Sealing Pointer-Based Optimizations Behind Pure Functions

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

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 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 [de Moura et al. 2015] and Coq [Coq Development Team 2019], 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.

1 INTRODUCTION

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 is dramatically simplified by persistent datastructures. 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 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.

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; Simon Hudon, simon.hudon@gmail.com, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, 15213; Leonardo de Moura, leonardo@microsoft.org, Microsoft Research, One Microsoft Way, Redmond, WA, 98052.
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 trans-
parency. 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 addi-
tional 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 opaque 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 function-
ally 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:

```lean
1 def List.head : \forall \alpha (xs : List \alpha) (pf : xs \neq []) \alpha
2 | [], (pf : [] \neq []) => absurd rfl pf
3 | x::_, _ => x
```

In addition to the list \( (xs : List \alpha) \), the function \texttt{List.head} takes an additional argument \( (pf : \text{xs} \neq []) \) constituting a proof that the list \texttt{xs} is not empty. Note that the type \texttt{xs} \neq [] \texttt{depends on} the term \texttt{xs}, hence the name \texttt{dependent types}. The function body starts by jointly pattern-matching
on \texttt{xs} and \texttt{pf}. In the [] branch (Line 2), the type of \texttt{pf} becomes [], which contradicts the
reflexivity of equality \texttt{rfl : [] = []}. The \texttt{absurd} function takes two contradictory facts as inputs
and lets us produce a term of any type we wish, in this case \( \alpha \). 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 that we think improves readability to include: if
\((x y : \alpha) (p : x = y) (h : r x)\), then \(p \Rightarrow h\) 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 $\alpha$ we can form the type $\|\alpha\|$ such that for any $(x : \alpha)$, $|x|$ has type $\|\alpha\|$, and $\forall (x y : \alpha), |x| = |y|$. If $\beta$ is a subsingleton, then we can lift a function $f : \alpha \to \beta$ to a function $\text{Squash}.\text{lift} f : \|\alpha\| \to \beta$ such that $\forall (x : \alpha)$, $\text{Squash}.\text{lift} f |x| = f x$. 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.

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:

```haskell
def StateM \(\sigma\) \(\alpha\) := \(\sigma\) \to \(\alpha \times \sigma\)
def get : StateM \(\sigma\) \(\sigma\) := \(\lambda s \Rightarrow (s, s)\)
def set (s : \(\sigma\)) : StateM Unit := \(\lambda _ \Rightarrow ((, s)\)
def modify (f : \(\sigma\) \to \(\sigma\)) : StateM Unit := \(\lambda s \Rightarrow ((, f s)\)
def pure (x : \(\alpha\)) : StateM \(\sigma\) \(\alpha\) := \(\lambda s \Rightarrow (x, s)\)

def bind (c_1 : StateM \(\sigma\) \(\alpha\)) (c_2 : \(\alpha\) \to StateM \(\sigma\) \(\beta\)) : StateM \(\sigma\) \(\beta\) :=
\(\lambda s \Rightarrow let (x, s) := c_1 s; c_2 x s\)

We also adopt Haskell’s *do* notation, so that e.g. do $s \leftarrow \text{get}; \text{set} (f s); \text{pure} \text{true}$ is sugar for $\lambda s \Rightarrow \text{bind} (\text{set} (f s)) (\lambda _ \Rightarrow \text{pure} \text{true})$, which itself is equivalent to $\lambda s \Rightarrow (\text{true}, f s)$.

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:

```haskell
class HasDefault \(\alpha\) := Type := (default : \(\alpha\))
with example instances:

instance : HasDefault Nat := \{ \text{default} := 0 \}
instance : HasDefault (Option \(\alpha\)) := \{ \text{default} := \text{none} \}
```
We can define a function `default` that takes a `HasDefault \(\alpha\)` instance as an *instance-implicit* argument, indicated by square brackets:

```haskell
def default \(\alpha\) [HasDefault \(\alpha\)] := HasDefault.default \(\alpha\)
```
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:

```haskell
class Subsingleton \(\alpha\) := Prop := (h : \(\forall (x y : \alpha), x = y\)
```
Recall from above that $\|\alpha\|$ is a subsingleton for all types $\alpha$. Another subsingleton that we use is the result of applying a function to a given input argument:

```haskell
structure Result \(f\) \(\alpha\) \(\beta\) \(x\) := (output : \(\beta\)) (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 $\alpha$ and $\beta$ are both subsingletons, then so is the state monad computation $\text{StateM} \alpha \beta := \alpha \to \beta \times \alpha$.

We also need a type to represent decidable propositions:

```lean
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:

```lean
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:

```lean
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 $r : \alpha \to \alpha \to \text{Bool}$ on two terms $(t_1, t_2 : \alpha)$. If $t_1$ and $t_2$ have the same address in memory, then they must be the same object, and hence $r \ t_1 \ t_2$ 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:

```lean
def withPtrEq (x y : \alpha) (k : Unit \to \text{Bool}) (h : x = y \to k () = true) : \text{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 $(h : x = y \to k () = true)$ 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, \(\text{withPtrEq } x\ y \ (λ\_ \Rightarrow f\ x\ y)\ h\) will be compiled into the following low-level IR code (presented as pseudocode):

\[
\text{if ptrAddr } x == \text{ptrAddr } y \text{ then true else f x y}
\]

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:

\[
def \text{withPtrRel} (r : \alpha → \alpha → \text{Bool}) (h : \forall (x : \alpha), r x x = \text{true}) : \alpha → \alpha → \text{Bool} := \\
λ (x y : \alpha) \Rightarrow \text{withPtrEq} x y (λ\_ \Rightarrow r x y) (λ (p : x = y) \Rightarrow p \triangleright h x)
\]

### 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:

\[
\text{inductive } \text{Term} : \text{Type} \\
| \text{one} : \text{Term} \\
| \text{add} : \text{Term} → \text{Term} → \text{Term}
\]

along with the following function to generate a term tower:

\[
def \text{tower} : \text{Nat} → \text{Term} \\
| 0 \Rightarrow \text{one} \\
| n+1 \Rightarrow \text{let } t := \text{tower } n; \text{add } t t
\]

Figure 1 shows both the graph and the tree representations of \(\text{tower } 4\). The relevant point is that the size of the graph is \(Θ(n)\) whereas the size of the unfolded tree is \(Θ(2^n)\). 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. For example, the following pure functional equality test will require \(Θ(2^n)\) time to even confirm that two pointer-identical towers of height \(n\) are equal:

\[
def \text{termEqPure} : \text{Term} → \text{Term} → \text{Bool} \\
| \text{one}, \text{one} \Rightarrow \text{true} \\
| \text{add } x_1 y_1, \text{add } x_2 y_2 \Rightarrow \text{termEqPure } x_1 x_2 \&\& \text{termEqPure } y_1 y_2 \\
| _, _, \Rightarrow \text{false}
\]

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

\[
def \text{termEqPureRef1} : \forall (t : \text{Term}), \text{termEqPure } t\ t = \text{true} \\
| \text{one} \Rightarrow \text{refl} \\
| \text{add } t_1 t_2 \Rightarrow
\]
In general, \( \text{tower } n \) 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.

For two towers that are each of the form \( \text{add} \ (\text{tower } n) \ (\text{tower } n) \), where all four towers are pointer equal but where the two outermost add operations are not. On this example, the simplistic \( \text{termEqOneOff} \) 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 \( \text{withPtrEq} \) at each recursive call.

```
let h₁ : \text{termEqPure}_1 t₁ t₁ = true := \text{termEqPure}_1 t₁;
let h₂ : \text{termEqPure}_2 t₂ t₂ = true := \text{termEqPure}_2 t₂;
show (\text{termEqPure}_1 t₁ t₁ \&\& \text{termEqPure}_2 t₂ t₂) = true
  from h₁.symm \& h₂.symm \& rfl
```

def \text{termEqOneOff} : \text{Term} \to \text{Term} \to \text{Bool} := \text{withPtrRel} \text{termEqPure}_1 \text{termEqPure}_2

However, any deviation from perfect sharing would cause the speedups from \( \text{termEqOneOff} \) to evaporate. Figure 2 shows two towers that are each of the form \( \text{add} \ (\text{tower } n) \ (\text{tower } n) \), where all four towers are pointer equal but where the two outermost add operations are not. Then since \( \text{termEqOneOff}_1 t₁ t₁ \) falls back on \( \text{termEqPure}_1 t₁ t₁ \) 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 \( \text{withPtrEq} \) 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 example is simplified considerably
by bundling the boolean equality test with a proof of its correctness using the `Decidable` type introduced in §2. First, we need the following generic helper function `withPtrEqDecEq` that tries to decide \( x = y \) by passing a provided thunk to `withPtrEq`:

```lean
1 def withPtrEqDecEq (x y : α) (k : Unit → Decidable (x = y)) : Decidable (x = y) :=
2 let kb : Unit → Bool := λ_ => toBool (k ());
3 let kbRfl : x = y → kb () = true := toBoolEqTrue (k ());
4 let b : Bool := withPtrEq x y kb kbRfl;
5 condEq b
6 (λ h : b = true) => isTrue (ofToBoolEqTrue (k ()) h))
7 (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 ()) h))` 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`:

```lean
1 def termDecEqAux : \( t_1 t_2 : \text{Term} \), Decidable (t_1 = t_2)
2 | one, one => isTrue rfl
3 | add x_1 y_1, add x_2 y_2 =>
4 | match withPtrEqDecEq x_1 x_2 (λ _ => termDecEqAux x_1 x_2) with
5 | | isTrue h_1 =>
6 | | | match withPtrEqDecEq y_1 y_2 (λ _ => termDecEqAux y_1 y_2) with
7 | | | | isTrue h_2 => isTrue (h_1 ▷ h_2 ▷ rfl)
8 | | | | isFalse h_2 => isFalse #
9 | | | | ifFalse h_1 => isFalse #
10 | | add x_1 y_1 => isFalse #
11 | | add x_2 y_2 => 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:

```lean
1 def termDecEq : \( t_1 t_2 : \text{Term} \), Decidable (t_1 = t_2) :=
2 \lambda t_1 t_2 \Rightarrow withPtrEqDecEq t_1 t_2 (λ _ => termDecEqAux t_1 t_2)
```

and extract the Boolean equality test from it:

```lean
1 def termEqRec (t_1 t_2 : \text{Term}) : \text{Bool} := toBool (termDecEq t_1 t_2)
```

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 prove that two 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.

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, or alternatively by defining a new type that packages a `Term`, a hash code, and a proof that the code indeed agrees with the naive hash of the term. We advocate this approach in general despite any downsides, though we present an alternative that does not rely on it in §5. Unless mentioned otherwise, we assume from now on that `Term` has been intrusively extended to include its hash code, and that this field is always compared before the children inside `termEqRec`. Note that to avoid clutter, we do not show the extra field when pattern matching.

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:

```haskell
def evalNat : Term → StateM (HashMap Term Nat) Nat
| t => do
  map ← get;
  match map.find? t with
    Note that as written this function requires well-founded recursion and in particular generates proof obligations for the recursive calls, but it could also be written (though less clearly) in terms of structural recursion only.
Fig. 3. A term of the form \( \text{add} \left( \text{tower } n \right) \left( \text{tower } n \right) \) 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.

\[
\begin{align*}
| \text{some } n & \Rightarrow \text{pure } n \\
| \text{none} & \Rightarrow \\
& \text{match } t \text{ with} \\
| \text{one} & \Rightarrow \text{pure } 1 \\
| \text{add } t_1 \text{ } t_2 & \Rightarrow \text{do} \\
& n_1 \leftarrow \text{evalNat } t_1; \quad n_2 \leftarrow \text{evalNat } t_2; \\
& \text{let } n := n_1 + n_2; \\
& \text{modify } (\lambda \text{ map } \Rightarrow \text{map.insert } t \text{ } n); \\
& \text{pure } n
\end{align*}
\]

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 \( \text{mul} \) constructor and distribute multiplication over addition with e.g. \( \text{distrib} \left( \text{mul } t \left( \text{add } t_1 \text{ } t_2 \right) \right) \) reducing to \( \text{add} \left( \text{distrib } \left( \text{mul } t \text{ } t_1 \right) \right) \left( \text{distrib } \left( \text{mul } t \text{ } t_2 \right) \right) \). However, this approach will still take exponential time when traversing a term that contains two pointer-disjoint towers as shown in Figure 3. 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.

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. Although the additional primitives we will introduce in §5 allow implementing such a sharing function with the right asymptotics, many runtimes (including Poly/ML and Lean) already include a generic, high-performance implementation of it that applies to terms of any type and that scales linearly in the graph size of the terms. 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

\[
\begin{align*}
\text{def shareCommon } (x : \alpha ) & : \alpha := x
\end{align*}
\]

Viewed as a pure function, it is simply the identity function, yet just as for \( \text{withPtrEq} \), 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:

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

In practice, it is wasteful to apply `shareCommon` from scratch each time. To accommodate incrementally sharing the common data across multiple terms, we introduce a new primitive type `ShareCommon.State : Type` and the more general `withShareCommon`:

```lean
def ShareCommon.State : Type := Unit
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

```lean
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 equivalent to `do x ← withShareCommon x; f x`. 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. We have found this to be a satisfactory solution for all use cases we have encountered in practice while implementing automation for Lean in Lean itself. 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. As we will see, a notable downside of both extensions is that they require a reference implementation for the functions 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 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, 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:

\[
\text{inductive \textit{PtrEqResult} (x y : \alpha) : Type} \\
\text{\hspace{1em} | unknown : \textit{PtrEqResult}} \\
\text{\hspace{1em} | yesEqual : x = y \rightarrow \textit{PtrEqResult}}
\]

and introduce a second primitive, \textit{withPtrEqResult}:

\[
\text{def withPtrEqResult \text{[Subsingleton \beta]} (x y : \alpha) (k : \textit{PtrEqResult x y \rightarrow \beta}) : \beta := k \, \text{unknown}}
\]

This primitive differs from the original \textit{withPtrEq} in two ways. First, rather than returning a boolean, the continuation \(k\) can return any subsingleton type \(\beta\). Second, rather than taking an argument of type \textit{Unit}, the continuation either gets no information (\text{unknown}) or a proof that the two elements are equal (\text{yesEqual} \(h : x = y\)). We will see shortly why this proof is necessary. The reference implementation simply evaluates the continuation \(k\) on \text{unknown}. As for \textit{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 \text{unknown} or \text{yesEqual} depending on the result. More specifically, Lean will compile \textit{withPtrEqResult} \(x y k\) into the following low-level IR code (presented as pseudocode):

\[
\text{if ptrAddr x == ptrAddr y then } k \text{ yesEqual else } k \text{ unknown}
\]

Note that the \text{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 \textit{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 \text{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 \textit{withPtrEqResult}. Define an entry of the list to be a pair of an input \(x\) and a \(\text{Result f x}\) (see §2):

\[
\text{structure Entry (f : \alpha \rightarrow \beta) : Type := (input : \alpha) (result : Result f input)}
\]

Here is the implementation:

1. \[
\text{def evalReadImpreciseListCacheOneOff (x}_0 : \alpha) : \text{List (Entry f)} \rightarrow \text{Result f x}_0
\]
2. \[
\text{| [] => Result.mk (f x}_0 \text{) rfl -- rfl is the reflexivity proof of type f x}_0 = f x}_0
\]
3. \[
\text{| (Entry.mk x r)::es =>}
\]
4. \[
\text{withPtrEqResult x x}_0 \text{ (\lambda (pr : \text{PtrEqResult (x = x}_0)) =>}
\]
5. \[
\text{match x, pr, r with}
\]
6. \[
\text{| _, yesEqual rfl, r => r}
\]
7. \[
\text{| _, unknown _, _ => evalReadImpreciseListCacheOneOff es}
\]

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 \(pr : \text{PtrEqResult (x = x}_0\text{)}\) and the result \(r : \text{Result f x}\) (Line 5). In the first branch (Line 6), the continuation finds a proof that \(x = x_0\), which since \(x\) is being matched on as well, becomes the
reflexivity proof of $x_0 = x_0$. In this branch, $r$ has type $\text{Result } f \ x_0$ and so it suffices to return it. In the branch where $\text{pr}$ 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 $\text{evalReadImpreciseListCacheOneOff}$ 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 $\text{withPtrEqResult}$ requires that the return type be a subsingleton. We can address this limitation by taking an additional argument $g : \text{List } (\text{Entry } f) \rightarrow \gamma$ for some subsingleton $\gamma$, 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 $\text{StateM}$ to simplify the notation (see §2):

```haskell
def evalImpreciseBucketAux [Subsingleton $\gamma$] (x_0 : $\alpha$) (k : $\text{StateM } \gamma$ (Result f x_0))
  (update : Entry f $\rightarrow$ $\text{StateM } \gamma$ Unit) : List (Entry f) $\rightarrow$ $\text{StateM } \gamma$ (Result f x_0)
| [] => do
  r ← k;
  update (Result.mk x_0 r);
  pure r
| (Entry.mk x r)::es =>
  withPtrEqResult x x_0 ($\lambda$ (pr : PtrEqResult (x = x_0)) =>
    match x, pr, r with
    | x, yesEqual rfl, r => pure r
    | x, unknown _, r => evalImpreciseBucketAux es)
```

```haskell
def evalImpreciseBucket (x : $\alpha$) (k : $\text{StateM } ||\text{List } (\text{Entry } f)||$ (Result f x))
  : $\text{StateM } ||\text{List } (\text{Entry } f)||$ (Result f x) := do
b ← get;
Squash.lift b
  ($\lambda$ es => evalImpreciseBucketAux x k
    ($\lambda$ x => modifySquash ($\lambda$ xs => x :: xs)) es)
```

### 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 some 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:

```haskell
def withPtrAddr [Subsingleton $\beta$] (x : $\alpha$) (k : Addr $\rightarrow$ $\beta$) : $\beta$ := k 0
```

where $\text{Addr}$ is a fixed-size numeric type that is big enough to store any pointer address. The reference implementation of $\text{withPtrAddr}$ 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 \( \text{withPtrAddr} \ x \ k \) into the following low-level IR (pseudo)code: \( k \ (\text{ptrAddr} \ x) \). Since the return type \( \beta \) 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 \( \text{withPtrAddr} \) 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 \( \text{evalImpreciseBucket} \) 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:

```lean
def PtrCache (f : \( \alpha \to \beta \)) : Type := ||Array (List (Entry f))||
```

As for \( \text{evalInsertImpreciseListCache} \), when we query a \( \text{PtrCache} \ f \) for a given value \( (x : \alpha) \), we will return an element of the subsingleton type \( \text{Result} \ f \ x \times \text{PtrCache} \ f \), so that we may inspect pointer addresses freely using \( \text{withPtrAddr} \). The function itself is relatively straightforward:

```lean
def evalPtrCache (x : \( \alpha \)) (k : StateM (PtrCache f) (Result f x)) : StateM (PtrCache f) (Result f x) := do s ← get;
withPtrAddr x (\( \lambda \ u \to \)
Squash.lift s (\( \lambda \) 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 (\( \lambda \) buckets => Array.modify buckets i (\( \lambda \) es => e :: es));
        let es := Array.get! buckets i;
        evalImpreciseBucketAux 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 \( (x : \alpha) \) 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 \( (x : \alpha) \) 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 \( \text{evalPtrCache} \) from §5.2 to traverse a term in linear time without the intrusive hash. As our example, we will implement an alternative version of \( \text{evalNat} \) (see §4.3).

Before we can even state the type of the new version, we need to provide a reference implementation. Although we could use our previous \( \text{evalNat} \) for this role, in general one will want to implement a naive version to simplify the proof obligations:

```lean
def evalNatNaive : Term \to \text{Nat}
| one => 1
| add t1 t2 => evalNatNaive t1 + evalNatNaive t2
```
The reference version will never be executed and so its performance is irrelevant. It is also convenient to give a name to the pointer address caching monad for a function $f$:

```haskell
def PtrCacheM (f : $\alpha \rightarrow \beta$) (x : $\alpha$) := StateM (PtrCache f) (Result f x)
```

Now we can implement `evalNatPtrCache` as follows:

```haskell
1 def evalNatPtrCache : $\forall$ (t : Term), PtrCacheM evalNatNaive t
2 1 | one => pure (Result.mk 1 rfl) -- `1 = evalNatNaive one` by definition
3 1 | add t1 t2 => do
4 2 Result.mk r1 hr1 ← evalPtrCache t1 (evalNatPtrCache t1);
5 2 Result.mk r2 hr2 ← evalPtrCache t2 (evalNatPtrCache t2);
6 1 let output : Nat := r1 + r2;
7 1 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 $1 = \text{evalNatNaive one}$, which is `rfl` since it holds by definition. Otherwise, if the term is `add t1 t2` (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 t1 t2) = evalNatNaive t1 + evalNatNaive t2` 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. Experiments comparing many variations of `evalNat` on different term graphs can be found in the supplementary material.

6 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 \textit{from the bottom up} using an explicit graph representation, whereas if you \textit{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 \texttt{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#. 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.

Lastly, 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 it the cost is linear in the size of the array.

7 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.

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

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

Coq Development Team. 2019. *The Coq reference manual: Release 8.9.1*. INRIA.

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.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vuillon. 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 Glaubert, 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 Stemrnan, 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 Glaubert (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.

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.