Type Checking in Lean 4

This document exists to help readers better understand Lean's kernel, clarify the trust assumptions involved in using Lean, and serve as a resource for those who wish to write their own external type checkers for Lean's kernel language.

Foreword

The author would like to thank:

Mario Carneiro

Leonardo de Moura

Sebastian Ullrich

Gabriel Ebner

Jonathan Protzenko

The Lean Zulip community.

Each and every contributor to Lean4, Mathlib, and the broader Lean ecosystem.

This book is dedicated to the memory of Philip Elliot Bailey *(31 January, 1987 - 9 December, 2023)*.

What is the kernel?

The kernel is an implementation of Lean's logic in software; a computer program with the minimum amount of machinery required to construct elements of Lean's logical language and check those elements for correctness. The major components are:

- A sort of names used for addressing.
- A sort of universe levels.
- A sort of expressions (lambdas, variables, etc.)
- A sort of declarations (axioms, definitions, theorems, inductive types, etc.)
- Environments, which are maps of names to declarations.
- Functionality for manipulating expressions. For example bound variable substitution and substitution of universe parameters.
- Core operations used in type checking, including type inference, reduction, and definitional equality checking.
- Functionality for manipulating and checking inductive type declarations. For example, generating a type's recursors (elimination rules), and checking whether a type's constructors agree with the type's specification.
- Optional kernel extensions which permit the operations above to be performed on nat and string literals.

The purpose of isolating a small kernel and requiring Lean definitions to be translated to a minimal kernel language is to increase the trustworthiness of the proof system. Lean's design allows users to interact with a full-featured proof assistant which offers nice things like robust metaprogramming, rich editor support, and extensible syntax, while also permitting extraction of constructed proof terms into a form that can be verified without having to trust the correctness of the code that implements the higher level features that makes Lean (the proof assistant) productive and pleasant to use.

In section 1.2.3 of the *Certified Programming with Dependent Types*, Adam Chlipala defines what is sometimes referred to as the de Bruijn criterion, or de Bruijn principle.

> Proof assistants satisfy the “de Bruijn criterion” when they produce proof terms in small kernel languages, even when they use complicated and extensible procedures to seek out proofs in the first place. These core languages have feature complexity on par

> with what you find in proposals for formal foundations for mathematics (e.g., ZF set theory). To believe a proof, we can ignore the possibility of bugs during search and just rely on a (relatively small) proof-checking kernel that we apply to the result of the search.

Lean's kernel is small enough that developers can write their own implementation and independently check proofs in Lean by using an exporter[1]. Lean's export format contains enough information about the exported declarations that users can optionally restrict their implementation to certain subsets of the full kernel. For example, users interested in the core functionality of inference, reduction, and definitional equality may opt out of implementing the functionality for checking inductive specifications.

In addition to the list of items above, external type checkers will also need a parser for Lean's export format, and a pretty printer, for input and output respectively. The parser and pretty printer are not part of the kernel, but they are important if one wants to have interesting interactions with the kernel.

[1] Writing your own type checker is not an afternoon project, but it is well within the realm of what is achievable for citizen scientists.

Trust

A big part of Lean's value proposition is the ability to construct mathematical proofs, including proofs about program correctness. A common question from users is how much trust, and in what exactly, is involved in trusting Lean.

An answer to this question has two parts: what users need to trust in order to trust proofs in Lean, and what users need to trust in order to trust executable programs obtained by compiling a Lean program.

Concretely, the distinction is that proofs (which includes statements about programs) and uncompiled programs can be expressed directly in Lean's kernel language and checked by an implementation of the kernel. They do not need to be compiled to an executable, therefore the trust is limited to whatever implementation of the kernel they're being checked with, and the Lean compiler does not become part of the trusted code base.

Trusting the correctness of compiled Lean programs requires trust in Lean's compiler, which is separate from the kernel and is not part of Lean's core logic. There is a distinction between trusting *statements about programs* in Lean, and trusting *programs produced by the Lean compiler*. Statements about Lean programs are proofs, and fall into the category that only requires trust in the kernel. Trusting that proofs about a program *extend to the behavior of a compiled program* brings the compiler into the trusted code base.

NOTE: Tactics and other metaprograms, even tactics that are compiled, do *not* need to be trusted *at all*; they are untrusted code which is used to produce kernel terms for use by something else. A proposition `P` can be proved in Lean using an arbitrarily complex compiled metaprogram without expanding the trusted code base beyond the kernel, because the metaprogram is required to produce a proof expressed in Lean's kernel language.

- These statements hold for proofs that are exported. To satisfy more ~~pedantic~~ vigilant readers, this does necessarily entail some degree of trust in, for example, the operating system on the computer used to run the exporter and verifier, the hardware, etc.
- For proofs that are not exported, users are additionally trusting the elements of Lean outside the kernel (the elaborator, parser, etc.).

A more itemized list

A more itemized description of the trust involved in Lean 4 comes from a post by Mario

Carneiro on the Lean Zulip.

In general:

1. You trust that the lean logic is sound (author's note: this would include any kernel extensions, like those for Nat and String)
2. If you didn't prove the program correct, you trust that the elaborator has converted your input into the lean expression denoting the program you expect.
3. If you did prove the program correct, you trust that the proofs about the program have been checked (use external checkers to eliminate this)
4. You trust that the hardware / firmware / OS software running all of these things didn't break or lie to you
5. (When running the program) You trust that the hardware / firmware / OS software faithfully executes the program according to spec and there are no debuggers or magnets on the hard drive or cosmic rays messing with your output

For compiled executables:

6. You trust that any compiler overrides (extern / implemented_by) do not violate the lean logic (i.e. the model matches the implementation)
7. You trust the lean compiler (which lowered the lean code to C) to preserve the semantics of the program
8. You trust clang / LLVM to convert the C program into an executable with the same semantics

---

The first set of points applies to both proofs and compiled executables, while the second set applies specifically to compiled executable programs.

Trust for external checkers

1. You're still trusting Lean's logic is sound.
2. You're trusting that the developers of the external checker properly implemented the program.

3. You're trusting the implementing language's compiler or interpreter. If you run multiple external checkers, you can think of them as circles in a venn diagram; you're trusting that the part where the circles intersect is free of soundness issues.

4. For the Nat and String kernel extensions, you're probably trusting a bignum library and the UTF-8 string type of the implementing language.

The advantages of using external checkers are:

- Users can check their results with something that is completely disjoint from the Lean ecosystem, and is not dependent on any parts of Lean's code base.

- External checkers can be written to take advantage of mature compilers or interpreters.

- For kernel extensions, users can cross-check the results of multiple bignum/string implementations.

- Using the export feature is the only way to get out of trusting the parts of Lean outside the kernel, so there's a benefit to doing this even if the export file is checked by something like lean4lean. Users worried about fallout from misuse of Lean's metaprogramming features are therefore encouraged to use the export feature.

Unsafe declarations

Lean's vernacular allows users to write declarations marked as `unsafe`, which are permitted to do things that are normally forbidden. For example, Lean permits the following definition:

```
unsafe def y : Nat := y
```

Unsafe declarations are not exported[1], do not need to be trusted, and (for the record) are not permitted in proofs, even in the vernacular. Permitting unsafe declarations in the vernacular is still beneficial for Lean users, because it gives users more freedom when writing code that is used to produce proofs but doesn't have to be a proof in and of itself.

The aesop library provides us with an excellent real world example. Aesop is an automation framework; it helps users generate proofs. At some point in development, the authors of aesop felt that the best way to express a certain part of their system was with a mutually defined inductive type, seen here. It just so happens that this set of inductive type has an invalid occurrence of one of the types being declared within Lean's theory, and would not be permitted by Lean's kernel, so it needs to be marked `unsafe`.

Permitting this definition as an `unsafe` declaration is still a win-win. The Aesop developers were able to use Lean to write their library the way they wanted, in Lean, without having to call out to (and learn) a separate metaprogramming DSL, they didn't have to jump through hoops to satisfy the kernel, and users of aesop can still export and verify the proofs produced *by* aesop without having to verify aesop itself.

[1] There's technically nothing preventing an unsafe declaration from being put in an export file (especially since the exporter is not a trusted component), but checks run by the kernel will prevent unsafe declarations from being added to the environment if they are actually unsafe. A properly implemented type checker would throw an error if it received an export file declaring the aesop library code described above.

Adversarial inputs

A topic that often accompanies the more general trust question is Lean's robustness against adversarial inputs.

A correct type checker will restrict the input it receives to the rules of Lean's type system under whatever axioms the operator allows. If the operator restricts the permitted axioms to the three "official" ones (`propext` , `Quot.sound` , `Classical.choice`), an input file should not be able to offer a proof of the prelude's `False` which is accepted by the type checker under any circumstances.

However, a minimal type checker will not actively protect against inputs which provide Lean declarations that are logically sound, but are designed to fool a human operator. For example, redefining deep dependencies an adversary knows will not be examined by a referee, or introducing unicode lookalikes to produce a pretty printer output that conceals modification of key definitions.

The idea that "a user might think a theorem has been formally proved, while in fact he or she is misled about what it is that the system has actually done" is addressed by the idea of Pollack consistency and is explored in this publication[1] by Freek Wiedijk.

Note that there is nothing in principle preventing developers from writing software or extending a type checker to provide protection against such attacks, it's just not captured by the minimal functionality required by the kernel. However, the extent to which Lean's users have embraced its powerful custom syntax and macro systems may pose some challenges for those interested in improving the story here. Readers should consider this somewhat of an open issue for future work

[1] Freek Wiedijk. Pollack-inconsistency. Electronic Notes in Theoretical Computer Science, 285:85–100, 2012

Export format

An exporter is a program which emits Lean declarations using the kernel language, for consumption by external type checkers. Producing an export file is a complete exit from the Lean ecosystem; the data in the file can be checked with entirely external software, and the exporter itself is not a trusted component. Rather than inspecting the export file itself to see whether the declarations were exported as the developer intended, the exported declarations are checked by the external checker, and are displayed back to the user by a pretty printer, which produces output far more readable than the export file. Readers can (and are encouraged to) write their own external checkers for Lean export files.

The official exporter is lean4export.

The master branch of the official exporter uses the same base format as lean 3 here, with the addition of the new-to-lean4 items, which are projections, literals, and explicitly support for let expressions. The new stuff is outlined in the lean4export readme.

A slightly modified version of the export format supported by this fork of lean4export is described below. These modifications export reducibility hints, quotient declarations, recursors, and iota reduction rules (rec rules). These additional exports were added to give more flexibility in implementation and for performance, but they also allow for the development of more minimized software used for experimentation, and can make bootstrapping and testing a checker easier for developers.

There are also ongoing discussions about how best to evolve the export format.

(ver 0.1.2)

For clarity, some of the compound items are decorated here with a name, for example `(name : T)`, but they appear in the export file as just an element of `T`.

The export scheme for mutual and nested inductives is as follows:

- `Inductive.inductiveNames` contains the names of all types in the `mutual .. end` block. The names of any other inductive types used in a nested (but not mutual) construction will not be included.
- `Inductive.constructorNames` contains the names of all constructors for THAT inductive type, and no others (no constructors of the other types in a mutual block, and no constructors from any nested construction).

NOTE: readers writing their own parsers and/or checkers should initialize names[0] as the anonymous name, and levels[0] as universe zero, as they are not emitted by the exporter, but are expected to occupy the name and level indices for 0.

```
File ::= ExportFormatVersion Item*

ExportFormatVersion ::= nat '.' nat '.' nat

Item ::= Name | Universe | Expr | RecRule | Declaration

Declaration ::=
    | Axiom
    | Quotient
    | Definition
    | Theorem
    | Inductive
    | Constructor
    | Recursor

nidx, uidx, eidx, ridx ::= nat

Name ::=
  | nidx "#NS" nidx string
  | nidx "#NI" nidx nat

Universe ::=
  | uidx "#US"  uidx
  | uidx "#UM"  uidx uidx
  | uidx "#UIM" uidx uidx
  | uidx "#UP"  nidx

Expr ::=
  | eidx "#EV"  nat
  | eidx "#ES"  uidx
  | eidx "#EC"  nidx uidx*
  | eidx "#EA"  eidx eidx
  | eidx "#EL"  Info nidx eidx eidx
  | eidx "#EP"  Info nidx eidx eidx
  | eidx "#EZ"  Info nidx eidx eidx eidx
  | eidx "#EJ"  nidx nat eidx
  | eidx "#ELN" nat
  | eidx "#ELS" (hexhex)*
  -- metadata node w/o extensions
  | eidx "#EM" mptr eidx

Info ::= "#BD" | "#BI" | "#BS" | "#BC"

Hint ::= "O" | "A" | "R" nat

RecRule ::= ridx "#RR" (ctorName : nidx) (nFields : nat) (val : eidx)

Axiom ::= "#AX" (name : nidx) (type : eidx) (uparams : uidx*)

Def ::= "#DEF" (name : nidx) (type : eidx) (value : eidx) (hint : Hint)
(uparams : uidx*)
```

```
Theorem ::= "#THM" (name : nidx) (type : eidx) (value : eidx) (uparams: uidx*)

Quotient ::= "#QUOT" (name : nidx) (type : eidx) (uparams : uidx*)

Inductive ::=
  "#IND"
  (name : nidx)
  (type : eidx)
  (isRecursive: 0 | 1)
  (isNested : 0 | 1)
  (numParams: nat)
  (numIndices: nat)
  (numInductives: nat)
  (inductiveNames: nidx {numInductives})
  (numConstructors : nat)
  (constructorNames : nidx {numConstructors})
  (uparams: uidx*)

Constructor ::=
  "#CTOR"
  (name : nidx)
  (type : eidx)
  (parentInductive : nidx)
  (constructorIndex : nat)
  (numParams : nat)
  (numFields : nat)
  (uparams: uidx*)

Recursor ::=
  "#REC"
  (name : nidx)
  (type : eidx)
  (numInductives : nat)
  (inductiveNames: nidx {numInductives})
  (numParams : nat)
  (numIndices : nat)
  (numMotives : nat)
  (numMinors : nat)
  (numRules : nat)
  (recRules : ridx {numRules})
  (k : 1 | 0)
  (uparams : uidx*)
```

Kernel concepts

This chapter begins with a high level road map, which everyone should read, then covers some ideas needed to understand the kernel's nuts and bolts, and is less focused on theory. The ideas in the later sections do not need to be completely understood up-front, so readers should feel free to skim them and come back as needed.

The big picture

To give the reader a road map, the entire procedure of checking an export file consists of these steps:

- Parse an export file, yielding a collection of components for each primitive sort: names, levels, and expressions, as well as a collection of declarations.
- The collection of parsed declarations represents an environment, which is a mapping from each declaration's name to the declaration itself; these are the actual targets of the type checking process.
- For each declaration in the environment, the kernel requires that the declaration is not already declared in the environment, has no duplicate universe parameters, that the declaration's type is actually a type and not a value (that `infer declar.ty` returns an expression `Sort <n>`), and that the declaration's type has no free variables.
- For definitions, theorems, and opaque declarations, assert that inferring the type of the definition's value yields an expression which is definitionally equal to the type the user assigned to the declaration. This is where the rubber meets the road in terms of asserting that proofs are correct, and for theorems, this is the step that corresponds to "the user says this is a proof of `P`, does the value actually constitute a valid proof of `P`".
- For inductive declarations, their constructors, and recursors, check that they are properly formed and comply with the rules of Lean's type theory (more on this later).
- If the export file includes the primitive declarations for quotient types, ensure those declarations have the correct types, and that the `Eq` type exists, and is defined properly (since quotients rely on equality).
- Finally, pretty print any declarations requested by the user, so they can check that the declarations checked match the declarations they exported.

Clarifying language

Type, Sort, Prop

`Prop` refers to `Sort 0`

`Type n` refers to `Sort (n+1)`

`Sort n` is how these are actually represented in the kernel, and can always be used.

The reason why `Type <N>` and `Prop` are sometimes used instead of always adhering to `Sort n` is that elements of `Type <N>` have certain important qualities and behaviors that are not observed by those of `Prop` and vice versa.

Example: elements of `Prop` can be considered for definitional proof irrelevance, while elements of `Type _` can use large elimination without needing to satisfy other tests.

Level/universe and Sort

The terms "level" and "universe" are basically synonymous; they refer to the same kind of kernel object.

A small distinction that's sometimes made is that "universe parameter" may be implicitly restricted to levels that are variables/parameters. This is because "universe parameters" refers to the set of levels that parameterize a Lean declaration, which can only be identifiers, and are therefore restricted to identifiers. If this doesn't mean anything to you, don't worry about it for now. As an example, a Lean declaration may begin with `def Foo.{u} ..` meaning "a definition parameterized by the universe variable `u` ", but it may not begin with `def Foo.{1} ..` , because `1` is an explicit level, and not a parameter.

On the other hand, `Sort _` is an expression that represents a level.

Value vs. type

Expressions can be either values or types. Readers are probably familiar with the idea that `Nat.zero` is a value, while `Nat` is a type. An expression `e` is a value or "value level

expression" if `infer e != Sort _`. An expression `e` is a type or "type level expression" if `infer(e) = Sort _`.

Parameters vs. indices

The distinction between a parameter and index comes into play when dealing with inductive types. Roughly speaking, elements of a telescope that come before the colon in a declaration are parameters, and elements that come after the colon are indices:

```
       parameter ----v          v---- index
inductive Nat.le (n : Nat) : Nat → Prop
```

The distinction is non-negligible within the kernel, because parameters are fixed within a declaration, while indices are not.

Instantiation and abstraction

Instantiation refers to substitution of bound variables for the appropriate arguments. Abstraction refers to replacement of free variables with the appropriate bound variable when replacing binders. Lean's kernel uses deBruijn indices for bound variables and unique identifiers for free variables.

For our purposes, a free variable is a variable in an expression that refers to a binder which has been "opened", and is no longer immediately available to us, so we replace the corresponding bound variable with a free variable that has some information about the binder we're leaving behind.

To illustrate, let's say we have some lambda expression `(fun (x : A) => <body>)` and we're doing type inference. Type inference has to traverse into the `<body>` part of the expression, which may contain a bound variable that refers to `x`. When we traverse into the body, we can either add `x` to some stateful context of binders and take the whole stateful context into the body with us, or we can temporarily replace all of the bound variables that refer to `x` with a free variable, allowing us to traverse into the body without having to carry any additional context.

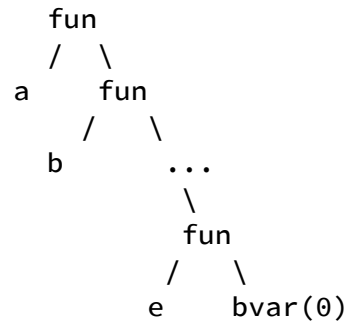
If we eventually come back to where we were before we opened the binder, abstraction allows us to replace all of the free variables that were once bound variables referring to `x` with new bound variables that again refer to `x`, with the correct deBruijn indices.

Implementing free variable abstraction

For deBruijn levels, the free variables keep track of a number that says "I am a free variable representing the nth bound variable *from the top of the telescope*".

This is the opposite of a deBruijn index, which is a number indicating "the nth bound variable from the bottom of the telescope".

Top and bottom here refer to visualizing the expression's telescope as a tree:



For example, with a lambda `fun (a b c d e) => bvar(0)`, the bound variable refers to `e`, by referencing "the 0th from the bottom".

In the lambda expression `fun (a b c d e) => fvar(4)`, the free variable is a deBruijn level representing `e` again, but this time as "the 4th from the top of the telescope".

Why the distinction? When we create a free variable during strong reduction, we know a couple of things: we know that the free variable we're about to sub in might get moved around by further reduction, we know how many open binder are *ABOVE* us (because we had to visit them to get here), and we know we might need to quote/abstract this expression to replace the binders, meaning we need to re-bind the free variable. However, in that moment, we do NOT know how many binders remain below us, so we cannot say how many variables from the bottom that variable might be when it's eventually abstracted/quoted.

For implementations using unique identifiers to tag free variables, this problem is solved by having the actual telescope that's being reconstructed during abstraction. As long as you have the expression and a list of the uniquely-tagged free variables, you can abstract, because the position of the free variables within the list indicates their binder position.

Weak and strong reduction

The implementation details of Lean's reduction strategies is discussed in another chapter; this section is specifically to clarify the difference between the general concepts of weak and strong reduction.

Weak reduction

Weak reduction refers to reduction that stops at binders which do not have an argument applied to them. By binders, we mean lambda, pi, and let expressions.

For example, weak reduction can reduce `(fun (x y : Nat) => y + x) (0 : Nat)` to `(fun (y : Nat) => y + 0)`, but can do no further reduction.

When we say or 'weak head normal form reduction', or just reduction without specifically identifying it as 'strong', we're talking about weak reduction. Strong reduction just happens as a byproduct of applying weak reduction after we've opened a binder somewhere else.

Strong reduction

Strong reduction refers to reduction under open binders; when we run across a binder without an accompanying argument (like a lambda expression with no `app` node applying an argument), we can traverse into the body and potentially do further reduction by creating and substituting in a free variable. Strong reduction is needed for type inference and definitional equality checking. For type inference, we also need the ability to "re-close" open terms, replacing free variables with the correct bound variables after some reduction has been done in the body. This is not as simple as just replacing it with the same bound variable as before, because bound variables may have shifted, invalidating their old deBruijn index relative to the new rebuilt expression.

As with weak reduction, strong reduction can still reduce `(fun (x y : Nat) => y + x) (0 : Nat)` to `(fun (y : Nat) => y + 0)`, and instead of getting stuck, it can continue by substituting `y` for a free variable, reducing the expression further to `((fVar id, y, Nat) + 0)`, and `(fvar id, y, Nat)`.

As long as we keep the free variable information around *somewhere*, we can re-combine that information with the reduced `(fVar id, y, Nat)` to recreate `(fun (y : Nat) =>`

```
    bvar(0))
```

Names

The first of the kernel's primitive types is `Name`, which is sort of what it sounds like; it provides kernel items with a way of addressing things.

```
Name ::= anonymous | str Name String | num Name Nat
```

Elements of the `Name` type are displayed as dot-separated names, which users of Lean are probably familiar with. For example, `num (str (anonymous) "foo") 7` is displayed as `foo.7`.

Implementation notes

The implementation of names assumes UTF-8 strings, with characters as unicode scalars (these assumptions about the implementing language's string type are also important for the string literal kernel extension).

Some information on the lexical structure of names can be found here

The exporter does not explicitly output the anonymous name, and expects it to be the 0th element of the imported names.

Universe levels

This section will describe universe levels from an implementation perspective, and will cover what readers need to know when it comes to type checking Lean declarations. More in-depth treatment of their role in Lean's type theory can be found in TPIL4, or section 2 of Mario Carneiro's thesis

The syntax for universe levels is as follows:

```
Level ::= Zero | Succ Level | Max Level Level | IMax Level Level | Param Name
```

Properties of the `Level` type that readers should take note of are the existence of a partial order on universe levels, the presence of variables (the `Param` constructor), and the distinction between `Max` and `IMax`.

`Max` simply constructs a universe level that represents the larger of the left and right arguments. For example, `Max(1, 2)` simplifies to `2`, and `Max(u, u+1)` simplifies to `u+1`. The `IMax` constructor represents the larger of the left and right arguments, *unless* the right argument simplifies to `Zero`, in which case the entire `IMax` resolves to `0`.

The important part about `IMax` is its interaction with the type inference procedure to ensure that, for example, `forall (x y : Sort 3), Nat` is inferred as `Sort 4`, but `forall (x y : Sort 3), True` is inferred as `Prop`.

Partial order on levels

Lean's `Level` type is equipped with a partial order, meaning there's a "less than or equals" test we can perform on pairs of levels. The rather nice implementation below comes from Gabriel Ebner's Lean 3 checker trepplein. While there are quite a few cases that need to be covered, the only complex matches are those relying on `cases`, which checks whether `x ≤ y` by examining whether `x ≤ y` holds when a parameter `p` is substituted for `Zero`, and when `p` is substituted for `Succ p`.

```
  leq (x y : Level) (balance : Integer): bool :=
    Zero, _ if balance >= 0 => true
    _, Zero if balance < 0 => false
    Param(i), Param(j) => i == j && balance >= 0
    Param(_), Zero => false
    Zero, Param(_) => balance >= 0
    Succ(l1_), _ => leq l1_ l2 (balance - 1)
    _, Succ(l2_) => leq l1 l2_ (balance + 1)

    -- descend left
    Max(a, b), _ => (leq a l2 balance) && (leq b l2 balance)

    -- descend right
    (Param(_) | Zero), Max(a, b) => (leq l1 a balance) || (leq l1 b balance)

    -- imax
    IMax(a1, b1), IMax(a2, b2) if a1 == a2 && b1 == b2 => true
    IMax(_, p @ Param(_)), _ => cases(p)
    _, IMax(_, p @ Param(_)) => cases(p)
    IMax(a, IMax(b, c)), _ => leq Max(IMax(a, c), IMax(b, c)) l2 balance
    IMax(a, Max(b, c)), _ => leq (simplify Max(IMax(a, b), IMax(a, c))) l2
balance
    _, IMax(a, IMax(b, c)) => leq l1 Max(IMax(a, c), IMax(b, c)) balance
    _, IMax(a, Max(b, c)) => leq l1 (simplify Max(IMax(a, b), IMax(a, c)))
balance


  cases l1 l2 p: bool :=
    leq (simplify $ subst l1 p zero) (simplify $ subst l2 p zero)
    ∧
    leq (simplify $ subst l1 p (Succ p)) (simplify $ subst l2 p (Succ p))
```

Equality for levels

The `Level` type recognizes equality by antisymmetry, meaning two levels `l1` and `l2` are equal if `l1 ≤ l2` and `l2 ≤ l1`.

Implementation notes

Be aware that the exporter does not export `Zero`, but it is assumed to be the 0th element of `Level`.

For what it's worth, the implementation of `Level` does not have a large impact on performance, so don't feel the need to aggressively optimize here.

Expressions

Complete syntax

Expressions will be explained in more detail below, but just to get it out in the open, the complete syntax for expressions, including the string and nat literal extensions, is as follows:

```
Expr ::=
  | boundVariable
  | freeVariable
  | const
  | sort
  | app
  | lambda
  | forall
  | let
  | proj
  | literal

BinderInfo ::= Default | Implicit | InstanceImplicit | StrictImplicit

const ::= Name, Level*
sort ::= Level
app ::= Expr Expr
-- a deBruijn index
boundVariable ::= Nat
lambda ::= Name, (binderType : Expr), BinderInfo, (body : Expr)
forall ::= Name, (binderType : Expr), BinderInfo, (body : Expr)
let ::= Name, (binderType : Expr), (val : Expr) (body : Expr)
proj ::= Name Nat Expr
literal ::= natLit | stringLit

-- Arbitrary precision nat/unsigned integer
natLit ::= Nat
-- UTF-8 string
stringLit ::= String

-- fvarId can be implemented by unique names or deBruijn levels;
-- unique names are more versatile, deBruijn levels have better
-- cache behavior
freeVariable ::= Name, Expr, BinderInfo, fvarId
```

Some notes:

- The `Nat` used by nat literals should be an arbitrary precision natural/bignum.

- The expressions that have binders (lambda, pi, let, free variable) can just as easily bundle the three arguments (binder_name, binder_type, binder_style) as one argument `Binder`, where a binder is `Binder ::= Name BinderInfo Expr`. In the pseudocode that appears elsewhere I will usually treat them as though they have that property, because it's easier to read.
- Free variable identifiers can be either unique identifiers, or they can be deBruijn levels.
- The expression type used in Lean proper also has an `mdata` constructor, which declares an expression with attached metadata. This metadata does not effect the expression's behavior in the kernel, so we do not include this constructor.

Binder information

Expressions constructed with the lambda, pi, let, and free variable constructors contain binder information, in the form of a name, a binder "style", and the binder's type. The binder's name and style are only for use by the pretty printer, and do not alter the core procedures of inference, reduction, or equality checking. In the pretty printer however, the binder style may alter the output depending on the pretty printer options. For example, the user may or may not want to display implicit or instance implicits (typeclass variables) in the output.

Sort

`sort` is simply a wrapper around a level, allowing it to be used as an expression.

Bound variables

Bound variables are implemented as natural numbers representing deBruijn indices.

Free variables

Free variables are used to convey information about bound variables in situations where the binder is currently unavailable. Usually this is because the kernel has traversed into the body of a binding expression, and has opted not to carry a structured context of the binding information, instead choosing to temporarily swap out the bound variable for a free variable, with the option of swapping in a new (maybe different) bound variable to

reconstruct the binder. This unavailability description may sound vague, but a literal explanation that might help is that expressions are implemented as trees without any kind of parent pointer, so when we descend into child nodes (especially across function boundaries), we end up just losing sight of the elements above where we currently are in a given expression.

When an open expression is closed by reconstructing binders, the bindings may have changed, invalidating previously valid deBruijn indices. The use of unique names or deBruijn levels allow this re-closing of binders to be done in a way that compensates for any changes and ensures the new deBruijn indices of the re-bound variables are valid with respect the reconstructed telescope (see this section).

Going forward, we may use some form of the term "free variable identifier" to refer to the objects in whatever scheme (unique IDs or deBruijn levels) an implementation may be using.

`Const`

The `const` constructor is how an expression refers to another declaration in the environment, it must do so by reference.

In example below, `def plusOne` creates a `Definition` declaration, which is checked, then admitted to the environment. Declarations cannot be placed directly in expressions, so when the type of `plusOne_eq_succ` invokes the previous declaration `plusOne`, it must do so by name. An expression is created: `Expr.const (plusOne, [])`, and when the kernel finds this `const` expression, it can look up the declaration referred to by name, `plusOne`, in the environment:

```
def plusOne : Nat -> Nat := fun x => x + 1

theorem plusOne_eq_succ (n : Nat) : plusOne n = n.succ := rfl
```

Expressions created with the `const` constructor also carry a list of levels which are substituted into any unfolded or inferred declarations taken from the environment by looking up the definition the `const` expression refers to. For example, inferring the type of `const List [Level.param(x)]` involves looking up the declaration for `List` in the current environment, retrieving its type and universe parameters, then substituting `x` for the universe parameter with which `List` was initially declared.

Lambda, Pi

`lambda` and `pi` expressions (Lean proper uses the name `forallE` instead of `pi`) refer to function abstraction and the "forall" binder (dependent function types) respectively.

```
  binderName      body
      |            |
fun (foo : Bar) => 0
            |
        binderType

-- `BinderInfo` is reflected by the style of brackets used to
-- surround the binder.
```

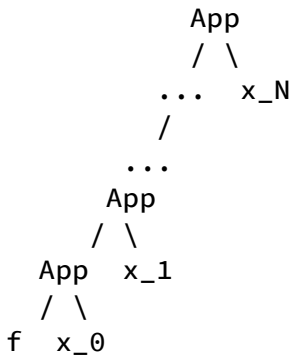
Let

`let` is exactly what it sounds like. While `let` expressions are binders, they do not have a `BinderInfo`, their binder info is treated as `Default`.

```
  binderName      val
      |            |
let (foo : Bar) := 0; foo
            |          |
        binderType     .... body
```

App

`app` expressions represent the application of an argument to a function. App nodes are binary (have only two children, a single function and an single argument), so `f x_0 x_1 .. x_N` is represented by `App(App(App(f, x_0), x_1)..., x_N)`, or visualized as a tree:



An exceedingly common kernel operation for manipulating expressions is folding and unfolding sequences of applications, getting `(f, [x_0, x_1, .., x_N])` from the tree

structure above, or folding `f, [x_0, x_1, .., x_N]` into the tree above.

Projections

The `proj` constructor represents structure projections. Inductive types that are not recursive, have only one constructor, and have no indices can be structures.

The constructor takes a name, which is the name of the type, a natural number indicating the field being projected, and the actual structure the projection is being applied to.

Be aware that in the kernel, projection indices are 0-based, despite being 1-based in Lean's vernacular, where 0 is the first non-parameter argument to the constructor.

For example, the kernel expression `proj Prod 0 (@Prod.mk A B a b)` would project the `a`, because it is the 0th field after skipping the parameters `A` and `B`.

While the behavior offered by `proj` can be accomplished by using the type's recursor, `proj` more efficiently handles frequent kernel operations.

Literals

Lean's kernel optionally supports arbitrary precision Nat and String literals. As needed, the kernel can transform a nat literal `n` to `Nat.zero` or `Nat.succ m`, or convert a string literal `s` to `String.mk List.nil` or `String.mk (List.cons (Char.ofNat _) ...)`.

String literals are lazily converted to lists of characters for testing definitional equality, and when they appear as the major premise in reduction of a recursor.

Nat literals are supported in the same positions as strings (definitional equality and major premises of a recursor application), but the kernel also provide support for addition, multiplication, exponentiation, subtraction, mod, division, as well as boolean equality and "less than or equal" comparisons on nat literals.

Implementing Expressions

A few noteworthy points about the expression type for readers interested in writing their own kernel in whole or in part...

Stored data

Expressions need to store some data inline or cache it somewhere to prevent prohibitively expensive recomputation. For example, creating an expression `app x y` needs to calculate and then store the hash digest of the resulting app expression, and it needs to do so by getting the cached hash digests of `x` and `y` instead of traversing the entire tree of `x` to recursively calculate the digest of `x`, then doing the same for `y`.

The data you will probably want to store inline are the hash digest, the number of loose bound variables in an expression, and whether or not the expression has free variables. The latter two are useful for optimizing instantiation and abstraction respectively.

An example "smart constructor" for `app` expressions would be:

```
def mkApp x y:
  let hash := hash x.storedHash y.storedHash
  let numLooseBvars := max x.numLooseBvars y.numLooseBvars
  let hasFvars := x.hasFvars || y.hasFvars
  .app x y (cachedData := hash numLooseBVars hasFVars)
```

No deep copies

Expressions should be implemented such that child expressions used to construct some parent expression are not deep copied. Put another way, creating an expression `app x y` should not recursively copy the elements of `x` and `y`, rather it should take some kind of reference, whether it's a pointer, integer index, reference to a garbage collected object, reference counted object, or otherwise (any of these strategies should deliver acceptable performance). If your default strategy for constructing expressions involves deep copies, you will not be able to construct any nontrivial environments without consuming massive amounts of memory.

Example implementation for number of loose bound variables

```
numLooseBVars e:
    match e with
    | Sort | Const | FVar | StringLit | NatLit => 0
    | Var dbjIdx => dbjIdx + 1,
    | App fun arg => max fun.numLooseBvars arg.numLooseBvars
    | Pi binder body | Lambda binder body =>
    |   max binder.numLooseBVars (body.numLooseBVars - 1)
    | Let binder val body =>
    |   max (max binder.numLooseBvars val.numLooseBvars) (body.numLooseBvars -
1)
    | Proj _ _ structure => structure.numLooseBvars
```

For `Var` expressions, the number of loose bound variables is the deBruijn index plus one, because we're counting the number of binders that would need to be applied for that variable to no longer be loose (the `+1` is because deBruijn indices are 0-based). For the expression `Var(0)`, one binder needs to be placed above the bound variable in order for the variable to no longer be loose. For `Var(3)`, we need four:

```
--  3 2 1 0
fun a b c d => Var(3)
```

When we create a new binder (lambda, pi, or let), we can subtract 1 (using saturating/natural number subtraction) from the number of loose bvars in the body, because the body is now under one additional binder.

Declarations

Declarations are the big ticket items, and are the last domain elements we need to define.

```
declarationInfo ::= Name, (universe params : List Level), (type : Expr)

declar ::=
  Axiom declarationInfo
  | Definition declarationInfo (value : Expr) ReducibilityHint
  | Theorem declarationInfo (value : Expr)
  | Opaque declarationInfo (value : Expr)
  | Quot declarationInfo
  | InductiveType
      declarationInfo
      is_recursive: Bool
      num_params: Nat
      num_indices: Nat
      -- The name of this type, and any others in a mutual block
      allIndNames: Name+
      -- The names of the constructors for *this* type only,
      -- not including the constructors for mutuals that may
      -- be in this block.
      constructorNames: Name*

  | Constructor
      declarationInfo
      (inductiveName : Name)
      (numParams : Nat)
      (numFields : Nat)

  | Recursor
        declarationInfo
        numParams : Nat
        numIndices : Nat
        numMotives : Nat
        numMinors : Nat
        RecRule+
        isK : Bool

RecRule ::= (constructor name : Name), (number of constructor args : Nat), (val
: Expr)
```

Checking a declaration

For all declarations, the following preliminary checks are performed before any additional procedures specific to certain kinds of declaration:

- The universe parameters in the declaration's `declarationInfo` must not have duplicates. For example, a declaration `def Foo.{u, v, u} ...` would be prohibited.
- The declaration's type must not have free variables; all variables in a "finished" declaration must correspond to a binder.
- The declaration's type must be a type (`infer declarationInfo.type` must produce a `Sort`). In Lean, a declaration `def Foo : Nat.succ := ..` is not permitted; `Nat.succ` is a value, not a type.

Axiom

The only checks done against axioms are those done for all declarations which ensure the `declarationInfo` passes muster. If an axiom has a valid set of universe parameters and a valid type with no free variables, it is admitted to the environment.

Quot

The `Quot` declarations are `Quot` , `Quot.mk` , `Quot.ind` , and `Quot.lift` . These declarations have prescribed types which are known to be sound within Lean's theory, so the environment's quotient declarations must match those types exactly. These types are hard-coded into kernel implementations since they are not prohibitively complex.

Definition, theorem, opaque

Definition, theorem, and opaque are interesting in that they both a type and a value. Checking these declarations involves inferring a type for the declaration's value, then asserting that the inferred type is definitionally equal to the ascribed type in the `declarationInfo` .

In the case of a theorem, the `declarationInfo` 's type is what the user claims the type is, and therefore what the user is claiming to prove, while the value is what the user has offered as a proof of that type. Inferring the type of the received value amounts to checking what the proof is actually a proof of, and the definitional equality assertion ensures that the thing the value proves is actually what the user intended to prove.

Reducibility hints

Reducibility hints contain information about how a declaration should be unfolded. An `abbreviation` will generally always be unfolded, `opaque` will not be unfolded, and `regular N` might be unfolded depending on the value of `N`. The `regular` reducibility hints correspond to a definition's "height", which refers to the number of declarations that definition uses to define itself. A definition `x` with a value that refers to definition `y` will have a height value greater than `y`.

The Secret Life of Inductive Types

Inductive

For clarity, the whole shebang of "an inductive declaration" is a type, a list of constructors, and a list of recursors. The declaration's type and constructors are specified by the user, and the recursor is derived from those elements. Each recursor also gets a list of "recursor rules", also known as computation rules, which are value level expressions used in iota reduction (a fancy word for pattern matching). Going forward, we will do our best do distinguish between "an inductive type" and "an inductive declaration".

Lean's kernel natively supports mutual inductive declarations, in which case there is a list of (type, list constructor) pairs. The kernel supports nested inductive declarations by temporarily transforming them to mutual inductives (more on this below).

Inductive types

The kernel requires the "inductive type" part of an inductive declaration to actually be a type, and not a value (`infer ty` must produce some `sort <n>`). For mutual inductives, the types being declared must all be in the same universe and have the same parameters.

Constructor

For any constructor of an inductive type, the following checks are enforced by the kernel:

- The constructor's type/telescope has to share the same parameters as the type of the inductive being declared.
- For the non-parameter elements of the constructor type's telescope, the binder type must actually be a type (must infer as `Sort _`).
- For any non-parameter element of the constructor type's telescope, the element's inferred sort must be less than or equal to the inductive type's sort, or the inductive type being declared has to be a prop.
- No argument to the constructor may contain a non-positive occurrence of the type being declared (readers can explore this issue in depth here).
- The end of the constructor's telescope must be a valid application of arguments to the

type being declared. For example, we require the `List.cons ..` constructor to end with `.. -> List A`, and it would be an error for `List.cons` to end with `.. -> Nat`

Nested inductives

Checking nested inductives is a more laborious procedure that involves temporarily specializing the nested parts of the inductive types in a mutual block so that we just have a "normal" (non-nested) set of mutual inductives, checking the specialized types, then unspecializing everything and admitting those types.

Consider this definition of S-expressions, with the nested construction `Array Sexpr`:

```
inductive Sexpr
| atom (c : Char) : Sexpr
| ofArray : Array Sexpr -> Sexpr
```

Zooming out, the process of checking a nested inductive declaration has three steps:

1. Convert the nested inductive declaration to a mutual inductive declaration by specializing the "container types" in which the current type is being nested. If the container type is itself defined in terms of other types, we'll need to reach those components for specialization as well. In the example above, we use `Array` as a container type, and `Array` is defined in terms of `List`, so we need to treat both `Array` and `List` as container types.

2. Do the normal checks and construction steps for a mutual inductive type.

3. Convert the specialized nested types back to the original form (un-specializing), adding the recovered/unspecialized declarations to the environment.

An example of this specialization would be the conversion of the `Sexpr` nested inductive above as:

```
mutual
  inductive Sexpr
    | atom : Char -> Sexpr
    | ofList : ListSexpr -> Sexpr

  inductive ListSexpr
    | nil : ListSexpr
    | cons : Sexpr -> ListSexpr -> ListSexpr

  inductive ArraySexpr
    | mk : ListSexpr -> ArraySexpr
end
```

Then recovering the original inductive declaration in the process of checking these types. To clarify, when we say "specialize", the new `ListSexpr` and `ArraySexpr` types above are specialized in the sense that they're defined only as lists and arrays of `Sexpr`, as opposed to being generic over some arbitrary type as with the regular `List` type.

Recursors

For now, see iota reduction in the section on reduction

Type Inference

Type inference is a procedure for determining the type of a given expression, and is one of the core functionalities of Lean's kernel. Type inference is how we determine that `Nat.zero` is an element of the type `Nat`, or that `(fun (x : Char) => var(0))` is an element of the type `Char -> Char`.

This section begins by examining the simplest complete procedure for type inference, then the more performant but slightly more complex version of each procedure.

We will also look at a number of additional correctness assertions that Lean's kernel makes during type inference.

Bound variables

If you're following Lean's implementation and using the locally nameless approach, you should not run into bound variables during type inference, because all open binders will be instantiated with the appropriate free variables.

When we come across a binder, we need to traverse into the body to determine the body's type. There are two main approaches one can take to preserve the information about the binder type; the one used by Lean proper is to create a free variable that retains the binder's type information, replace the corresponding bound variables with the free variable using instantiation, and then enter the body. This is nice, because we don't have to keep track of a separate piece of state for a typing context.

For closure-based implementations, you will generally have a separate typing context that keeps track of the open binders; running into a bound variable then means that you will index into your typing context to get the type of that variable.

Free variables

When a free variable is created, it's given the type information from the binder it represents, so we can just take that type information as the result of inference.

```
infer FVar id binder:
  binder.type
```

Function application

```
infer App(f, arg):
  match (whnf $ infer f) with
  | Pi binder body =>
    assert! defEq(binder.type, infer arg)
    instantiate(body, arg)
  | _ => error
```

The additional assertion needed here is that the type of `arg` matches the type of `binder`. For example, in the expression

`(fun (n : Nat) => 2 * n) 10`, we would need to assert that `defEq(Nat, infer(10))`.

While existing implementations prefer to perform this check inline, one could potentially store this equality assertion for processing elsewhere.

Lambda

```
infer Lambda(binder, body):
  assert! infersAsSort(binder.type)
  let binderFvar := fvar(binder)
  let bodyType := infer $ instantiate(body, binderFVar)
  Pi binder (abstract bodyType binderFVar)
```

Pi

```
infer Pi binder body:
  let l := inferSortOf binder
  let r := inferSortOf $ instantiate body (fvar(binder))
  imax(l, r)

inferSortOf e:
  match (whnf (infer e)) with
  | sort level => level
  | _ => error
```

Sort

The type of any `Sort n` is just `Sort (n+1)`.

```
infer Sort level:
  Sort (succ level)
```

Const

`const` expressions are used to refer to other declarations by name, and any other declaration referred to must have been previously declared and had its type checked. Since we therefore already know what the type of the referred to declaration is, we can just look it up in the environment. We do have to substitute in the current declaration's universe levels for the indexed definition's universe parameters however.

```
infer Const name levels:
  let knownType := environment[name].type
  substituteLevels (e := knownType) (ks := knownType.uparams) (vs := levels)
```

Let

```
infer Let binder val body:
  assert! inferSortOf binder
  assert! defEq(infer(val), binder.type)
  infer (instantiate body val)
```

Proj

We're trying to infer the type of something like `Proj (projIdx := 0) (structure := Prod.mk A B (a : A) (b : B))`.

Start by inferring the type of the structure offered; from that we can get the structure name and look up the structure and constructor type in the environment.

Traverse the constructor type's telescope, substituting the parameters of `Prod.mk` into the telescope for the constructor type. If we looked up the constructor type `A -> B -> (a : A) -> (b : B) -> Prod A B`, substitute A and B, leaving the telescope `(a : A) -> (b : B) -> Prod A B`.

The remaining parts of the constructor's telescope represent the structure's fields and have the type information in the binder, so we can just examine `telescope[projIdx]` and take the binder type. We do have to take care of one more thing; because later structure fields can depend on earlier structure fields, we need to instantiate the rest of the telescope (the body at each stage) with `proj thisFieldIdx s` where `s` is the original structure in the proj expression we're trying to infer.

```
infer Projection(projIdx, structure):
  let structType := whnf (infer structure)
  let (const structTyName levels) tyArgs := structType.unfoldApps
  let InductiveInfo := env[structTyName]
  -- This inductive should only have the one constructor since it's claiming to
be a structure.
  let ConstructorInfo := env[InductiveInfo.constructorNames[0]]

  let mut constructorType := substLevels ConstructorInfo.type (newLevels :=
levels)

  for tyArg in tyArgs.take constructorType.numParams
    match (whnf constructorType) with
      | pi _ body => inst body tyArg
      | _ => error

  for i in [0:projIdx]
    match (whnf constructorType) with
      | pi _ body => inst body (proj i structure)
      | _ => error

  match (whnf constructorType) with
    | pi binder _=> binder.type
    | _ => error
```

Nat literals

Nat literals infer as the constant referring to the declaration `Nat`.

```
infer NatLiteral _:
  Const(Nat, [])
```

String literals

String literals infer as the constant referring to the declaration `String`.

```
infer StringLiteral _:
  Const(String, [])
```

Definitional equality

Definitional equality is implemented as a function that takes two expressions as input, and returns `true` if the two expressions are definitionally equal within Lean's theory, and `false` if they are not.

Within the kernel, definitional equality is important simply because it's a necessary part of type checking. Definitional equality is still an important concept for Lean users who do not venture into the kernel, because definitional equalities are comparatively nice to work with in Lean's vernacular; for any `a` and `b` that are definitionally equal, Lean doesn't need any prompting or additional input from the user to determine whether two expressions are equal.

There are two big-picture parts of implementing the definitional equality procedure. First, the individual tests that are used to check for different definitional equalities. For readers who are just interested in understanding definitional equality from the perspective of an end user, this is probably what you want to know.

Readers interested in writing a type checker should also understand how the individual checks are composed along with reduction and caching to make the problem tractable; naively running each check and reducing along the way is likely to yield unacceptable performance results.

Sort equality

Two `Sort` expressions are definitionally equal if the levels they represent are equal by antisymmetry using the partial order on levels.

```
defEq (Sort x) (Sort y):
  x ≤ y ∧ y ≤ x
```

Const equality

Two `Const` expressions are definitionally equal if their names are identical, and their levels are equal under antisymmetry.

```
defEq (Const n xs) (Const m ys):
  n == m ∧ forall (x, y) in (xs, ys), antisymmEq x y

  -- also assert that xs and ys have the same length if your `zip` doesn't do
so.
```

Bound Variables

For implementations using a substitution-based strategy like locally nameless (if you're following the C++ or lean4lean implementations, this is you), encountering a bound variable is an error; bound variables should have been replaced during weak reduction if they referred to an argument, or they should have been replaced with a free variable via strong reduction as part of a definitional equality check for a pi or lambda expression.

For closure-based implementations, look up the elements corresponding to the bound variables and assert that they are definitionally equal.

Free Variables

Two free variables are definitionally equal if they have the same identifier (unique ID or deBruijn level). Assertions about the equality of the binder types should have been performed wherever the free variables were constructed (like the definitional equality check for pi/lambda expressions), so it is not necessary to re-check that now.

```
defEqFVar (id1, _) (id2, _):
  id1 == id2
```

App

Two application expressions are definitionally equal if their function component and argument components are definitionally equal.

```
defEqApp (App f a) (App g b):
  defEq f g && defEq a b
```

Pi

Two Pi expressions are definitionally equal if their binder types are definitionally equal, and their body types, after substituting in the appropriate free variable, are definitionally equal.

```
defEq (Pi s a) (Pi t b)
  if defEq s.type t.type
  then
    let thisFvar := fvar s
    defEq (inst a thisFvar) (inst b thisFvar)
  else
    false
```

Lambda

Lambda uses the same test as Pi:

```
defEq (Lambda s a) (Lambda t b)
  if defEq s.type t.type
  then
    let thisFvar := fvar s
    defEq (inst a thisFvar) (inst b thisFvar)
  else
    false
```

Structural eta

Lean recognizes definitional equality of two elements `x` and `y` if they're both instances of some structure type, and the fields are definitionally equal using the following procedure comparing the constructor arguments of one and the projected fields of the other:

```
defEqEtaStruct x y:
  let (yF, yArgs) := unfoldApps y
  if
    yF is a constructor for an inductive type `T`
    && `T` can be a struct
    && yArgs.len == T.numParams + T.numFields
    && defEq (infer x) (infer y)
  then
    forall i in 0..t.numFields, defEq Proj(i+T.numParams, x)
yArgs[i+T.numParams]

    -- we add `T.numParams` to the index because we only want
    -- to test the non-param arguments. we already know the
    -- parameters are defEq because the inferred types are
    -- definitionally equal.
```

The more pedestrian case of congruence `T.mk a .. N = T.mk x .. M` if `[a, .., N] = [x, .., M]`, is simply handled by the `App` test.

Unit-like equality

Lean recognizes definitional equality of two elements `x: S p_0 .. p_N` and `y: T p_0 .. p_M` under the following conditions:

- `S` is an inductive type
- `S` has no indices
- `S` has only one constructor which takes no arguments other than the parameters of `S`, `p_0 .. p_N`
- The types `S p_0 .. p_N` and `T p0 .. p_M` are definitionally equal

Intuitively this definitional equality is fine, because all of the information that elements of these types can convey is captured by their types, and we're requiring those types to be definitionally equal.

Eta expansion

```
defEqEtaExpansion x y : bool :=
  match x, (whnf $ infer y) with
  | Lambda .., Pi binder _ => defEq x (App (Lambda binder (Var 0)) y)
  | _, _ => false
```

The lambda created on the right, `(fun _ => $0) y` trivially reduces to `y`, but the addition of the lambda binder gives the `x` and `y'` a chance to match with the rest of the definitional equality procedure.

Proof irrelevant equality

Lean treats proof irrelevant equality as definitional. For example, Lean's definitional equality procedure treats any two proofs of `2 + 2 = 4` as definitionally equal expressions.

If a type `T` infers as `Sort 0`, we know it's a proof, because it is an element of `Prop` (remember that `Prop` is `Sort 0`).

```
defEqByProofIrrelevance p q :
  infer(p) == S ∧
  infer(q) == T ∧
  infer(S) == Sort(0) ∧
  infer(T) == Sort(0) ∧
  defEq(S, T)
```

If `p` is a proof of type `A` and `q` is a proof of type `B`, then if `A` is definitionally equal to `B`, `p` and `q` are definitionally equal by proof irrelevance.

Natural numbers (nat literals)

Two nat literals are definitionally equal if they can be reduced to `Nat.zero`, or they can be reduced as (`Nat.succ x`, `Nat.succ y`), where `x` and `y` are definitionally equal.

```
match X, Y with
| Nat.zero, Nat.zero => true
| NatLit 0, NatLit 0 => true
| Nat.succ x, NatLit (y+1) => defEq x (NatLit y)
| NatLit (x+1), Nat.succ y => defEq (NatLit x) y
| NatLit (x+1), NatLit (y+1) => x == y
| _, _ => false
```

String literal

```
StringLit(s), App(String.mk, a)
```

The string literal `s` is converted to an application of `Const(String.mk, [])` to a `List Char`. Because Lean's `Char` type is used to represent unicode scalar values, their integer representation is a 32-bit unsigned integer.

To illustrate, the string literal "ok", which uses two characters corresponding to the 32 bit unsigned integers `111` and `107` is converted to:

(String.mk (((List.cons Char) (Char.ofNat.[] NatLit(111))) (((List.cons Char) (Char.ofNat NatLit(107))) (List.nil Char))))

Lazy delta reduction and congruence

The available kernel implementations implement a "lazy delta reduction" procedure as part of the definitional equality check, which unfolds definitions lazily using reducibility hints and checks for congruence when things look promising. This is a much more efficient strategy than eagerly reducing both expressions completely before checking for definitional equality.

If we have two expressions `a` and `b`, where `a` is an application of a definition with height 10, and `b` is an application of a definition with height 12, the lazy delta procedure takes the more efficient route of unfolding `b` to try and get closer to `a`, as opposed to unfolding both of them completely, or blindly choosing one side to unfold.

If the lazy delta procedure finds two expressions which are an application of a `const` expression to arguments, and the `const` expressions refer to the same declaration, the expressions are checked for congruence (whether they're the same consts applied to definitionally equal arguments). Congruence failures are cached, and for readers writing their own kernel, caching these failures turns out to be a performance critical optimization, since the congruence check involves a potentially expensive call to `def_eq_args`.

Syntactic equality (also structural or pointer equality)

Two expressions are definitionally equal if they refer to exactly the same implementing object, as long as the type checker ensures that two objects are equal if and only if they are constructed from the same components (where the relevant constructors are those for Name, Level, and Expr).

Reduction

Reduction is about nudging expressions toward their normal form so we can determine whether expressions are definitionally equal. For example, we need to perform beta reduction to determine that `(fun x => x) Nat.zero` is definitionally equal to `Nat.zero`, and delta reduction to determine that `id List.nil` is definitionally equal to `List.nil`.

Reduction in Lean's kernel has two properties that introduce concerns which sometimes go unaddressed in basic textbook treatments of the topic. First, reduction in some cases is interleaved with inference. Among other things, this means reduction may need to be performed with open terms, even though the reduction procedures themselves are not creating free variables. Second, `const` expressions which are applied to multiple arguments may need to be considered together with those arguments during reduction (as in iota reduction), so sequences of applications need to be unfolded together at the beginning of reduction.

Beta reduction

Beta reduction is about reducing function application. Concretely, the reduction:

```
(fun x => x) a    ~~>    a
```

An implementation of beta reduction must despine an expression to collect any arguments in `app` expressions, check whether the expression to which they're applied is a lambda, then substitute the appropriate argument for any appearances of the corresponding bound variable in the function body:

```
betaReduce e:
  betaReduceAux e.unfoldApps

betaReduceAux f args:
  match f, args with
  | lambda _ body, arg :: rest => betaReduceAux (inst body arg) rest
  | _, _ => foldApps f args
```

An important performance optimization for the instantiation (substitution) component of beta reduction is what's sometimes referred to as "generalized beta reduction", which involves gathering the arguments that have a corresponding lambda and substituting them in all at once. This optimization means that for `n` sequential lambda expressions with applied arguments, we only perform one traversal of the expression to substitute the

appropriate arguments, instead of `n` traversals.

```
betaReduce e:
  betaReduceAux e.unfoldApps []

betaReduceAux f remArgs argsToApply:
  match f, remArgs with
  | lambda _ body, arg :: rest => betaReduceAux body rest (arg :: argsToApply)
  | _, _ => foldApps (inst f argsToApply) remArgs
```

Zeta reduction (reducing `let` expressions)

Zeta reduction is a fancy name for reduction of `let` expressions. Concretely, reducing

```
let (x : T) := y; x + 1    ~~>    (y : T) + 1
```

An implementation can be as simple as:

```
reduce Let _ val body:
  instantiate body val
```

Delta reduction (definition unfolding)

Delta reduction refers to unfolding definitions (and theorems). Delta reduction is done by replacing a `const ..` expr with the referenced declaration's value, after swapping out the declaration's generic universe parameters for the ones that are relevant to the current context.

If the current environment contains a definition `x` which was declared with universe parameters `u*` and value `v`, then we may delta reduce an expression `Const(x, w*)` by replacing it with `val`, then substituting the universe parameters `u*` for those in `w*`.

```
deltaReduce Const name levels:
  if environment[name] == (d : Declar) && d.val == v then
  substituteLevels (e := v) (ks := d.uparams) (vs := levels)
```

If we had to remove any applied arguments to reach the `const` expression that was delta reduced, those arguments should be reapplied to the reduced definition.

Projection reduction

A `proj` expression has a natural number index indicating the field to be projected, and another expression which is the structure itself. The actual expression comprising the structure should be a sequence of arguments applied to `const` referencing a constructor.

Keep in mind that for fully elaborated terms, the arguments to the constructor will include any parameters, so an instance of the `Prod` constructor would be e.g. `Prod.mk A B (a : A) (b : B)`.

The natural number indicating the projected field is 0-based, where 0 is the first *non-parameter* argument to the constructor, since a projection cannot be used to access the structure's parameters.

With this in mind, it becomes clear that once we despine the constructor's arguments into `(constructor, [arg0, .., argN])`, we can reduce the projection by simply taking the argument at position `i + num_params`, where `num_params` is what it sounds like, the number of parameters for the structure type.

```
reduce proj fieldIdx structure:
  let (constructorApp, args) := unfoldApps (whnf structure)
  let num_params := environment[constructorApp].num_params
  args[fieldIdx + numParams]

  -- Following our `Prod` example, constructorApp will be `Const(Prod.mk, [u,
v])`
  -- args will be `[A, B, a, b]`
```

Special case for projections: String literals

The kernel extension for string literals introduces one special case in projection reduction, and one in iota reduction.

Projection reduction for string literals: Because the projection expression's structure might reduce to a string literal (Lean's `String` type is defined as a structure with one field, which is a `List Char`)

If the structure reduces as a `StringLit (s)`, we convert that to `String.mk (.. : List Char)` and proceed as usual for projection reduction.

Nat literal reduction

The kernel extension for nat literals includes reduction of `Nat.succ` as well as the binary operations of addition, subtraction, multiplication, exponentiation, division, mod, boolean equality, and boolean less than or equal.

If the expression being reduced is `Const(Nat.succ, []) n` where `n` can be reduced to a nat literal `n'`, we reduce to `NatLit(n'+1)`

If the expression being reduced is `Const(Nat.<binop>, []) x y` where `x` and `y` can be reduced to nat literals `x'` and `y'`, we apply the native version of the appropriate `<binop>` to `x'` and `y'`, returning the resulting nat literal.

Examples:

```
Const(Nat.succ, []) NatLit(100) ~> NatLit(100+1)

Const(Nat.add, []) NatLit(2) NatLit(3) ~> NatLit(2+3)

Const(Nat.add, []) (Const Nat.succ [], NatLit(10)) NatLit(3) ~> NatLit(11+3)
```

Iota reduction (pattern matching)

Iota reduction is about applying reduction strategies that are specific to, and derived from, a given inductive declaration. What we're talking about is application of an inductive declaration's recursor (or the special case of `Quot` which we'll see later).

Each recursor has a set of "recursor rules", one recursor rule for each constructor. In contrast to the recursor, which presents as a type, these recursor rules are value level expressions showing how to eliminate an element of type `T` created with constructor `T.c`. For example, `Nat.rec` has a recursor rule for `Nat.zero`, and another for `Nat.succ`.

For an inductive declaration `T`, one of the elements demanded by `T`'s recursor is an actual `(t : T)`, which is the thing we're eliminating. This `(t : T)` argument is known as the "major premise". Iota reduction performs pattern matching by taking apart the major premise to see what constructor was used to make `t`, then retrieving and applying the corresponding recursor rule from the environment.

Because the recursor's type signature also demands the parameters, motives, and minor premises required, we don't need to change the arguments to the recursor to perform reduction on e.g. `Nat.zero` as opposed to `Nat.succ`.

In practice, it's sometimes necessary to do some initial manipulation to expose the constructor used to create the major premise, since it may not be found as a direct application of a constructor. For example, a `NatLit(n)` expression will need to be transformed into either `Nat.zero`, or `App Const(Nat.succ, []) ...`. For structures, we may also perform structural eta expansion, transforming an element `(t : T)` into `T.mk t.1 .. t.N`, thereby exposing the application of the `mk` constructor, permitting iota reduction to proceed (if we can't figure out what constructor was used to create the major premise, reduction fails).

List.rec type

```
forall
  {α : Type.{u}}
  {motive : (List.{u} α) -> Sort.{u_1}},
  (motive (List.nil.{u} α)) ->
  (forall (head : α) (tail : List.{u} α), (motive tail) -> (motive (List.cons.
{u} α head tail))) -> (forall (t : List.{u} α), motive t)
```

List.nil rec rule

```
fun
  (α : Type.{u})
  (motive : (List.{u} α) -> Sort.{u_1})
  (nilCase : motive (List.nil.{u} α))
  (consCase : forall (head : α) (tail : List.{u} α), (motive tail) -> (motive
(List.cons.{u} α head tail))) =>
  nilCase
```

List.cons rec rule

```
fun
  (α : Type.{u})
  (motive : (List.{u} α) -> Sort.{u_1})
  (nilCase : motive (List.nil.{u} α))
  (consCase : forall (head : α) (tail : List.{u} α), (motive tail) -> (motive
(List.cons.{u} α head tail)))
  (head : α)
  (tail : List.{u} α) =>
  consCase head tail (List.rec.{u_1, u} α motive nilCase consCase tail)
```

k-like reduction

For some inductive types, known as "subsingleton eliminators", we can proceed with iota reduction even when the major premise's constructor is not directly exposed, as long as we know its type. This may be the case when, for example, the major premise appears as a free variable. This is known as k-like reduction, and is permitted because all elements of a subsingleton eliminator are identical.

To be a subsingleton eliminator, an inductive declaration must be an inductive prop, must not be a mutual or nested inductive, must have exactly one constructor, and the sole constructor must take only the type's parameters as arguments (it cannot "hide" any information that isn't fully captured in its type signature).

For example, the value of any element of the type `Eq Char 'x'` is fully determined just by its type, because all elements of this type are identical.

If iota reduction finds a major premise which is a subsingleton eliminator, it is permissible to substitute the major premise for an application of the type's constructor, because that is the only element the free variable could actually be. For example, a major premise which is a free variable of type `Eq Char 'a'` may be substituted for an explicitly constructed `Eq.refl Char 'a'`.

Getting to the nuts and bolts, if we neglected to look for and apply k-like reduction, free variables that are subsingleton eliminators would fail to identify the corresponding recursor rule, iota reduction would fail, and certain conversions expected to succeed would no longer succeed.

`Quot` reduction; `Quot.ind` and `Quot.lift`

`Quot` introduces two special cases which need to be handled by the kernel, one for `Quot.ind`, and one for `Quot.lift`.

Both `Quot.ind` and `Quot.lift` deal with application of a function `f` to an argument `(a : α)`, where the `a` is a component of some `Quot r`, formed with `Quot.mk r a`.

To execute the reduction, we need to pull out the argument that is the `f` element and the argument that is the `Quot` where we can find `(a : α)`, then apply the function `f` to `a`. Finally, we reapply any arguments that were part of some outer expression not related to the invocation of `Quot.ind` or `Quot.lift`.

Since this is only a reduction step, we rely on the type checking phases done elsewhere to provide assurances that the expression as a whole is well typed.

The type signatures for `Quot.ind` and `Quot.mk` are recreated below, mapping the elements of the telescope to what we should find as the arguments. The elements with a `*` are the ones we're interested in for reduction.

```
Quotient primitive Quot.ind.{u} : ∀ {α : Sort u} {r : α → α → Prop}
  {β : Quot r → Prop}, (∀ (a : α), β (Quot.mk r a)) → ∀ (q : Quot r), β q

  0  |-> {α : Sort u}
  1  |-> {r : α → α → Prop}
  2  |-> {β : Quot r → Prop}
  3* |-> (∀ (a : α), β (Quot.mk r a))
  4* |-> (q : Quot r)
  ...


Quotient primitive Quot.lift.{u, v} : {α : Sort u} →
  {r : α → α → Prop} → {β : Sort v} → (f : α → β) →
  (∀ (a b : α), r a b → f a = f b) → Quot r → β

  0  |-> {α : Sort u}
  1  |-> {r : α → α → Prop}
  2  |-> {β : Sort v}
  3* |-> (f : α → β)
  4  |-> (∀ (a b : α), r a b → f a = f b)
  5* |-> Quot r
  ...
```

Future work and open issues

File format

There is an open rfc here about moving the export file format to json, which would be a major version change.

Ensuring Nat/String are defined properly.

The Lean community has not yet settled on a particular solution for determining whether the declarations in an export file for `Nat`, `String`, and the operations covered by the relevant kernel extensions match those expected by extensions in a way that does not pull the exporter into the trusted code.

An approach similar to that taken for `Eq` and the `Quot` declarations (defining them by hand within the type checker, then asserting they're the same) is not feasible due to the complexity of the fully elaborated terms for the supported binary operations on `Nat`.

Improving Pollack consistency

Lean 4 offers very powerful facilities for defining custom syntax, macros, and pretty printer behaviors, and almost every aspect of Lean 4's internals is available to users. These elements of Lean's design were effective responses to real world feedback from the mathlib community during Lean 3's lifetime.

While these features were important factors in Lean's success as a tool for enabling large formalization efforts, they are also in tension with Lean4's Pollack consistency[1], or lack thereof. Without replicating the macro and syntax extension capabilities in the pretty printer, type checkers cannot consistently read terms back to the user in a form that is recognizable. However, the idea of adding these features to a pretty printer is an unappealing expansion of the trusted code base. An alternative approach is to drop the pretty printer in favor of a trusted parser (ala metamath zero), but Lean's parser can be modified on the fly in userspace with custom syntax declarations.

As Lean matures and adoption increases, there is likely to be a push for progress in the

development of techniques and practices that allow users to take advantage of Lean's extensibility while sacrificing the least degree of Pollack consistency.

Forward reasoning

Existing type checkers implement a form of backward reasoning; an alternate strategy for type checking is to accept and check forward reasoning chains worked out by an external program, potentially allowing for an even simpler type checker.

[1] Freek Wiedijk. Pollack-inconsistency. Electronic Notes in Theoretical Computer Science, 285:85–100, 2012

Further reading

- Mario Carneiro's thesis on Lean's type theory
- Lean's official kernel: https://github.com/leanprover/lean4/tree/master/src/kernel
- lean4lean, a reimplementation of Lean's kernel in Lean4.
- lean4export, the recommended/official exporter for Lean 4 environments.
- lean4checker
- Peter Dybjer's original description of the inductive types implemented by Lean: P. Dybjer. Inductive families. Formal aspects of computing, 6(4):440–465, 1994.
- Conor McBride and James McKinna's paper *I am not a number: I am a free variable* describing the locally nameless approach to bound/free variables
- Information about non-positive occurrences in inductive types, from *Counterexamples in Type Systems*: https://counterexamples.org/strict-positivity.html?highlight=posi#positivity-strict-and-otherwise
- Freek Wiedijk. Pollack-inconsistency. Electronic Notes in Theoretical Computer Science, 285:85–100, 2012