

# Agda Backends: A survey and a UHC backend prototype

Author: Philipp Hausmann

`<p.hausmann@students.uu.nl>`

Supervisors: Wouter Swierstra `<w.s.swierstra.uu.nl>`

Atze Dijkstra `<atze@uu.nl>`

Department of Information and Computing Sciences  
Utrecht University

November 27, 2014

# Table of Contents

## 1 Agda Introduction

## 2 Existing Backends

- MAlonzo backend
- JS backend
- Epic backend
- Optimizations

## 3 UHC Backend

# Agda Introduction

- Why dependent types?

# Agda Introduction

- Why dependent types?
- **head** :: forall a . **List** a -> a  
  **head** (x:xs) = x  
  **head** [] = **error** "something went wrong..."

# Agda Introduction

- Why dependent types?
- **head** :: forall a . **List** a -> a  
  **head** (x:xs) = x  
  **head** [] = **error** "something\_went\_wrong..."
- Runtime crashes are possible in Haskell!

# Agda Introduction

- How to make sure at compile time that this doesn't happen?
- We need to encode the length of lists in the type

# Agda Introduction

- How to make sure at compile time that this doesn't happen?
- We need to encode the length of lists in the type

```
data Nat : Set where  
  zero : Nat  
  succ : Nat → Nat
```

# Agda Introduction

- How to make sure at compile time that this doesn't happen?
- We need to encode the length of lists in the type

```
data Nat : Set where
```

```
  zero : Nat
```

```
  succ : Nat → Nat
```

```
data Vec : (A : Set) → (n : Nat) → Set where
```

```
  nil : ∀ {A} → Vec A zero
```

```
  cons : ∀ {A n} → A → Vec A n → Vec A (succ n)
```



# Cont.

We can now write the head function in Agda

`head1` :  $\forall \{A\} n \rightarrow \text{Vec } A\ n \rightarrow A$

`head1` (`cons` `x` `xs`) = `x`

`head1` `nil` = ????

# Cont.

We can now write the head function in Agda

`head1` :  $\forall \{A\} n \rightarrow \text{Vec } A \ n \rightarrow A$

`head1` (`cons` `x` `xs`) = `x`

`head1` `nil` = ????

This will not type check!

## Cont.

We can now write the head function in Agda

`head1` :  $\forall \{A\} n \rightarrow \text{Vec } A \ n \rightarrow A$

`head1` (`cons` `x` `xs`) = `x`

`head1` `nil` = ????

This will not type check!

`head2` :  $\forall \{A\} n \rightarrow \text{Vec } A \ (\text{succ } n) \rightarrow A$

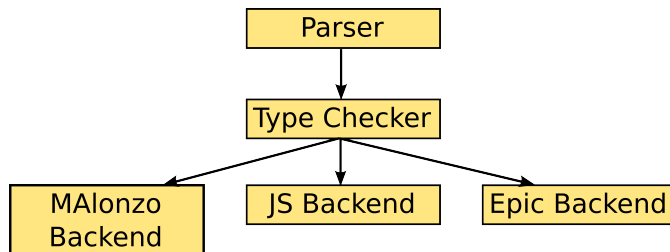
`head2` (`cons` `x` `xs`) = `x`

The typechecker now knows that the nil-case cannot happen!

# Agda Characteristics

- Values can be used as types
- Types cannot influence value of an expression
- Functions need to be total
- Single language for programs, specifications and proofs
- Typechecking requires evaluation

# Agda Architecture



## MAlonzo backend

# MAlonzo backend

- Targets Haskell
- Maintained
- Relies on GHC for optimizations

# MAlonzo - Code Generation

```
vecToStr : ∀ {A m} → (A → String)
           → Vec A m → String
vecToStr f [] = ""
vecToStr f (x :: xs) = ", " ++ ((f x)
                                ++ (vecToStr f xs))
```



# MAlonzo - Code Generation

```
d55 v0 v1 v2 v3
= MAlonzo.RTE. mazCoerce
  (d_1_55 (MAlonzo.RTE. mazCoerce v0)
    (MAlonzo.RTE. mazCoerce v1)
    (MAlonzo.RTE. mazCoerce v2)
    (MAlonzo.RTE. mazCoerce v3))
where d_1_55 v0 v1 v2 (C51 v3 v4 v5)
  = MAlonzo.RTE. mazCoerce
    (d33 (MAlonzo.RTE. mazCoerce " ,␣")
      (MAlonzo.RTE. mazCoerce
        (d33 (MAlonzo.RTE. mazCoerce (v2 (MAlonzo.RTE. mazCoerce v4)))
          (MAlonzo.RTE. mazCoerce
            (d55 (MAlonzo.RTE. mazCoerce v0) (MAlonzo.RTE. mazCoerce v3)
              (MAlonzo.RTE. mazCoerce v2)
              (MAlonzo.RTE. mazCoerce v5)))))))
  d_1_55 v0 v1 v2 v3 = MAlonzo.RTE. mazIncompleteMatch name55
```

# MAlonzo - Summary

- Produces 'strange' haskell code
- Can lead to size blow-up
  - 84 lines Agda - 250'000 lines Haskell - 300 Mb executable (CITE)
- Relies on GHC for optimization
- But generated code is not always suited for optimization!

# MAlonzo - FFI

- Provides simple FFI to haskell
- Very limited
  - No class support
  - Can't export Agda datatypes
  - Not automatic

# MALonzo - FFI

```
{-# IMPORT Data.List #-}
```

```
data List : (A : Set) -> Set where
```

```
  nil : ∀ {A} → List A
```

```
  cons : ∀ {A} → A → List A → List A
```

```
{-# COMPILED_DATA List Data.List nil cons #-}
```

```
postulate
```

```
  head : ∀ {A} → List A -> A
```

```
{-# COMPILED head Data.List.head #-}
```

# JS backend

# JS backend

- Targets Javascript
- Not maintained
- Very similar to MAlonzo

## Epic backend

# Epic backend

- Targets Epic
- Not maintained



# Epic

- Untyped-lambda calculus with some extensions
- Intended as building block for compilers
- Also not maintained

# Epic Language

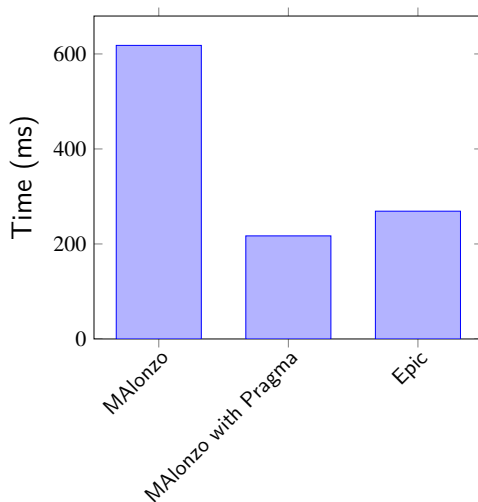
Epic Language		
$t$	$::=$	$x$ Variable
		$t \vec{t}$ Application
		$\lambda x \rightarrow t$ Abstraction
		$\text{Con } i \vec{t}$ Constructor application
		$\text{if } t \text{ then } t \text{ else } t$ if-then-else
		$\text{case } t \text{ of } \vec{a} \vec{t}$ Case expression
		$\text{let } x = t \text{ in } t$ Let expression
		$\text{lazy } t$ Suspended term
		$i$ Integer constants

# Optimizations

# Nat - Primitive Data

- `data Nat : Set where`  
     `zero : Nat`  
     `succ : Nat -> Nat`  
     `{-# BUILTIN NATURAL Nat #-}`
- Naive translation is horribly slow
- Can be transformed into arbitrary precision Integers
- Automatic detection of Nat-like datatypes in Epic backend

# Primitive Data - Performance



# Smashing

- Consider the following Agda Code:

```
data Equality {A : Set} (x : A) : A -> Set where
  refl : Equality x x
  plusAssoc : (n m k : Nat)
    → Equality (n + (m + k)) ((n + m) + k)
  plusAssoc zero m k = refl
  plusAssoc (suc n) m k = cong suc (plusAssoc n m k)
```

# Smashing

- Consider the following Agda Code:

```
data Equality {A : Set} (x : A) : A -> Set where
  refl : Equality x x
  plusAssoc : (n m k : Nat)
    → Equality (n + (m + k)) ((n + m) + k)
  plusAssoc zero m k = refl
  plusAssoc (suc n) m k = cong suc (plusAssoc n m k)
```

- The above definition of `plusAssoc` is linear in it's input.

# Smashing

- Consider the following Agda Code:

```
data Equality {A : Set} (x : A) : A -> Set where
  refl : Equality x x
  plusAssoc : (n m k : Nat)
    → Equality (n + (m + k)) ((n + m) + k)
  plusAssoc zero m k = refl
  plusAssoc (suc n) m k = cong suc (plusAssoc n m k)
```

- The above definition of `plusAssoc` is linear in it's input.
- But it will always return the same value.



# Smashing

- Consider the following Agda Code:

```
data Equality {A : Set} (x : A) : A -> Set where
  refl : Equality x x
  plusAssoc : (n m k : Nat)
    → Equality (n + (m + k)) ((n + m) + k)
  plusAssoc zero m k = refl
  plusAssoc (suc n) m k = cong suc (plusAssoc n m k)
```

- The above definition of `plusAssoc` is linear in it's input.
- But it will always return the same value.
- We can just replace the body by the `refl` constructor at runtime.

# Comparison

	MAlonzo (HS)	Epic	Javascript
Forcing	No	Yes	No
Erasure	No	Yes	No
Smashing	No	Yes	Yes
Primitive Data	Nat only	Yes	Nat only
Maintained	Yes	No	No
User Documentation	Usable	Bad	Bad

# How to fix these issues?

- How can we solve this problem?

# How to fix these issues?

- How can we solve this problem?
- Let's write another backend :-)

# How to fix these issues?

- How can we solve this problem?
- Let's write another backend :-)
- What would be a good target language?

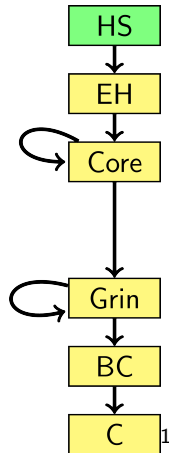
# How to fix these issues?

- How can we solve this problem?
- Let's write another backend :-)
- What would be a good target language?
- Untyped, functional, maintained

# How to fix these issues?

- How can we solve this problem?
- Let's write another backend :-)
- What would be a good target language?
- Untyped, functional, maintained
- UHC Core fits that bill!

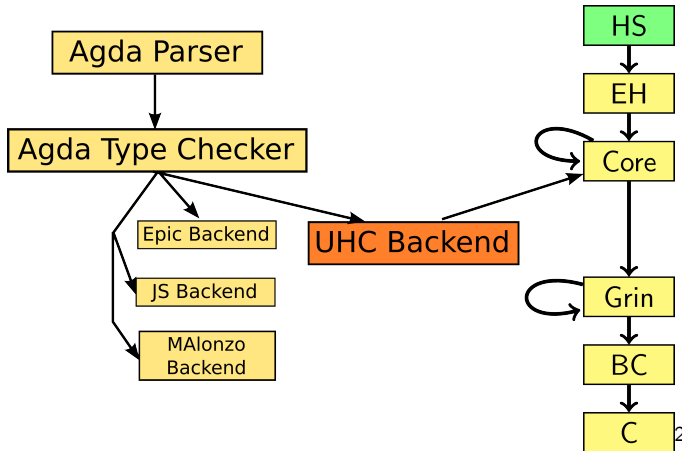
# UHC Compiler



<sup>1</sup>Dijkstra, Fokker, and Swierstra, 2009.



# UHC Backend



<sup>2</sup>Dijkstra et al., 2009.

# Epic vs UHC Core

Epic Language	UHC Core
$t ::= x$	$t ::= x$
$t \vec{t}$	$t t$
$\lambda x \rightarrow t$	$\lambda x \rightarrow t$
$\text{Con } i \vec{t}$	$\text{Con } i \vec{t}$
$\text{if } t \text{ then } t \text{ else } t$	
$\text{case } t \text{ of } \vec{a} \vec{t}$	$\text{case } t \text{ of } \vec{a} \vec{t}$
$\text{let } x = t \text{ in } t$	$\text{let } x = t \text{ in } t$
	$\text{let! } x = t \text{ in } t$
$\text{lazy } t$	
$i$	$i$

# UHC Backend

- Idea: Take Agda Epic backend and combine it with UHC
- A lot of the Epic backend can be reused

# UHC Backend - Challenges

- Agda is a moving target
- UHC Core was not intended as public API
- Undocumented assumptions inside UHC

# UHC Backend - Challenges

- Agda is a moving target
- UHC Core was not intended as public API
- Undocumented assumptions inside UHC

```
case x of
  []      -> a
  (x : xs) -> b
```

is not the same as

```
case x of
  (x : xs) -> b
  []      -> a
```

# UHC Backend - What works?

- (Dependent) datatypes, functions
- Compiling single Agda modules
- Agda - Haskell FFI, but involves manual work

# Demonstration

# UHC Backend - Future work

- Support whole Agda language
  - Multiple modules
  - Complete IO bindings
  - Agda Standard Library
- Optimizations
- Improve Agda - Haskell FFI
- Agda support for Cabal
- Contracts for FFI



Thank you!

Questions?

# References I



Benke, M. (n.d.). Alonzo—a compiler for agda.






Brady, E. (2012). Epic—a library for generating compilers. In *Proceedings of the 12th international conference on trends in functional programming* (pp. 33–48). TFP’11. Madrid, Spain: Springer-Verlag. doi:10.1007/978-3-642-32037-8\_3



Brady, E., McBride, C., & McKinna, J. (2004). Inductive families need not store their indices. In S. Berardi, M. Coppo, & F. Damiani (Eds.), *Types for proofs and programs* (Vol. 3085, pp. 115–129). Lecture Notes in Computer Science. Springer Berlin Heidelberg. doi:10.1007/978-3-540-24849-1\_8

## References II

-  Dijkstra, A., Fokker, J., & Swierstra, S. D. (2009). The architecture of the utrecht haskell compiler. *In Proceedings of the 2nd acm sigplan symposium on haskell* (pp. 93–104). ACM.
-  Fredriksson, O. & Gustafsson, D. (2011). A totally epic backend for agda.
-  Jeffrey, A. (2013). Dependently typed web client applications. *In K. Sagonas (Ed.), Practical aspects of declarative languages* (Vol. 7752, pp. 228–243). Lecture Notes in Computer Science. Springer Berlin Heidelberg.  
doi:10.1007/978-3-642-45284-0\_16

## References III



Osera, P.-M., Sjöberg, V., & Zdancewic, S. (2012). Dependent interoperability. In *Proceedings of the sixth workshop on programming languages meets program verification* (pp. 3–14). PLPV '12. Philadelphia, Pennsylvania, USA: ACM. doi:10.1145/2103776.2103779