

Introducing Well-founded Recursion

Eric Mertens

Galois Inc

Tech Talk: June 15, 2010

This Talk

- 1 Introduction to Agda
- 2 Implementing Quicksort
- 3 Defining Relations
- 4 Building a better recursion
- 5 Back to Implementing Quicksort

The Agda Programming Language

- Agda is a dependently typed functional programming language.
- Agda is a proof assistant based on intuitionistic type theory.
- Agda has similar syntax to Haskell.

Programs as Proofs

- Types are propositions.

Programs as Proofs

- Types are propositions.
- Inhabited types are true propositions.

```
data  $\top$  : Set where unit :  $\top$ 
```

Programs as Proofs

- Types are propositions.
- Inhabited types are true propositions.

data \top : Set **where** unit : \top

- Uninhabited types are false propositions.

data \perp : Set **where**

Programs as Proofs

- Types are propositions.
- Inhabited types are true propositions.

data \top : Set **where** unit : \top

- Uninhabited types are false propositions.

data \perp : Set **where**

- Implementations are proofs.

The Price of Consistency

- Programs must be total.

The Price of Consistency

- Programs must be total.

$$\text{head} : \text{List } (1 \equiv 2) \rightarrow (1 \equiv 2)$$
$$\text{head } (x :: xs) = x$$

The Price of Consistency

- Programs must be total.

$$\begin{aligned} \text{head} &: \text{List } (1 \equiv 2) \rightarrow (1 \equiv 2) \\ \text{head } (x :: xs) &= x \end{aligned}$$

- Programs must terminate.

The Price of Consistency

- Programs must be total.

$$\begin{aligned} \text{head} &: \text{List } (1 \equiv 2) \rightarrow (1 \equiv 2) \\ \text{head } (x :: xs) &= x \end{aligned}$$

- Programs must terminate.

$$\begin{aligned} \text{absurd} &: \perp \\ \text{absurd} &= \text{absurd} \end{aligned}$$

The Termination Checker

- Agda uses a termination checking algorithm when loading source files.
- Structural recursion is supported by the termination checker.
- Structural recursion allows us to make recursive calls on a structurally smaller argument.

The Termination Checker

- Agda uses a termination checking algorithm when loading source files.
- Structural recursion is supported by the termination checker.
- Structural recursion allows us to make recursive calls on a structurally smaller argument.
- Example

$$\text{foldl} : \{a\ b : \text{Set}\} \rightarrow \text{List } a \rightarrow (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow b$$

$$\text{foldl } []\ f\ z = z$$

$$\text{foldl } (x :: xs)\ f\ z = \text{foldl } xs\ f\ (f\ z\ x)$$

The Termination Checker

- Agda uses a termination checking algorithm when loading source files.
- Structural recursion is supported by the termination checker.
- Structural recursion allows us to make recursive calls on a structurally smaller argument.
- Example

$$\text{foldl} : \{a\ b : \text{Set}\} \rightarrow \text{List } a \rightarrow (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow b$$

$$\text{foldl } []\ f\ z = z$$

$$\text{foldl } (x :: xs)\ f\ z = \text{foldl } xs\ f\ (f\ z\ x)$$

- But not all recursive functions are structurally recursive...

Additional Termination Checker Features

- Agda can find termination orders across mutually recursive functions.

Additional Termination Checker Features

- Agda can find termination orders across mutually recursive functions.
- Agda can find lexicographic termination orders.

Additional Termination Checker Features

- Agda can find termination orders across mutually recursive functions.
- Agda can find lexicographic termination orders.
- Example

$$\text{ack} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$$
$$\text{ack } 0 \ n = 1$$
$$\text{ack } (\text{suc } m) \ 0 = \text{ack } m \ 1$$
$$\text{ack } (\text{suc } m) \ (\text{suc } n) = \text{ack } m \ (\text{ack } (\text{suc } m) \ n)$$

Additional Termination Checker Features

- Agda can find termination orders across mutually recursive functions.
- Agda can find lexicographic termination orders.
- Example

$$\text{ack} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$$
$$\text{ack } 0 \ n = 1$$
$$\text{ack } (\text{suc } m) \ 0 = \text{ack } m \ 1$$
$$\text{ack } (\text{suc } m) \ (\text{suc } n) = \text{ack } m \ (\text{ack } (\text{suc } m) \ n)$$

- Is this enough?

- 1 Introduction to Agda
- 2 Implementing Quicksort**
- 3 Defining Relations
- 4 Building a better recursion
- 5 Back to Implementing Quicksort

Quicksort in Haskell

- Naïve functional-programming quicksort

```
quicksort :: (a -> a -> Bool) -> List a -> List a
```

```
quicksort p [] = []
```

```
quicksort p (x : xs) = case partition (p x) xs of  
  (small, big) -> small' ++ [x] ++ big'
```

where

```
  small' = quicksort p small
```

```
  big' = quicksort p big
```

Quicksort in Haskell

- Naïve functional-programming quicksort

```
quicksort :: (a -> a -> Bool) -> List a -> List a
```

```
quicksort p [] = []
```

```
quicksort p (x : xs) = case partition (p x) xs of
  (small, big) -> small' ++ [x] ++ big'
```

where

```
  small' = quicksort p small
```

```
  big' = quicksort p big
```

- Why does this terminate (on finite lists)?

Quicksort in Agda

- Naïve functional-programming quicksort in Agda

```
quicksort : {A : Set} → (A → A → Bool) → List A → List A
```

```
quicksort p [] = []
```

```
quicksort p (x :: xs) with partition (p x) xs
```

```
... | small, big = small' ++ [x] ++ big'
```

where

```
small' = quicksort p small
```

```
big'   = quicksort p big
```

Quicksort in Agda

- Naïve functional-programming quicksort in Agda

```
quicksort : {A : Set} → (A → A → Bool) → List A → List A
```

```
quicksort p [] = []
```

```
quicksort p (x :: xs) with partition (p x) xs
```

```
... | small, big = small' ++ [x] ++ big'
```

where

```
small' = quicksort p small
```

```
big'   = quicksort p big
```

- The termination checker fails on this function.

Well-founded Recursion

- Recursive calls are on "smaller" arguments.
- No infinite descending chains of recursive calls; we always reach a smallest element.
- We can capture this relation between elements in Agda.

Well-founded Relations

- A relation is well-founded if it contains no infinite descending chains.

Well-founded Relations

- A relation is well-founded if it contains no infinite descending chains.
- Example: "Less than" on natural numbers ($<$).

Well-founded Relations

- A relation is well-founded if it contains no infinite descending chains.
- Example: "Less than" on natural numbers ($<$).
- $0 < 1 < 2 < 5 < 8 < 10$

Well-founded Relations

- A relation is well-founded if it contains no infinite descending chains.
- Example: "Less than" on natural numbers ($<$).
- $0 < 1 < 2 < 5 < 8 < 10$
- Example: "Proper subset" on sets (\subset).

Well-founded Relations

- A relation is well-founded if it contains no infinite descending chains.
- Example: "Less than" on natural numbers ($<$).
- $0 < 1 < 2 < 5 < 8 < 10$
- Example: "Proper subset" on sets (\subset).
- $\{\} \subset \{2\} \subset \{1,2,5\} \subset \{1,2,3,4,5\}$

Well-founded Relations

- A relation is well-founded if it contains no infinite descending chains.
- Example: "Less than" on natural numbers ($<$).
- $0 < 1 < 2 < 5 < 8 < 10$
- Example: "Proper subset" on sets (\subset).
- $\{\} \subset \{2\} \subset \{1,2,5\} \subset \{1,2,3,4,5\}$
- False example: "Less than or equal to" on natural numbers (\leq).

Well-founded Relations

- A relation is well-founded if it contains no infinite descending chains.
- Example: "Less than" on natural numbers ($<$).
- $0 < 1 < 2 < 5 < 8 < 10$
- Example: "Proper subset" on sets (\subset).
- $\{\} \subset \{2\} \subset \{1,2,5\} \subset \{1,2,3,4,5\}$
- False example: "Less than or equal to" on natural numbers (\leq).
- $\dots \leq 42 \leq 42 \leq 42$

- 1 Introduction to Agda
- 2 Implementing Quicksort
- 3 Defining Relations**
- 4 Building a better recursion
- 5 Back to Implementing Quicksort

Relations in Agda

- Characterizing relations:

$$\text{Rel} : \text{Set} \rightarrow \text{Set}_1$$
$$\text{Rel } A = A \rightarrow A \rightarrow \text{Set}$$

Relations in Agda

- Characterizing relations:

$$\text{Rel} : \text{Set} \rightarrow \text{Set}_1$$

$$\text{Rel } A = A \rightarrow A \rightarrow \text{Set}$$

- Defining a new relation (inductively defined type family):

data `_<_` (m : \mathbb{N}) : $\mathbb{N} \rightarrow \text{Set}$ **where**

`<-base` : $m < \text{suc } m$

`<-step` : $\{n : \mathbb{N}\} \rightarrow m < n \rightarrow m < \text{suc } n$

Relations in Agda

- Characterizing relations:

$$\text{Rel} : \text{Set} \rightarrow \text{Set}_1$$

$$\text{Rel } A = A \rightarrow A \rightarrow \text{Set}$$

- Defining a new relation (inductively defined type family):

data $_ < _$ (m : \mathbb{N}) : $\mathbb{N} \rightarrow \text{Set}$ **where**

<-base : m < suc m

<-step : {n : \mathbb{N} } \rightarrow m < n \rightarrow m < suc n

- Defining a negation:

$$_ \not< _ : \text{Rel } \mathbb{N}$$

$$a \not< b = a < b \rightarrow \perp$$

Examples using relations

- Inhabiting a relation:

example₁ : 3 < 5

example₁ = ?

Goal: 3 < 5

Examples using relations

- Inhabiting a relation:

example₁ : 3 < 5

example₁ = <-step ?

Goal: 3 < 4

Examples using relations

- Inhabiting a relation:

example₁ : 3 < 5

example₁ = <-step <-base

Done

Examples using relations

- Inhabiting a relation:

example₁ : 3 < 5

example₁ = <-step <-base

- Uninhabited relation:

example₂ : 5 $\not<$ 2

example₂ x

x : 5 < 2

Examples using relations

- Inhabiting a relation:

example₁ : 3 < 5

example₁ = <-step <-base

- Uninhabited relation:

example₂ : 5 $\not<$ 2

example₂ (<-step x)

x : 5 < 1

Examples using relations

- Inhabiting a relation:

example₁ : 3 < 5

example₁ = <-step <-base

- Uninhabited relation:

example₂ : 5 $\not<$ 2

example₂ (<-step (<-step x))

x : 5 < 0

Examples using relations

- Inhabiting a relation:

example₁ : 3 < 5

example₁ = <-step <-base

- Uninhabited relation:

example₂ : 5 $\not<$ 2

example₂ (<-step (<-step ()))

Done

- 1 Introduction to Agda
- 2 Implementing Quicksort
- 3 Defining Relations
- 4 Building a better recursion**
- 5 Back to Implementing Quicksort

Accessibility

- We need a way to encode what it means for a relation to be well-founded.

Accessibility

- We need a way to encode what it means for a relation to be well-founded.
- An element, x , of a set, A , is "accessible" with respect to a relation, $<$, iff all of the elements, y , are less-than x are accessible.

```

module WF {A : Set} (_<_ : Rel A) where
  data Acc (x : A) : Set where
    acc : ( $\forall$  y  $\rightarrow$  y < x  $\rightarrow$  Acc y)  $\rightarrow$  Acc x
  
```

Accessibility

- We need a way to encode what it means for a relation to be well-founded.
- An element, x , of a set, A , is "accessible" with respect to a relation, $<$, iff all of the elements, y , are less-than x are accessible.
- A relation is well-founded iff all elements in the set are accessible.

```

module WF { A : Set } ( _ <_ : Rel A ) where
  data Acc ( x : A ) : Set where
    acc : (  $\forall$  y  $\rightarrow$  y < x  $\rightarrow$  Acc y )  $\rightarrow$  Acc x
  
```

Well-founded : Set

Well-founded = \forall x \rightarrow Acc x

The less-than relation on \mathbb{N} is well-founded

- We implement this with structural recursion.

$<-N\text{-wf}$: Well-founded $_ < _$
 $<-N\text{-wf } x = \text{acc } (\text{aux } x)$

where

$\text{aux} : \forall x y \rightarrow y < x \rightarrow \text{Acc } _ < _ y$

$\text{aux} . (\text{suc } y) y < \text{-base} = <-N\text{-wf } y$

$\text{aux} . (\text{suc } x) y (< \text{-step } \{x\} y < x) = \text{aux } x y y < x$

The less-than relation on \mathbb{N} is well-founded

- We implement this with structural recursion.

$\text{<-}\mathbb{N}\text{-wf} : \text{Well-founded } _ < _$
 $\text{<-}\mathbb{N}\text{-wf } x = \text{acc } (\text{aux } x)$

where

$\text{aux} : \forall x y \rightarrow y < x \rightarrow \text{Acc } _ < _ y$
 $\text{aux} . (\text{suc } y) y \text{ <-base} = \text{<-}\mathbb{N}\text{-wf } y$
 $\text{aux} . (\text{suc } x) y (\text{<-step } \{x\} y < x) = \text{aux } x y y < x$

- This is rather complicated; let's reuse it!

Building new well-founded relations

- Let's build new well-founded relations from old ones.

Building new well-founded relations

- Let's build new well-founded relations from old ones.

module Inverse-image-Well-founded {A B}

(_<_ : Rel B) (f : A → B) **where**

< : Rel A

ii-acc : $\forall \{x\} \rightarrow \text{Acc } _<_ (f\ x) \rightarrow \text{Acc } _<_ x$

ii-wf : Well-founded _<_ → Well-founded _<_

Building new well-founded relations

- Let's build new well-founded relations from old ones.

module Inverse-image-Well-founded {A B}

(_<_ : Rel B) (f : A → B) **where**

< : Rel A

$x < y = f\ x < f\ y$

ii-acc : $\forall \{x\} \rightarrow \text{Acc } _<_ (f\ x) \rightarrow \text{Acc } _<_ x$

ii-wf : Well-founded _<_ → Well-founded _<_

Building new well-founded relations

- Let's build new well-founded relations from old ones.

module Inverse-image-Well-founded {A B}

(_<_ : Rel B) (f : A → B) **where**

< : Rel A

$x < y = f\ x < f\ y$

ii-acc : $\forall \{x\} \rightarrow \text{Acc } _<_ (f\ x) \rightarrow \text{Acc } _<_ x$

ii-acc (acc g) = acc ($\lambda\ y\ f\ y < f\ x \rightarrow$ ii-acc (g (f y) f y < f x))

ii-wf : Well-founded _<_ → Well-founded _<_

Building new well-founded relations

- Let's build new well-founded relations from old ones.

module Inverse-image-Well-founded {A B}

(_<_ : Rel B) (f : A → B) **where**

< : Rel A

$x < y = f\ x < f\ y$

ii-acc : $\forall \{x\} \rightarrow \text{Acc } _<_ (f\ x) \rightarrow \text{Acc } _<_ x$

ii-acc (acc g) = acc ($\lambda y\ fy < fx \rightarrow$ ii-acc (g (f y) fy < fx))

ii-wf : Well-founded $_<_ \rightarrow$ Well-founded $_<_$

ii-wf wf x = ii-acc (wf (f x))

Shorter lists

- Let's instantiate this module for our quicksort case.
- $_ < _ : \text{Rel } \mathbb{N}$
- $\text{length} : \text{List } A \rightarrow \mathbb{N}$
- $_ \prec _ : \text{Rel } (\text{List } A)$

```

module <-on-length-Well-founded { A } where
  open Inverse-image-Well-founded { List A }  $\_ < \_$  length public
  wf : Well-founded  $\_ \prec \_$ 
  wf = ii-wf <- $\mathbb{N}$ -wf

```

- 1 Introduction to Agda
- 2 Implementing Quicksort
- 3 Defining Relations
- 4 Building a better recursion
- 5 Back to Implementing Quicksort**

Partitioned lists don't grow!

- Now we show that partition does not make lists longer.

module PartitionLemma {A} **where**

$_ \rightsquigarrow _ : \text{Rel (List A)}$

$x \rightsquigarrow y = \text{length } x < (1 + \text{length } y)$

partition-size : (p : A → Bool) (xs : List A)

→ proj₁ (partition p xs) \rightsquigarrow xs

× proj₂ (partition p xs) \rightsquigarrow xs

Partitioned lists don't grow!

- Now we show that partition does not make lists longer.

module PartitionLemma {A} **where**

$_ \preccurlyeq _ : \text{Rel (List A)}$

$x \preccurlyeq y = \text{length } x < (1 + \text{length } y)$

partition-size : (p : A → Bool) (xs : List A)

→ proj₁ (partition p xs) \preccurlyeq xs

× proj₂ (partition p xs) \preccurlyeq xs

partition-size p [] = <-base, <-base

partition-size p (x :: xs)

with p x | partition p xs | partition-size p xs

... | true | as, bs | as-size, bs-size = s<s as-size, <-step bs-size

... | false | as, bs | as-size, bs-size = <-step as-size, s<s bs-size

Quicksort take two

```

module Quick {A} (p : A → A → Bool) where
open <-on-length-Well-founded; open PartitionLemma
quicksort' : (xs : List A)                → List A

```

```

quicksort : List A → List A
quicksort xs = quicksort' xs

```

Quicksort take two

```

module Quick {A} (p : A → A → Bool) where
open <-on-length-Well-founded; open PartitionLemma
quicksort' : (xs : List A)                → List A
quicksort' [] = []
quicksort' (x :: xs)
  with partition (p x) xs
... | small, big                    = small' ++ [x] ++ big'
  where
    small' = quicksort' small
    big'   = quicksort' big

quicksort : List A → List A
quicksort xs = quicksort' xs

```

Quicksort take two

```

module Quick {A} (p : A → A → Bool) where
open <-on-length-Well-founded; open PartitionLemma
quicksort' : (xs : List A) → Acc _<_ xs → List A
quicksort' [] _ = []
quicksort' (x :: xs) (acc g)
  with partition (p x) xs | partition-size (p x) xs
... | small, big | small-size, big-size = small' ++ [x] ++ big'
  where
    small' = quicksort' small (g small small-size)
    big'   = quicksort' big (g big big-size)

quicksort : List A → List A
quicksort xs = quicksort' xs (wf xs)

```

Other techniques for building well-founded relations

- The subrelation of a well-founded relation is well-founded.
- The transitive closure of a well-founded relation is well-founded.
- The lexicographic product of a well-founded relation is well-founded.