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

this itemized list should be expanded into actual text

- benchmark system engineering and scaling
- benchmarking of anything resembling proofs would be a major research project
- the languages (of types/expressions and of declarations) of all 4 are quite similar in their surface expressivity, even though all possess a myriad of specific features that are unshared

2 Methodology

This is really documenting the 'experiment'. The actual details of the thinking behind the design is in Section 5.

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'

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:7
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

3 Results

4 Discussion

5 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 [2]: 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[1] 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 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 proving 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.

5.1 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.

[1] Not to be confused with *the* Grammatical Framework [5], which aims to be a more general meta-framework for implementing and translating between grammars.

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.

$\langle expr \rangle$:= x
 | n
 | $\langle expr \rangle$ '+' $\langle expr \rangle$
 | $\langle expr \rangle$ '*' $\langle expr \rangle$

$\langle stmt \rangle$:= $\langle var \rangle$ '=' $\langle expr \rangle$
 | 'while' $\langle expr \rangle$ 'do' $\langle stmt \rangle$
 | $\langle stmt \rangle$ ';' $\langle stmt \rangle$

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

```
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*[1], 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 5.2 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[3], as demonstrated below.

```
class Assign expr stmt | stmt -> expr where
  assign :: String -> expr -> stmt

class While expr stmt | stmt -> expr where
  while :: expr -> stmt -> stmt
```

5.2 Designing The Language

Reed: Too informal, just brain dumping.

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, though their treatment of binding forms and top level definitions diverges.

5.2.1 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 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.

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

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

```
-- | Pi types.
class Pi cell tm | tm → cell where
  -- | Create a pi type over a list of binding cells.
  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 type-classes.

```
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 a form of semi-implicits[2], 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).

5.2.2 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. The largest source of divergence is type signatures. Languages that prioritize dependent pattern matching typically opt to have standalone type signatures: this allow for top-level pattern matches, which in turn makes it much easier to infer motives[4]. Conversely, languages oriented around tactics 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:

```
  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

[2] 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.

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.

```
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.

Unfortunately, we are not out of the woods yet: like binders, there are a plethora of different top-level definition forms. The key insight we had was that top level definitions *are binding forms*, and that both should share the same underlying language. This perspective lets us simplify how we view some more baroque top-level bindings. As a contrived example, consider the following Agda definition:

Reed: unicode :(

```
private instance abstract @irr @mixed foo bar : Nat -> _
foo _ = 0
bar zero = true
bar (suc _) = false
```

In the 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, along with a compound body which consists of a simple definition and a pattern-matching definition.

Reed: Present the panbench grammar in BNF

6 Conclusion

References

1 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.

2 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.

3 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.

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

5 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.