A Brief Overview of Agda –
A Functional Language with Dependent Types

Ana Bove, Peter Dybjer, and Ulf Norell

e-mail: {`bove,peterd,ulfn`}`@chalmers.se`
Chalmers University of Technology, Gothenburg, Sweden

Abstract. We give an overview of Agda, the latest in a series of dependently typed programming languages developed in Gothenburg. Agda is based on Martin-Löf's intuitionistic type theory but extends it with numerous programming language features. It supports a wide range of inductive data types, including inductive families and inductive-recursive types, with associated flexible pattern-matching. Unlike other proof assistants, Agda is not tactic-based. Instead it has an Emacs-based interface which allows programming by gradual refinement of incomplete type-correct terms.

1   Introduction

A dependently typed programming language and proof assistant. Agda is a functional programming language with dependent types. It is an extension of Martin-Löf's intuitionistic type theory [12, 13] with numerous features which are useful for practical programming. Agda is also a proof assistant. By the Curry-Howard identification, we can represent logical propositions by types. A proposition is proved by writing a program of the corresponding type. However, Agda is primarily being developed as a programming language and not as a proof assistant.

Agda is the latest in a series of implementations of intensional type theory which have been developed in Gothenburg (beginning with the ALF-system) since 1990. The current version (Agda 2) has been designed and implemented by Ulf Norell and is a complete redesign of the original Agda system. Like its predecessors, the current Agda supports a wide range of inductive data types, pattern matching, termination checking, and comes with an interface for programming and proving by direct manipulation of proof terms. On the other hand, the new Agda goes beyond the earlier systems in several respects: flexibility of pattern-matching, more powerful module system, flexible and attractive concrete syntax (using unicode), etc.

A system for functional programmers. A programmer familiar with a standard functional language such as Haskell or OCaml will find it easy to get started with Agda. Like in ordinary functional languages, programming (and proving) consists of defining data types and recursive functions. Moreover, users familiar with Haskell's generalised algebraic data types (GADTs) will find it easy to use Agda's inductive families [5].

The Agda wiki. More information about Agda can be found on the Agda wiki [1]. There are tutorials [15, 3], a guide to editing, type checking, and compiling Agda code, a link to the standard library, and much else. There is also a link to Norell's PhD thesis [14] with a language definition and detailed discussions of the features of Agda.

2 Agda Features

We begin by listing the logically significant parts of Agda.

Logical framework. The core of Agda is Martin-Löf's logical framework [13] which gives us the type Set and dependent function types $(x : A) \to B$ (using Agda's syntax). Agda's logical framework also provides record types and a countable sequence of larger universes $\mathsf{Set} = \mathsf{Set}_0, \mathsf{Set}_1, \mathsf{Set}_2, \ldots$.

Data type definitions. Agda supports a rich family of strictly positive inductive and inductive-recursive data types and families. Agda checks that the data type definitions are well-formed according to a discipline similar to that in [6, 7].

Recursive function definitions. One of Agda's main features is its flexible pattern matching for inductive families. A coverage checker makes sure the patterns cover all possible cases. As in Martin-Löf type theory, all functions definable in Agda must terminate, which is ensured by the termination checker.

Codata. The current version of Agda also provides coinductive data types. This feature is however somewhat experimental and not yet stable.

Agda also provides several features to make it useful in practice:

Concrete syntax. The concrete syntax of Agda is much inspired by Haskell, but also contains a few distinctive features such as mixfix operators and full support for unicode identifiers and keywords.

Implicit arguments. The mechanism for implicit arguments allows the omission of parts of the programs that can be inferred by the typechecker.

Module system. Agda's module system supports separate compilation and allows parametrised modules. Together with Agda's record types, the module system provides a powerful mechanism for structuring larger developments.

Compilation. There is a simple compiler that compiles Agda programs via Haskell and allows Haskell functions to be called from within Agda.

Emacs interface. Using Agda's Emacs interface, programs can be developed incrementally, leaving parts of the program unfinished. By type checking the unfinished program, the programmer can get useful information on how to fill in the missing parts. The Emacs interface also provides syntax highlighting and code navigation facilities.

3 Agda and some Related Languages

Agda and Martin-Löf type theory. Agda is an extension of Martin-Löf's type theory. An implementation of the latter in Agda can be found on the Agda wiki [1]. Meaning explanations of foundational interest for type theory have been provided by Martin-Löf [11, 12], and all constructions in Agda (except codata) are intended to satisfy them. Agda is thus a predicative theory.

Agda and Coq. The most well-known system with dependent types which is based on the Curry-Howard identification is Coq [2]. Coq is an implementation of the Calculus of Inductive Constructions, an extension of the Calculus of Constructions [4] with inductive (but not inductive-recursive) types. Unlike Agda, Coq has an impredicative universe Prop. Moreover, for the purpose of program extraction, there is a distinction between Prop and Set in Coq which is not present in Agda. There are many other differences between Agda and Coq. For example, Agda's pattern matching for inductive families is more flexible than Coq's. On the other hand, Coq supports tactical theorem proving in the tradition of LCF [10], but Agda does not.

Agda and Haskell. Haskell has GADTs, a feature which mimics inductive families by representing them by type-indexed types. A fundamental difference is that Haskell allows partial general recursive functions and non-strictly positive data types. Hence, logic cannot be obtained by the Curry-Howard correspondence.

Other languages with dependent types. There are nowadays a number of functional languages with dependent types (some with and some without general recursion). Among these McBride's Epigram [8] is closest in spirit to Agda.

4 Example: Equational Proofs in Commutative Monoids

We will now show some of the code for a module which decides equality in commutative monoids. This is an example of reflection, a technique which makes it possible to program and use efficient verified decision procedure inside the system. Reflection was for example used extensively by Gonthier in his proof of the four colour theorem [9].

An example of a commutative monoid is the natural numbers with addition. Thus our decision procedure can automatically prove arithmetic equations such as

```
∀ n m → (n + m) + n ≡ m + (n + n).
```

The above is a valid type in Agda syntax. To prove it in Agda we create a file `Example.agda` with the following content:

```
module Example where

open import Data.Nat
open import Relation.Binary.PropositionalEquality
```

```
prf : ∀ n m → (n + m) + n ≡ m + (n + n)
prf n m = ?
```

Natural numbers and propositional equality are imported from the standard library and opened to make their content available. Finally, we declare a proof object `prf`, the type of which represents the proposition to be proved; here $\forall\, x \to B$ is an abbreviation of $(x : A) \to B$ which does not explicitly mention the argument type. The final line is the incomplete definition of `prf`: it is a function of two arguments, but we do not yet know how to build a proof of the equation so we leave a "?" in the right hand side. The "?" is a placeholder that can be stepwise refined to obtain a complete proof.

In this way we can manually build a proof of the equation from associativity and commutativity of +, and basic properties of equality which can be found in the standard library. Manual equational reasoning however can become tedious for complex equations. We shall therefore write a general procedure for equational reasoning in commutative monoids, and show how to use it for proving the equation above.

Decision procedure for commutative monoids. First we define monoid expressions as an inductive family indexed by the number of variables:

```
data Expr n : Set where
  var  : Fin n → Expr n
  _⊕_  : Expr n → Expr n → Expr n
  zero : Expr n
```

`Fin n` is a finite set with `n` elements; there are at most `n` variables. Note that infix (and mixfix) operators are declared by using underscores to indicate where the arguments should go.

To decide whether two monoid expressions are equal we normalise them and compare the results. The normalisation function is

```
norm : ∀ {n} → Expr n → Expr n
```

Note that the first argument (the number of variables) is enclosed in braces, which signifies that it is implicit. To define this function we employ normalisation by evaluation, that is, we first interpret the expressions in a domain of "values", and then reify these values into normal expressions. Below, we omit the definitions of `eval` and `reify` and give only their types:

```
norm = reify ∘ eval

eval  : ∀ {n} → Expr n → NF n
reify : ∀ {n} → NF n → Expr n
```

The values in `NF n` are vectors recording the number of occurrences of each variable:

```
NF : ℕ → Set
NF n = Vec ℕ n
```

Next we define the type of equations between monoid expressions:

```
data Eqn n : Set where
  _==_ : Expr n → Expr n → Eqn n
```

We can define our arithmetic equation above as follows:

```
eqn₁ : Eqn 2
eqn₁ = build 2  λ a b → a ⊕ b ⊕ a == b ⊕ (a ⊕ a)
```

where we have used an auxiliary function `build` which builds an equation in `Eqn n` from an `n`-place curried function by applying it to variables.

Equations will be proved by normalising both sides:

```
simpl : ∀ {n} → Eqn n → Eqn n
simpl (e₁ == e₂) = norm e₁ == norm e₂
```

We are now ready to define a general decision procedure for arbitrary commutative monoids (the complete definition is given later):

```
prove : ∀ {n} (eqn : Eqn n) ρ → Prf (simpl eqn) ρ → Prf eqn ρ
```

The function takes an equation and an environment in which to interpret it, and builds a proof of the equation given a proof of its normal form. The definition of `Prf` will be given below.

We can instantiate this procedure to the commutative monoid of natural numbers and apply it to our equation, an environment with the two variables, and a proof of the normalised equation. Since the two sides of the equation will be equal after normalisation we prove it by reflexivity:

```
prf : ∀ n m → (n + m) + n ≡ m + (n + n)
prf n m = prove eqn₁ (n :: m :: []) ≡-refl
```

The `prove` function is defined in a module `Semantics` which is parametrised over an arbitrary commutative monoid

```
module Semantics (CM : CommutativeMonoid) where

  open CommutativeMonoid CM renaming (carrier to C)
```

Opening the `CommutativeMonoid` module brings into scope the carrier `C` with its equality relation `_≈_` and the monoid operations `_●_` and ε. A monoid expression is interpreted as a function from an environment containing values for the variables to an element of `C`.

```
Env : ℕ → Set
Env n = Vec C n

[[_]] : ∀ {n} → Expr n → Env n → C
[[ var i ]] ρ = lookup i ρ
[[ a ⊕ b ]] ρ = [[ a ]] ρ ● [[ b ]] ρ
[[ zero  ]] ρ = ε
```

Equations are also interpreted:

```
Prf : ∀ {n} → Eqn n → Env n → Set
Prf (e₁ == e₂) ρ = ⟦ e₁ ⟧ ρ ≈ ⟦ e₂ ⟧ ρ
```

One can prove that the normalisation function is sound in the sense that the normal form is semantically equal to the original expression in any environment:

```
sound : ∀ {n} (e : Expr n) ρ → ⟦ e ⟧ ρ ≈ ⟦ norm e ⟧ ρ
```

Hence, to prove an equation it suffices to prove the normalised version. The proof uses the module `Relation.Binary.EqReasoning` from the standard library for the equational reasoning:

```
prove : ∀ {n} (eqn : Eqn n) ρ → Prf (simpl eqn) ρ → Prf eqn ρ
prove (e₁ == e₂) ρ prf =
  begin ⟦ e₁ ⟧ ρ          ≈⟨ sound e₁ ρ ⟩
        ⟦ norm e₁ ⟧ ρ   ≈⟨ prf ⟩
        ⟦ norm e₂ ⟧ ρ   ≈⟨ sym (sound e₂ ρ) ⟩
        ⟦ e₂ ⟧ ρ
  □
```

The complete code is available on the Agda wiki [1].

References

1. Agda wiki page. `http://wiki.portal.chalmers.se/agda/`
2. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Develpment. Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.
3. A. Bove and P. Dybjer. Dependent types at work. In L. Barbosa, A. Bove, A. Pardo, and J. S. Pinto, editors, *LerNet ALFA Summer School 2008,*, LNCS 5520, to appear, pages 57–99. Springer, 2009.
4. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
5. P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1994.
6. P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.
7. P. Dybjer and A. Setzer. Indexed induction-recursion. *Journal of Logic and Algebraic Programming*, 66(1):1–49, January 2006.
8. Epigram homepage. `www.e-pig.org`
9. G. Gonthier. The four colour theorem: Engineering of a formal proof. In *ASCM*, page 333, 2007.
10. M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*. Springer-Verlag, LNCS 70, 1979.
11. P. Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.
12. P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.
13. B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.
14. U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology, 2007.
15. U. Norell. Dependently typed programming in Agda. In *Lecture Notes from the Summer School in Advanced Functional Programming*, 2008. To appear.