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 26, 2014

Table of Contents

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 : ∀ {A n} → Vec A n → A
head1 (cons x xs) = x
head1 nil = ????
```

Cont.

We can now write the head function in Agda

```
head1 : ∀ {A n} → Vec A n → A
head1 (cons x xs) = x
head1 nil = ????
```

This will not type check!

Cont.

We can now write the head function in Agda

$\mathsf{head1} : \forall \{A\ n\} \to \mathsf{Vec}\ A\ n \to A$
$\mathsf{head1}\ (\mathsf{cons}\ x\ xs) = x$
$\mathsf{head1}\ \mathsf{nil} = ????$

This will not type check!

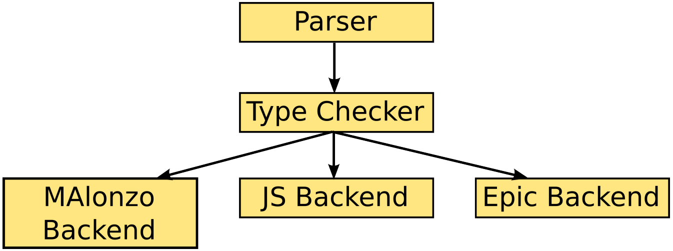
$\mathsf{head2} : \forall \{A\ n\} \to \mathsf{Vec}\ A\ (\mathsf{succ}\ n) \to A$
$\mathsf{head2}\ (\mathsf{cons}\ x\ xs) = x$

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

Agda Summary

- Values can be used as types
- Types cannot influence value of an expression
- Functions need to be total

Agda Architecture



MAlonzo backend

MAlonzo backend

- Targets Haskell
- Maintained
- Relies on GHC for optimizations

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 #-}
```

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)

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

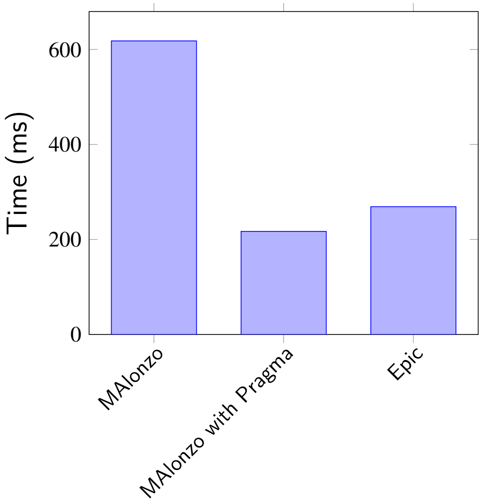
$$
\begin{array}{lcl}
t & ::= & x \\
& | & t\ \vec{t} \\
& | & \lambda x \rightarrow t \\
& | & \mathsf{Con}\ i\ \vec{t} \\
& | & \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \\
& | & \mathsf{case}\ t\ \mathsf{of}\ \vec{alt} \\
& | & \mathsf{let}\ x = t\ \mathsf{in}\ t \\
\\
& | & \mathsf{lazy}\ t \\
& | & t\ !\ i \\
& | & i
\end{array}
$$

Epic - Nat Optimizations

-
  ```
  data Nat : Set where
      zero : Nat
      succ : Nat -> Nat
  {-# BUILTIN NATURAL Nat #-}
  ```

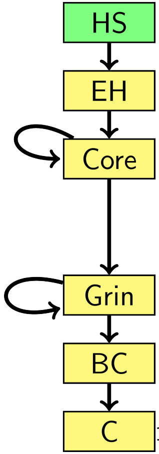
- Naive translation is horribly slow
- Can be transformed into arbitrary precision Integers
- Automatic detection of Nat-like datatypes

Nat Performance



Comparison

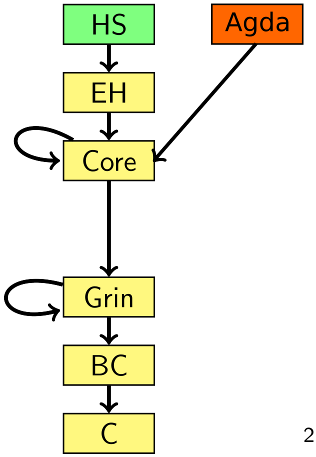
UHC Compiler



[1]

[1] Dijkstra, Fokker, and Swierstra, 2009.

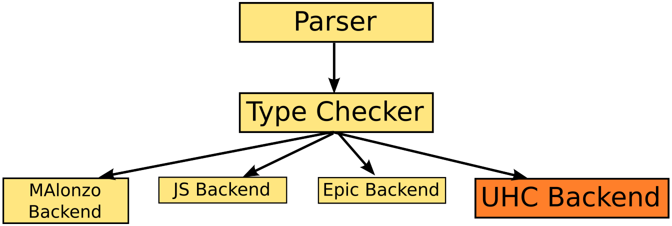
UHC Compiler


$$\begin{tikzcd}
HS \arrow[d] & Agda \arrow[dl] \\
EH \arrow[d] & \\
Core \arrow[loop left] \arrow[d] & \\
Grin \arrow[loop left] \arrow[d] & \\
BC \arrow[d] & \\
C &
\end{tikzcd}$$

[2]

[2]Dijkstra et al., 2009.

UHC Backend


Parser
Type Checker
MAlonzo Backend
JS Backend
Epic Backend
UHC Backend

Epic vs UHC Core

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

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) dataypes, 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 totaly 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