Extracting LLVM Intermediate Representation from Agda

Jochem Broekhoff
Supervisors: Jesper Cockx, Lucas Escot
EEMCS, Delft University of Technology, The Netherlands

June 27, 2022

A Dissertation
Submitted to EEMCS faculty
Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Extracting LLVM Intermediate Representation from Agda

JOCHEM BROEKHOFF, Delft University of Technology, The Netherlands

Agda, a promising dependently typed function language, needs more mainstream adoption. By the process of code extraction, we compile proven Agda code into a popular existing language, allowing smooth integration with existing workflows. Due to Agda's pluggable nature, this process is relatively straightforward. We implement a solution in Haskell and perform an empirical benchmark analysis. We show that LLVM's Intermediate Representation language is a usable and promising target, although some optimizations are necessary before broader application. More indirect paths towards LLVM IR appear more suitable, because of the large translation gap.

Additional Key Words and Phrases: code extraction, Agda backend, LLVM Intermediate Representation

1 INTRODUCTION

Agda is a dependently typed functional programming language, part of the class of languages that embrace programming without side-effects. What distinguishes Agda from popular functional languages, such as Haskell, OCaml or F#, is its type system. Dependent types allow programmers to reason about correctness of their programs in a very fine way. In general, a value whose type is dependent has a type which depends on the actual value. This allows logic and more generic constraints to be encoded in types. For example, a list of items could have a type which captures the length, or similarly, one might write a function which takes such a list and returns another list whose length is guaranteed to be equal or less. This is, in the most general sense, not possible in non-dependently typed languages. Even though languages such as C++ have extensive macro processing and compile-time type instantiation and verification engines, dependent types are still more powerful.

Dependent types are useful for example to prove at compilation time that a program will never error[1]. Stronger yet, it can be used to write general-purpose proofs of any logic statement, by the Curry-Howard correspondence [Howard 1980]. Throughout the main body of this paper, several examples will be given to demonstrate this in more detail.

Other dependent functional languages exist, such as the newer Idris2 and the older Coq are popular alternatives. Executing an Agda program is normally done by compiling the source code into another (executable) format. This step normally happens via an intermediary language, which is why the process is also referred to as *code extraction.* Two compilation backends are provided out-of-the-box: Haskell and JavaScript [Agda Team 2022]. As the compiler implements a highly pluggable backend architecture, it allows the development of new and experimental backends with relative ease. This is exactly what we have used to research the applicability of using the LLVM intermediate representation (IR) as a code extraction target for Agda.

The choice for LLVM IR is particularly interesting because it is part of a large family of software [LLVM Team 2022d], which we would like to tap into. This is similar to how Agda by default compiles to Haskell, which opens up the possibility of interfacing with the entire Haskell ecosystem. Similarly, compiling to LLVM IR allows us to leverage many optimizations that have been implemented by the LLVM pipeline. Moreover, as LLVM IR works in C-compatible mode by default, we can link with native system libraries.

Although there are many similarities, there are at least as many differences. LLVM is a low-level language, situated somewhere in between pure assembly and C. This inherently means that we

[1]That is of course unless the programmer explicitly disables safeguards.

Author's address: Jochem Broekhoff, Delft University of Technology, Van Mourik Broekmanweg 6, Delft, The Netherlands, 2628 XE.

gain generality, while discarding high-level coherency and relation details. A safe assumption is that the GHC Haskell compiler is significantly more intelligent than the compiler backend we are te implement. Thus, it is unlikely that our backend will outperform Agda's default choice.

In order to systematically investigate the usefulness of such a rudimentary backend, the following research question is posed: 'Is LLVM IR a practical compilation backend for Agda?' Furthermore, it can be divided into several sub-questions, which will all be answered individually:

(1) To what extent can, and need, Agda types be mapped onto LLVM's type system?
(2) Which Agda features are necessary to answer the main research question?
(3) From an architectural perspective, how is the backend structured?
(4) How does the implementation compare time-wise to backends from the peer group and the built-in ones?

The content of this paper is structured as follows. First, we give some more in-depth background information in section 2. Section 3 briefly explains the method we used. Then, implementation details are outlined in section 4, followed by empirical results demonstrated and discussed in section 5. Then, we draw a conclusion in section 6, where we also mention possibilities for future work. Some related work is discussed in section 7. We end with a short statement on responsible research in section 8.

2 BACKGROUND

In this section we introduce the necessary background information about compilers, Agda, thunks and laziness. Specifically regarding Agda, we refer to Agda Team [2022]; van der Walt [2012] as excellent sources which introduce the language by many examples.

2.1 Compiler backends

It is in the interest of the language to support multiple backends, as this allows smoother integration by end-users in existing workflows. To give an impression, the competing languages Idris2 and Coq compile to C, Scheme [Idris2 Team 2020], and OCaml [Annenkov et al. 2021; Letouzey 2008], respectively. Even though Agda may often just be used to run proofs and play with the type system, it is perfectly usable for real-world applications. New backends not only provide library writers new mechanisms to deliver libraries with guarantees such as safety and finite-time termination. Moreover, it allows deeper integration with existing workflows of users, by providing a foreign-function interface (FFI). Using this FFI, it is not even necessary for the end-users to be programming anything in Agda.

Agda implements a pluggable backend architecture. This means that developers can write a new or custom backend for any target language without requiring deep knowledge of other internals. Conventionally, the heavy-lifting such as type-checking and high-level optimizations are done in the front-end phases of the compiler. This leaves the backend with a generic representation of the input program to map onto its specific target language.

This project will provide such a backend for the LLVM Intermediate Representation (LLVM IR) language. LLVM itself is the name, not an abbreviation, of the project of a set of "reusable compiler and toolchain technologies" [LLVM Team 2022d]. Well-known projects belonging to LLVM are the Clang C/C++ compiler [LLVM Team 2022a], the LLDB debugger [LLVM Team 2022c] and the LLD linker [LLVM Team 2022b]. Separate from this, the LLVM Core project provides the necessary resources for the LLVM IR language, an integral part of the entire LLVM project family. It is a static-single assignment (SSA) format in which many parts of a typical LLVM pipeline communicate. The Clang compiler, for example, parses the C/C++ code and emits LLVM IR for further processing by several optimization passes, followed by a final translation to machine code. Similarly, the

Fig. 1. Grammars. Some parameters are named for clarity.

(a) Agda treeless

```
⟨Term⟩    ::= Var ⟨idx⟩
            | Prim ⟨Prim⟩
            | Def ⟨QName⟩
            | App subj:⟨Term⟩ args:⟨Term*⟩
            | Lam ⟨Term⟩
            | Lit ⟨Literal⟩
            | Con ⟨QName⟩
            | Let var:⟨Term⟩ body:⟨Term⟩
            | Case ⟨idx⟩ ⟨CaseInfo⟩ ⟨Term⟩ ⟨Alt*⟩

⟨Prim⟩    ::= Add | Add64 | Sub | Sub64 | ...

⟨QName⟩ ::=  a fully qualified identifier

⟨CaseInfo⟩ ::=  case meta-information

⟨Literal⟩ ::= Nat ⟨integer⟩
            | Word64 ⟨integer64⟩
            | Float ⟨double⟩
            | String ⟨string⟩
            | Char ⟨char⟩
            | QName ⟨QName⟩

⟨Alt⟩     ::= ⟨match value⟩ body:⟨TTerm⟩
```

(b) Mid-Level Intermediate Representation

```
⟨module⟩  =  ⟨Entry*⟩

⟨Ident⟩   ::= Ident ⟨string⟩
            | IdentRaw ⟨string⟩

⟨idx⟩     =  ⟨integer⟩

⟨Entry⟩   ::= Thunk ⟨Ident⟩ private:⟨bool⟩ ⟨Thunk⟩
            | Direct ⟨Ident⟩ pushArg:⟨bool⟩ ⟨Body⟩
            | Main ref:⟨Ident⟩
            | Alias ⟨Ident⟩ aliasOf:⟨Ident⟩

⟨Thunk⟩   ::= Delay ⟨Body⟩
            | Value ⟨Value⟩

⟨Body⟩    ::= MakeValue ⟨Value⟩
            | Apply subj:⟨Arg⟩ args:⟨Arg*⟩
            | Case subj:⟨idx⟩ alts:⟨Alts⟩ fallback:⟨Body⟩
            | Error ⟨string⟩

⟨Arg⟩     ::= Extern ⟨Ident⟩
            | Record ⟨idx⟩
            | Erased

⟨Alts⟩    ::= Data ( ident:⟨Ident⟩, arity:⟨integer⟩, ⟨Body⟩ )*
            | Nat ( ⟨integer⟩, ⟨Body⟩ )*

⟨Value⟩   ::= Data id:⟨integer64⟩ arity:⟨integer⟩
            | Function ⟨Ident⟩
            | Literal ⟨Literal⟩
```

Agda2LLVM backend will emit LLVM IR which is to be optimized, assembled and linked by further processing parts, e.g. Clang.

2.2 Quick Intro to Agda and its Syntaxes

Agda internally uses several representations of the input, some of which the backend can use. The most simplified syntax is called the *treeless* form [Agda Team 2022], which is a very low-level syntax which only supports the lambda form and `case` and `let` constructs, including some primitive operations. Most of its grammar is listed in grammar 1a. As the documentation mentions: this syntax is "not used for type checking", but "intended to be used as an input for the compiler backends". Roughly this syntax level is what other functional language compilers mostly operate on, in terms of actual translation to imperative or more machine-specific code. Our challenge is to compile this to LLVM IR. GHC, Haskell's compiler, solves a similar problem by leveraging its Spineless Tagless G-Machine (STGM) [Peyton Jones 1992], followed by a translation through C-- [Peyton Jones et al. 1999] to machine code.

Although Agda has overlap with Haskell, there is still a non-empty difference [Agda2HS Team 2021]. Natural numbers in Agda, represented by the built-in type ℕ, are defined as the Peano numbers, as is shown in listing 1. Fortunately, to mitigate severe overhead of actually using Peano numbers at runtime, Agda provides built-ins which allow backends to leverage the target's numeric primitives. This not only holds for natural numbers, but a wide range of similar primitives.

Continuing on the natural number example, we can use it to define a vector type, a list type of fixed length. Listing 2 shows the Agda implementation. Observe that the empty (`[]`) constructor returns a vector with elements of the required type, and a length set to zero. Correspondingly, the

Listing (1) The natural numbers defined in Agda, as Peano numbers.

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

Listing (2) A vector with its length embedded in the type.

```
data Vec (A : Set) : (l : Nat) → Set where
  [] : Vec A zero
  _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)
```

Fig. 3. Two states of a Thunk, expressed as a Haskell data type.

```
data Thunk a =
    Forced { evaluatedValue :: a }
  | Delay { evaluator :: (Context -> a), context :: Context }
```

cons (`_::_`) constructor returns a vector that is one longer (by `suc n`, the successor of `n`) than the vector to which an element is concatenated. The previous length `n` is derived implicitly, which is why this meta-variable is defined inside curly braces instead of parentheses. Note that the name of the cons constructor contains two underscores: these indicate that the constructor can be used as an operator, with operands at the positions of the underscores. Constructing a vector can thus be done like `x :: y :: []`, which has the type `Vec A (suc (suc zero))`, where `A` is the type of `x` and `y`.

2.3 Theory Recap: Thunks & Laziness

In the implementation we describe in section 4, but also in the MLIR syntax we just listed, we heavily rely on the concept of thunks. A thunk is, according to Raymond [1991] in his dictionary: "an expression, frozen together with its environment, for later evaluation if and when needed", and "the process of unfreezing these thunks is called 'forcing'."

We use thunks to achieve laziness. Being lazy means not executing code that is not necessary *right now*, but only when it *really* is needed, i.e. when the program cannot otherwise continue its execution. Passing arguments to a function which ends up not using the argument, means that all code related to that argument is never executed. This allows working with infinite data structures, such as lists of infinite length, as the list's elements are only computed one-by-one. Where in imperative languages such as Java, complete pipeline processing frameworks exist[2], lazy languages such as Haskell or Agda provide most of the logic implicitly.

In practical terms, a thunk can be represented by a simple data structure, as demonstrated in figure 3, listed in Haskell syntax[3]. A thunk can only exist in two states: either it is already forced and a value is readily available, or it is not yet forced and thus still delayed. In that case, only a reference to a function that can produce the value is stored, alongside some context information. The value of such a context heavily depends on implementation details, but in general it is a capture of the context where the thunk was instantiated. This way, the evaluator can for example access local variables.

When a force point in the program is reached, and the thunk appears to still be delayed, it will be forced. The evaluator is executed, given its context, to produce a value. The thunk is then atomically replaced with a `Forced` instance, which includes the just computed value. Because thunks can be *shared*, replacing it with a forced value instantly propagates the change to all other locations where the thunk is used.

[2]For example, Java Stream, introduced in version 8.
[3]Haskell itself is already lazy, so thunks are already implicitly used. This code just demonstrates the different thunk states.

3 METHOD

The method exists of two parts: implementing the backend (3.3) and analyzing its performance characteristics by the means of running benchmarks (3.4), in order to answer the research questions. Before we describe these two, we take note of the actual scope that applies to the implementation (3.1), as a fully ready-to-use implementation cannot be expected. Additionally, we mention some supporting material that may be useful for reading or repeating an implementation (3.2).

3.1 Scope

Agda offers a plethora of features. Although function bodies are abstracted away in the treeless form we deal with, not all treeless features have to be supported. Besides function definitions, we must also deal with primitives and constructors for data types. In the implementation section about these different kinds of definitions (4.4), we mention how we use them exactly.

We actually were able to implement support for almost all treeless terms, but opted to disregard some of its literals. Additionally, we decided to only implement a subset of all possible primitive operations. Both are for the same reason, namely that literals, on which primitive operations occur, are largely redundant for analysis sake. Of course, for broad practical use, this does not suffice.

3.2 Supporting Material for Implementing

First and foremost, Agda already comes with several backends [Agda Team 2022], which serve as a good starting point for a new implementation. In addition, our supervisor had prepared a repository with an example implementation of a stand-alone backend which compiles to Scheme [Cockx 2022]. We took inspiration from both works, because the Agda documentation on internals is somewhat lacking. Fortunately, our supervisor is an active contributor to the Agda code base.

On the other hand, the LLVM IR language is fully documented, accompanied by plenty examples [LLVM Team 2022e]. We suggest scanning this language reference to those who are unfamiliar with LLVM IR. In addition, the Clang compiler can emit LLVM IR instead of machine code, and can thus be used to inspect how Clang would translate a particular construct in C/C++ to LLVM IR.

3.3 Work Plan

The first preparatory block of work concerns devising reference implementation matter. Based on simple programs, such as a constant value function containing a literal, or the identity function, we manually wrote a `.ll` file with a working implementation. To aid this process, we wrote some parts in C and viewed the IR emitted by Clang to get ideas about how to implement constructs such as C-style structs and unions. During this manual process, most of the (re-usable) core functionality can be determined already, including which parts may need to be abstracted into a separate runtime support library.

Based on the experience gathered in the 1-2 weeks these tasks took to complete, we estimated which features were feasible to implement. The majority of the remaining time was spent implementing as many feature as possible in actual Haskell code.

3.4 Benchmark Procedure

To evaluate the performance of our backend with respect to other Agda backends or itself, we establish a common benchmarking procedure. The procedure is straightforwardly divided into four steps which are executed for each backend, for each fixture, for each input:

(1) Create a clean temporary working directory.
(2) Devise the input source code according to the input configuration, placing it in the fresh temporary directory.

(3) Compile the code with optimal configuration flags.
(4) Execute the output several times, collecting timings. Discard the first result, as this is a warm-up. Take the median of all remaining points as the final measured value.

To ease the process of constructing inputs, a templating facility can be used to dynamically emit the Agda sources to be compiled. Data points gathered from all executions of these steps can then simply be plotted directly.

4 IMPLEMENTATION

Corresponding to the methodology, we will describe most relevant implementation details in this section. Due to paper length constraints, we cannot cover every aspect, as this would also require an extensive introduction into the preexisting Agda backend. We will instead describe the transformation process on a high level, diving into more details and nuance where necessary. Understanding the full transformations is also not essential to the interpretation of the results, but can be helpful discussing the results. For those who want to read the full transformation, we refer to the two main implementation files[4] in the repository[5].

We start off by hooking into Agda's backend plug-in mechanism, which requires a specification of the backend's steps and some meta data. In the latter, we specify the name and some command-line options we want to expose, such as a strict/lazy flag or debug/release mode. The steps we implement are *compileDef*, *postModule* and *postCompile*, where the results of a previous step are aggregated and passed on to the next. All logic is implemented in the same programming language as Agda itself, namely Haskell.

From the highest point of view, the pipeline that is formed by the steps corresponds approximately to one divided in three stages, each of which we discuss in a separate subsection. First, the treeless syntax is transformed into a mid-level intermediate representation (4.4), which is then immediately transformed into LLVM IR itself (4.5). The last step is bundling all of this with a runtime library, which is then fed into the Clang compiler, to produce the final executable (4.7).

Before we dive into the details, we first explain why an additional intermediate stage makes sense (4.1). We follow this by giving some examples, as introduction (4.2).

4.1 Why an Intermediate Stage?

The two-stage design, transforming treeless into MLIR, and transforming that into LLVM IR, comes into existence naturally. There are two reasons for this.

First, LLVM's IR is notoriously verbose, as it lies in between C and an assembly language: there are proper function calls and variables, but that is about it. Most expressions in, say C, will correspond to numerous lines in LLVM IR. One of the reasons for this, is that memory access and transfer must be stated explicitly, expressed in one operation per line. Thus, each subexpression in general corresponds to at least one line. The verbosity of LLVM IR leads to the desire for abstraction, as repeating patterns can be identified. Each concept that can be distilled from the total LLVM IR, can correspond to a notion in the MLIR.

The second and most important reason, is that another intermediate language allows for better reasoning about a program. MLIR introduces the concept of thunks, which are not conceptualized in the treeless syntax. And for good reason: treeless just serves as a general enhanced λ-calculus, as a neutral carrier of Agda code. Since thunks form an essential concept, transforming treeless into MLIR directly gives a different perspective on the program.

[4]`src/Agda/Compiler/LLVM/ToAbstractIntermediate.hs` and `src/Agda/Compiler/LLVM/AbstracToLLVM.hs`
[5]https://github.com/jochembroekhoff/agda2llvm

Fig. 4. Example: the identity function

Listing (3) Agda

```
id : Nat -> Nat
id x = x
```

Listing (4) *Treeless*

```
id = λ $0
-- = Lam (Var 0)
```

Listing (5) MLIR (optimized)

```
(Thunk "id" private=True
  (Value
    (Function "id.lam.0")))
(Direct "id.lam.0" push=True
  (Apply (Record 0) []))
```

Fig. 5. Example: factorial function

Listing (6) Agda

```
fac : Nat -> Nat
fac zero = 1
fac n@(suc m) = n * fac m
```

Listing (7) *Treeless*

```
fac = λ case.nat $0
  #0 -> #1
  _ ->
    let _-_ $0 #1
    in _*_ $1 (fac $0)
```

4.2 Examples

To give an impression of what (simple) Agda programs and their treeless and MLIR equivalences look like, we give some examples. The textual representation of the treeless syntax is not strictly defined, but follows a general Haskell-esque style. Here, a hashtag (#x) indicates a literal natural number x, and a dollar ($$n$) refers to the nth-back variable in scope (de Bruijn index). Lambdas (λ) and **`let`** push a new variable in scope.

For the first example, an identity function (listings 3–5), the type signature is not fully representative of a real-world implementation. For brevity, we did not make the function polymorphic, but instead constrained it to `Nat` only. The problem is, if we were to use a proper polymorphic signature, such as `id : {A : Set} -> A -> A`, both the treeless code and the MLIR become needlessly complex for these demonstration purposes. This is due to the fact that the `{A : Set}` named parameter is implicit (this is what the curly braces indicate). To the programmer, it is in fact implicit, but from the perspective of a backend, it really is a concrete parameter. In practice, its value is often a placeholder (*Erased*), as the value is not used anywhere in the function, merely for type checking.

The second example, the factorial function (listings 6–7), might appear to have an unexpected treeless form. In this case, Agda optimized the natural numbers into a case expression for literals. Without this optimization, the case would switch on `zero/0` and `suc/1` data. However, because the constructors are replaced with actual literals, no instances of the `Nat` data are passed around at runtime. The full constructor names are only used to prove correctness to Agda, which, in turn—apparently, lowers the implementation to literal numbers. Due to space constraints, we cannot list the MLIR syntax here, as it is already very verbose. We have instead included it in appendix C.

4.3 The Record at Runtime

We often use the words *record* or *current frame* in relation to variables. More precisely, by means of a numeric index, called a de Bruin index. The record can be viewed as a stack of frames, which can only ever be appended to. Only implicitly, frames are popped as they are collected by the GC. In all calls, a record is present. Notably when allocating a delayed thunk, the record forms the context for the thunk's evaluator.

From the perspective of the treeless syntax, there are three occasions where frames are pushed to the record: lambdas (λ), **`let`**-expressions and in a successful case match. Matching on some data, e.g., `(a , b)` first pushes the entire value, then a, then b, with respective de Bruijn-indices 2, 1, and 0.

4.4 From Agda's Treeless Syntax to Mid-Level Intermediate Representation

The first stage is transforming Agda's structure, primarily the *treeless* syntax, into the MLIR, of which the grammar is given in figure 1b. Starting from the *compileDef* callback, which is invoked for each Agda definition, the translation is started. In case any of the definitions is the main one, an additional *Main* entry is emitted, referring to the particular definition.

As an Agda definition comprises all aspects of the language, we need to do different things, based on the particular case. We handle the cases *Function*, *Primitive* and *Constructor*, which cover almost all language features. A handful others are available, but simply skipped over.

Function definitions are only handed to us in Agda's internal syntax. We instead operate on the treeless syntax, which is why we ask Agda to transform it for us, if applicable at all. A function is defined by its name and a treeless body term (*T.Term*), as listed in figure 1a. The goal is to map the body to an MLIR body (*M.Body*), which is wrapped into a named entry (*M.Entry*). We achieve this by applying a handful of transformation rules on the body term, which are described in full in appendix A. This results in *M.Ident*, the full identifier of the current function, and some auxiliary entries which are produced as by-products in each internal transformation step.

At the top-level, each function is emitted as a thunk. That means, from the perspective of MLIR, if a function is called, it *just* returns a thunk, describing its *actual* body. Consequently, applying to a function first requires the applier to force the thunk to which is applies. Similarly, when a case analysis is done, the scrutinee (the variable which is matched on) must be forced.

Primitives are implemented in a comparable way. As primitives do not have a body[6], their name serves as the starting point for the transformation. The result of the transformation is merely an alias entry, which requires an external definition that is later linked in. We implemented relevant primitives in an auxiliary LLVM file.[7] Primitives that we did *not* implement will therefore cause a link-time error, later down the line.

We regard *Constructors* as syntactic sugar for a function of a similar signature, which returns the instance of the data type. There are however still two minor challenges in this regard.

First, a data instance captures the record to extract from at match-site, which we rather do not if not necessary. This can be prevented only in case of zero-arity constructors. Thus, we dedicate a special case for these, which is implemented as the following *M.Entry*: $\text{Thunk}\ q\ \text{False}\ (\text{Delay Data}\{\dots\})$ where q is the base name of the constructor and $\{\dots\}$ contains the constructor meta-data.

In all other cases, we need to progressively generate the curried lambda, which results in three different entries that are emitted. The entry that is exported, which has the name of the constructor, returns a thunk to the root lambda. The root lambda entry is part of the other lambda entries, which are direct values which simply refer to the next level, while recording the argument. The final lambda does not return another lambda, but the value instead, capturing the current argument stack.

[6]In practice they sometimes do, but this is only intended to be used to prove the program correct.
[7]See `data/AgdaPrim.ll`, and `data/gen/AgdaPrimWrap.ll` which is generated by the `gen_agda_prim.py` script.

Listing 8. Demonstration of constants, declarations and definitions in LLVM IR.

```
@str.Hello = private constant [i8 x 6] c"Hello\00"
declare %struct.my_struct* @get_from_my_stuff(%struct.my_stuff*, i64)
define %struct.my_struct* @example(%struct.my_stuff* %stuff) {
  %v = call %struct.my_struct* @get_from_my_stuff(%struct.my_stuff* %stuff, i64 0)
  ret %struct.my_struct* %v
}
```

4.5 Mid-Level Intermediate Representation to LLVM IR

Whereas the translation described in the previous subsection was relatively straightforward, the translation from MLIR to LLVM IR requires a little more attention. This process is split into two, first a small optimization pass, followed by the actual transformations. The latter would take too much space in this paper to cover in detail, thus we only cover its essentials. Unfortunately, we also cannot show any real pieces of emitted LLVM IR, as in practice the ratio of Agda code to LLVM IR is in the order of $1 : 30$.

4.5.1 A Small Optimization. We find one trivial optimization that can be applied to entries in the MLIR, converting any value that is generated in a delayed thunk to a pre-evaluated thunk:

$$\textsc{Opt-Entry-Thunk-Const}\quad \frac{}{\text{Thunk } q\ p\ (\text{Delay } (\text{MakeValue } v)) \longrightarrow \text{Thunk } q\ p\ (\text{Value } v)}$$

4.5.2 MLIR Entries to LLVM IR Entries. Now, each MLIR module, which is a list of entries, can be transformed into one or more LLVM entries. For our intents and purposes, an LLVM entry is one of the following: definition of a function, declaration of a function, definition of a constant value. The naming difference between 'definition' and 'declaration' of a function is subtle, but captures a significant difference: a definition contains an implementation, whereas a definition is just information about a function. Only at link time, when the system linker links all objects[8], definitions and declarations are brought together. Listing 8 demonstrates the differences. For other constructs that are relevant to the paper, we will give brief explanations, but for most details we refer to the reference manual [LLVM Team 2022e].

All four kinds of entries in MLIR, *Thunk*, *Direct*, *Main* and *Alias* correspond to at least one LLVM definition. As *M.Thunk* and *M.Direct* bear some body, additional entries may be generated if necessary, because inner bodies are lifted out. The latter two, *M.Main* and *M.Alias* are simply short wrappers for the reference they indicate. The main entry, of which only one may exist[9], calls the runtime library with a reference to the *actual* main implementation. Similarly, an alias calls the function of which it is an alias, simply passing all arguments and the return value along.

Thunk entries can either be delayed (*M.Delay*) or immediate (*M.Value*). At runtime, any delayed thunk will be replaced with an immediate thunk once it is forced. However, if we know in advance that the value of a particular thunk is fixed, we can return a thunk which does not need forcing (see Opt-Entry-Thunk-Const). In general, a thunk entry just returns a thunk pointer, which is completely opaque to the caller: it must not care what the state of the thunk is.

Assuming a thunk is delayed, and thus includes a body, its body needs to be lifted into its own definition. This auxiliary definition is the evaluator function for the thunk, i.e. the function that is executed once the thunk is forced. LLVM does not implement any closure or lambda mechanisms,

[8]An object, a `.o` file on Linux, is machine code that is yet to be bundled into either a library or a stand-alone executable.
[9]Because the entry is exported globally as the standardized symbol `main`.

Fig. 6. Thunk body lifting demonstrated.

Listing (9) MLIR

```
true = Thunk.Delay public
  MakeValue Data true/0
```

Listing (10) MLIR (optimized)

```
true = Thunk.Value public
  Data true/0
```

Listing (11) LLVM IR (simplified, from unoptimized MLIR)

```
define %thunk* @true(%frame* %record) {
  %t = call @alloc.thunk.eval(@true--body)
  ret %thunk* %t
}
define internal
%value* @true--body(%frame* %record, %thunk* %arg) {
  %v = call @alloc.value.data("true", i64 0, %frame* %record)
  ret %value* %v
}
```

Fig. 7. Pseudocode implementations for the remaining MLIR entries.

Listing (12) *M.Apply*

```
gather subject thunk
gather argument thunks
dispatch to runtime helper
```

Listing (13) *M.Case*

```
call case id helper
use llvm switch statement
continue exec to branch
```

Listing (14) *M.Error*

```
<print the error>
system exit(1)
```

which is why we need to implement it manually. The example from figure 6 demonstrates this behavior. There, it is also shown that the creation of a thunk instance captures the current record pointer.

In the same example, we show what the evaluation implementation looks like for a *M.MakeValue* body. The signature of the function is identical for all implementations (*M.MakeValue*, *M.Apply*, *M.Case*, *M.Error*), only the instructions leading up to the construction of %v differ. Figure 7 describes, in pseudocode, the implementations for each of these three remaining constructs.

4.6 Dealing With or Without Laziness

For theoretical analysis, it is interesting to explore the effects of disabling laziness in a lazy language, as in practice the popular functional languages are strict. Fortunately, we do not have to write two complete separate transformation rules for each runtime approach. In fact, there is only a single difference at one point in the entire process. Namely, at the point where normally a delayed thunk would be returned, we instead call the evaluator *right away* and put its result into the thunk we return. To the caller, a thunk is still returned. The caller may pass this thunk along, until at some point it is forced in order to be used. As thunks are fully opaque to the program, they simply appear as being already forced, and thus there is no need for any other modification elsewhere.

4.7 Final Assembly via Clang

Finally, from the *postCompile* backend callback, the final output binary can be assembled. Here, all LLVM modules are collected and written to disk at an intermediate storage location. This is achieved by pretty-printing the in-memory representation into the standardized format.

In addition, we generate a auxiliary LLVM file. This contains alias definitions for built-ins[10], which are referenced by the runtime library. We need to delay this generation, because the built-ins are dynamic, while the runtime library is static.

The runtime library is simply a collection of LLVM files, which are included as data files in the compiler's resources. These are passed along to Clang identically to how the locally emitted files are. Depending on command line flags, different implementations may be selected. The signatures of the functions found in the runtime library can be found in appendix B.

The paths to all emitted files are collected and passed to Clang as command-line arguments. We need to configure Clang with some more arguments. First, as we use the Boehm GC library, we add the flag `-lgc`, which tells Clang to link the `gc` library. Second, to enable all optimizations LLVM has to offer us, we add the flags `-flto` and `-O3`. The former indicates the wish to enable link-time optimization (LTO), the latter sets the optimization level to 3 (max). Last, in case the Agda compiler is called with the explicit request to not output a binary, i.e. no main function, but a library instead, we add the flag `--shared`, which requests Clang to generate a shared library[11].

After the Clang process exits, we end up with either an executable binary (`./out.bin`) or a shared library (`out.so`). These are the results of the compilation, and thus we are finished.

4.8 Static Allocation Optimization

At the top-level of each LLVM file, we end up with many public functions that allocate and return delayed thunks. Generally speaking, the allocation and configuration of a delayed thunk includes two steps: setting the reference to the evaluator, and capturing the current record. We observe that capturing the current record does not have any benefit for these top-level thunks. It is impossible for the evaluator of such thunks to reference *anything*, as there must always first be some instruction that introduces variables in scope, whether this is explicitly done via a `let` construct, or implicitly by a λ. Thus, the current record can be nullified and stored that way.

Consequently, we observe that both values that are stored into the delayed thunk are in fact constant: the reference to the evaluator is the memory address of the implementing function, and the record pointer is null. Fortunately, it is possible to express static memory allocation in LLVM IR. We leverage this feature by emitting such allocations for the type of thunks we observed earlier. This optimization should have an effect on the amount of memory allocations done, and thus should also affect the effective running time. Therefore, we analyze the effect of this particular optimization in the benchmarks.

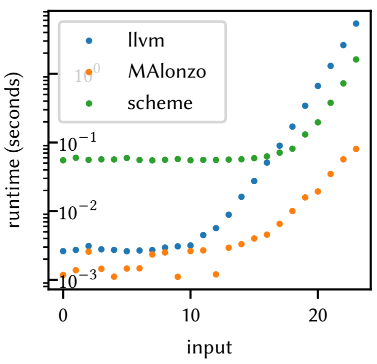
5 RESULTS & DISCUSSION

By means of the benchmark procedure outlined in section 3.4, we test our implementation against several others[12]. We subject the implementations to three different *fixtures*. In ConsumePow2, the input is given as a Peano number, to which power we raise 2, after which the resulting number is traversed (consumed). In QuickSort, we run the quick sort algorithm [Hoare 1961] on a list produced from the linear range $[0, n]$ where n is the input, then lookup the nth element (effectively traversing the entire sorted list). Finally, in Triples, we compute the sum of the bag $\{x + y + z \mid x, y, z \in \{1 \dots n\} \wedge x^2 + y^2 = z^2\}$, where n is again the input. For the full source of the different benchmark fixtures, we refer to the repository[13].

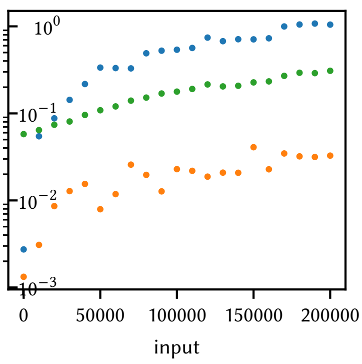
[10]Built-ins are Agda definitions which are marked with a pragma to indicate a particular behavior, such as booleans.
[11]E.g., `.so` or `.dll` files.
[12]All benchmarks were run on the same machine: AMD Ryzen 5 1600 (6c/12t) with 8 GiB memory free, running Linux Zen kernel 5.18.
[13]https://gitlab.ewi.tudelft.nl/jcockx/agda-extractors, directory `benchmarks/fixtures`.

Fig. 8. Benchmark results comparing different backends with their default configuration.

(a) *ConsumePow2* fixture.



(b) *QuickSort* fixture.



(c) *Triples* fixture.

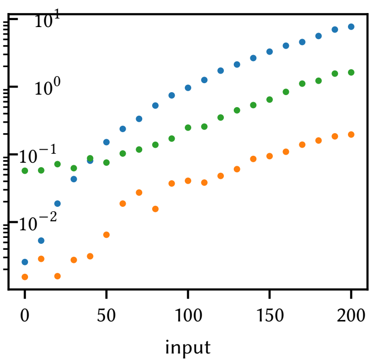
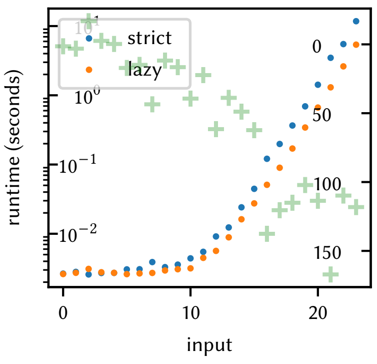
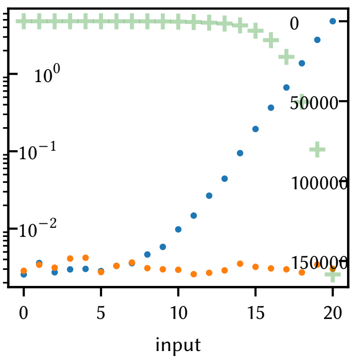


Fig. 9. Benchmark results comparing strict evaluation with the default lazy evaluation.

(a) *ConsumePow2* fixture.



(b) *QuickSort* fixture.



(c) *Triples* fixture.

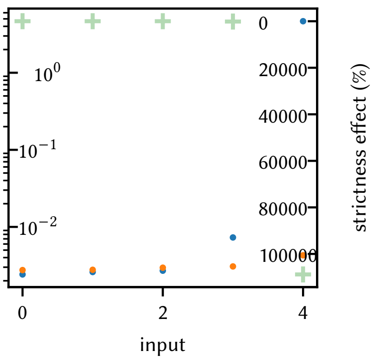
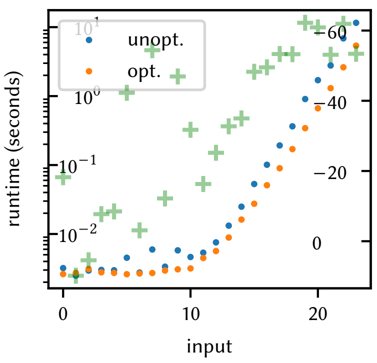
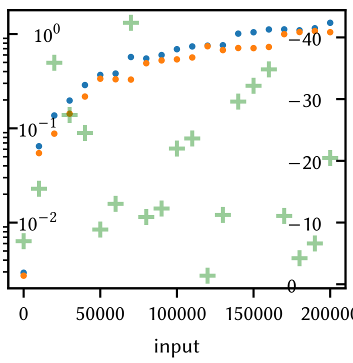


Fig. 10. Benchmark results for the effect of the static thunk allocation optimization.

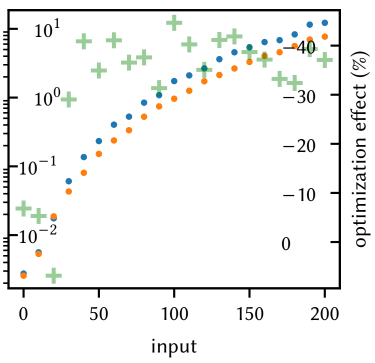
(a) *ConsumePow2* fixture.



(b) *QuickSort* fixture.



(c) *Triples* fixture.



The first result set, is visualized in figure 8. Here, we compare three different backends: our Agda2LLVM implementation, the Haskell one ('MAlonzo') and our supervisor's Scheme one. Note that all graphs have the left Y-axis in logarithmic scale, which allows better distinction for small runtime differences. The ConsumePow2 fixture's inputs are linearly increasing, but are effectively exponential, as the benchmark consumes a number that is 2 to the power of the input.

Next, we analyze the relative effect on running time when switching to lazy evaluation, shown in figure 9. It only makes sense to compare performance between different configurations of our benchmark. Even though the Scheme backend can easily be configured to use strict evaluation, it is not relevant to this paper. The input space has been severely reduced for the QuickSort and Triples fixtures, otherwise it would take too long to run the benchmarks.

Finally, we measure the performance effect that the static thunk allocation optimization has. This optimization was described in section 4.8. Both this and the previous set of measurements are to be compared in relative terms only.

As we had relatively little time for the entirety of the research, the implementation was kickstarted by working out a minimal working example in C code. This, in turn, was compiled to LLVM IR and formed the basis of our implementation. Although not completely unavailable, we expect this to influence the implementation and results negatively. However, it is difficult to estimate to what extent.

Furthermore, there are many optimizations left to be implemented. We have run all benchmarks with the system stack set to unlimited size[14]. However, by default, stack-overflow errors would occur before all benchmarks could be completed. Tail-call optimization (TCO) seems the most promising solution to mitigate this, as it should largely prevent the unbounded stack growth in many occasions for self-recursive, or even mutually-recursive functions.

6 CONCLUSIONS & FUTURE WORK

We implemented a new backend for the Agda language and evaluated its performance with respect to the default backends and the new Scheme one. The main question in this regard was 'Is LLVM IR a practical compilation backend for Agda?'. In some aspects, the answer is *yes*, while in other regards it is *no*. We answer the sub-questions in order.

Agda's dependent type system has not posed any difficulty during implementation. We opted for a fully type-erased runtime approach, ensuring that there is little common ground necessary between Agda and LLVM, only for the primitive values. This was completely safe to do, as we can rely on Agda's analyzer that no runtime errors can occur *unexpectedly*.

The benchmarks we have run show that with the features we implemented, it is possible to do meaningful comparisons. In fact, we implemented most Agda features. Notable exceptions are support for a foreign-function interface and support for all primitive data types and operations among them. In that aspect, we only implemented support for natural numbers. We observe that many of Agda's important features are in fact largely abstracted away by the treeless representation.

The introduction of a new auxiliary intermediate representation, the Mid-Level Intermediate Representation (MLIR), served us well in adding conceptual support for thunks to Agda's treeless syntax. It allowed us to reason, albeit to a small extent, about optimizations *within* MLIR. Otherwise, our pipeline exists of four distinct stages: (1) Agda Treeless (2) to MLIR (3) to LLVM IR (4) Clang.

From the benchmarks, we observe that our backend in general does not outperform MAlonzo. This is largely to be expected, as Agda is heavily inspired by Haskell's syntax and thus largely maps onto it cleanly. Consequently, most of the original meaning of the program is kept, which allows GHC (Haskell's compiler) to apply all of its optimizations, which are tailored for lazy functional

[14]See manpages on `setrlimit.2`.

languages. On the other hand, for small inputs our backend is in line with MAlonzo, namely beating Scheme. This is, however, simply explained by the fact that Scheme has a large start-up overhead. In strict evaluation setting, we observe that enabling strict evaluation almost always leads to worse execution times. Only for very small inputs, there is a tiny speed-up noticeable. Both were expected, because under small inputs, strict evaluation wins because it saves on calls to thunk evaluators, while for large inputs, the program is evaluated more than actually is necessary. Lastly, our optimization of static memory allocation made significant impact on running time: 40% speed-ups are measured, or sometimes even higher, see figure 10. This is very promising, as this was the only significant optimization.

Regarding the fact that we can answer all sub-questions positively, we must conclude that it was practical to compile to LLVM IR. Unfortunately, the approach we took was perhaps too straightforward, partially due to the constrained research time. Although we are already able to run many programs with a relatively simple compilation pipeline, more generic efforts of compiling functional languages to imperative languages seem more promising. In that regard, LLVM IR *is* a very suitable target, as it is one of the lowest-level interfaces to compilers for imperative languages. Its theoretical performance is extremely high, as it is *the* format used by state-of-the-art C/C++ compilers. The challenge of mapping functional to imperative still remains.

The usage of an auxiliary intermediate format, MLIR, naturally came to existence in the first few weeks of the project. When adding more optimizations, it appears logical that more intermediate formats would arise. This leads to a more and more indirect path to LLVM, which largely is the approach that related work takes (see section 7). Therefore, we must also note that an indirect path, with numerous intermediate stages, *seems* very suitable.

We observe that more work could be put towards bringing existing imperative optimizations to our pipeline, as the current state of implementation is by no means desired. Notably tail-call optimization is an optimization that appears almost crucial. Furthermore, there are several optimizations in the area of memory management that seem obvious: (1) move to a different memory model, reducing the reliance on a garbage collector (2) leverage LLVM's garbage collector primitives to gain more insight into allocations (3) perform more static allocations.

7 RELATED WORK

Agda has never had a native LLVM backend, however it has already been possible to obtain an LLVM translation of any Agda code. In some related work, we notice two different approaches.

First, Agda compiles, to Haskell by default. In turn, Haskell can compile to LLVM [Terei and Chakravarty 2010]. This LLVM backend for Haskell is still experimental. We also know that the Haskell compiler is not well-known for its speed. In relation to Agda, this is even more relevant, as Haskell will do a lot of redundant work, which has already been done by the Agda type checker.

Secondly, the older GRIN framework [Boquist and Johnsson 1997] has gotten support for LLVM due to the Utrecht Haskell Compiler (UHC)[15] extending its GRIN-based pipeline to LLVM [Dijkstra et al. 2008; van Schie 2008]. Slightly more recently, an Agda backend was written, which was able to emit GRIN [Turk 2010]. In a similar line, Frederiksson and Gustafsson [2011] have implemented a compiler from Agda to the Epic language. Not long ago, extensive efforts were made by Hausmann [2015] to fully integrate Agda with UHC, inspired by the Epic backend. All of these appear very promising, especially the UHC implementation, which solves some problems that plague MAlonzo. Unfortunately, the UHC project is not actively maintained, contrary to GHC.

[15]Previously known as Essential Haskell Compiler (EHC).

8 RESPONSIBLE RESEARCH

Reproducibility of results and ethical considerations are of high priority in our research. Therefore, we make the full source code of the project available open source, to be found at https://github.com/jochembroekhoff/agda2llvm.

As described in the implementation (section 4), we mainly rely on the Agda and LLVM compiler infrastructures. Both projects are publicly available as open source projects. Similarly, all our (transitive) dependencies are licensed as open source and publicly available.

Part of this paper includes empirical results (section 5), which were obtained by the predetermined method (section 3). This procedure has been implemented in the form of an ad-hoc framework, separate from the project repository. Built in collaboration with the entire peer group, it is publicly available at https://gitlab.ewi.tudelft.nl/jcockx/agda-extractors.

Finally, ethical concerns could not be identified. Our implementation is not intended for processing of personal data, and, if used for that purpose, it is fully open source.

ACKNOWLEDGMENTS

To my supervisors, Jesper Cockx and Lucas Escot, for guiding me through this process and giving me critical feedback on this paper. To my peer group, Hector, Louis, Lukas and Matteo, for providing me insightful feedback in all aspects of this project. To some of my friends who reviewed this paper.

REFERENCES

Agda Team. 2022. *Agda v2.6.2.1 documentation.* Retrieved June 15, 2022 from https://agda.readthedocs.io/en/v2.6.2.1/
Agda2HS Team. 2021. *GitHub Repository of agda/agda2hs.* Retrieved May 19, 2022 from https://github.com/agda/agda2hs
Danil Annenkov, Mikkel Milo, and Bas Spitters. 2021. Code Extraction from Coq to ML-like languages. (2021).
Urban Boquist and Thomas Johnsson. 1997. The GRIN project: A highly optimising back end for lazy functional languages. In *Implementation of Functional Languages*, Werner Kluge (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 58–84.
Jesper Cockx. 2022. *GitHub Repository of jespercockx/agda2scheme.* Retrieved June 18, 2022 from https://github.com/jespercockx/agda2scheme
Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. 2008. The Structure of the Essential Haskell Compiler, or Coping with Compiler Complexity. In *Implementation and Application of Functional Languages*, Olaf Chitil, Zoltán Horváth, and Viktória Zsók (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 57–74.
Olle Frederiksson and Daniel Gustafsson. 2011. *A totaly Epic backend for Agda.* Master's thesis. Chalmers University of Technology, University of Gothenburg.
Philipp Hausmann. 2015. *The Agda UHC Backend.* Master's thesis. Universiteit Utrecht.
C. A. R. Hoare. 1961. Algorithm 64: Quicksort. *Commun. ACM* 4, 7 (jul 1961), 321. https://doi.org/10.1145/366622.366644
William A. Howard. 1980. The Formulae-as-Types Notation of Constructors. *To H.B. Curry: essays on combinatory logic, lambda calculus and formalisms* 44 (1980), 479–490.
Idris2 Team. 2020. *Documentation for the Idris 2 Language.* Retrieved May 19, 2022 from https://idris2.readthedocs.io/en/latest
Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Logic and Theory of Algorithms*, Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe (Eds.). Springer, Berlin, Heidelberg, 359–369.
LLVM Team. 2022a. *Clang: a C langauge family frontend for LLVM.* Retrieved May 19, 2022 from https://clang.llvm.org
LLVM Team. 2022b. *LLD - The LLVM Linker.* Retrieved May 19, 2022 from https://lld.llvm.org
LLVM Team. 2022c. *The LLDB Debugger.* Retrieved May 19, 2022 from https://lldb.llvm.org
LLVM Team. 2022d. *The LLVM Compiler Infrastructure.* Retrieved May 19, 2022 from https://llvm.org
LLVM Team. 2022e. *LLVM Language Reference Manual.* Retrieved May 27, 2022 from https://llvm.org/docs/LangRef.html
Simon L. Peyton Jones. 1992. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming* 2, 2 (1992), 127–202. https://doi.org/10.1017/S0956796800000319
Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. 1999. C- -: A Portable Assembly Language that Supports Garbage Collection. In *Principles and Practice of Declarative Programming*, Gopalan Nadathur (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–28.
Eric S. Raymond. 1991. The New Hacker's Dictionary.
David A. Terei and Manuel M.T. Chakravarty. 2010. An LlVM Backend for GHC. *SIGPLAN Not.* 45, 11 (sep 2010), 109–120. https://doi.org/10.1145/2088456.1863538
Remi Turk. 2010. *A modern back-end for a dependently typed language.* Master's thesis. Universiteit van Amsterdam.

Paul van der Walt. 2012. *Reflection in Agda.* Master's thesis. Universiteit Utrecht.
John van Schie. 2008. *Compiling Haskell to LLVM.* Master's thesis. Universiteit Utrecht.

A RULES FROM TREELESS TO MID-LEVEL INTERMEDIATE REPRESENTATION

This informational appendix contains all transformation rules that are used in practice to transform Agda's treeless syntax into our Mid-Level Intermediate Representation. To read and verify these transformations, we suggest and note the following.

Each transformation rule reads as a logic statement, from a source type to a target type, which should become clear from context. For ease of reading, please keep aside a copy of the full treeless grammar from the Agda manual [Agda Team 2022], or the summarized version from figure 1a, and the grammar of the MLIR in figure 1b. There are some preconditions above the line, which ensure the transformation below the line applies. If there are no preconditions, i.e. there is nothing above the dividing line, the transformation trivially applies. Above all, a transformation can only hold if the left-hand side of the transformation rule successfully matches the value in question. If for some input there is no rule that applies, it means that the feature is not supported.

Most transformation occurs in a given context, which is indicated by the variables in front of the turnstile (⊢). Transformations which discard their context, simply do not list a variable. For other variables, types should be unambiguous by the grammar, and are often inferable by their given name. We overset ranged variables with a bar. On the 'from-side' of an arrow (the data destruction), underscores are used to indicate that a data field is explicitly disregarded. We use the @-symbol to give name to a destruction, e.g. x@Let y means that x represents Let y.

We introduce two symbols: ⋄ and ▲. ⋄ is an operator used to concatenate identifiers with a magical character infixed, to prevent theoretical name clashes. We use ▲ to indicate the emission of an MLIR entry, which are to be collected at the root transformation (the external interface). In more technical terms, this is a monadic write operation.

Finally, we assume the existence of the functions: gets and modify. As their names imply, these get and modify the value of an implicitly assumed counter. The updated counter values are only propagated if calling transformations actually hold.

The transformations are separated in five clusters. The auxiliary helpers are there for convenience and should be self-explanatory. The external interface are two rules which are the start rules for a full conventional transformation, as used in the implementation, one for each Agda definition type. The last three clusters contain the practical body rules, which are differentiated by their purpose, as the name above the arrow suggests.

Auxiliary helpers:

$$\textsc{Aux-Conv-QName-Ident}\quad \frac{}{\vdash \mathsf{QName}\ q \longrightarrow \mathsf{Ident}\ q}\ (\textsc{implicit})$$

$$\textsc{Aux-Monad-State-Next}\quad \frac{i \leftarrow \mathsf{gets} \qquad \mathsf{modify}\ (+1)}{\mathsf{next} \longrightarrow i} \qquad \textsc{Aux-Desugar-Let}\quad \frac{}{\mathsf{desugarLet}\ (\mathsf{Let}\ t\ b) \longrightarrow \mathsf{App}\ (\mathsf{Let}\ b)\ [t]}$$

$$\textsc{Aux-Lift-Arg}\quad \mathsf{liftArg}\ K\ k = \frac{n \leftarrow \mathsf{next} \qquad q' = q \diamond \mathtt{lift} \diamond k \diamond n \qquad t' \overset{\text{Body}}{\longleftarrow} q' \vdash t \qquad \blacktriangle\mathsf{Thunk}\ q'\ \mathsf{True}\ b'}{q \vdash t@K\ \_ \longrightarrow \mathsf{Extern}\ q'}$$

External interface:

$$\textsc{Transform-Function}\quad \frac{t' \xleftarrow{\text{Body}} q \vdash t \qquad \blacktriangle\text{Thunk}\ q\ \text{False}\ (\text{Delay}\ t')}{q \vdash t \longrightarrow q} \qquad \textsc{Transform-Primitive}\quad \frac{q_3 = \texttt{agda.prim.} \diamond q_2 \qquad \blacktriangle\text{Alias}\ q_1\ q_3}{q_1 \vdash q_2 \longrightarrow q_1}$$

Rules [QName $\vdash$ Term $\xrightarrow{\text{Body}}$ Entry]:

$$\textsc{Body-App}\quad \frac{s' \xleftarrow{\text{Arg}} q \vdash s \qquad \overline{a'} \xleftarrow{\text{Arg}} q \vdash \overline{a}}{q \vdash \text{App}\ s\ \overline{a} \longrightarrow \text{Apply}\ s'\ \overline{a'}} \qquad \textsc{Body-Lam}\quad \frac{\begin{array}{cc} n \leftarrow \text{next} & q' = q \diamond \texttt{lam} \diamond n \\ b' \leftarrow q' \vdash b & \blacktriangle\text{Direct}\ q'\ \text{False}\ b' \end{array}}{q \vdash \text{Lam}\ b \longrightarrow \text{MakeValue}\ (\text{Function}\ q')}$$

$$\textsc{Body-Lit}\quad \frac{}{\vdash \text{Lit}\ l \longrightarrow \text{MakeValue}\ (\text{Literal}\ l)} \qquad \textsc{Body-Let}\quad \frac{t' \leftarrow q \vdash \text{desugarLet}\ l}{q \vdash l@\text{Let}\ \_ \longrightarrow t')} \qquad \textsc{Body-Coerce}\quad \frac{t' \leftarrow q \vdash t}{q \vdash \text{Coerce}\ t \longrightarrow t'}$$

$$\textsc{Body-Case-Data}\quad \frac{f' \leftarrow q \vdash f \qquad \overline{a'} \xleftarrow{\text{Alt}} q \vdash \overline{a}}{q \vdash \text{Case}\ i\ \text{Data}\ f\ \overline{a} \longrightarrow \text{Case}\ i\ (\text{Data}\ \overline{p})\ f'} \qquad \textsc{Body-Case-Nat}\quad \frac{f' \leftarrow q \vdash f \qquad \overline{a'} \xleftarrow{\text{Alt}} q \vdash \overline{a}}{q \vdash \text{Case}\ i\ \text{Nat}\ f\ \overline{a} \longrightarrow \text{Case}\ i\ (\text{Nat}\ \overline{a'})\ f'}$$

$$\textsc{Body-Error-Unreachable}\quad \frac{}{\vdash \text{Error Unreachable} \longrightarrow \text{Error}\ \texttt{unreachable}} \qquad \textsc{Body-Error-Meta}\quad \frac{}{\vdash \text{Error}\ (\text{Meta}\ m) \longrightarrow \text{Error}\ m}$$

Rules [QName $\vdash$ Alt $\xrightarrow{\text{Alt}}$ Alt]:

$$\textsc{Alt-Con}\quad \frac{b' \leftarrow q \vdash b}{q \vdash \text{Con}\ c\ n\ b \longrightarrow (c, n, b')} \qquad \textsc{Alt-Lit}\quad \frac{b' \leftarrow q \vdash b}{q \vdash \text{Lit}\ (\text{Nat}\ n)\ b \longrightarrow (n, b')}$$

Rules [QName $\vdash$ Term $\xrightarrow{\text{Arg}}$ Arg]:

$$\textsc{Arg-Var}\quad \frac{}{\vdash \text{Var}\ i \longrightarrow \text{Record}\ i} \qquad \textsc{Arg-Prim}\quad \frac{p' \leftarrow \text{primIdent}\ p}{\vdash \text{Prim}\ p \longrightarrow \text{Extern}\ (\texttt{agda.prim.} \diamond p')} \qquad \textsc{Arg-Def, Arg-Con}\quad \frac{}{\vdash (\text{Def}|\text{Con})\ q \longrightarrow \text{Extern}\ q}$$

$$\textsc{Arg-Let}\quad \frac{t' \leftarrow q \vdash \text{desugarLet}\ l}{q \vdash l@\text{Let}\ \_ \longrightarrow t'} \qquad \textsc{Arg-Erased}\quad \frac{}{\vdash \text{Erased} \longrightarrow \text{Erased}} \qquad \textsc{Arg-Coerce}\quad \frac{t' \leftarrow q \vdash t}{q \vdash \text{Coerce}\ t \longrightarrow t'}$$

$$\begin{array}{llll} \textsc{Arg-App} & \textsc{Arg-Lam} & \textsc{Arg-Lit} & \textsc{Arg-Case} \\ \text{liftArg App}\ \texttt{appl} & \text{liftArg Lam}\ \texttt{lam} & \text{liftArg Lit}\ \texttt{lit} & \text{liftArg Case}\ \texttt{case} \end{array}$$

B RUNTIME HEADER

The following listing contains the actual source text that is prefixed to each file generated by the compiler, before being fed into Clang. It contains all necessary type and function declarations to link with the runtime library.

```
;; Agda basic structures

```

```
; Evaluation Info
%agda.struct.eval =
  type { %agda.struct.value* (%agda.struct.frame*, %agda.struct.thunk*)*, %agda.struct.frame* }

; Value
%agda.struct.value = type { i64, [2 x i64] }
%agda.struct.value.fn = type { i64, %agda.struct.eval } ; tag=0
%agda.struct.value.data = type { i64, %agda.data.base* } ; tag=1
%agda.struct.value.lit_nat = type { i64, i64 } ; tag=2
%agda.struct.value.lit_w64 = type { i64, i64 } ; tag=3
%agda.struct.value.lit_f64 = type { i64, double } ; tag=4
%agda.struct.value.lit_str = type { i64, i8* } ; tag=5
%agda.struct.value.lit_chr = type { i64, i8 } ; tag=6

; Thunks
%agda.struct.thunk = type { i64, [16 x i8] }
%agda.struct.thunk.eval = type { i64, %agda.struct.eval } ; evaluated=false
%agda.struct.thunk.value = type { i64, %agda.struct.value* } ; evaluated=true

; Frame
%agda.struct.frame = type { %agda.struct.thunk*, %agda.struct.frame* }

; Data base
%agda.data.base = type { i64, i64, %agda.struct.frame* }

;; Agda allocators

declare
%agda.struct.value*
@agda.alloc.value()

declare
%agda.struct.thunk*
@agda.alloc.thunk()

declare
%agda.data.base*
@agda.alloc.data(i64)

;; Agda evaluation

declare
%agda.struct.value*
@agda.eval.force(%agda.struct.thunk*)

declare
%agda.struct.value*
@agda.eval.appl.n(%agda.struct.thunk*, ...)

declare
i64
@agda.eval.case.data(%agda.struct.thunk*)

declare
i64
@agda.eval.case.lit_nat(%agda.struct.thunk*)
```

```

declare
%agda.struct.value*
@agda.eval.main(%agda.struct.thunk*(%agda.struct.frame*)*)

;; Agda stack

declare
void
@agda.record.push_replace(%agda.struct.frame**, %agda.struct.thunk*)

declare
%agda.struct.thunk*
@agda.record.get(%agda.struct.frame*, i64)

declare
%agda.struct.frame*
@agda.record.extract(%agda.struct.frame*, i64, %agda.struct.thunk*)
```

C MLIR OF THE FACTORIAL FUNCTION

The listing below contains the MLIR corresponding to the factorial function example, as given in figure 5. Note that this MLIR is not optimized, i.e. all thunks are delayed, even though some could be optimized to be immediate.

```
(Thunk "fac.lift.appl.3" private=True
  (Delay
    (Appl
      (Ext "fac")
      [Record 0]))
(Direct "fac.lam.2" push=True
  (Appl
    (Ext "agda.prim.mul")
    [
      Record 1,
      Ext "fac.lift.appl.3"
    ]))
(Thunk "fac.lift.lam.1" private=True
  (Delay
    (MakeValue (Function "fac.lam.2")))
(Thunk "fac.lift.lit.5" private=True
  (Delay
    (MakeValue (Lit 1)))
(Thunk "fac.lift.appl.4" private=True
  (Delay
    (Appl
      (Ext "agda.prim.sub")
      [
        Record 0,
        Ext "fac.lift.lit.5"
      ])))
(Direct "fac.lam.0" push=True
  (Case subj=0
    (Nat [
      (0, MakeValue (Lit 1))
      ])
    (Appl
      (Ext "fac.lift.lam.1")
      [
        Ext "fac.lift.appl.4"
      ])))
(Thunk "fac" private=False
  (Delay
    (MakeValue (Function "fac.lam.0"))))
```