Panbench: A Comparative Benchmarking Tool for Dependently-Typed Languages

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Abstract

We benchmark four proof assistants (Agda, Idris 2, Lean 4 and Rocq) through a single test suite. We focus our benchmarks on the basic features that all systems based on a similar foundations (dependent type theory) have in common. We do this by creating an "over language" in which to express all the information we need to be able to output *correct and idiomatic syntax* for each of our targets. Our benchmarks further focus on "basic engineering" of these systems: how do they handle long identifiers, long lines, large records, large data declarations, and so on.

Our benchmarks reveals both flaws and successes in all systems. We give a thorough analysis of the results.

We also detail the design of our extensible system. It is designed so that additional tests and additional system versions can easily be added. A side effect of this work is a better understanding of the common abstract syntactic structures of all four systems.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Add keywords

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Production-grade implementations of dependently typed programming languages are complicated pieces of software that implement many intricate and potentially expensive algorithms. As such, large amounts of engineering effort has been dedicated to optimizing these components. Unfortunately, engineering time is a finite resource, and this necessarily means that other parts of these systems get comparatively less attention. This often results in easy-to-miss performance problems: we have heard anecdotes from a proof assistant developer that a naïve $O(n^2)$ fresh name generation algorithm used for pretty-printing resulted in 100x slowdowns in some pathological cases.

This suggests that a benchmarking suite that focuses on these simpler components could reveal some (comparatively) easy potential performance gains. Moreover, such a benchmarking suite would also be valuable for developers of new dependently typed languages, as it is much easier to optimize with a performance goal in mind. This is an instance of the classic $m \times n$ language tooling problem: constructing a suite of m benchmarks for n languages directly requires a quadratic amount of work up front, and adding either a new test case or a new language to the suite requires an additional linear amount of effort.

Like most $m \times n$ tooling problems, the solution is to introduce a mediating tool. In our case, we ought to write all of the benchmarks in an intermediate language, and then translate that intermediate language to the target languages in question. There are existing languages like Dedukti[2] or Informath[1] that attempt to act as an intermediary between popular proof assistants, but these tools typically focus on translating the *content* of proofs, not exact syntactic structure. To fill this gap, we have created the Panbench system, which consists of:

1. An extensible embedded Haskell DSL that encodes the concrete syntax of a typical dependently typed language.

Reed: Citations here

2. A series of compilers for that DSL to Agda, Idris 2, Lean 4, and Rocq.
3. A benchmarking harness that can perform sandboxed builds of multiple revisions Agda, Idris 2, Lean 4, Rocq.
4. An incremental build system that can produce benchmarking reports as static HTML files or PGF plots[1].

2 Methodology

This is really documenting the 'experiment'. The actual details of the thinking behind the design is in Section 6.

this itemized list should be expanded into actual text

- single language of tests
- document the setup of tests, high level
- document the setup of testing infrastructure, high level
- linear / exponential scaling up of test 'size'

3 Results

4 Discussion

5 Infrastructure

One of the major goals of Panbench is to make performance analysis as low-cost as possible for language developers. Meeting this goal requires a large amount of supporting infrastructure: simply generating benchmarks is not very useful if you cannot run them nor analyze their outcomes. After some discussion, we concluded that any successful language benchmarking system should meet the following criteria:

1. It must provide infrastructure for performing sandboxed builds of compilers from source. Asking potential users to set up four different toolchains presents an extremely large barrier to adoption. Moreover, if we rely on user-provided binaries, then we have no hope of obtaining reproducible results, which in turn makes any insights far less actionable.
2. It must allow for multiple revisions of the same tool to be installed simultaneously. This enables developers to easily look for performance regressions, and quantify the impact of optimizations.
3. It must allow for multiple copies of the *same* version tool to be installed with different build configurations. This allows developers to look for performance regressions induced by different compiler versions/optimizations.
4. It must be able to be run locally on a developers machine. Cloud-based tools are often cumbersome to use and debug, which in turn lowers adoption.
5. It must present a declarative interface for creating benchmarking environments and running benchmarks. Sandboxed builds of tools are somewhat moot if we cannot trust that a benchmark was run with the correct configuration.

[1] All plots in this paper were produced directly by Panbench!

6. It must present performance results in a self-contained format that is easy to understand and share. Performance statistics that require large amounts of post-processing or dedicated tools to view can not be easily shared with developers, which in turn makes the data less actionable.

Of these criteria, the first four present the largest engineering challenge, and are tantamount to constructing a meta-build system that is able orchestrate *other* build systems. We approached the problem by constructing a bespoke content-addressed system atop of Shake [8], which we discuss in section 5.1. The final two criteria also presented some unforseen difficulties, which we detail in 5.2.

5.1 The Panbench Build System

As noted earlier, we strongly believe that any benchmarking system should provide infrastructure for installing multiple versions of reproducibly built software. Initially, we intended to build this infrastructure for Panbench atop of Nix [4]. This is seemingly a perfect fit; after all, Nix was designed to facilitate almost exactly this use-case. However, after further deliberation, we came to the conclusion that Nix did not quite meet our needs for the following reasons:

1. Nix does not work natively on Windows. Performance problems can be operating system specific, so ruling out an OS that has a large user base that is often overlooked in testing seems unwise[2].
2. Nix adds a barrier to adoption. Installing Nix is a somewhat invasive process, especially on MacOS[3]. We believe that it is somewhat unreasonable to ask developers to add users and modify their root directory to run a benchmarking tool, and strongly suspect that this would hamper adoption.

With the obvious option exhausted, we opted to create our own Nix-inspired build system based atop Shake [8]. This avoids the aforementioned problems with Nix: Shake works on Windows, and only requires potential users to install a Haskell toolchain.

The details of content-addressed build systems are a deep topic unto themselves, so we will only describe the key points. Systems like Nix use an *input-addressing* scheme, wherein the results of a build are stored on disk prefixed by a hash of all build inputs. Crucially, this lets the build system know where to store the result of the build before the build is run, which avoids vicious cycles where the result of a build depends on its own hash. However, most input-addressed systems also require that the hash of the inputs *solely* determines the output. On its face, this is a reasonable ask, but taking this idea to its logical conclusion requires one to remove *any* dependency on the outer environment, which in turn forces on re-package the entirety of the software stack all the way down to `libc`. This is an admirable goal in its own right, but is actually somewhat counterproductive for our use case: there is a very real chance that we might end up benchmarking our sub-par repackaging of some obscure C dependency four layers down the stack.

To avoid this cascading series of dependency issues, Panbench takes a hybrid approach, wherein builds are input-addressed, but are allowed to also depend on the external environment. Additionally, the results of builds are also hashed, and stored out-of-band inside of a

[2] Currently, Panbench does not support Windows, but this is an artifact of prioritization, and not a fundamental restriction.
[3] The situation is even worse on x86-64 Macs, which most Nix installers simply do not support.

Shake database. This hash is used to perform invalidate downstream results, and also as a fingerprint to identify if two benchmarks were created from the same binary. This enables a pay-as-you go approach to reproducibility, which we hope will result in a lower adoption cost. Moreover, we can achieve fully reproducible builds by using Nix as a meta-meta build system to compile Panbench: this is how we obtained the results presented in Section 3.

5.2 Running Benchmarks and Generating Reports

As noted in the introduction to this section, we believe that benchmarking tools should present a *declarative* interface for writing not just single benchmarking cases, but entire benchmarking suites and their corresponding environments. Panbench accomplishes this by *also* implementing the benchmark execution framework atop Shake. This lets us easily integrate the process of tool installation with environment setup, but introduces its own set of engineering challenges.

The crux of the problem is that performance tests are extremely sensitive to the current load on the system. This is largely at odds with the goals of a build system, which is to completely saturate all system resources to try to complete a build as fast as possible. This can be avoided via careful use of locks, but we are then faced with another, larger problem. Haskell is a garbage collected language, and running the GC can put a pretty heavy load on the system. Moreover, the GHC runtime system is very well engineered, and is free to run the garbage collector inside of `safe` FFI calls, and waiting for process completion is marked as `safe`.

To work around this, we take opt to eschew existing process interaction libraries, and implement the benchmark spawning code in C[4]. This lets us take the rather extreme step of linking against the GHC runtime system so that we can call `rts_pause`, which pauses all other Haskell threads and GC sweeps until `rts_resume` is called.

Initially, we thought that this was the only concern that would arise by tightly integrating the build system with the benchmark executor. However, our initial benchmarks on Linux systems displayed some very strange behaviour, wherein the max resident set size reported by `getrusage` and `wait4` would consistently always report a reading of approximately 2 gigabytes halfway through a full benchmarking suite.

6 The Design of Panbench

At its core, Panbench is a tool for producing grammatically well-formed concrete syntax across multiple different languages. Crucially, Panbench does *not* require that the syntax produced is well-typed or even well-scoped: if it did, then it would be impossible to benchmark how systems perform when they encounter errors. This seemingly places Panbench in stark contrast with other software tools for working with the meta-theoretic properties of type systems, which are typically concerned only with well-typed terms.

Reed: Cite something

However, core task of Panbench is not that different from the task of a logical framework [5]: Both systems exist to manipulate judgements, inference rules, and derivations: Panbench just works with *grammatical* judgements and production rules rather than typing judgments and inference rules. In this sense Panbench is a *grammatical* framework[5] rather than a logical one.

[4] This is why Panbench does not currently support Windows.
[5] Not to be confused with *the* Grammatical Framework [9], which aims to be a more natural-language focused meta-framework for implementing and translating between grammars.

This similarity let us build Panbench atop well-understood design principles. In particular, a mechanized logical framework typically consists of two layers:

1. A layer for defining judgements àla relations.
2. A logic programming layer for synthesizing derivations.

To use a logical framework, one first encodes a language by laying out all of the judgements. Then, one needs to prove an adequacy theorem on the side that shows that their encoding of the judgements actually aligns with the language. However, if one wanted to mechanize this adequacy proof, then a staged third layer that consists of a more traditional proof assistant would be required.

If we take this skeleton of a design and transpose it to work with grammatical constructs rather than logical ones, we will also obtain three layers:

1. A layer for defining grammars àla relations.
2. A logic programming layer for synthesizing derivations.
3. A staged functional programming layer for proving "adequacy" results.

In this case, an adequacy result for a given language $\mathcal{L}$ is a constructive proof that all grammatical derivations written within the framework can be expressed within the concrete syntax of a language $\mathcal{L}$. However, the computational content of such a proof essentially amounts to a compiler written in the functional programming layer.

6.1 Implementing The Grammatical Framework

Implementing a bespoke hybrid of a logic and functional programming language is no small feat, and also requires prospective users to learn yet another single-purpose tool. Luckily, there already exists a popular, industrial-grade hybrid logic/functional programming language in wide use: GHC Haskell.

At first glance, Haskell does not contain a logic programming language. However, if we enable enough GHC extensions, the typeclass system can be made *just* rich enough to encode a simply-typed logical framework. The key insight is that we can encode each production rule using multi-parameter type classes with a single method. Moreover, we can encode our constructive adequacy proofs for a given set of production rules as instances that translate each of the productions in the abstract grammar to productions in the syntax of an actual language.

As a concrete example, consider the grammar of the following simple imperative language.

$$
\begin{aligned}
\langle \mathit{expr} \rangle &::= \mathrm{x} \\
&\mid\ \mathrm{n} \\
&\mid\ \langle \mathit{expr} \rangle\ \text{`+'}\ \langle \mathit{expr} \rangle \\
&\mid\ \langle \mathit{expr} \rangle\ \text{`*'}\ \langle \mathit{expr} \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle \mathit{stmt} \rangle &::= \langle \mathit{var} \rangle\ \text{`='}\ \langle \mathit{expr} \rangle \\
&\mid\ \text{`}\mathtt{while}\text{'}\ \langle \mathit{expr} \rangle\ \text{`}\mathtt{do}\text{'}\ \langle \mathit{stmt} \rangle \\
&\mid\ \langle \mathit{stmt} \rangle\ \text{`;'}\ \langle \mathit{stmt} \rangle
\end{aligned}
$$

We can then encode this grammar with the following set of multi-parameter typeclasses:

Listing 1 An example tagless encoding.

```
class Var expr where
  var :: String → expr
```

```
class Lit expr where
  lit :: Int → expr

class Add expr where
  add :: expr → expr → expr

class Mul expr where
  mul :: expr → expr → expr

class Assign expr stmt where
  assign :: String → expr → stmt

class While expr stmt where
  while :: expr → stmt → stmt

class AndThen stmt where
  andThen :: stmt → stmt → stmt
```

This style of language encoding is typically known as the untyped variant of *finally tagless*[3], and is well-known technique. However, our encoding is a slight refinement of the usual tagless style. In particular, we restrict ourselves to a single class per production rule, whereas other tagless encodings often use a class per syntactic category. This more fine-grained approach allows us to encode grammatical constructs that are only supported by a subset of our target grammars; see section 6.2 for examples.

Unfortunately, the encoding above has some serious ergonomic issues. In particular, expressions like `assign "x" (lit 4)` will result in an unsolved metavariable for `expr`, as there may be multiple choices of `expr` to use when solving the `Assign ?expr stmt` constraint. Luckily, we can resolve ambiguities of this form through judicious use of functional dependencies[6], as demonstrated below.

Listing 2 A tagless encoding with functional dependencies.

```
class Assign expr stmt | stmt → expr where
  assign :: String → expr → stmt

class While expr stmt | stmt → expr where
  while :: expr → stmt → stmt
```

6.2 Designing The Language

Now that we've fleshed out how we are going to encode our grammatical framework into our host language, it's time to design our idealized abstract grammar. All of our target languages roughly agree on a subset of the grammar of non-binding terms: the main sources of divergence are binding forms and top-level definitions[6]. This is ultimately unsurprisingly: dependent type theories are fundamentally theories of binding and substitution, so we would expect some variation in how our target languages present the core of their underlying theories.

[6] As we shall see in section 6.2.2, the syntactical divergence of top-level definitions is essentially about binders as well.

This presents an interesting language design problem. Our idealized grammar will need to find some syntactic overlap between all of our four target languages. Additionally, we would also like for our solution to be (reasonably) extensible. Finding the core set of grammatical primitives to accomplish this task is surprisingly tricky, and requires a close analysis of fine structure of binding.

6.2.1 Binding Forms

As users of dependently typed languages are well aware, a binding form carries much more information than just a variable name and a type. Moreover, this extra information can have a large impact on typechecking performance, as is the case with implicit/visible arguments. To make matters worse, languages often offer multiple syntactic options for writing the same binding form, as is evidenced by the prevalence of multi-binders like $(x\ y\ z : A) \to B$. Though such binding forms are often equivalent to their single-binder counterparts as *abstract* syntax, they may have different performance characteristics, so we cannot simply lower them to a uniform single-binding representation. To account for these variations, we have designed a sub-language dedicated solely to binding forms. This language classifies the various binding features along three separate axes: binding arity, binding annotations, and binding modifiers.

Binding arities and annotations are relatively self-explanatory, and classify the number of names bound, along with the type of annotation allowed. Our target languages all have their binding arities falling into one of three classes: n-ary, unary, or nullary. We can similarly characterise annotations into three categories: required, optional, or forbidden.

This language of binders is encoded in the implementation as a single class `Binder` that is parameterised by higher-kinded types `arity :: Type -> Type` and `ann :: Type -> Type`, and provide standardized types for all three binding arities and annotations.

Listing 3 The basic binding constructs in Panbench.

```
class Binder arity nm ann tm cell | cell → nm tm where
  binder :: arity nm → ann tm → cell

-- | No annotation or arity.
data None nm = None

-- | A single annotation or singular arity.
newtype Single a = Single { unSingle :: a }

-- | Multi-binders.
type Multi = []

-- | Infix operator for an annotated binder with a single name.
(.:) :: (Binder Single nm Single tm cell) => nm → tm → cell
nm .: tp = binder (Single nm) (Single tp)

-- | Infix operator for an annotated binder.
(.:*) :: (Binder arity nm Single tm cell) => arity nm → tm → cell
nms .:* tp = binder nms (Single tp)
```

Production rules that involve binding forms are encoded as classes that are parametric over a notion of a binding cell, as demonstrated below.

Listing 4 The Panbench class for Π-types.

```
class Pi cell tm | tm → cell where
```

```
  pi :: [cell] → tm → tm
```

Decoupling the grammar of binding forms from the grammar of binders themselves allows us to be somewhat polymorphic over the language of binding forms when writing generators. This in turn means that we can potentially re-use generators when extending Panbench with new target grammars that may support only a subset of the binding features present in our four target grammars.

Binding modifiers, on the other hand, require a bit more explanation. A binding modifier captures features like implicit arguments, which do not change the number of names bound nor their annotations, but rather how those bound names get treated by the rest of the system. Currently, Panbench only supports visibility-related modifiers, but we have designed the system so that it is easy to extend with new modifiers; EG: quantities in Idris 2 or irrelevance annotations in Agda.

The language of binding modifiers is implemented as the following set of Haskell typeclasses.

Listing 5 Typeclasses for binding modifiers.

```
class Implicit cell where
  implicit :: cell → cell

class SemiImplicit cell where
  semiImplicit :: cell → cell
```

This is a case where the granular tagless encoding shines. Both Lean 4 and Rocq have a form of semi-implicits[7], whereas Idris 2 and Agda have no such notion. Decomposing the language of binding modifiers into granular pieces lets us write benchmarks that explicitly require support for features like semi-implicits. Had we used a monolithic class that encodes the entire language of modifiers, we would have to resort to runtime errors (or, even worse, dubious attempts at translation).

6.2.2 Top-Level Definitions

The question of top-level definitions is much thornier, and there seems to be less agreement on how they ought to be structured. Luckily, we can re-apply many of the lessons we learned in our treatment of binders; after all, definitions are "just" top-level binding forms! This perspective lets us simplify how we view some more baroque top-level bindings. As a contrived example, consider the following signature for a pair of top-level Agda definitions.

Listing 6 A complicated Agda signature.

```
private instance abstract @irr @mixed foo bar : Nat → _
```

In our language of binders, this definition consists of a 2-ary annotated binding of the names `foo`, `bar` that has had a sequence of binding modifiers applied to it.

Unfortunately, this insight does not offer a complete solution. Notably, our four target grammar differ significantly in how their treatment of type signatures. prioritize dependent pattern matching (EG: Agda, Idris 2) typically opt to have standalone type signatures: this allow for top-level pattern matches, which in turn makes it much easier to infer motives[7].

[7] We use the term *semi-implicit* argument to refer to an implicit argument that is not eagerly instantiated. In Rocq these are known as non-maximal implicits.

Conversely, languages oriented around tactics (EG: Lean 4, Rocq) typically opt for in-line type signatures and pattern-matching expressions. This appears to be largely independent of Miranda/ML syntax split: Lean 4 borrows large parts of its syntax from Haskell, yet still opts for in-line signatures.

This presents us with a design decision: should our idealized grammar use inline or standalone signatures? As long as we can (easily) translate from one style to the other, we have a genuine decision to make. We have opted for the former as standalone signatures offer variations that languages with inline signatures cannot handle. As a concrete example, consider the following Agda declaration:

Listing 7 A definition with mismatched names.

```
id : (A : Type) → A → A
id B x = x
```

In particular, note that we have bound first argument to a different name. Translating this to a corresponding Rocq declaration then forces us to choose to use either the name from the signature or the term. Conversely, using in-line signatures does not lead us to having to make an unforced choice when translating to a separate signature, as we can simply duplicate the name in both the signature and term.

However, inline signatures are not completely without fault, and cause some edge cases with binding modifiers. As an example, consider the following two variants of the identity function in Agda.

Listing 8 Two variants of the identity function.

```
id : {A : Type} → A → A
id x = x

id' : {A : Type} → A → A
id' {A} x = x
```

Both definitions mark the `A` argument as an implicit, but the second definition *also* binds it in the declaration. When we pass to inline type signatures, we lose this extra layer of distinction. To account for this, we were forced to refine the visibility modifier system to distinguish between "bound" and "unbound" modifiers. This extra distinction has not proved to be too onerous in practice, and we still believe that inline signatures are the correct choice for our application.

We have encoded this decision in our idealized grammar by introducing a notion of a "left-hand-side" of a definition, which consists of a collection of names to be defined, and a scope to define them under. This means that we view definitions like ??? not as functions $\mathrm{id} : (A : \mathrm{Type}) \to A \to A$ but rather as *bindings* $A : \mathrm{Type}, x : A \vdash \mathrm{id} : A$ in non-empty contexts. This shift in perspective has the added benefit of making the interface to other forms of parameterised definitions entirely uniform; for instance, a parameterised record is simply just a record with a non-empty left-hand side.

Reed: link to the previous listing

In Panbench, definitions and their corresponding left-hand sides are encoded via the following set of typeclasses.

Listing 9 Definitions and left-hand sides.

```
class Definition lhs tm defn | defn → lhs tm where
  (.=) :: lhs → tm → defn

class DataDefinition lhs ctor defn | defn → lhs ctor where
```

```
  data_ :: lhs → [ctor] → defn

class RecordDefinition lhs nm fld defn | defn → lhs nm fld where
  record_ :: lhs → name → [fld] → defn
```

7 Conclusion

References

1. February 2026. URL: https://github.com/GrammaticalFramework/informath.
2. Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a logical framework based on the $\lambda\pi$-calculus modulo theory. (arXiv:2311.07185), November 2023. arXiv:2311.07185 [cs]. URL: http://arxiv.org/abs/2311.07185, doi:10.48550/arXiv.2311.07185.
3. Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009. doi:10.1017/S0956796809007205.
4. Eelco Dolstra and The Nix contributors. Nix. URL: https://github.com/NixOS/nix.
5. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993. doi:10.1145/138027.138060.
6. Mark P. Jones. Type classes with functional dependencies. In Gert Smolka, editor, *Programming Languages and Systems*, page 230–244, Berlin, Heidelberg, 2000. Springer. doi:10.1007/3-540-46425-5_15.
7. Conor Mcbride and James Mckinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, January 2004. doi:10.1017/S0956796803004829.
8. Neil Mitchell. Shake before building: replacing make with haskell. *SIGPLAN Not.*, 47(9):55–66, 2012. doi:10.1145/2398856.2364538.
9. Aarne Ranta. *Grammatical framework: programming with multilingual grammars*. CSLI studies in computational linguistics. CSLI Publications, Center for the Study of Language and Information, Stanford, Calif, 2011.