Advanced functional programming

Agda I – Introduction

Paige Randall North

13 March 2024

Utrecht University

Background

Haskell vs Agda

Haskell:

- simply-typed lambda calculus
- with support for partial functions / nontermination (`undefined`)

Haskell vs Agda

Haskell:

- simply-typed lambda calculus
- with support for partial functions / nontermination (`undefined`)

Agda:

- **dependently-typed** lambda calculus
- with requirement that every function is total / every computation terminates

Similar languages

Theoretical:

- Martin-Löf type theory (1970s)
- Calculus of constructions (1980s)
- ...

Implemented:

- Coq (1989)
- Agda (2007)
- Lean (2013)
- ...

Proof assistants

Agda (and Coq, Lean) is not only a language but a *proof assistant* (also called an *interactive theorem prover*).

That means that the editor includes more assistance for writing a program/proof than usual.

Dependent types

Dependent types

In a simply typed language like Haskell, the input to functions can only be terms (in Haskell, values).

I.e.: functions go from types to types.

```
not ::  bool -> bool
```

Dependent types

In a simply typed language like Haskell, the input to functions can only be terms (in Haskell, values).

I.e.: functions go from types to types.

```
not :: bool -> bool
```

There are also *kinds* in Haskell.

```
[] :: * -> *
```

But Haskell keeps kinds and types *separate*.

Dependent types

In a dependent type theory, we do not keep kinds and types separate.[1]

Kinds are also types, and so they can be combined.

```
divisors :: Nat -> *
```

[1] Technically, there is a universe hierarchy to prevent paradoxes.

Dependent types

In a dependent type theory, we do not keep kinds and types separate.[1]

Kinds are also types, and so they can be combined.

```
divisors :  Nat -> Set
```

(In Agda, we write `Set` instead of `*` and `:` instead of `::`.)

There is a lot of effort to add this functionality to Haskell, e.g. `DataKinds`.

[1] Technically, there is a universe hierarchy to prevent paradoxes.

An important dependent type

We can consider the function

```
f:  Set -> Set
f A = A -> A -> Set
```

If we want to consider a 'polymorphic' term of this type, i.e. a function that takes in an `A : Set` and returns an `A -> A -> Set`, we write it as:

```
?  :  (A : Set) -> f A
```

or

```
?  :  (A : Set) -> A -> A -> Set
```

This is called a *dependent* function.

An important dependent type

My favorite dependent type (defined inductively) is

```
_≡_ :  (A : Set) -> A -> A -> Set
```

Now we can write the type of functors in Agda as

```
record Fun :  Set where
  field
    F0 :  Set -> Set
    F1 :  {A B : Set} -> {A -> B} -> F0 A -> F0 B
    id_law :  {A : Set} -> F1 (id A) ≡ id (F1 A)
    comp_law :  {A B C : Set} -> {f :  A -> B} -> {g :  B -> C}
               -> F1 g.f ≡ (F1 g).(F1 f)
```

A term of `Fun` is then a functor *satisfying the functor laws.*

Why dependent types

In general, types correspond to program specifications.[2]

A term is a program meeting the specification.

[2]See Curry-Howard correspondence, Brouwer–Heyting–Kolmogorov interpretation, proofs-as-programs.

Why dependent types

In general, types correspond to program specifications.[2]

A term is a program meeting the specification.

Dependent types allow us to write programs that are *correct-by-construction*.

[2] See Curry-Howard correspondence, Brouwer–Heyting–Kolmogorov interpretation, proofs-as-programs.

Program extraction

Once you have a term of type Fun, you can *extract* the underlying program (i.e., erase the correctness proofs).

You can extract Agda programs/proofs to Haskell or JavaScript.

Termination

Termination

Every program is Agda terminates.

Equivalently, every function is total.

This is because our programs are often proofs

e.g.: `id_law : {A : Set} -> F1 (id A) ≡ id (F1 A)`

and we expect `id_law` to be defined at each `A : Set`.

Simulating nontermination

We can of course *simulate* nontermination by using the `Maybe` monad (or more purpose-built solutions), as in Haskell.

Useful info

Useful info

Installation

- Official Agda installation instructions
- I am using `VS Code` with the `agda-mode` extension and its Agda language server.

Resources

- Dependently Typed Programming in Agda by Ulf Norell and James Chapman
- CS410 Advanced Functional Programming at Strathclyde, by Fredrik Nordvall Forsberg

Questions?