

Agda (programming language)

Agda is a dependently typed functional programming language originally developed by Ulf Norell at Chalmers University of Technology with implementation described in his PhD thesis.^[2] The original Agda system was developed at Chalmers by Catarina Coquand in 1999.^[3] The current version, originally known as Agda 2, is a full rewrite, which should be considered a new language that shares a name and tradition.

Agda is also a proof assistant based on the propositions-as-types paradigm, but unlike Coq, has no separate tactics language, and proofs are written in a functional programming style. The language has ordinary programming constructs such as data types, pattern matching, records, let expressions and modules, and a Haskell-like syntax. The system has Emacs and Atom interfaces^{[4][5]} but can also be run in batch mode from the command line.


Agda is based on Zhaohui Luo's unified theory of dependent types (UTT),^[6] a type theory similar to Martin-Löf type theory.

Agda is named after the Swedish song "Hönan Agda", written by Cornelis Vreeswijk,^[7] which is about a hen named Agda. This alludes to the naming of Coq.

<h2>Contents</h2>
<h3>Features</h3>
<ul style="list-style-type: none"><u>Inductive types</u><u>Dependently typed pattern matching</u><u>Metavariables</u><u>Proof automation</u><u>Termination checking</u><u>Standard library</u><u>Unicode</u><u>Backends</u>
<h3>See also</h3>
<h3>References</h3>
<h3>External links</h3>

Features

Agda

	
Paradigm	<u>Functional</u>
Designed by	Ulf Norell; Catarina Coquand (1.0)
Developer	Ulf Norell; Catarina Coquand (1.0)
First appeared	2007 (1.0 in 1999)
Stable release	2.6.2 / June 19, 2021
Typing discipline	<u>strong</u> , <u>static</u> , <u>dependent</u> , <u>nominal</u> , <u>manifest</u> , <u>inferred</u>
Implementation language	<u>Haskell</u>
OS	<u>Cross-platform</u>
License	<u>BSD-like</u> ^[1]
Filename extensions	<u>.agda</u> , <u>.lagda</u> , <u>.lagda.md</u> , <u>.lagda.rst</u> , <u>.lagda.tex</u>
Website	<u>wiki.portal</u> <u>.chalmers.se</u> <u>/agda</u> (<u>http://wiki.portal.chalmers.se/agda</u>)
Influenced by	
<u>Coq</u> , <u>Epigram</u> , <u>Haskell</u>	
Influenced	
<u>Idris</u>	

Inductive types

The main way of defining data types in Agda is via inductive data types which are similar to algebraic data types in non-dependently typed programming languages.

Here is a definition of Peano numbers in Agda:

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

Basically, it means that there are two ways to construct a value of type \mathbb{N} , representing a natural number. To begin, `zero` is a natural number, and if `n` is a natural number, then `suc n`, standing for the successor of `n`, is a natural number too.

Here is a definition of the "less than or equal" relation between two natural numbers:

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : {n : ℕ} → zero ≤ n
  s≤s : {n m : ℕ} → n ≤ m → suc n ≤ suc m
```

The first constructor, `z≤n`, corresponds to the axiom that zero is less than or equal to any natural number. The second constructor, `s≤s`, corresponds to an inference rule, allowing to turn a proof of $n \leq m$ into a proof of $\text{suc } n \leq \text{suc } m$.^[8] So the value `s≤s {zero} {suc zero} (z≤n {suc zero})` is a proof that one (the successor of zero), is less than or equal to two (the successor of one). The parameters provided in curly brackets may be omitted if they can be inferred.

Dependently typed pattern matching

In core type theory, induction and recursion principles are used to prove theorems about inductive types. In Agda, dependently typed pattern matching is used instead. For example, natural number addition can be defined like this:

```
add zero n = n
add (suc m) n = suc (add m n)
```

This way of writing recursive functions/inductive proofs is more natural than applying raw induction principles. In Agda, dependently typed pattern matching is a primitive of the language; the core language lacks the induction/recursion principles that pattern matching translates to.

Metavariables

One of the distinctive features of Agda, when compared with other similar systems such as Coq, is heavy reliance on metavariables for program construction. For example, one can write functions like this in Agda:

```
add : ℕ → ℕ → ℕ
add x y = ?
```

? here is a metavariable. When interacting with the system in emacs mode, it will show the user expected type and allow them to refine the metavariable, i.e., to replace it with more detailed code. This feature allows incremental program construction in a way similar to tactics-based proof assistants such as Coq.

Proof automation

Programming in pure type theory involves a lot of tedious and repetitive proofs. Although Agda has no separate tactics language, it is possible to program useful tactics within Agda itself. Typically, this works by writing an Agda function that optionally returns a proof of some property of interest. A tactic is then constructed by running this function at type-checking time, for example using the following auxiliary definitions:

```
data Maybe (A : Set) : Set where
  Just : A → Maybe A
  Nothing : Maybe A

data isJust {A : Set} : Maybe A → Set where
  auto : ∀ {x} → isJust (Just x)

Tactic : ∀ {A : Set} (x : Maybe A) → isJust x → A
Tactic Nothing ()
Tactic (Just x) auto = x
```

Given a function `check-even : (n : ℕ) → Maybe (Even n)` that inputs a number and optionally returns a proof of its evenness, a tactic can then be constructed as follows:

```
check-even-tactic : {n : ℕ} → isJust (check-even n) → Even n
check-even-tactic {n} = Tactic (check-even n)

lemma0 : Even zero
lemma0 = check-even-tactic auto

lemma2 : Even (suc (suc zero))
lemma2 = check-even-tactic auto
```

The actual proof of each lemma will be automatically constructed at type-checking time. If the tactic fails, type-checking will fail.

Additionally, to write more complex tactics, Agda has support for automation via reflection. The reflection mechanism allows one to quote program fragments into – or unquote them from – the abstract syntax tree. The way reflection is used is similar to the way Template Haskell works.^[9]

Another mechanism for proof automation is proof search action in emacs mode. It enumerates possible proof terms (limited to 5 seconds), and if one of the terms fits the specification, it will be put in the meta variable where the action is invoked. This action accepts hints, e.g., which theorems and from which modules can be used, whether the action can use pattern matching, etc.^[10]

Termination checking

Agda is a total language, i.e., each program in it must terminate and all possible patterns must be matched. Without this feature, the logic behind the language becomes inconsistent, and it becomes possible to prove arbitrary statements. For termination checking, Agda uses the approach of the Foetus termination checker.^[11]

Standard library

Agda has an extensive de facto standard library, which includes many useful definitions and theorems about basic data structures, such as natural numbers, lists, and vectors. The library is in beta, and is under active development.

Unicode

One of the more notable features of Agda is a heavy reliance on Unicode in program source code. The standard emacs mode uses shortcuts for input, such as `\Sigma` for Σ .

Backends

There are two compiler backends, MAonzo for Haskell and one for JavaScript.

See also

- Coq
- HOL (proof assistant)
- Idris (programming language)
- Isabelle (proof assistant)

References

1. Agda license file (<http://code.haskell.org/Agda/LICENSE>)
2. Ulf Norell. Towards a practical programming language based on dependent type theory. PhD Thesis. Chalmers University of Technology, 2007. [1] (<http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf>)
3. "Agda: An Interactive Proof Editor" (<https://web.archive.org/web/20111008115843/http://ocvs.cfv.jp/Agda/index.html>). Archived from [the original](http://ocvs.cfv.jp/Agda/index.html) (<http://ocvs.cfv.jp/Agda/index.html>) on 2011-10-08. Retrieved 2014-10-20.
4. Coquand, Catarina; Synek, Dan; Takeyama, Makoto. *An Emacs interface for type directed support constructing proofs and programs* (https://web.archive.org/web/20110722063632/https://mailserver.di.unipi.it/ricerca/proceedings/ETAPS05/uitp/uitp_p05.pdf) (PDF). European Joint Conferences on Theory and Practice of Software 2005. Archived from [the original](http://mailserver.di.unipi.it/ricerca/proceedings/ETAPS05/uitp/uitp_p05.pdf) (http://mailserver.di.unipi.it/ricerca/proceedings/ETAPS05/uitp/uitp_p05.pdf) (PDF) on 2011-07-22.
5. "agda-mode on Atom" (<https://atom.io/packages/agda-mode>). Retrieved 7 April 2017.
6. Luo, Zhaohui. *Computation and reasoning: a type theory for computer science*. Oxford University Press, Inc., 1994.
7. "[Agda] origin of \"Agda\"? (Agda mailing list)" (<https://lists.chalmers.se/pipermail/agda/2016/008867.html>). Retrieved 24 Oct 2020.
8. "Nat from Agda standard library" (<https://github.com/agda/agda-stdlib/blob/master/src/Data/Nat.agda>). *GitHub*. Retrieved 2014-07-20.
9. Van Der Walt, Paul, and Wouter Swierstra. "Engineering proof by reflection in Agda." In *Implementation and Application of Functional Languages*, pp. 157-173. Springer Berlin Heidelberg, 2013. [2] (<http://hal.inria.fr/docs/00/98/76/10/PDF/ReflectionProofs.pdf>)

10. Kokke, Pepijn, and Wouter Swierstra. "Auto in Agda." (<http://www.staff.science.uu.nl/~swier004/publications/2015-mpc.pdf>)
11. Abel, Andreas. "foetus – Termination checker for simple functional programs." *Programming Lab Report* 474 (1998). [3] (<http://www.tcs.informatik.uni-muenchen.de/~abel/foetus.pdf>)

External links

- [Official website](http://wiki.portal.chalmers.se/agda) (<http://wiki.portal.chalmers.se/agda>)
 - [Dependently Typed Programming in Agda](http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf) (<http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf>), by [Ulf Norell](#)
 - [A Brief Overview of Agda](http://www.cse.chalmers.se/~ulfn/papers/tphols09/tutorial.pdf) (<http://www.cse.chalmers.se/~ulfn/papers/tphols09/tutorial.pdf>), by Ana Bove, Peter Dybjer, and [Ulf Norell](#)
 - [Introduction to Agda](https://www.youtube.com/playlist?p=B7F836675DCE009C) (<https://www.youtube.com/playlist?p=B7F836675DCE009C>), a five-part YouTube playlist by Daniel Peebles
 - [Brutal \[Meta\]Introduction to Dependent Types in Agda](http://oxij.org/note/BrutalDepTypes/) (<http://oxij.org/note/BrutalDepTypes/>)
 - [Agda Tutorial: "explore programming in Agda without theoretical background"](http://people.inf.elte.hu/divip/AgdaTutorial/Index.html) (<http://people.inf.elte.hu/divip/AgdaTutorial/Index.html>)
-

Retrieved from "[https://en.wikipedia.org/w/index.php?title=Agda_\(programming_language\)&oldid=1067138752](https://en.wikipedia.org/w/index.php?title=Agda_(programming_language)&oldid=1067138752)"

This page was last edited on 21 January 2022, at 23:22 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.