The Utrecht Agda Compiler

Submitted to TFP 2015

Philipp Hausmann, Atze Dijkstra, and Wouter Swierstra

Universiteit Utrecht, Netherlands
`http://www.uu.nl`

Abstract. This paper describes our experience implementing the Utrecht Agda Compiler, a new Agda compiler targeting the Utrecht Haskell Compiler's core language. Besides presenting the translation scheme, this paper describes the key issues in the compiler's development. The Utrecht Agda Compiler not only addresses several limitations and shortcomings of the existing backends, but provides a platform for further research in the compilation of dependently typed programming languages.

Keywords: dependently-typed programming, Agda, UHC, code generation

1 Introduction

There are many different dependently typed programming languages and proof assistants, including Coq [22], Agda [16], Cayenne [4], Twelf [19] and Idris [7]. Yet most of these systems focus on theorem proving rather than efficient execution; very few have a mature compiler. As a result, there are very few examples of large systems developed using programming languages with dependent types.

In this paper we report our initial steps in developing a compiler for Agda. Leveraging parts of the existing Utrecht Haskell Compiler [10,3] (UHC), allowed us to develop a rapid prototype at relatively low cost. This paper reports our progress so far:

– We survey the existing Agda backends and the design criteria we set ourselves for our compiler (Section 2).
– We will describe our experience implementing a new compiler backend for Agda. This paper not only focuses on the difficulties we encountered, but also gives some advice for future backend developers. The principal algorithms implemented are not novel and many elements have been described before, but we do provide a more detailed formal semantics of the translation from Agda to UHC's Core language (Section 3) and the problems we have encountered (Section 4).
– We have updated and significantly extended the collection of executable Agda programs (Section 4.2). These examples provide a regression test suite for our own compiler, but may be valuable to other backend developers too. We hope that this collection and our new compiler will eventually inspire more Agda developers to write executable programs, instead of libraries that are never run.

– We have started exploring how this new backend can be used to interface between Agda and Haskell. Having a shared intermediate language for both the Utrecht Agda Compiler and the Utrecht Haskell Compiler makes it possible to share a common binary format and exchange data between the two languages freely (Section 5).

2 Background

2.1 Existing Backends

The most notable compiler for Agda is MAlonzo [14]. Its target language is Haskell, using the Glasgow Haskell Compiler (GHC) to produce executables from the generated Haskell code. While a large part of the Agda language can easily be translated into valid Haskell code, this does not hold for the entire language; after all, Haskell is not dependently typed. The MAlonzo compiler works around this problem by inserting (unsafe) type coercions. This coercions increase the code size significantly, as can be seen in the Hello World example in Figure 1.

This behavior, together with the lack of optimizations in MAlonzo, can lead to a blowup in the size of the generated Haskell code [1]. The inserted type-coercions also prevent GHC from applying certain type-directed optimization, which is unfortunate as MAlonzo relies solely on GHC for optimizing the generated code. Although these coercions can have a performance impact, they do not affect the correctness of the program, as the source code has already been type checked by Agda.

MAlonzo also provides a Foreign Function Interface (FFI) to Haskell. Using this FFI, programmers can call Haskell functions from Agda, export Agda functions to Haskell and reuse Haskell data types in Agda. Only the common subset of both programming languages can be used in the FFI. Dependently typed functions, for example, are not supported. The current FFI relies on the programmer to specify the exact mapping between Agda and Haskell using pragmas; Figure 2 contains a small example illustrating all these features.

Besides the MAlonzo compiler, there are several more experimental Agda backends targeting JavaScript [11] and the Epic language [17,6]. The Epic language itself is a simple untyped lambda calculus, extended with pattern matching and data types.

2.2 What should be the target language?

While targeting Haskell directly is certainly a viable approach, the numerous coercions inserted by MAlonzo push GHC to its limits. There are examples of modest Agda files generating huge Haskell files, that require unacceptable compilation time. We felt that it might be worthwhile to explore alternative approaches.

We explicitly wanted to avoid creating a full compiler from scratch; instead, we wanted to reuse existing infrastructure to deal with the low-level aspects of compilation. Foremost, reusing an existing language as our target language saves a lot of work and allows us to leverage all existing tooling and infrastructure. Secondly, we hope to experiment further with a Foreign Function Interface between two high-level languages such as Agda and Haskell. Having a common target language is almost a necessity for such experiments.

```
main = run (putStrLn (show (10 + 10)))
```

(a) A small Agda program.

```
main = d1
name1 = "HelloWorld.main"
d1
  = MAlonzo.RTE.mazCoerce
      (MAlonzo.Code.IO.d21
         (MAlonzo.RTE.mazCoerce MAlonzo.Code.Agda.Primitive.d3)
         (MAlonzo.RTE.mazCoerce MAlonzo.Code.Data.Unit.Base.d3)
         (MAlonzo.RTE.mazCoerce
            (MAlonzo.Code.IO.d75
               (MAlonzo.RTE.mazCoerce
                  (MAlonzo.Code.Data.Nat.Show.d11
                     (MAlonzo.RTE.mazCoerce
                        (MAlonzo.Code.Data.Nat.Base.d14
                           (MAlonzo.RTE.mazCoerce
                              (MAlonzo.Code.Data.Nat.Base.↩
                                  ↪ mazIntegerToNat (10 :: ↩
                                  ↪ Integer)))
                           (MAlonzo.RTE.mazCoerce
                              (MAlonzo.Code.Data.Nat.Base.↩
                                  ↪ mazIntegerToNat
                                 (10 :: Integer))))))))))
```

(b) The generated Haskell code for the above Agda program. The mazCoerce expressions are type coercions.

Fig. 1: A example Agda program and it's translation to Haskell.

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
-- Bind the Agda List data type to the Haskell List data type
{-# COMPILED_DATA List Data.List.List [] (:) #-}

postulate
  _++_ : ∀ {A} → List A → List A → List A
-- Bind the Agda ++ operator to Haskell's ++
{-# COMPILED _++_ (\_ -> (Data.List.++)) #-}

reverse : ∀ {A} → List A → List A
reverse [] = []
reverse (x :: xs) = reverse xs ++ (x :: [])
-- Export the Agda reverse function, callable from Haskell
{-# COMPILED_EXPORT reverse reverse #-}
```

Fig. 2: An Agda example using the Foreign Function Interface (FFI).

Adapting the existing Epic backend would have been a possibility. Epic compiles directly to C, and does not target any other high-level language. As a result, it would have severely limited the possibility of supporting the existing FFI with Haskell: any call to Haskell code would always have to go through Haskell's FFI with C.

The other obvious candidate for an intermediate language would have been GHC Core [18]. GHC Core, however, is a typed intermediate language, based on System FC [21]. While all Haskell's high-level language features, including type families [8] and GADTs [20], can be translated to System FC, it is *not* dependently typed. As a result, translating Agda to System FC is not a trivial exercise.

This mismatch in type systems is the same problem the MAlonzo backend exhibits. We could solve the problem in the same way: inserting numerous type coercions. These coercions, however, will also cause the same problems already present in the current MAlonzo backend. We could, perhaps, do a slightly better job by only inserting those type coercions that are strictly necessary, as opposed to naively inserting them everywhere where they *might* be required. Coq's extraction mechanism takes this approach [12]. By replicating an ML type checker inside the Coq compiler, the extraction mechanism only inserts coercions at places where the generated ML code would not type check otherwise. Nonetheless, following the same approach would have required a relatively large effort and would have seriously limited our freedom to experiment and modify the target language to suit our needs.

Although there have been proposals for a dependently-typed core language [2], there is no mature implementation that we could use off the shelf. Instead, we chose to resolve this mismatch between type systems by targeting an untyped core language.

The Utrecht Haskell Compiler (UHC), developed by Dijkstra et al. [10], has a Core language similar to Epic. The main difference is that UHC Core is non-strict. UHC is also used to a large degree for experimenting with new ideas and backward compatibility is not as important as for example for GHC. As its name suggests, the Utrecht Haskell Compiler also compiles Haskell to executables via UHC Core. Hence, an Agda backend targeting UHC Core is a suitable vehicle for exploring the possibility of mixed Agda-Haskell developments.

3 Translating between Agda and UHC

Both Agda and UHC consist of a series of transformations between intermediate languages; starting from the high-level Agda or Haskell input and going towards more simplistic core languages. Our compiler links Agda's Internal Syntax language with the UHC Core language; reusing Agda's type checker and UHC's code generation.

While UHC's code generation already existed, its compilation pipeline required significant changes to support our use case. We will describe these changes in more detail in Section 4.

Reusing Agda's type checker by itself is easy; the hard part has been translating Agda's Internal Syntax language to the UHC Core language. We were able to reuse some code from the existing Agda Epic backend, but differences in the target language and new Agda features posed new challenges. For example, the support for varying arity functions described later in this section is new compared to the Epic backend.

In the remainder of this section we will give a formal description of the translation from Agda's Internal Syntax to UHC Core.

3.1 UHC Core

UHC Core is one of the intermediate languages used by UHC, and forms the target language of our compiler. We use a slightly simplified version of UHC Core for describing the translation from Agda's Internal Syntax to UHC Core, which we will call UHC SCore. The grammar of UHC SCore is given in Figure 3. All the UHC SCore terms in the rest of this paper will be written in red to distinguish them from terms in Agda's Internal Syntax.

Unlike UHC Core, in UHC SCore we treat all let bindings as possibly recursive. We also assume that case terms drive the evaluation, whereas in UHC Core a special let-strict term is used for this purpose. Finally, constructors are named, rather than numbered, and may be partially applied.

Converting UHC SCore expressions to UHC Core is straightforward; e.g. all UHC SCore case expressions simply have to be wrapped in a let-strict term.

3.2 Translating Agda's Internal Syntax to UHC SCore

The input for our Compiler is Agda's Internal Syntax (AIS), which is produced by the existing Agda frontend and presented in Figure 4. To help distinguish AIS from UHC SCore, we will use the color blue for AIS expressions in the rest of this paper.

We will now explain the compilation from AIS to UHC SCore. Figure 5 contains the complete formal translation semantics. In the following subsections, we will go through all constructs of AIS and explain the translation in detail. The translation from the AIS term A to the UHC SCore term B is written as $A \rhd B$.

Terms	t	$::=$	x	Variable
		$\mid$	$t\ \vec{t}$	Application
		$\mid$	$\lambda\vec{x} \to t$	Lambda Abstraction
		$\mid$	$\text{Con}\ n\ \vec{t}$	Constructor Application
		$\mid$	$\text{case}\ t\ \text{else}\ \vec{alt}\ \text{or}\ t$	Case with default
		$\mid$	$\text{let}\ x\ =\ t\ \text{in}\ t$	Let
		$\mid$	$\top$	Unit
		$\mid$	$\bot$	Failure
Alternatives	alt	$::=$	$\text{Con}\ n\ \vec{x}\ \to\ t$	Constructors

Fig. 3: The abstract syntax of UHC SCore, a simplified version of UHC Core. The notation $\vec{t}$, $\vec{alt}$, and $\vec{v}$ refers to a list of terms, alternatives, and variables respectively.

Agda Internal Syntax

Program	p	$::=$	$[(n, def)]$	Program
Name	n			
Definition	def	$::=$	Data	Data type
		$\mid$	$\text{Fun}\ c$	Function
		$\mid$	Axiom	Axiom
CaseTree	c	$::=$	$\text{Case}\ \alpha\ \vec{b}$	Inspect
		$\mid$	$\text{Done}\ \beta\ t$	Finished
		$\mid$	Fail	Absurd case
Branch	b	$::=$	$\text{Con}\ n\ \psi\ c$	Constructor
		$\mid$	$\text{CatchAll}\ c$	Default
Term	t	$::=$	$\text{Var}\ \alpha\ \vec{t}$	Variable / Application
		$\mid$	$\text{Def}\ n\ \vec{t}$	Application / Projection
		$\mid$	$\lambda \to t$	Abstraction
		$\mid$	$\text{Con}\ n\ \vec{t}$	Constructor Application
		$\mid$	$\Pi\ t\ t$	(Dependent) Function Type
		$\mid$	$\text{Set}\ t$	Set
		$\mid$	$\text{Level}\ l$	Level
Level	l	$::=$	Z	Zero
		$\mid$	$\text{S}\ l$	Successor
		$\mid$	$l \sqcup l$	Join

Fig. 4: The abstract syntax of Agda. Irrelevant parts of the syntax have been omitted for brevity.

$$\frac{}{\mathsf{Axiom} \rhd \bot}\ \text{E-Def-Axiom} \qquad \frac{}{\mathsf{Data} \rhd \top}\ \text{E-Def-Data} \qquad \frac{[\,]; \bot; c \rhd c'}{\mathsf{Fun}\ c \rhd c'}\ \text{E-Def-Fun}$$

$$\frac{|\Gamma| \geq \alpha \qquad \Gamma; \phi; \vec{b} \rhd \phi' \qquad \Gamma; \phi'; \alpha; \vec{b} \rhd \vec{b}'}{\Gamma[x_0..x_\alpha..x_n]; \phi; \mathsf{Case}\ \alpha\ \vec{b} \rhd \mathsf{case}\ x_\alpha\ \mathsf{of}\ \vec{b}'\ \mathsf{else}\ \phi'}\ \text{E-Case1}$$

$$\frac{|\Gamma| < \alpha \qquad \text{let } x \text{ be fresh} \qquad \Gamma \mathbin{+\!\!+} [x]; (\phi\ x); \mathsf{Case}\ \alpha\ \vec{b} \rhd t'}{\Gamma; \phi; \mathsf{Case}\ \alpha\ \vec{b} \rhd \lambda x \to t'}\ \text{E-Case2}$$

$$\frac{\Gamma; \phi; c \rhd c'}{\Gamma; \phi; [b_0, b_1, ..(\mathsf{CatchAll}\ c).., b_n] \rhd c'}\ \text{E-CatchAll} \qquad \frac{\vec{b} \text{ contains no } \mathsf{CatchAll}}{\Gamma; \phi; \vec{b} \rhd \phi}\ \text{E-NoCatchAll}$$

$$\frac{[x_0..x_{\alpha-1}] \mathbin{+\!\!+} [y_0..y_\psi] \mathbin{+\!\!+} [x_{\alpha+1}..x_n]; \phi; c \rhd c' \qquad \text{let } y_0..y_\psi \text{ be fresh}}{[x_0..x_\alpha..x_n]; \phi; \alpha; \mathsf{Con}\ n\ \psi\ c \rhd [\mathsf{Con}\ n\ y_0..y_\psi\ c']}\ \text{E-Branch-Con}$$

$$\frac{}{\Gamma; \phi; \alpha\ \mathsf{CatchAll}\quad \mathsf{c} \rhd [\,]}\ \text{E-Branch-CatchAll}$$

$$\frac{}{\Gamma; \phi; \mathsf{Fail} \rhd \bot}\ \text{E-Fail} \qquad \frac{|\Gamma| \geq \beta \qquad (\mathit{reverse}\ \Gamma); t \rhd t'}{\Gamma; \phi; \mathsf{Done}\ \beta\ t \rhd t'}\ \text{E-Done1}$$

$$\frac{|\Gamma| < \beta \qquad \text{let } x \text{ be fresh} \qquad \Gamma \mathbin{+\!\!+} [x]; \phi; \mathsf{Done}\ \beta\ t \rhd d'}{\Gamma; \phi; \mathsf{Done}\ \beta\ t \rhd \lambda x \to d'}\ \text{E-Done2}$$

$$\frac{\Gamma; \vec{t} \rhd \vec{t'}}{\Gamma[x_0..x_\alpha..x_n]; \mathsf{Var}\ \alpha\ \vec{t} \rhd x_\alpha\ \vec{t'}}\ \text{E-VarApp} \qquad \frac{\Gamma \mathbin{+\!\!+} [x]; t \rhd t' \qquad \text{let } x \text{ be fresh}}{\Gamma; \lambda \to t \rhd \lambda x \to t'}\ \text{E-Abs}$$

$$\frac{\Gamma; \vec{t} \rhd \vec{t'}}{\Gamma\ \mathsf{Def}\ n\ \vec{t} \rhd n\ \vec{t'}}\ \text{E-DefApp}$$

$$\frac{\Gamma; \vec{x} \rhd \vec{x}'}{\Gamma; \mathsf{Con}\ n\ x \rhd \mathsf{Con}\ n\ \vec{x}'}\ \text{E-Con} \qquad \frac{}{\Gamma; \Pi\ t_1\ t_2 \rhd \top}\ \text{E-Pi} \qquad \frac{}{\Gamma; \mathsf{Set}\ l \rhd \top}\ \text{E-Set}$$

$$\frac{l \rhd l'}{\Gamma; \mathsf{Set}\ l \rhd l'}\ \text{E-Level} \qquad \frac{}{\mathsf{Z} \rhd \top}\ \text{E-Level-Z} \qquad \frac{}{\mathsf{S}\ l \rhd \top}\ \text{E-Level-S} \qquad \frac{}{l_1 \sqcup l_2 \rhd \top}\ \text{E-Level-Join}$$

Fig. 5: The formal description of the translation from Agda's internal Syntax to UHC SCore.

Axioms Axioms are one of the simplest definitions to translate. They arise from postulates, for example:

```
    postulate axiom−0≡1 : 0 ≡ 1
```

Postulates can be used to introduce new assumptions by declaring their type, without providing the associated definition. Such postulates are only interesting during proofs; if such an axiom needs to be evaluated at runtime, the program will crash. We hence translate it to the crashing UHC SCore expression $\bot$, as can be seen in rule E-Def-Axiom.

Functions and case splitting trees The AIS Fun construct introduces a new function. Most such functions are defined using pattern matching, just as in other functional languages like Haskell or OCaml. During type checking this patterns are converted to case splitting trees, as first described by Augustsson [5]. The AIS case tree generated by Agda's type checker is the input to our compiler.

Nameless references AIS uses a locally nameless representation for variable references [15,13,9]. UHC SCore on the other hand relies on named references; thus we need to convert de Bruijn levels and de Bruijn indices to names in our translation. The usage of both, de Bruijn levels *and* indices is not an accident. It is motivated by the varying arity feature of Agda explained later in this section. The conversion from de Bruijn levels and indices to names is achieved by passing down an environment Γ, containing the mapping from levels/indices to names. Later, we will use this environment to look up the corresponding name of a nameless reference. Initially, we set Γ to the empty environment in rule E-Def-Fun, as the function has not yet received any arguments or defined any local variables.

Case splitting trees Unlike UHC SCore case expressions, which scrutinize exactly one variable at a time, a case tree can inspect any number of variables. The case splitting tree translation of an example Agda function can be seen in Figure 6.

A case tree consists of a non-terminal Case node that scrutinizes one variable at a time; and Done and Fail terminal nodes representing the result of the given branch.

Case non-terminals Case ψ $\vec{b}$ non-terminals use the argument at the de Bruijn level ψ to branch on. De Bruijn levels count arguments from the outermost towards the innermost binder; as the environment Γ grows to the right, the name has to be looked up at position ψ from the left.

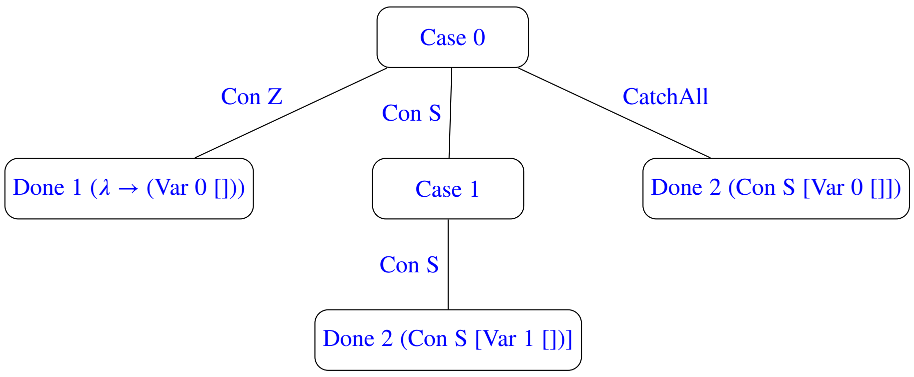
In the example case tree in Figure 6, the root non-terminal branches on the first argument of the function `f`. If the first variable has the value Con S, another Case non-terminal is encountered and the second argument of the function is examined. A case tree may contain any number of such Case non-terminals.

Each Case ψ $\vec{b}$ non-terminal is translated to one case expression. Given a way to translate the branches $\vec{b}$ to UHC SCore alternatives and assuming that the environment Γ already contains the variable matched on and ignoring non-exhaustive cases, the translation is defined as follows:

$$\frac{\Gamma; \vec{b} \rhd \vec{b}'}{\Gamma[x_0..x_\alpha..x_n]; \text{Case } \alpha\ \vec{b} \rhd \text{case } x_\alpha \text{ of } \vec{b}'} \text{ E-Case1'}$$

```
f : ℕ → ℕ → ℕ
f Z = λ y → y
f (S x) (S y) = S x
f x y = (S y)
```

(a) Example Agda function.



(b) Case-tree translation of the above Agda function. The branches $\vec{b}$ of a Case non-terminal are drawn as edges, and case trees as nodes.

Fig. 6: A simple example of the case splitting tree translation.

Varying arity One peculiarity of Agda complicating the translation to UHC SCore is that, in contrast to Haskell, Agda doesn't require all functions clauses to take the same number of arguments. Agda function clauses may accept a varying number of arguments; for instance, when assigning a dependent type to functions such as `printf`, the value of one argument may determine the number of subsequent arguments.

This behavior makes the conventional way of compiling functions by putting as many lambda abstractions as there are function arguments around the whole function body impossible. Instead, we need to check at all Case non-terminals if enough lambda-abstractions are already present; if not, an additional lambda abstraction needs to be inserted. To this purpose, we insert the rule E-Case2", which insert lambda-abstractions until the environment Γ has the required size. Rule E-Case1" needs the additional constraint $|\Gamma| \geq \alpha$.

$$\frac{|\Gamma| \geq \alpha \qquad \Gamma; \vec{b} \rhd \vec{b}'}{\Gamma[x_0..x_\alpha..x_n]; \text{Case } \alpha\ \vec{b} \rhd \text{case } x_\alpha \text{ of } \vec{b}'}\ \text{E-Case1''}$$

$$\frac{|\Gamma| < \alpha \qquad \text{let } x \text{ be fresh} \qquad \Gamma +\!\!+ [x]; \text{Case } \alpha\ \vec{b} \rhd t'}{\Gamma; \text{Case } \alpha\ \vec{b} \rhd \lambda x \to t'}\ \text{E-Case2''}$$

The varying arity feature motivates using de Bruijn levels in the case splitting trees; with de Bruijn indices it would be impossible to tell when additional lambdas need to be inserted.

Constructor branches Previously, we assumed that branches could be translated to UHC SCore alternatives, but have not gone into any details. A Con ψ c branch translates directly to an UHC SCore alternative. The number ψ records the number of arguments the constructor takes; for each argument, we generate a fresh name. The current case scrutinee inside the environment Γ is replaced by the list of fresh names. This makes the pattern matched variables available to be used in the child case splitting tree stored inside the current Con non-terminal. The body of the generated UHC SCore alternative is the sub tree c, translated to UHC SCore.

$$\frac{[x_0..x_{\alpha-1}] +\!\!+ [y_0..y_\psi] +\!\!+ [x_{\alpha+1}..x_n]; \phi; c \rhd c' \qquad \text{let } y_0..y_\psi \text{ be fresh}}{[x_0..x_\alpha..x_n]; \phi; \alpha; \text{Con } n\ \psi\ c \rhd [\text{Con } n\ y_0..y_\psi\ c']}\ \text{E-Branch-Con}$$

Note that this replacement inside the environment can change the size and indexes of variables inside Γ. This is already taken into account when the case splitting tress are generated and therefore requires no further work.

CatchAll branches So far we have ignored the CatchAll c branches. These come into play when none of the other branches match, and provide a default return value. This default c in itself is another case splitting tree.

CatchAll branches apply to a whole sub tree. To come back to the example in Figure 6, the CatchAll branch at the root node also applies to the Case 1 non-terminal. This can be useful when a CatchAll applies to many Case non-terminals. For example, when compiling the following Agda functions, a failing pattern match on the first *or* second argument will yield the same result:

```
f : ℕ → ℕ → ℕ
f zero zero = zero
f x y = x
```

Our earlier rules E-Case1” and E-Case2” plainly ignore CatchAll branches. To fix this, we need to pass around the default value to use when generating UHC SCore case expressions in E-Case1”. To this end, we pass down the value ϕ, which contains the UHC SCore term to use as default value. Initially, we set ϕ to $\bot$, as evaluating it at runtime corresponds to a pattern match failure.

In E-Case1”, we may need to update the default value ϕ. If the current non-terminal contains a CatchAll branch, rule E-CatchAll triggers and returns the new default value. If there is no CatchAll branch, rule E-NoCatchAll fires and the current default value is used unchanged. The updated rule E-Case1 is as follows:

$$\frac{|\Gamma| \geq \alpha \qquad \Gamma; \phi; \vec{b} \rhd \phi' \qquad \Gamma; \phi'; \alpha; \vec{b} \rhd \vec{b}'}{\Gamma[x_0..x_\alpha..x_n]; \phi; \text{Case } \alpha\ \vec{b} \rhd \text{case } x_\alpha \text{ of } \vec{b}' \text{ else } \phi'}\ \text{E-Case1}$$

The rule E-Case2” also needs to be updated to pass the default value ϕ. Doing this correctly requires careful attention, though. The rule E-Case2” introduces a new lambda, which the UHC SCore expression in ϕ does not expect. The expression ϕ has been compiled in the environment Γ, where the corresponding CatchAll branch has been defined, rather than the extended environment $\Gamma \mathbin{+\!\!+} [x]$. To remedy this, the ϕ needs to ignore the newly-introduced lambda. This is achieved by applying the default value immediately to the newly introduced lambda argument before passing it on in the term $(\phi\ x)$. The changed rule E-Case2 looks like this:

$$\frac{|\Gamma| < \alpha \qquad \text{let } x \text{ be fresh} \qquad \Gamma \mathbin{+\!\!+} [x]; (\phi\ x); \text{Case } \alpha\ \vec{b} \rhd t'}{\Gamma; \phi; \text{Case } \alpha\ \vec{b} \rhd \lambda x \to t'}\ \text{E-Case2}$$

Terminal nodes So far we have discussed how to translate the non-terminals of case splitting trees, but not the terminal nodes. First, there is the Fail non-terminal. It is inserted whenever the Agda type checker believes that this node should never be reached. These nodes correspond to absurd patterns in Agda. As Fail terminal nodes should never be reached at runtime, we translate them to the UHC SCore term $\bot$:

$$\frac{}{\Gamma; \phi; \text{Fail} \rhd \bot}\ \text{E-Fail}$$

Second, a Done β t non-terminal represents a successful pattern match, containing the body of the function for this terminal node. As a result of the problem with varying arities discussed earlier, we need the value β to know how many arguments the function body expects to have already been abstracted. Similar to how we adapted the rules E-Case1 and E-Case2, we need to insert additional lambdas if necessary using the helper rule E-Done2:

$$\frac{|\Gamma| < \beta \qquad \text{let } x \text{ be fresh} \qquad \Gamma \mathbin{+\!\!+} [x]; \phi; \text{Done } \beta\ t \rhd d'}{\Gamma; \phi; \text{Done } \beta\ t \rhd \lambda x \to d'}\ \text{E-Done2}$$

While the above rule ensures that the environment is complete, it is not the correct environment yet to use for translating the function body. The case splitting trees may have to use de Bruijn levels to support the varying arity feature; for the function body itself using de Bruijn indices is more convenient.

The two different counting schemes are exact opposites in a way. De Bruijn indices count from the innermost towards the outermost binder; de Bruijn levels count from the outermost towards the innermost binder. Taking advantage of this relationship, the correct environment for the function body can be easily computed by reversing the accumulated environment Γ.

$$\frac{|\Gamma| \geq \beta \qquad (\mathit{reverse}\ \Gamma); t \rhd t'}{\Gamma; \phi; \mathsf{Done}\ \beta\ t \rhd t'}\ \text{E-Done1}$$

Term translation Now that we have translated the case splitting trees, we can finally do the same for the actual function bodies. First, let us discuss the terms actually influencing runtime behavior as opposed to terms only relevant for type checking.

AIS uses de Bruin indices to refer to local variables and arguments, and names for module-level definitions. As both de Bruijn indices and names are used to represent variables, there are two different versions of variable access: Var to refer to locally bound variables by index, and Def to refer to global definitions accessed by name. Both versions also double as application and take a list of arguments. If the list of arguments is empty, these two constructs are simply variables without further arguments. Both versions can easily be translated to the UHC SCore variable access and application terms; using the environment Γ in the Var case to map the indices to the corresponding generated names.

Lambda abstractions and constructor application are the remaining two terms contributing to the runtime semantics. They have a direct counterpart in UHC SCore and thus can be translated easily.

Type translation The remaining terms Π, Set and Level are significant for type checking only. In Agda, a value of type Set or Level cannot be inspected or pattern matched. As Agda enforces that it is impossible to observe any value of these types, they cannot affect the runtime semantics. For executing a program, it is thus safe to replace all occurrences of such values by the unit value $\top$.

One could also be tempted to completely remove any values of these kind. This could potentially alter the semantics of the translated program. Agda does not evaluate expressions under lambdas; dropping lambda abstractions taking type expressions could remove evaluation-blocking lambdas. A partial erasure of types is possible in a sound way. Saturated function applications for example can always be optimized in this way. A more detailed description of when such types may be soundly erased can be found in previous work by Letouzey [12].

3.3 Example

The translation of the Agda example from Figure 6 is shown in Figure 7.

One side-effect of the inherited default values can be seen in the repetition of the term $(\lambda f \rightarrow \mathsf{Con}\ \mathsf{S}\ f)$ on line 6 and 8. The higher the sub tree in question and the larger the default expression, the more significant this duplication becomes. To avoid this problem, our compiler shares the default expression between all occurrences using a let expression.

```
f = λa → case a of
    Con Z → (λb → b)
    Con S c → (λd →
        case d of
            Con S e → Con S c
            else ((λf → Con S f) d)
        )
    else (λf → Con S f)
```

Fig. 7: Translation to UHC SCore of the example Agda function from Figure 6.

4 Lessons learned

4.1 Adapting UHC to our needs

Build System At the start of this project, UHC was a standalone Haskell compiler. It had not been used before as component of another compiler. Using UHC as part of our Agda compiler required using the build system in an unsupported way and was fairly fragile. The long dependency lists of both Agda and UHC, combined with the custom configuration and build system of UHC made versioning conflicts in the Cabal Package Database a recurring problem.

The proper solution would be to port UHC to a cabal-based build system, but this would require major effort. Instead, we opted to build a subset of UHC into a proper cabal library, which can be distributed on Hackage easily. We can then use this fragment of UHC to compile Agda code.

UHC Core In contrast to the build system, exposing the UHC Core language as a public API not only required technical changes, but also posed challenges in understanding the behavior of the generated code. Due to its origin as an internal intermediate language, there is no formal specification of UHC Core. While most of the syntactic constructs are self-explanatory, code generated from Agda could sometimes exhibit unexpected behavior. To diagnose errors, we sometimes resorted to inspecting the output generated from Haskell programs to see how this differed from the code our Agda compiler generated. For example, UHC Core requires case alternatives to be in lexicographical order. This is not immediately obvious from generated code. Specific compiler invariants such as this one are typically poorly documented and are easy to break.

4.2 Testing our Compiler

As with all sufficiently complex software, verifying the correctness of our Agda compiler is non-trivial. We have not tried to formally prove the correctness of our compiler. Instead, we rely on a test suite to validate the correctness of our compiler.

To this end, we have created a test suite consisting of 30 example Agda programs in total. Each of these programs can be run, and the output can be compared to a golden standard. Many of the example programs were not written from scratch, but instead adapted for our purpose or ported from Haskell.

This collection, very roughly, consists of two kinds of programs. The first kind are small Agda programs, each testing a single Agda feature. These programs also serve as regression tests to avoid re-introducing fixed bugs.

The second kind of test programs are more complicated programs, which involve more computations. The intent here is to stress test all parts of the compiler and runtime system. For example, we have a port of the interactive Eliza[1] program from GHC's nofib benchmark suite.

These test cases target both our compiler and the existing MAlonzo compiler. This allows us to compare the output and use the MAlonzo backend as baseline for development.

4.3 Searching the needle in the haystack - how to locate bugs

While the test set can guide us where to look for bugs, it often does not provide sufficient information to pinpoint a bug exactly. The problem is exacerbated by the call-by-need semantics of UHC Core, which makes it much harder to diagnose why generated code crashes. Fortunately, UHC already incorporates a tracing mechanism reporting the progress of evaluation. This proved helpful in some situations, but only to a limited degree. Due to it's low-level nature, it tends to be very verbose. For example, the simple program in Figure 8a generates $250,000$ lines of tracing output.

To improve matters, we implemented a simple tracing facility in our compiler, that inserts debugging information into the generated code. If tracing is disabled, the generated code is not modified and no performance penalty is incurred.

The current implementation traces evaluation of function bodies and arguments. An excerpt from a trace of the same Agda program can be seen in Figure 8b. The `Eval fun` lines indicate that the body of the named function is evaluated, whereas `Eval arg` lines indicate which argument is currently being evaluated.

This tracing facility proved essential in finding the exact causes of multiple bugs, where we had failed previously to find the offending code due to the size of the test cases involved. Especially when implementing advanced compiler features, tracing becomes necessary to analyze bugs triggered in corner cases effectively.

5 FFI

Our new Agda compiler features a basic Foreign Function Interface (FFI) to Haskell, similar to the FFI of the MAlonzo compiler explained in Section 2.1.

The primitives work by using pragmas to instruct the compiler to use a specific UHC Core representation for Agda data types or postulates.

Crucially, all pragmas work on the UHC Core level and *not* on the Haskell representation. But because Haskell data types and functions are translated in a stable and

[1] `https://github.com/ghc/nofib/tree/master/spectral/eliza`

```
main = run (putStrLn (show (10 + 20)))
```

(a) An Agda program printing the number 30.

```
Eval fun: Agda.PrintNat....main...
Eval fun: Agda.IO....run...
Eval arg: Agda.IO....run...  :: 2
Eval fun: Agda.IO....putStrLn...
Eval fun: Agda.IO....putStrLn...
Eval fun: Agda.IO....IO__>>...
...
Eval arg: Agda.IO....IO__>>...  :: 2
Eval fun: Agda.IO....o-16...
Eval fun: Agda.Coinduction....o...
Eval arg: Agda.Coinduction....o...  :: 0
Eval fun: Agda.IO....IO_return...
30
```

(b) Excerpt from the trace of the above program. The numbers trailing the `Eval arg` lines indicate the position of the argument being evaluated.

Fig. 8: Demonstration of the Agda UHC Compiler tracing facility for a minimal Agda program.

deterministic way to UHC Core, the UHC Core level FFI can be leveraged to interact with Haskell code.

An example of the data type bindings is given in Figure 9. The code in this figure binds the AgdaList data type to the UHC Core data type HsList. This binding, although expressed on the UHC Core level, effectively also acts as a binding to the Haskell HsList data type.

Functions, on the other hand, pose a slightly more complicated problem. Agda explicitly passes around of types; for example, the argument {A : Set} is used to pass the type of list elements around in the append function _++_ . Haskell instead uses an implicit encoding of types, which are not explicitly passed around in either Haskell itself or UHC Core. To bridge the gap between these two approaches, all arguments of type Set have to be discarded when calling Haskell functions from Agda. This is done by wrapping the UHC representation of Haskell functions inside a lambda and discarding all unnecessary arguments before calling the actual function.

```
data HsList a = HsNil | HsCons a (HsList a)

hsAppend :: HsList a -> HsList a -> HsList a
hsAppend HsNil ys = ys
hsAppend (HsCons x xs) ys = HsCons x (hsAppend xs ys)
```

(a) A simple Haskell data type and function.

```
data AgdaList (A : Set) : Set where
  [] : AgdaList A
  _::_ : A → AgdaList A → AgdaList A
-- Bind the Agda List data type to a Haskell List data type
{-# COMPILED_CORE_DATA AgdaList HsList HsNil HsCons #-}

postulate
  _++_ : {A : Set} → AgdaList A → AgdaList A → AgdaList A
-- Bind the Agda ++ operator to the Haskell hsAppend
{-# COMPILED_CORE _++_ (\_ -> (hsAppend)) #-}
```

(b) An Agda FFI binding to the above Haskell definitions.

Fig. 9: The FFI interface from Agda to UHC Core and Haskell.

6 Related work

A lot of work has been done on compiling dependently typed languages in the last few years, without which this project would not have been feasible.

Other Agda Compilers The existing Epic backend [17] was very important in the development of our compiler. We managed to reuse some parts of it directly; on other occasions, we could follow a very similar design. It is the most ambitious existing Agda compiler in terms of optimizations, and clearly shows the viability of compiling Agda to an untyped core language. Sadly, it has not been actively maintained and does not support all current Agda features. Furthermore, as we remarked previously, it is hard to support the existing FFI to Haskell using the Epic backend.

The MAlonzo backend, written by Benke [14], on the other other hand can interface freely with Haskell libraries. Compared to the Epic backend, MAlonzo is better maintained and somewhat more reliable; it does not support many of the optimizations that the Epic backend does implement.

Last but not least, there is also a JavaScript backend [11]. It supports some of the optimizations of the Epic backend and has a mature FFI to JavaScript. The motivation for targeting JavaScript was to use Agda for developing web applications. Using JavaScript as a target language for running code locally, however, may not be the best choice.

Coq Leaving the world of Agda, Coq supports the extraction of OCaml, Haskell and Scheme from Gallina. It suffers from similar problems as compiling Agda to Haskell as discussed in Section 2.1.

Idris Idris, created by Brady [7], is the dependently typed language aimed specifically at writing real-world runnable programs. To this end, it currently features a compiler targeting the LLVM intermediate language. This is a different design compared to the extractions to high-level languages used by Agda and Coq, as it targets a low-level language. While this approach may yield favorable performance, it severely limits the possibility of interacting with other high-level languages – an important constraint to support the existing FFI between Agda and Haskell.

7 Discussion

We have developed a working compiler for Agda, targeting UHC Core. We are able to compile the whole Agda standard library, together with a test suite of executable Agda programs. Our compiler has been integrated into the development version of Agda and is now a part of Agda.

In the future, we would like to investigate further optimizations inspired by the Epic backend. Initial performance benchmarks indicate that our compiler does not outperform the other backends. This is partially due to the lack of optimizations done by UHC itself, especially compared to more mature compilers like GHC.

Nonetheless, we have a new Agda backend which is suitable for further exploration. It already serves as the foundation for our currently ongoing research into creating a more sophisticated Agda FFI.

This paper shows how to tackle some of the challenges inherent in writing a compiler for Agda, piggybacking on the existing technology provided by UHC. We hope that providing a more robust backend for Agda, with excellent interoperability with an existing Haskell compiler, will provide the technology for more ‘real world’ dependently typed programs.

References

1. [agda] size limit on generated code?, https://lists.chalmers.se/pipermail/agda/2014/006990.html
2. Altenkirch, T., Danielsson, N.A., Löh, A., Oury, N.: ΠΣ: Dependent Types without the Sugar, pp. 40–55. No. 6009 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, http://link.springer.com/chapter/10.1007/978-3-642-12251-4_5
3. Atze Dijkstra: Stepping through haskell, http://foswiki.cs.uu.nl/foswiki/pub/Ehc/SteppingThroughHaskell/20051006-2023-phd.pdf
4. Augustsson, L.: Cayenne — A Language with Dependent Types, pp. 240–267. No. 1608 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, http://link.springer.com/chapter/10.1007/10704973_6
5. Augustsson, L.: A compiler for lazy ML. In: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming. pp. 218–227. LFP '84, ACM, http://doi.acm.org/10.1145/800055.802038
6. Brady, E.: Epic—A Library for Generating Compilers, pp. 33–48. No. 7193 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, http://link.springer.com/chapter/10.1007/978-3-642-32037-8_3
7. Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation 23(05), 552–593, http://journals.cambridge.org/article_S095679681300018X
8. Chakravarty, M.M.T., Keller, G., Jones, S.P., Marlow, S.: Associated types with class. In: Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 1–13. POPL '05, ACM, http://doi.acm.org/10.1145/1040305.1040306
9. Charguéraud, A.: The locally nameless representation 49(3), 363–408, http://link.springer.com/article/10.1007/s10817-011-9225-2
10. Dijkstra, A., Fokker, J., Swierstra, S.D.: The architecture of the utrecht haskell compiler. In: Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell. pp. 93–104. Haskell '09, ACM, http://doi.acm.org/10.1145/1596638.1596650
11. Jeffrey, A.: Dependently Typed Web Client Applications, pp. 228–243. No. 7752 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, http://link.springer.com/chapter/10.1007/978-3-642-45284-0_16
12. Letouzey, P.: A New Extraction for Coq, pp. 200–219. No. 2646 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, http://link.springer.com/chapter/10.1007/3-540-39185-1_12
13. Löh, A., McBride, C., Swierstra, W.: A tutorial implementation of a dependently typed lambda calculus 102(2), 177–207, http://dx.doi.org/10.3233/FI-2010-304
14. Marcin Benke: Alonzo - a compiler for agda, http://www.mimuw.edu.pl/~ben/Papers/TYPES07-alonzo.pdf
15. McBride, C., McKinna, J.: Functional pearl: I am not a number–i am a free variable. In: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell. pp. 1–9. Haskell '04, ACM, http://doi.acm.org/10.1145/1017472.1017477
16. Norell, U.: Towards a practical programming language based on dependent type theory
17. Olle Fredriksson, Daniel Gustafsson: A totally epic backend for agda, http://publications.lib.chalmers.se/records/fulltext/146807.pdf
18. Peyton Jones, S., Marlow, S.: Secrets of the glasgow haskell compiler inliner 12(5), 393–434, http://dx.doi.org/10.1017/S0956796802004331
19. Pfenning, F., Schürmann, C.: System Description: Twelf — A Meta-Logical Framework for Deductive Systems, pp. 202–206. No. 1632 in Lecture Notes in Computer Science, Springer

Berlin Heidelberg, `http://link.springer.com/chapter/10.1007/3-540-48660-7_14`

20. Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for GADTs. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 341–352. ICFP '09, ACM, `http://doi.acm.org/10.1145/1596550.1596599`
21. Sulzmann, M., Chakravarty, M.M.T., Jones, S.P., Donnelly, K.: System f with type equality coercions. In: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation. pp. 53–66. TLDI '07, ACM, `http://doi.acm.org/10.1145/1190315.1190324`
22. The Coq development team: The coq proof assistant reference manual, `http://coq.inria.fr`, version 8.0