A Certified Interpreter for the List Machine Benchmark

Samuel Feitosa*
samuel.feitosa@ifsc.edu.br
Departamento de Informática
Caçador, Santa Catarina, Brazil

Rodrigo Ribeiro
rodrigo.ribeiro@ufop.edu.br
Prog. Pós Graduação em Ciência da Computação
Ouro Preto, Minas Gerais, Brazil

ABSTRACT

This is the abstract...

CCS CONCEPTS

• **Software and its engineering** → **Semantics**; **Interpreters**; • **Theory of computation** → **Type theory**.

KEYWORDS

Dependent types, formal semantics

ACM Reference Format:
Samuel Feitosa and Rodrigo Ribeiro. 2020. A Certified Interpreter for the List Machine Benchmark. In *SBLP '20: Brazilian Symposium on Programming Languages, October 19–23, 2020, Natal, Brazil.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

2 AN OVERVIEW OF AGDA

Agda is a dependently-typed functional programming language based on Martin-Löf intuitionistic type theory [2]. Function types and an infinite hierarchy of types of types, Set l, where l is a natural number, are built-in. Everything else is a user-defined type. The type Set, also known as Set$_0$, is the type of all "small" types, such as Bool, String and List Bool. The type Set$_1$ is the type of Set and "others like it", such as Set → Bool, String → Set, and Set → Set. We have that Set l is an element of the type Set ($l + 1$), for every $l \geqslant 0$. This stratification of types is used to keep Agda consistent as a logical theory [4].

An ordinary (non-dependent) function type is written $A \rightarrow B$ and a dependent one is written $(x : A) \rightarrow B$, where type B depends on x, or $\forall\ (x : A) \rightarrow B$. Agda allows the definition of *implicit parameters*, i.e., parameters whose values can be inferred from the context, by surrounding them in curly braces: $\forall\ \{x : A\} \rightarrow B$. To avoid clutter, we'll omit implicit arguments from the source code presentation. The reader can safely assume that every free variable in a type is an implicit parameter.

As an example of Agda code, consider the following data type of length-indexed lists, also known as vectors.

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ

data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

Constructor [] builds empty vectors. The cons-operator (_::_) inserts a new element in front of a vector of n elements (of type Vec A n) and returns a value of type Vec A (suc n). The Vec datatype is an example of a dependent type, i.e., a type that uses a value (that denotes its length). The usefulness of dependent types can be illustrated with the definition of a safe list head function: head can be defined to accept only non-empty vectors, i.e., values of type Vec A (suc n).

```
head : Vec A (suc n) → A
head (x :: xs) = x
```

In head's definition, constructor [] is not used. The Agda type-checker can figure out, from head's parameter type, that argument [] to head is not type-correct.

Another useful data type is the finite type, Fin[1], which is defined in Agda's standard library as:

```
data Fin : ℕ → Set where
  zero : ∀ {n} → Fin (suc n)
  suc : ∀ {n} → Fin n → Fin (suc n)
```

Type Fin n has exactly n inhabitants (elements), i.e., it is isomorphic to the set $\{0, \ldots, n-1\}$. An application of such type is to define a safe vector lookup function, which avoids the access of invalid positions.

```
lookup : ∀ {A n} → Fin n → Vec A n → A
lookup zero (x :: _) = x
lookup (suc idx) (_ :: xs) = lookup idx xs
```

Thanks to the propositions-as-types principle,[2] we can interpret types as logical formulas and terms as proofs. An example is the representation of equality as the following Agda type:

```
data _≡_ {l} {A : Set l} (x : A) : A → Set where
  refl : x ≡ x
```

This type is called propositional equality. It defines that there is a unique evidence for equality, constructor refl (for reflexivity), that asserts that the only value equal to x is itself. Given a predicate $P : A \rightarrow$ Set and a vector xs, the type All P xs is used to build proofs that P holds for all elements in xs and it is defined as:

[1] Note that Agda supports the overloading of data type constructor names. Constructor zero can refer to type ℕ or Fin, depending on the context where the name is used.
[2] It is also known as Curry-Howard "isomorphism" [4].

SBLP '20, October 19–23, 2020, Natal, Brazil

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
data All (P : A → Set) : Vec A n → Set where
  [] : All P []
  _ :: _ : ∀ {x xs} → P x → All P xs → All P (x :: xs)
```

The first constructor specifies that All P holds for the empty vector and constructor _ :: _ builds a proof of All P (x :: xs) from proofs of P x and All P xs. Since this type has the same structure of vectors, some functions on Vec have similar definitions for type All. As an example used in our formalization, consider the function lookup, which extracts a proof of P for the element at position v :: Fin n in a Vec:

```
lookup : {xs : Vec A n} → Fin n → All P xs → P x
lookup zero (px :: _) = px
lookup (suc idx) (_ :: pxs) = lookup idx pxs
```

An important application of dependent types is to encode programming languages syntax. The role of dependent types in this domain is to encode programs that only allow well-typed and well-scoped terms [1]. Intuitively, we encode the static semantics of the object language in the host language AST's constructor, leaving the responsibility of checking type safety to the host's language type checker. As an example, consider the following simple expression language.

```
data Expr : Set where
  True False : Expr
  Num : ℕ → Expr
  _ ∧ _ _ + _ : Expr → Expr → Expr
```

Using this data type,[3] we can construct expressions that denote terms that should not be considered well-typed like (Num *1*) + True. Using this approach, we need to specify the static semantics as another definition, which should consider all possible cases to avoid the definition of ill-typed terms.

A better approach is to combine the static semantics and language syntax into a single definition, as shown below.

```
data Ty : Set where
  Nat Bool : Ty
data Expr : Ty → Set where
  True False : Expr Bool
  Num : Nat → Expr Nat
  _ ∧ _ : Expr Bool → Expr Bool → Expr Bool
  _ + _ : Expr Nat → Expr Nat → Expr Nat
```

In this definition, the Expr type is indexed by a value of type Ty which indicates the type of the expression being built. In this approach, Agda's type system can enforce that only well-typed terms could be written. Agda's type checker will automatically reject a definition which uses the expression (Num *1*) + True.

For further information about Agda, see [3, 5].

REFERENCES

[1] Nick Benton, Chung-Kil Hur, Andrew J. Kennedy, and Conor Mcbride. 2012. Strongly Typed Term Representations in Coq. *J. Autom. Reason.* 49, 2 (Aug. 2012), 141–159. https://doi.org/10.1007/s10817-011-9219-0

[2] Per Martin-Löf. 1998. An intuitionistic theory of types. In *Twenty-five years of constructive type theory (Venice, 1995).* Oxford Logic Guides, Vol. 36. Oxford Univ. Press, New York, 127–172.

[3] Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09).* ACM, New York, NY, USA, 1–2. https://doi.org/10.1145/1481861.1481862

[4] Morten Heine Sørensen and Pawel Urzyczyn. 2006. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics).* Elsevier Science Inc., New York, NY, USA.

[5] Aaron Stump. 2016. *Verified Functional Programming in Agda.* Association for Computing Machinery and Morgan; Claypool, New York, NY, USA.

[3] Agda supports the definition of mixfix operators. We can use underscores to mark arguments positions.