` using basic lemmas (Lines 6-7). Note that we can pass *e.g.* `h : b = true` to `ofToBoolEqTrue (k ())` because the reference implementation of `withPtrEq` simply evaluates the thunk `k`, and so the result `b` returned by `withPtrEq` is definitionally equal to `toBool (k ())`.

Next, we can define the continuation `k` for decidable equality on `Terms`:

```
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 ▹ h₂ ▹ rfl)
    | isFalse h₂ => isFalse #
  | isFalse h₁ => isFalse #
| one,       add x y   => isFalse #
| add x y,   one       => isFalse #
```

This version is almost identical to the naive version `termEqPure`, except it calls `withPtrEqDecEq` for all recursive calls (passing itself as the continuation), and it also produces proofs in each of the branches that it is truly computing equality. Finally, we wrap this auxiliary function with a top-level pointer equality check:

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

and extract the Boolean equality test from it:

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

This construction is only a minor variation of the automatically-generated definitions already produced by pure functional languages (*e.g.* by Haskell's `deriving (Eq)`). Whereas `termEqOneOff` only provides speedups when comparing pointer-equal towers, `termEqRec` provides speedups exponential in the height of the shared pointer-equal base of two structurally equal towers, no matter how many non-pointer-equal constructors wrap the respective bases. Although this is an improvement over `termEqOneOff`, it will still take exponential time to determine that two pointer-disjoint towers of the same height are structurally equal. We revisit this scenario in §4.4.

4 TRAVERSING TERMS IN LINEAR TIME

We now show how to use the pointer equality optimizations discussed in §3 to traverse terms in linear time. As a running example, we consider the function that evaluates a `Term` into a natural number:

```
def evalNatNaive : Term → Nat
| one => 1
| add t₁ t₂ => evalNatNaive t₁ + evalNatNaive t₂
```

This section improves on the naïve version incrementally and culminates in §4.4 with the implementation of `evalNatRobust`, which scales linearly in the graph size rather than the tree size no matter the memory layout of the term.

4.1 Pure Functional Hash Maps

Pure functional hash maps—also called hash trees, hash tries, persistent hash maps, and hash array mapped tries—are a common datastructure in functional programming languages. They were introduced by Bagwell [2001] and are now part of the standard library in Lean4, Clojure [Hickey 2008] and Scala [Odersky et al. 2004]. They are also included in the `unordered-containers` package in Haskell. Finding, inserting and deleting each technically require $O(\log_B(n))$ time for a branching factor $B = 2^k$, though Bagwell [2001] simplifies this to $O(1)$ in his analysis.

Many functional languages based on reference counting—including Lean4, PVS [Owre et al. 1992], SISAL [McGraw et al. 1983], and SAC [Scholz 1994]—also support traditional hash maps that have the desired (amortized) $O(1)$ cost per operation as long as the map is not shared, *i.e.* its reference count is 1. In particular, the Lean4 standard library includes a hash map based on an array of buckets, and thanks to the optimizations described in Ullrich and de Moura [2019], the array will be updated destructively as long as the hash map is used linearly, which it is in all the examples that follow. For languages that do not support such destructive updates, the approach we now describe will allow traversing terms in either linear time or quasilinear time, depending on whether or not $O(\log_B(n))$ is considered $O(1)$.

4.2 Intrusive Hash Codes

A naive implementation of hashing a term requires a traversal and hence a single call will take exponential time on `tower n`. However, since hashing is a (pure) *unary* function of a term, we can hash terms in constant time by simply extending the `Term` type to store its hash code:

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

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

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

where `Addr` is a fixed-size numeric type that is big enough to store any pointer address. Alternatively, if there were more constructors, it may be more convenient to define a new type that packages a `Term`, a hash code, and (optionally) a proof that the stored hash code indeed agrees with the naive hash of the term:

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

def slowHash : TermNoHash → Addr

structure Term : Type :=
(term : TermNoHash) (hash : Addr) (hashOk : hash = slowHash term)
```

```
def fastHash : Term → Addr
| Term term hash hashOk => hash

def add : Term → Term → Term
| Term t1 h1 ok1, Term t2 h2 ok2 =>
  Term (TermNoHash.add t1 t2) (mixHash (fastHash t1) (fastHash t2)) #
```

We advocate intrusive hashing for most use cases, though we present an alternative that does not rely on it in §5. For the rest of §4, `Term` will refer to the first variant above, with its `add` constructor intrusively extended to include its hash code. Moreover, we assume that this field is always compared before the children inside `termEqRec`.

4.3 Traversing Near-Perfect Towers

By caching with a hash table that combines `termEqRec` with the intrusive hash code (§4.2), we can evaluate functions on both the tower of Figure 1a and the near-perfect tower of Figure 2 in (expected) linear time. For example, the following function that evaluates a `Term` as a natural number runs in linear time:[1]

```
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
```

It is not important that the function returns a scalar. On the two example terms above, this approach will scale with the graph size rather than the term size even if the function returns a new term, and even if the function recurses on (shallow) combinations of existing subterms— for example, if we add a `mul` constructor and distribute multiplication over addition with *e.g.* `distrib (mul t (add t1 t2))` reducing to `add (distrib (mul t t1)) (distrib (mul t t2))`. However, this approach will still take exponential time when traversing a term that contains two pointer-disjoint towers, like the term in Figure 3. Specifically, `evalNat` will evaluate the first tower efficiently, but then simply looking up the root of the second tower in the cache will fall back on structural equality, which in the absense of any pointer equalities will take exponential time. This limitation is similar to one alluded to at the end of §3.3. §4.4 presents the general solution that scales in the graph size rather than the term size no matter the shape of the term.

[1] Note that while Coq's trusted termination checker would accept the analogous Coq program as written, Lean requires well-founded recursion for this example and in particular generates proof obligations for the recursive calls. However, the example could be expressed (if less clearly) in terms of direct structural recursion only.

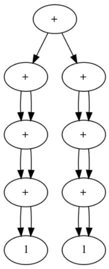


Fig. 3. A term of the form `add (tower n) (tower n)` where the two towers are pointer-disjoint. The simple caching approach of §4.3 will take exponential time on this example. §4.4 presents the general solution that scales in the graph size rather than the term size no matter the shape of the term.

4.4 Traversing Arbitrary Terms

We saw in §4.3 that as long as a term is (nearly) maximally shared, we can traverse it in linear time by caching with a hash table that uses pointer-accelerated equality and the intrusive hash. Thus, to traverse arbitrary terms in linear time, it suffices to be able to make a term maximally shared in linear time. We refer to this process as *sharing the common data* within a term, and it is akin to hash-consing after-the-fact. Terms are traversed and normalized in a bottom-up fashion while building a map from addresses of terms to addresses of equivalent but maximally-shared terms. Although the additional primitives we will introduce in §5 allow implementing such a sharing function that scales linearly in the graph size of the terms, many runtimes (including Poly/ML[2] and Lean) already include a generic, high-performance implementation of it that applies to terms of any type. Thus we can apply the same approach we took in §3.1 to seal the low-level implementation for sharing common data behind the (pure) polymorphic identity function. Specifically, we introduce a new primitive

```
def shareCommon (x : α) : α := x
```

Viewed as a pure function, it is simply the identity function, yet just as for `withPtrEq`, the compiler treats this definition as a new opaque primitive until reaching the low-level imperative IR, at which point it replaces it with a call to the runtime's `shareCommon` function. Since the correctness of the system already depends on the runtime's `shareCommon` implementation being functionally equivalent to the identity function, this transformation only requires a negligible amount of additional trust. Note that although this primitive does not require dependent types to be sealed by a pure function, it would not affect the asymptotics of traversing terms on its own without the additional ability to compare memory addresses.

By preceding the caching traversal of §4.3 with a call to `shareCommon`, we can traverse an arbitrary term `t : Term` in linear time. For example:

```
def evalNatRobust (t : Term) : Nat := (evalNat (shareCommon t) HashMap.empty).1
```

In practice, it is wasteful to apply `shareCommon` from scratch each time, since the map from addresses of terms to addresses of equivalent but maximally-shared terms will need to be rebuilt from scratch, and each term will need to be traversed again even if it was already made maximally shared by a previous `shareCommon` call. To accommodate incrementally sharing the common data across multiple terms, we introduce a new primitive type `ShareCommon.State : Type` (that wraps the map described above) and the more general `withShareCommon`:

[2] https://www.polyml.org/documentation/Reference/PolyMLStructure.html#shareCommonData. Accessed 5/30/2020.

```
def ShareCommon.State : Type := Unit -- wraps the map from addresses to addresses
def ShareCommon.State.empty : ShareCommon.State := ()
def withShareCommon (x : α) : StateM ShareCommon.State α := pure x
```

Here `withShareCommon` is a primitive that behaves like `shareCommon` above except it starts sharing the common data given the state it is passed and then returns the resulting state. Using the new primitive `withShareCommon`, we can now define the original `shareCommon` as

```
def shareCommon (x : α) : α := (withShareCommon x ShareCommon.State.empty).1
```

In Lean, `withShareCommon` satisfies the desirable property that `do x ← withShareCommon x; x ← withShareCommon x; f x` is algorithmatically equivalent to `do x ← withShareCommon x; f x`, since the second `withShareCommon` call will necessarily be a no-op. However, this property will not hold in general for languages such as OCaml and Haskell that use a moving (also known as compacting) garbage collector since objects may be moved at any time.

5 EXTENSIONS

In §4, we saw how to combine `withPtrEq`, intrusive hash codes, and a `shareCommon` primitive to traverse arbitrary terms in (expected) linear time while preserving functional equivalence with respect to a pure reference implementation. In our experience building automation for earlier versions of Lean, we found that unsafe versions of the methods in §4 yield very good performance in all our uses cases. Nonetheless, we now introduce two extensions to `withPtrEq` that may provide desirable trade-offs in certain contexts: `withPtrEqResult` of §5.1 allows giving up rather than recursing in the absence of pointer equality, while `withPtrAddr` of §5.2 allows using memory addresses directly as hash codes. We integrate these two extensions to implement `evalNatPtrCache` in §5.3, which is an alternative to `evalNatRobust` (§4.4) that can traverse terms in linear time without requiring intrusive hashes nor a call to the `shareCommon` primitive. As we will see, a notable downside of this alternative approach is that it requires a reference implementation for the function being cached. For this reason among others, we generally advocate the approach of §4.

5.1 Imprecise Equality Tests

One limitation of the approach of §4 is that even when a programmer knows that a term must be maximally shared in a particular context, it will still recurse into subterms when pointer equality fails to hold for two elements in the same hash bucket. However, this is rarely an issue in practice since it is highly unlikely that the hash codes of the subterms will also collide, and so `termEqRec` will still fail quickly. Nonetheless, we show how to apply the same methodology of §3.1 to seal an *imprecise* pointer equality test—one that gives up rather than recurses in the absence of pointer equality—behind a pure functional interface. Of course, arbitrary uses of imprecise pointer equality tests will not be sound in general. However, a use is clearly sound if the continuation returns an element of a subsingleton type (see §2), since there is only one value it could possibly return no matter how it computes the value internally. It turns out that this simple precondition is expressive enough to support our current needs.

To support imprecise equality tests, we define a new inductive type for the result of a pointer equality test:

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

and introduce a second primitive, `withPtrEqResult`:

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

This primitive differs from the original `withPtrEq` in two ways. First, rather than returning a boolean, the continuation `k` can instead return a subsingleton type β. Second, rather than taking an argument of type `Unit`, the continuation either gets no information (`unknown`) or a proof that the two elements are equal (`yesEqual (h : x = y)`). We will see shortly why this proof is necessary. The reference implementation simply evaluates the continuation `k` on `unknown`. As for `withPtrEq`, the compiler can treat this definition as a new opaque primitive until reaching the low-level imperative IR. At this point it can replace the implementation with code that first checks pointer equality, and then calls the continuation `k` on either `unknown` or `yesEqual` depending on the result. More specifically, Lean will compile `withPtrEqResult x y k` into the following low-level IR code (presented as pseudocode):

```
if ptrAddr x == ptrAddr y then k yesEqual else k unknown
```

Note that the `yesEqual` is just a constant for the runtime, as the proof itself has no runtime representation and is erased by the compiler. The soundness argument is similar to the one for `withPtrEq`. The runtime already relies on the assumption that two live objects with the same memory address must be equal. Thus, when pointer equality is detected and `k` is evaluated on `yesEqual`, `x` really does equal `y`. Moreover, since `k` returns a subsingleton, the same result (up to equality) will be returned no matter whether pointer equality is detected or not. Thus the low-level imperative version is functionally equivalent to the pure reference implementation.

Imprecise Association List Caches. We now show how to implement an imprecise association list cache for a function `f` using `withPtrEqResult`. Recall the subsingleton `Result` type from §2:

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

and define an entry of the association list to be a dependent pair of an input `x` and a `Result f x`:

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

We implement a function that looks up a `Result` in an association list of `Entry`s as follows:

```
1  def evalReadImpreciseListCacheOneOff (x0 : α) : List (Entry f) → Result f x0
2  | [] => Result.mk (f x0) rfl -- rfl is the reflexivity proof for f x0 = f x0
3  | (Entry.mk x r)::es =>
4    withPtrEqResult x x0 (λ (pr : PtrEqResult (x = x0)) =>
5      match x, pr, r with
6      | _, yesEqual rfl, r => r
7      | _, unknown _, _ => evalReadImpreciseListCacheOneOff es)
```

The three-way `match` at the end may seem strange at first to readers unfamiliar with dependent pattern matching; we will explain how it works shortly. If the list is empty, we simply evaluate `f` x_0 and return the result (Line 2). Otherwise (Line 3), we perform an imprecise pointer equality test on x_0 and the input `x` of the first entry (Line 4). The continuation then simultaneously pattern matches on `x`, the result of the pointer equality test `pr : PtrEqResult (x =` x_0`)` and the result `r : Result f x` (Line 5). In the first branch (Line 6), `pr` matches its second constructor, `yesEqual`, and the argument to `yesEqual` which would normally have type `x =` x_0 is able to reduce to the reflexivity proof `rfl :` x_0 `=` x_0 since we are matching on `x` simultaneously. In this branch, `r` has type `Result f` x_0 and so it suffices to return it. In the branch where `pr` does not contain a proof (Line 7), it simply recurses on the rest of the list. Note that there are no proof obligations besides the subsingleton requirement which is discharged by typeclass resolution.

The implementation above of evalReadImpreciseListCacheOneOff has two limitations. First, it only reads the list and does not return a new list on a cache miss. It cannot simply return the modified list in addition to the result, since withPtrEqResult requires that the return type be a subsingleton. We can address this limitation by taking an additional argument g : List (Entry f) ⟶ γ for some subsingleton γ, and returning g applied to the extended list in addition to the result. Second, it directly applies the function f on a cache miss, and cannot be made to query the pointer cache recursively on subterms. We can address this limitation by taking a continuation as an argument that itself may read and write to the cache. We present this version using the state monad StateM to simplify the notation (see §2):

```
1   def evalImpreciseBucket [Subsingleton γ] (x0 : α) (k : StateM γ (Result f x0))
2     (update : Entry f ⟶ StateM γ Unit) : List (Entry f) ⟶ StateM γ (Result f x0)
3   | [] => do
4     r ← k;
5     update (Entry.mk x0 r);
6     pure r
7   | (Entry.mk x r)::es =>
8     withPtrEqResult x x0 (λ (pr : PtrEqResult (x = x0)) =>
9       match x, pr, r with
10      | _, yesEqual rfl, r => pure r
11      | _, unknown _, r => evalImpreciseBucket es)
```

This function is the same as evalReadImpreciseListCacheOneOff except for two small changes to address the two limitations discussed in the previous paragraph. First, to allow querying the cache recursively on subterms, it calls the user-provided k to compute the result on a cache-miss (Line 4), rather than computing the function f from scratch. Second, to allow returning an updated association list, it applies the user-provided update function to the computed result, which may update the subsingleton state γ as appropriate. We will see an example use of evalImpreciseBucket in §5.2.

5.2 Pointer Address Hashing

The intrusive approach to hashing presented in §4.2 is simple and effective, and yet it may not be the best solution in all contexts. First, depending on the size of objects and the specifics of the runtime, the intrusive hash codes might impose an undesirable space overhead. Second, the intrusion imposes additional bookkeeping, both when defining the type and when proving properties about the program. Third, for some workloads it can be difficult to design a good structural hash function. Finally, in some situations it may be necessary to efficiently traverse existing terms of a type that lacks an intrusive hash, if only to convert these terms to a type that has one.

To support direct pointer address manipulations, we introduce the following new primitive:

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

where Addr is a fixed-size numeric type that is big enough to store any pointer address. The reference implementation of withPtrAddr simply calls the continuation k on the null address 0, but as usual, the compiler can treat this definition as a new opaque primitive until reaching the low-level imperative IR, at which point it can evaluate k on the actual memory address of x rather than the null address. More specifically, Lean will compile withPtrAddr x k into the following low-level IR (pseudo)code: k (ptrAddr x). Since the return type β is a subsingleton, k will return the same result no matter what address it is evaluated on. Thus the low-level version is functionally equivalent to the reference implementation.

Pointer Caches. We now show how to use `withPtrAddr` to implement a cache that uses pointer addresses as hash codes. To simplify the presentation, we will implement a simple array-based hash map, though the same approach could be used to implement a pure functional hash map as well. Resizing is also straightforward and so we omit it from our presentation. We will use `evalImpreciseBucket` (§5.1) for searching within each bucket, so that structural equality is avoided altogether. We first define a pointer cache for a function to be a squashed array of lists of entries for that function:

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

Since we squash the array in the definition of `PtrCache`, `PtrCache f` is a subsingleton type for any `f`. When we query a `PtrCache f` for a given value `(x : α)`, we will return an element of the subsingleton type `Result f x × PtrCache f`, so that we may inspect pointer addresses freely using `withPtrAddr`. Note that we are able to squash the array in the definition of `PtrCache` because we only query the `PtrCache` for subsingletons, and so the actual contents of the underlying array cannot affect the values we compute.

We implement the function as follows:

```
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 -- alt: store proof of nonempty in PtrCache type
      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))
```

As in §5.1, all of the proof obligations are reduced to establishing various types are subsingletons, and are discharged automatically by typeclass resolution.

Note that Lean uses reference counting, and so the address of an object is constant. Thus if a particular value `(x : α)` is inserted into a pointer cache, it will always be found when queried in the future. However, this invariant does not hold in languages with a moving (also known as compacting) garbage collector, and so there is a risk that a particular value `(x : α)` may be re-inserted into multiple different buckets without ever being found. Although this is only a performance risk and cannot affect referential transparency, it constitutes an additional reason for preferring the approach of §4.

5.3 Traversing Terms with Pointer Address Hashing

We now show how to use `evalPtrCache` from §5.2 to traverse a term in linear time without the intrusive hash. We consider the running example of §4 of evaluating a term into a natural number:

```
def evalNatNaive : Term → Nat
| one => 1
| add t₁ t₂ => evalNatNaive t₁ + evalNatNaive t₂
```

It is convenient to give a name to the pointer address caching monad for a function `f`:

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

Now we can implement `evalNatPtrCache` using the tools of §5.1 and §5.2 as follows:

```
1 def evalNatPtrCache : ∀ (t : Term), PtrCacheM evalNatNaive t
2 | one => pure (Result.mk 1 rfl) -- `evalNatNaive one = 1` by definition
3 | add t₁ t₂ => do
4   Result.mk r₁ hr₁ ← evalPtrCache t₁ (evalNatPtrCache t₁);
5   Result.mk r₂ hr₂ ← evalPtrCache t₂ (evalNatPtrCache t₂);
6   let output : Nat := r₁ + r₂;
7   let h : output = evalNatNaive t₁ + evalNatNaive t₂ := hr₁ ▹ hr₂ ▹ rfl;
8   pure (Result.mk output h)
```

If the term is one (Line 2), then it returns the number 1 along with a proof that `1 = evalNatNaive one`, which is `rfl` since it holds by definition. Otherwise, if the term is `add` t_1 t_2 (Line 3), it first searches for t_1 and t_2 in the pointer cache (Lines 4-5). For each child, it passes itself applied to that child as the pointer cache continuation, so if the child is not in the pointer cache, `evalNatPtrCache` will be called recursively on the child. Then it sums the resulting values together (Line 6), proves that the result is indeed faithful to `evalNatNaive` (Line 7), and bundles the output and the proof to return an element of type `Result evalNatNaive t` (Line 8). We are making use of the fact that `evalNatNaive (add` t_1 t_2`) = evalNatNaive` t_1 `+ evalNatNaive` t_2 holds by definition; this step would need to be stated and proved explicitly if using a more sophisticated reference implementation.

We note that `evalNatPtrCache` has an interesting advantage over the `evalNat` from §4.3: it will scale linearly on the example from Figure 3 without needing to preceed it by `shareCommon`, since it will effectively cache the two different towers separately.

6 EXPERIMENTS

We evaluate eight different variants of `evalNat` on two different towers. The eight variants we consider are as follows:

(1) `evalNatNoCache`: traverses recursively with no caching.
(2) `evalNatCacheSlowEqSlowHash`: caches but with naïve equality tests and naïve (unintrusive) hashing.
(3) `evalNatCacheSlowEqFastHash`: caches with the intrusive hash of §4.2 but naïve equality tests.
(4) `evalNatCacheFastEqSlowHash`: caches with the recursively-accelerated equality `termEqRec` of §3.3 but naïve unintrusive hashing.
(5) `evalNatCacheFastEqFastHash`: caches with the recursively-accelerated equality `termEqRec` of §3.3 and the intrusive hash of §4.2. This variant corresponds to the `evalNat` implementation in §4.3.
(6) `evalNatCacheFastEqFastHashRobust`: the robust version of `evalNatCacheFastEqFastHash` that first shares the common data. This variant corresponds to `evalNatRobust` in §4.4.
(7) `evalNatPtrCache`: uses the pointer cache described in §5.3, and corresponds to `evalNatPtrCache` of the same section.
(8) `evalNatPtrCacheRobust`: the same as `evalNatPtrCache` after first sharing the common data.

The first tower we consider is the simple maximally-shared tower of Figure 1. The results are shown in Figure 4, with both axes in log-scale. We see that as expected, the first four variants all take exponential time, whereas the latter four remain linear in n. The second tower we consider is that of Figure 3, which consists of two pointer-disjoint towers with a shared head. The results are shown in Figure 5, again with both axes in log-scale. In contrast to Figure 4, here we see that the `evalNatCacheFastEqFastHash` variant of §4.3 takes exponential time, for the reasons discussed in §4.3. The last three variants still take time linear in n, as expected. Code to reproduce all experiments is included in the supplementary material.

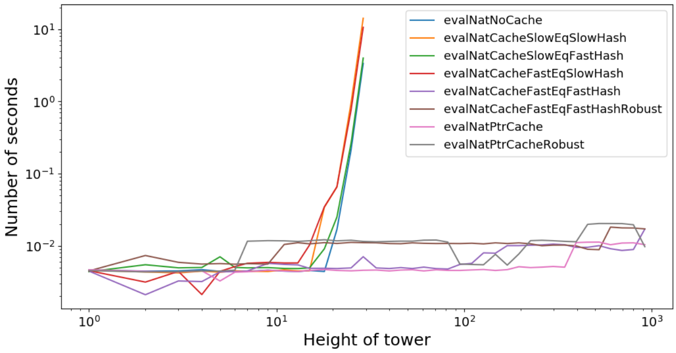


Fig. 4. Comparing eight different variants of evalNat on the simple maximally-shared tower of Figure 1. Both axes are in log-scale. We see that as expected, the first four variants (evalNatNoCache, evalNatCacheSlowEqSlowHash, evalNatCacheSlowEqFastHash, and evalNatCacheFastEqSlowHash) all take exponential time, whereas the latter four (evalNatCacheFastEqFastHash, evalNatCacheFastEqFastHashRobust, evalNatPtrCache, and evalNatPtrCacheRobust) remain linear in n.

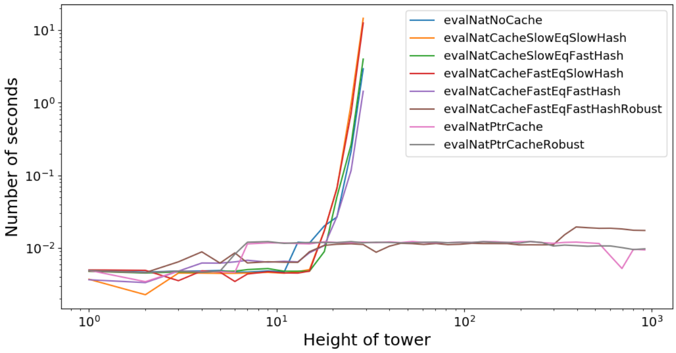


Fig. 5. Comparing eight different variants of evalNat on the tower of Figure 3 that consists of two pointer-disjoint towers with a shared head. Both axes are in log-scale. In contrast to Figure 4, here we see that the evalNatCacheFastEqFastHash variant of §4.3 takes exponential time, for the reasons discussed in §4.3. The last three variants (evalNatCacheFastEqFastHashRobust, evalNatPtrCache, and evalNatPtrCacheRobust) remain linear in n as expected.

7 DISCUSSION

A reduced-order binary decision diagram (ROBDD) is a canonical example of a datastructure that requires maintaining some kind of max-sharing invariant, *i.e.* that if two nodes in a graph are structurally equal then they must have the same unique identifier, where the identifier could be either an integer or a memory address. We note that in contrast to the problems we have considered in this paper, existing pure languages can construct ROBDDs from scratch and manipulate them without exponential blowup, *e.g.* by either of the two pure approaches used by [Braibant et al. 2014] to implement them in Gallina. The important distinction is that in existing pure languages, one can easily build ROBDDs *from the bottom up* using an explicit graph representation, whereas if you *start* with a term whose tree size is astronomically large, there is nothing you can do without the ability to compare memory addresses of subterms. It is also common practice within compilers to build and maintain compact representations of programs, e.g. with aggressive `let`-abstractions. This bottom-up style is appealing when it applies, but it is not feasible in interactive theorem provers. In contrast to compilation, where the input programs for the compiler stack are generally written explicitly by humans rather than being the output of other (meta-)programs, terms in interactive theorem provers are often the result of long chains of arbitrary, user-written meta-programs. There is no way to circumvent the need to exploit sharing in term trees without severely limiting the convenience or expressivity of the meta-programming frameworks.

In contrast to Lean which is directly compiled to C and which has its own runtime, Gallina code is generally executed by first extracting it to OCaml and then compiling the resulting OCaml program. The standard way of augmenting Gallina programs with access to impure features is to specify that particular Gallina functions should be extracted to particular (possibly impure) OCaml functions. This process is ad-hoc and unsafe in general, as the system itself cannot discern pure extraction instructions from impure ones. For example, Braibant et al. [2014] implement a naive BDD type in Gallina, extract it to an OCaml type that stores a unique identifier, extract the Gallina constructors to OCaml "smart" constructors that make use of a hash-consing library to guarantee maximal sharing, and extract the structural equality test on their BDD type to OCaml's physical (*i.e.* pointer) equality test. Thus when they execute their program, equality between BDDs is determined by comparing pointers only. However, their meta-logical soundness argument is subtle, and requires that the regular OCaml constructors are never used directly. Moreover, they give an example of a tempting smart constructor that would introduce inconsistencies between the original Gallina and extracted OCaml code. In contrast, the abstractions we have introduced can be used freely by users without any risk of impurity.

Pointer equality is a particularly delicate issue in Haskell. There are several reasons why an object may not even have the same address as itself, for example it might get duplicated during garbage collection, or it may live in two different un-evaluated thunks. In part because of these issues, checking pointer equality in Haskell is considered not only unsafe but "really" unsafe: indeed, the operation is named `reallyUnsafePtrEquality#`.[3] To support an analogue of memory addresses with more desirable properties, Jones et al. [1999] introduce the *stable name* abstraction for Haskell that allows fast equality, comparison, and hashing, and that is guaranteed to be stable over the lifetime of an object. However, creating a stable name for an object is not a pure operation, since *e.g.* the stable names of two objects might compare differently on different runs, and so the creation of stable names is still forced to be the `IO` monad.

Goubault [1994] proposed a runtime system for a functional language that would hash-cons all values to ensure maximal sharing at all times. The language could then have built-in support for datastructures such as maps that use memory addresses for ordering and equality. However, despite the promising empirical results reported in the paper, there is a general consensus that hash-consing is slow and wasteful on many workloads, especially for functional programming where it is particularly common to produce many transient objects. We also remark that several functional programming languages including Lean4, PVS [Owre et al. 1992], SISAL [McGraw et al. 1983], and SAC [Scholz 1994] have support for transforming functional array updates into destructive ones using reference counts, and hash-consing arrays would introduce undesired sharing and so prevent destructive updates from being applied. Hash-consing arrays is also inefficient in general, since the cost is linear in the size of the array.

[3] https://downloads.haskell.org/~ghc/8.8.2/docs/html/libraries/ghc-prim-0.5.3/GHC-Prim.html. Accessed 2/21/2020.

Lastly, the ACL2 theorem prover [Kaufmann and Moore 1997] is based on a subset of Common Lisp [Steele 1990] and includes structural equality as a primitive. Thus the runtime may choose to (recursively) accelerate it by short-circuiting in the pointer-equal case. Boyer and Hunt Jr [2006] also introduced a hash-consing framework for ACL2 akin to an opt-in version of that proposed by [Goubault 1994], with a new primitive `hons` that is a hash-consing version of the standard `cons` and a new primitive `hons-equal` that is like `equal` but that may use the hash-cons table to accelerate the check. Whereas ACL2 builds a fixed set of pointer-based optimizations into the language, our approach allows users to safely implement their own pointer-based optimizations on their own datatypes.

8 CONCLUSION

We have presented a new way to use dependent types to seal many pointer-based optimizations behind pure functional interfaces while requiring only a negligible amount of additional trust. We introduced primitives for conducting pointer equality tests (`withPtrEq` and `withPtrEqResult`), for sharing the common data across terms of arbitrary types (`withShareCommon`), and for directly observing pointer addresses (`withPtrAddr`). In all cases, the low-level imperative implementations of these primitives are functionally indistinguishable from their pure reference implementations. We also showed how to use these new primitives to achieve exponential speedups when traversing heavily-shared terms. We believe our work constitutes a significant step towards making pure functional programming a viable option for building high-performance systems for automated reasoning.

ACKNOWLEDGMENTS

We thank Sebastian Ullrich and Ryan Krueger for helpful feedback on early drafts.

REFERENCES

Thorsten Altenkirch and Ambrus Kaposi. 2016. Type theory in type theory using quotient inductive types. *ACM SIGPLAN Notices* 51, 1 (2016), 18–29.

Phil Bagwell. 2001. *Ideal hash trees.* Technical Report.

Maksym Bortin and Christoph Lüth. 2010. Structured Formal Development with Quotient Types in Isabelle/HOL. In *Intelligent Computer Mathematics*, Serge Autexier, Jacques Calmet, David Delahaye, Patrick D. F. Ion, Laurence Rideau, Renaud Rioboo, and Alan P. Sexton (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 34–48.

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 73–78.

Robert S Boyer and Warren A Hunt Jr. 2006. Function memoization and unique object representation for ACL2 functions. In *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*. 81–89.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming* 23, 5 (2013), 552–593.

Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. 2014. Implementing and reasoning about hash-consed data structures in Coq. *Journal of automated reasoning* 53, 3 (2014), 271–304.

Cyril Cohen. 2013. Pragmatic Quotient Types in Coq. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Springer, 213–228. https://doi.org/10.1007/978-3-642-39634-2_17

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388.

Margaret A Ellis and Bjarne Stroustrup. 1990. *The annotated C++ reference manual.* Addison-Wesley.

Phil Freeman. 2015. PureScript.

François Garillot. 2011. *Generic Proof Tools and Finite Group Theory.* Ph.D. Dissertation.

Jean Goubault. 1994. Implementing functional languages with fast equality, sets and maps: an exercise in hash consing. *Journées Francophones des Langages Applicatifs (JFLA'93)* (1994), 222–238.

Rich Hickey. 2008. The Clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*. 1–1.

Martin Hofmann. 1995. Extensional concepts in intensional type theory. (1995).

Gérard Huet. 1992. The Gallina specification language: A case study. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 229–240.

Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Simon Peyton Jones, Simon Marlow, and Conal Elliott. 1999. Stretching the storage manager: weak pointers and stable names in Haskell. In *Symposium on Implementation and Application of Functional Languages*. Springer, 37–58.

Matt Kaufmann and J. Strother Moore. 1997. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering* 23, 4 (1997), 203–213.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2018. The OCaml system release 4.07: Documentation and user's manual. (2018).

The mathlib Community. 2020. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 367–381. https://doi.org/10.1145/3372885.3373824

David CJ Matthews. 1985. Poly manual. *ACM SIGPLAN Notices* 20, 9 (1985), 52–76.

James McGraw, Stephen Skedzielewski, Stephen Allan, D Grit, R Oldehoeft, J Glauert, I Dobes, and P Hohensee. 1983. *SISAL: streams and iteration in a single-assignment language. Language reference manual, Version 1.* Technical Report. Lawrence Livermore National Lab., CA (USA).

Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.

Aleksey Nogin. 2002. Quotient Types: A Modular Approach. In *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics*, Victor Carreño, César Muñoz, and Sofiène Tashar (Eds.). Springer-Verlag, 263–280. Available at http://nogin.org/papers/quotients.html.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. *An overview of the Scala programming language*. Technical Report.

Sam Owre, John M Rushby, and Natarajan Shankar. 1992. PVS: A prototype verification system. In *International Conference on Automated Deduction*. Springer, 748–752.

Sven-Bodo Scholz. 1994. Single Assignment C - Functional Programming Using Imperative Style. In *In John Glauert (Ed.): Proceedings of the 6th International Workshop on the Implementation of Functional Languages. University of East Anglia.*

Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. 2020. Tabled Typeclass Resolution. *arXiv preprint arXiv:2001.04301* (2020).

Matthieu Sozeau and Nicolas Oury. 2008. First-class type classes. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 278–293.

Guy Steele. 1990. *Common LISP: the language*. Elsevier.

David Turner. 1986. An overview of Miranda. *ACM Sigplan Notices* 21, 12 (1986), 158–166.

Sebastian Ullrich and Leonardo de Moura. 2019. Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming. *arXiv preprint arXiv:1908.05647* (2019).

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

Philip Wadler. 1990. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*. 61–78.

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 60–76.