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 Mathematics of computing → Mathematical software performance

Keywords and phrases Benchmarking, dependent types, testing

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 feature 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, which entails that other parts of these systems get comparatively less attention. This often results in easy-to-miss performance problems: a proof assistant developer told us that a naïve $O(n^2)$ fresh name generation algorithm used for pretty-printing resulted in 100x slowdowns in some pathological cases.

Thus the need for a benchmarking suite for these simpler components. 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.

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 [3] or Informath [1] that attempt to act as an intermediary between 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.
2. A series of compilers for that DSL to Agda [2], Idris 2 [4], Lean 4 [7], and Rocq [10].

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

Concretely, we see our contributions as 1) concrete benchmarks and analysis thereof, 2) the supporting extensible infrastructure, and 3) the design of the embedded Panbench DSL for system-agnostic tests. All three required extensive work, including many design iterations, before settling on what is here.

Our paper is structured as follows. We detail our methodology (Section 2), both for how we came up with our tests and the experimental setup. Actual results (Section 3) are given and analyzed. We then take a step back and look at the results more globally, and speculate on some of the reasons for the weaknesses we found. Section 5 documents the engineering of the infrastructure parts of Panbench (build system, test harness and report generator). The *language framework* and its design is described in Section 6. Lastly we conclude.

2 Methodology

To perform reliable benchmarking, we need tooling and we need to design a test suite. For reproducibility, we also need to document the actual experimental setup. The tooling will be described later, now we focus on the design of the test suite and the experimental setup.

Category	**Details**	**Category**	**Details**
Syntax	Newlines	Datatypes	Parameters
	Parentheses		Indices
Names	Datatype		Params + Indices
	Data constructor		Constructors
	Definition	Records	Fields
	Definition lhs		Parameters
	Definition rhs	Dependency	Record Fields
	Lambda		Datatypes Parameters
	Pi		Definitions chains
	Record		Record chains
	Record constructor	Nesting	Id chain
	Record field		Id chain λ
Binders	Lambda		Let
	Implicits		Let addition
Misc	Postulates		Let functions
Conversion	Addition		

2.1 Designing tests

Let us assume that we possess the following tools: a) a single language of tests, b) a means to translate these tests to each of our four languages, b) a reproducible test harness, and d) a stable installation of the four languages. How would we design actual tests for common features?

As we said before, our aim here is to test “basic engineering”. For example, how does each system handle giant files (e.g.: one million empty lines), large names (thousands of

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

characters long) in each syntactic category, large numbers of fields in records, constructors in algebraic types, large numbers of parameters or indices, long dependency chains, and so on. We divide our tests into the following categories:

1. basic syntactic capabilities
2. long names in simple structures
3. large simple structures (data, record)
4. handling of nesting and dependency
5. conversion, postulates

where we vary the “size” of each.

At a higher level, we asked ourselves the question: for each aspect of a dependently typed language (lexical structure, grammatical structure, fundamental features of dependent languages), what could be made to “scale”? The only features of interest remained the ones that were clearly features of all four systems.

Note that we are *not* trying to create “intrinsically interesting” tests, but rather we want them to be *revealing* with respect to the need for basic infrastructure improvements.

For lack of space, we cannot show samples of all the tests, but we show just a few that should be *illustrative* of the rest[2]. The source of all tests can be found in the accompanying material. All tests below are from the snapshot of our “golden” test suite, uniformly run with a size parameter of 5.

Listing 1 Nesting: Id chain (Agda)

```
module IdChain where

f : {a : Set} (x : a) → a
f x = x

test : {a : Set} → a → a
test = f f f f f f
```

Listing 2 Nesting: Let add (Lean)

```
def n : Nat :=
  let x0 := 1
  let x1 := x0 + x0
  let x2 := x1 + x1
  let x3 := x2 + x2
  let x4 := x3 + x3
  x4
```

Listing 3 Binders: Lambda (Idris 2)

```
module Main

const5 : {A : Type} -> {B0, B1, B2, B3, B4, B5 : Type} ->
  A -> B0 -> B1 -> B2 -> B3 -> B4 -> B5 -> A
const5 = \a, b0, b1, b2, b3, b4, b5 => a
```

Listing 4 Dependency: Record Telescope (Rocq)

```
Module RecordTelescope.

Record Telescope (U : Type) (El : U -> Type) : Type := Tele
  { a0 : U
  ; a1 : forall (x0 : El a0), U
  ; a2 : forall (x0 : El a0) (x1 : El (a1 x0)), U
  ; a3 : forall (x0 : El a0) (x1 : El (a1 x0)) (x2 : El (a2 x0 x1))
    , U
  ; a4 : forall (x0 : El a0) (x1 : El (a1 x0)) (x2 : El (a2 x0 x1))
    (x3 : El (a3 x0 x1 x2)), U
```

[2] Very minor edits made to Panbench output in the listings to fit the page width.

```
  ; a5 : forall (x0 : El a0) (x1 : El (a1 x0)) (x2 : El (a2 x0 x1))
    (x3 : El (a3 x0 x1 x2)) (x4 : El (a4 x0 x1 x2 x3)), U
  }.

Arguments Tele {U} {El} _ _ _ _ _ _.

End RecordTelescope.
```

2.2 Experimental setup

All tests are run on a dedicated desktop machine running on NixOS 25.11. The CPU is an Intel i7-2600K with 1MB L2 cache, 8MB L3 cache, running at 3.4 Ghz, equipped with 24 Gigs of DDR3 memory. This box has no SSD drive, but this should only affect system time. When tests are run, no other (human) activity happens on the machine.

All tests are run with a time limit of 60 seconds. We also tried to use memory limits but, alas, Linux no longer reliably supports these. We ran all systems in their default configuration.

3 Results

Given that our test suite has 31 tests, each of which produces 3 different graphs, we have no room to display all 93 resulting graphs. We thus choose results that appear to be the most "interesting". Furthermore, other than in Table 1, we will not show the *System Time* as it correlates very strongly with memory use for our particular test suite.

System	User Time (s)	System Time (s)	Max RSS (MB)
Agda	0.02 (0.002)	0.01 (0.001)	64 (0.1)
Idris 2	0.58 (0.007)	0.10 (0.007)	248 (0.1)
Lean 4	0.14 (0.005)	0.04 (0.005)	307 (0.6)
Rocq	0.05 (0.004)	0.03 (0.003)	95 (0.05)

Figure 1 Start-up time and memory, mean with standard deviation in parentheses.

Start-up time and memory use varies wildly: from a super-fast (0.02s) and slim (64 MB) Agda to a 29 times slower Idris 2 and 4.8 times memory consumer in Lean 4. It is worth recalling that 0.1 seconds is the "instant" threshold.

A file with a header and a million blank lines is not intrinsically interesting. It is however very *revealing*: we see (Figure 2) that it takes Lean 4 and Rocq no noticeable time to deal with that, while already 100K lines causes both Agda and Idris 2 to slow down and consume significantly more memory (6.3Gigabytes for 10 million lines in Idris 2's case). Estimating the run-time for Agda and Idris 2 from the slope of the graphs, we get that both are linear.

What about *long identifiers*? Figure 3 shows what happens when we use increasingly long names for data types; other "long names", such as for constructors, field names of records, etc, show similar behaviour. Unlike for blank lines, all systems show an eventual increase in time and memory use, with Agda starting earlier than others. What is most interesting is that Lean 4 barely takes any more memory, even for extremely long (4 million characters) identifiers. Agda's time here too indicates it is linear in the number of characters.

Figure 4 tests simple data types (enumerations) with short constructor names. What is remarkable here is Idris 2: even at 2^{11} constructors, it takes no appreciable time or memory beyond start-up.

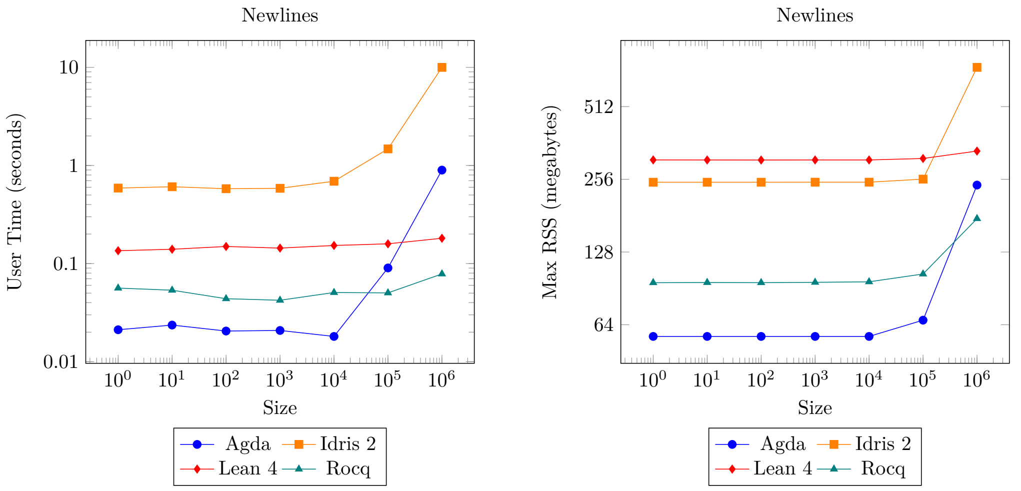

Figure 2 Blank lines

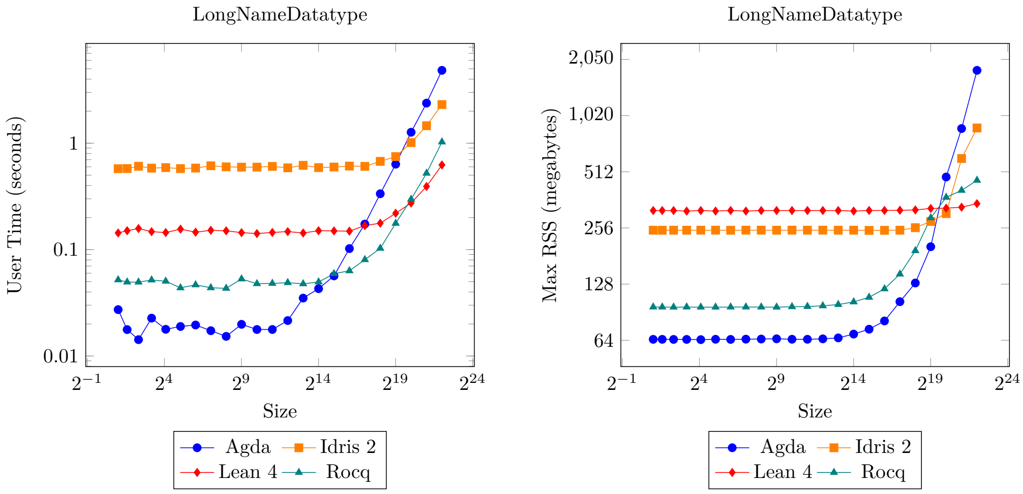

Figure 3 Long names (datatypes)

Figure 5, a test for a single lambda term that uses its first variable but has n other (unused) variables also declared (see Listing 3. This seems to be a “torture” test where systems perform very well until they hit a wall and suddenly time out. It appears that the underlying problem is using an enormous amount of memory.

Figure 6 is a test for “very dependent records”, i.e. a record where every later field depends on all previous ones, as shown in Listing 4. Of course, the code size itself is quadratic in the number of such fields. As expected, this rapidly takes quite a lot of time; less expected is the memory usage also goes up very significantly. This particular test likely needs finer sampling (and maybe larger timeouts) to understand the behaviour of each system.

Figure 7, corresponding to Listing 1, nested calls to an identity function, shows even more extreme behaviour: basically instantaneous until memory explosion and time out. This is well-known to be a problem for type systems based on Hindley-Milner inference, but it is less clear that it ought to be a weakness for bidirectional typing as well. Closer sampling (not shown) does not change this picture.

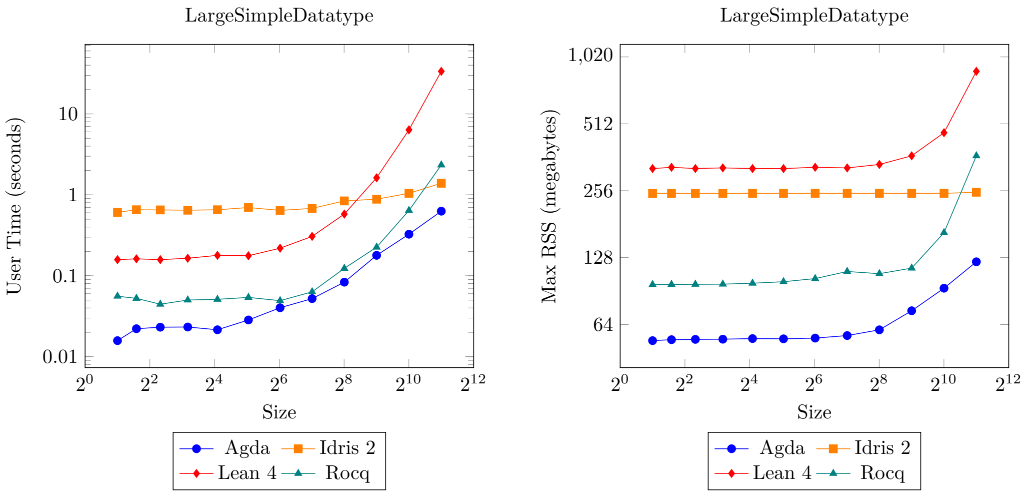

Figure 4 Enumerations

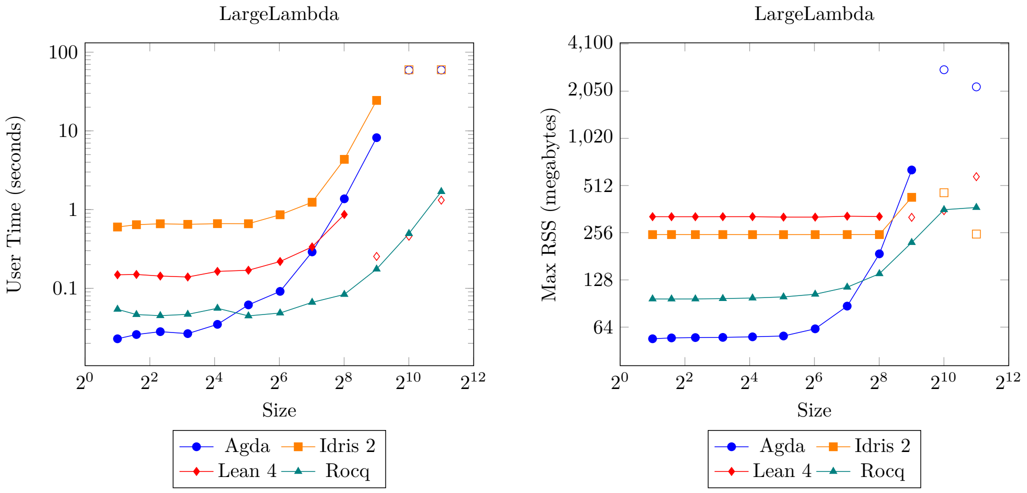

Figure 5 Lambda term with many variables

Figure 8, corresponding to Listing 2, nested lets doing very simple arithmetic. The contrast is remarkable: Agda and Idris 2 time out already at size 2^5, Lean 4 takes an increasing amount of time, while Rocq takes no time nor any memory! Note the enormous amount of memory taken by Agda when it times out.

Listing 5 Conversion: Addition (Idris)

```
conv : 5 + 5 + 5 + 5 + 5 + 0 = 25
conv = Refl
```

Figure 9, corresponding to Listing 5, does simple natural number arithmetic and ensures the correct result is obtained. Here the roles are inverted: Agda, Idris 2, and Lean 4 take no time while Rocq takes an increasing amount of time until timeout.

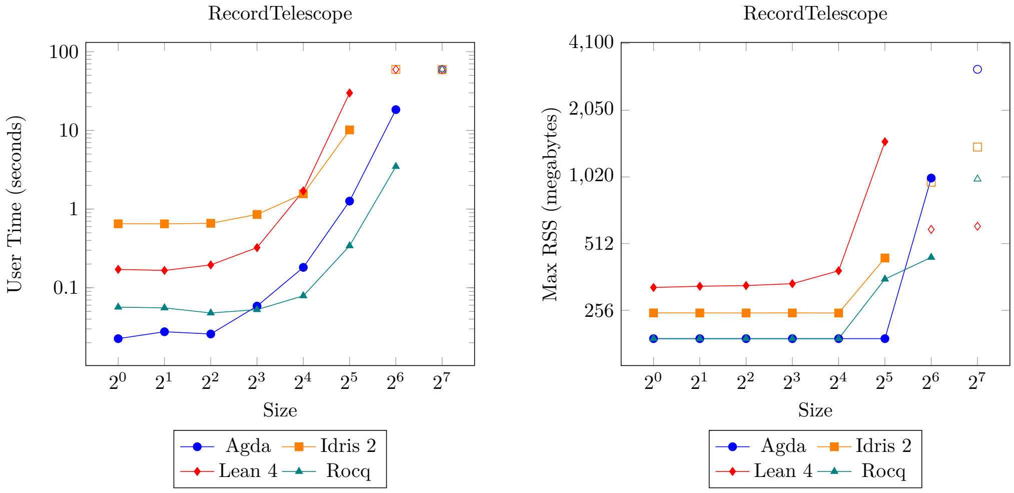

Figure 6 Record with increasingly dependent fields

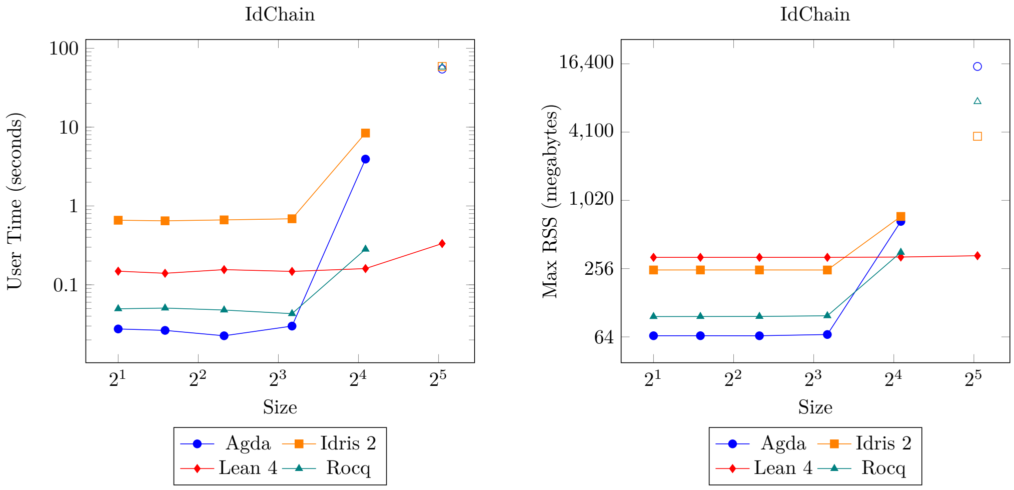

Figure 7 Chain of calls to identity function

4 Discussion

The previous section analyzed the results themselves. Here we speculate on why the results may be as they are. We have not (yet) verified any of our suspicions.

4.1 Agda

Most the results for Agda follow a general pattern: it is consistently the fastest and most memory efficient for small inputs, but has a couple of edge cases where it really struggles. Notably, Agda performs rather poorly on tests that put heavy pressure on the parser. This poor performance can be explained by the use of Haskell's `String` inside of Agda's lexer, which is a known performance pitfall.

Agda also struggles to typecheck let-bindings that introduce some non-trivial sharing. This is an unfortunate consequence of Agda's choice to inline let bindings during elaboration, which can easily result in exponential blowups if sharing is lost.

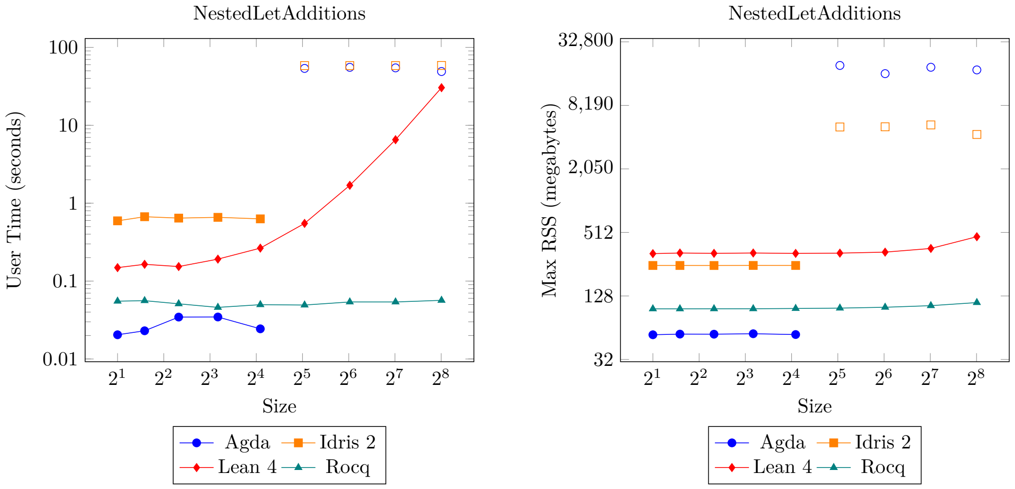

Figure 8 Nested let bindings, simple addition on rhs

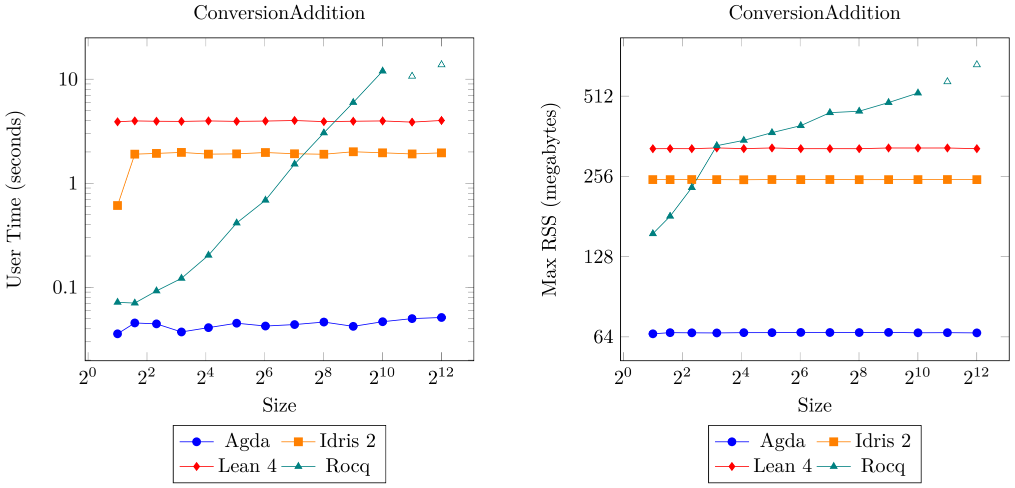

Figure 9 Conversion for Natural Number Addition

4.2 Idris 2

Unfortunately, Idris 2 struggles on a lot of these tests. It has the highest startup time of any systems we benchmarked, coming in at over half a second. We suspect the choice to use scheme as a runtime, which incurs a high interpreter startup cost.

Like Agda, Idris 2 struggles on benchmarks that stress-test the parser. Notably, it is the only system that times out when trying to parse files containing 10^7 newlines, and consumes over 6 gigabytes while doing so. We suspect Idris 2's `Text.Lexer` library. It also struggles with let-bindings, and times out when trying to elaborate a linear sequence of 1024 let bindings. This does not appear to have the same root cause as Agda's poor performance, and further investigation is required.

However, there are some bright spots. Idris 2 performs *exceptionally* well on all tests involving elaboration of datatypes and records once the startup time is accounted for.

4.3 Lean 4

Initially, we expected that Lean 4 would perform well. We were surprised by the number of time outs and hard crashes we encountered. Many tests involving records and datatypes hit hard limits for the number of fields, indices, or constructors. It also seems to struggle on elaboration tasks that involve large numbers of names: we suspect that this could be due to using a locally-nameless representation [6] internally.

On the bright side, Lean 4 does manage to elaborate the nested addition test, though it does seem to exhibit some exponential behaviour while doing so. It also is able to easily handle almost all of the parsing tests with ease, though it does stack overflow after 512 nested parenthesis. It was also the only system that was able to handle checking 32 iterated applications of the identity function, and did so with ease, taking approximately 0.25 seconds and a negligible amount of additional memory.

4.4 Rocq

Out of all of the systems we measured, Rocq was by far and away the winner. On most tests, it typically came in first or second place. Notably, it was the only system that was able to handle the nested addition test without exhibiting some sort of exponential blow-up. It also consistently outperforms other systems on parsing-heavy tasks, though it does stack overflow when presented with 16384 nested parenthesis.

However, there were still some surprises. Rocq was the only system that exhibited linear runtime on the addition conversion test: all other systems managed to run in essentially constant time. It also struggled to handle files that contained large numbers of very simple definitions, and took nearly 45 seconds to typecheck 4096 of them. We suspect that both of these may have to do with a sub-optimal approach to checking natural numbers.

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 reliably run them nor analyze their outcomes. We concluded that for ease of use and adoption, reliability and reproducibility, a language benchmarking system should:

1. Provide infrastructure for performing sandboxed builds of compilers from source.
2. Allow for multiple revisions of the same tool to be installed simultaneously.
3. Allow a single tool at a single version but with different build configurations.
4. Be able to be run locally on a developer's machine.
5. Present a declarative interface for creating benchmarking environments and running benchmarks.
6. Present performance results in a self-contained format that is easy to understand and share.

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 [13], 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 atop of Nix [8]. This is seemingly a perfect fit; after all, Nix was designed to facilitate almost exactly this use-case. However, we did not for the following reasons:

1. Nix does not work natively on Windows. Performance problems are often operating system specific, so ruling out a popular OS seems unwise[3].
2. Nix adds a barrier to adoption. Installing Nix is a somewhat invasive process, especially on MacOS[4]. We believe that it is unreasonable to ask developers to add users to their system and modify their root directory to run a benchmarking tool, and strongly suspect that this would hamper adoption.

Thus we opted to create our own Nix-inspired build system based atop Shake [13]. 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 one to 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 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 Shake database. This hash is used to invalidate downstream results, and also act 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 earlier, 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 at odds with one of the goals of a build system, which is to try

[3] Currently, Panbench does not support Windows, but this is an artifact of prioritization, and not a fundamental restriction.

[4] The situation is even worse on x86-64 Macs, which most Nix installers simply do not support.

to complete a build as fast as possible by using all available resources. 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 opt to eschew existing process interaction libraries, and implement the benchmark spawning code in C[5]. 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 report a reading of approximately 2 gigabytes halfway through a full benchmarking suite. We discovered the Linux preserves resource usage statistics across calls to `execve`. Consequentially, this means that we are unable to measure any max RSS that is lower than max RSS usage of Panbench itself. Luckily, our lowest baseline is Agda at 64 megs, and we managed to get the memory usage of Panbench itself down to approximately 10 megs via some careful optimization and GC tuning.

Currently, the statistics that Panbench gathers can then be rendered into standalone HTML files with `vega-lite` plots, or into TEX files containing PGF plots. We intend to add more visualization and statistic analysis tools as the need arises.

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, the core task of Panbench is not that different from that of a logical framework [9]: Both 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[6] rather than a logical one.

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 à la 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 that shows that their encoding of the judgements actually aligns with the language. However, mechanizing this adequacy proof would require a staged third layer consisting of a more traditional proof assistant.

[5] This is why Panbench does not currently support Windows.

[6] Not to be confused with *the* Grammatical Framework [14], which aims to be a more natural-language focused meta-framework for implementing and translating between grammars.

Transposing this skeleton design to grammatical constructs we will also obtain three layers:

1. A layer for defining grammars as 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. Given that this compiler outputs *concrete syntax*, it is implemented as a pretty-printer.

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.

⟨*expr*⟩ := x | n | ⟨*expr*⟩ '+' ⟨*expr*⟩

⟨*stmt*⟩ := ⟨*var*⟩ '=' ⟨*expr*⟩ | ⟨*stmt*⟩ ';' ⟨*stmt*⟩

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

Listing 6 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 Assign expr stmt where
  assign :: String → expr → stmt

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

This style of language encoding is typically known as the untyped variant of *finally tagless* [5]. However, our encoding is a slight refinement where we restrict ourselves to a single class per production rule. 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 [11], as demonstrated below.

Listing 7 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 on to design our idealized abstract grammar. 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[7]. This is ultimately unsurprising: 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.

This presents an interesting language design problem. Our idealized grammar will need to find "syntactic abstractions" common between 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 the fine structure of binding.

An analogy from linguistics might help contextualize the task: human languages vary on whether they are subject-verb-object (SVO) or SOV, amongst many other possibilities. This requires us to notice syntactic categories subject, object verb that are common to all, and that explicit renderings merely differ in the order. Similarly for singular and plural as modifiers. Our task is similar.

6.2.1 Binding Forms

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 further complicate matters, 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.

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

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 8 The core Panbench binder class.

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

Production rules involving binding forms are encoded as classes parametric over a notion of a binding cell.

Listing 9 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.

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 is designed to be extensible; e.g. quantities in Idris 2 or irrelevance annotations in Agda.

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

Listing 10 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[8], whereas Idris 2 and Agda do not. Decomposing the language of binding modifiers into granular pieces lets us write benchmarks that explicitly require support for features like semi-implicits. A monolithic class for the entire language of modifiers would have forced us into 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!

[8] 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.

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 11 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 they treat type signatures. Those that prioritize dependent pattern matching (e.g. 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 [12]. Conversely, languages oriented around tactics (e.g. Lean 4, Rocq) typically opt for in-line type signatures and pattern-matching expressions. This appears to be largely independent of the 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 12 A definition with mismatched names.

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

In particular, note that we have bound the 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. Using in-line signatures does not require this unforced choice.

However, inline signatures are not completely without fault, and cause some edge cases with binding modifiers. Consider the following two variants of the identity function.

Listing 13 Two variants of the identity function (Agda).

```
  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. Inline type signatures 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 Listing 13 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.

Listing 14 Encoding of 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
```

As an example, Listing 1 looks as show below in Panbench. We elide the monstrous type signature, which painfully documents all the language features used.

Listing 15 The IdChain test.

```
generator = GenModule "IdChain"
 [ ] \size →
 [ [unbound $ implicit ("a" .: builtin "Type"), "x" .: "a"]
            |- ("f" .: "a") .= "x"
 ,
 [unbound $ implicit ("a" .: builtin "Type")]
            |- "test" .: (None .:* "a" ‘arr‘ "a") .=
            foldr (\_ tm → app "f" [tm]) "f" [1..size]
 ]
```

7 Conclusion

This work started out as a final-year capstone project for a team of students. It showed that a single-source benchmark was possible, along with automated reporting. It also showed that the results of such benchmarking would be interesting. The current authors redesigned and rewrote the prototype into the current version.

Our benchmarking revealed multiple performance problems across all four systems. Luckily, most of our benchmarks had at least one system perform well. This suggests that the problems we uncovered should be relatively easy to fix, as maintainers can look to an existing implementation for inspiration.

Our infrastructure is open-source and we hope that others will be able to build on it. We believe that our *grammatical framework* is a useful abstraction, independently of benchmarking. We already have plans for it ourselves, for non-benchmarking applications.

We intend to extend both the Panbench benchmark suite and the supporting infrastructure. In particular, we are interested in adding more benchmarks that stress-test conversion and normalization performance. This would make the Panbench benchmark suite much more useful for the development of *new* proof assistants. Currently, the amount of work required to make an experimental proof assistant production-grade is measured in years, which is far too long. Being to test against an existing set of benchmarks that captures known issues would let the developers of these tools avoid falling into the same traps as their predecessors.

References

1 February 2026. URL: https://github.com/GrammaticalFramework/informath.

2 Agda Developers. Agda. URL: https://agda.readthedocs.io/.

3 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.

4 Edwin Brady. Idris 2: Quantitative Type Theory in Practice. pages 32460–33960, 2021. doi:10.4230/LIPICS.ECOOP.2021.9.

5 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.

6 Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, October 2012. doi:10.1007/s10817-011-9225-2.

7 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, page 625–635, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-79876-5_37.

8 Eelco Dolstra and The Nix contributors. Nix. URL: https://github.com/NixOS/nix.

9 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.

10 Hugo Herbelin, Pierre-Marie Pédrot, coqbot, Gaëtan Gilbert, Maxime Dénès, letouzey, Emilio Jesús Gallego Arias, Matthieu Sozeau, Théo Zimmermann, Enrico Tassi, Jean-Christophe Filliatre, Pierre Roux, Guillaume Melquiond, Arnaud Spiwack, Jason Gross, barras, Pierre Boutillier, Vincent Laporte, Jim Fehrle, Stéphane Glondu, Yves Bertot, Jasper Hugunin, Clément Pit-Claudel, Ali Caglayan, forestjulien, Pierre Courtieu, Frédéric Besson, Pierre Rousselin, MSoegtropIMC, and Yann Leray. Rocq, February 2026. doi:10.5281/zenodo.1003420.

11 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.

12 Conor Mcbride and James Mckinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, January 2004. doi:10.1017/S0956796803004829.

13 Neil Mitchell. Shake before building: replacing make with haskell. *SIGPLAN Not.*, 47(9):55–66, 2012. doi:10.1145/2398856.2364538.

14 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.