THE UNIVERSITY OF
NEW SOUTH WALES

SCIENTIA

MANU ET MENTE

SYDNEY · AUSTRALIA

School of Computer Science and Engineering
Faculty of Engineering

Towards a Verified Compiler from Cogent to LLVM

A thesis submitted for the degree of
Bachelor of Engineering (Honours)
in Software Engineering

Harrison Jay Scott
Supervised By
Christine Rizkallah

28 April 2021

Acknowledgements

Thank you to my supervisor, Christine Rizkallah, for being a guiding light over the past year that I have spent engrossed in this research.

I would also like to recognise the efforts of Kai Engelhardt and Liam O'Connor who introduced me to formal methods through their amazing teaching, it was this exposure early in my degree that ultimately led me to choose a project in this area.

I extend my thanks to everyone in the COGENT group at Data61 for their work which I have built on, as well as their interest, help, and feedback throughout my thesis. Additionally, I would like to thank the various UNSW students who worked on COGENT with and before me, especially Emmet Murray for being the one to encourage me to do a thesis related to COGENT.

Finally, I would like to express my deepest gratitude to my friends and family who have cheered me on from near and from afar over this past year.

Abstract

COGENT is a language aimed at reducing the cost of formally verifying systems code. A certifying compiler compiles COGENT code into C code and produces a proof that the C is a refinement of the COGENT code in the Isabelle/HOL theorem prover. For large COGENT programs, it takes a significant amount of time for Isabelle/HOL to check the generated proofs.

Firstly, this thesis presents an implementation for an LLVM backend that covers the entire COGENT language. LLVM is used by many modern compilers and allows for sophisticated optimisations that are not possible to represent directly on C.

The second component to this thesis proposes an approach for verifying the COGENT to LLVM compiler once and forall, which would provide an improvement to COGENT's existing certifying compilation approach. To ease compiler verification down to a denotational semantics of LLVM in the Coq proof assistant, we define a denotational semantics for COGENT and formalize it in Coq.

Finally, using the Coq, we develop the necessary underlying theory which will be used to formally verify the compiler backend.

Contents

Chapter 1

Introduction

COGENT is a purely functional programming language aimed at reducing the cost of verifying systems code (O'Connor et al., 2016). It comes with a *certifying compiler* that co-generates low-level (C) code, a *shallow embedding* in Isabelle/HOL (Nipkow and Klein, 2014), and a *refinement* proof between the shallow embedding and the generated C code. Low-level systems such as file systems can be implemented in COGENT (Amani et al., 2016). The functional correctness of these systems can be verified by reasoning about the shallow embedding. The refinement proof ensures that the verified properties carry over to the generated C code. By allowing reasoning about a HOL embedding rather than directly about C, COGENT reduces the cost of verifying systems.

COGENT is a higher level language than C that offers a greater level of abstraction, yet it remains practical for low-level systems development. Functional programming, variants, records, parametric polymorphism, and a rich linear type system are some of the high-level features of COGENT. In addition, COGENT programs may interoperate with C code via a foreign function interface (FFI), enabling lower-level systems development. COGENT is a restricted language, so the FFI allows parts of a COGENT system to be implemented in C. The FFI also enables a COGENT system to integrate with existing C systems.

While the current COGENT compiler targets C, a language that is familiar to systems engineers, the generated C code is unnecessarily complex and often less efficient than handcrafted C (O'Connor, 2019). Existing C compilers such as GCC and Clang are primarily designed to optimise handwritten C, rather than compiler-generated C code. Thus, instead of targeting C code, COGENT would benefit from a backend that targets an intermediate compiler language.

The LLVM Compiler *intermediate representation* (LLVM-IR) (Lattner and Adve, 2004) is developed specifically as an IR for compilers. It supports a Static Single Assignment (SSA) form (Rosen et al., 1988) that allows for better optimisations for both imperative and functional programming languages (Appel, 1998). LLVM-IR has a rich ecosystem and is a popular backend for many modern languages, including C++, Java, Python, Scala, Haskell, Rust, and Swift. To improve the runtime efficiency of compiled COGENT code we have developed an LLVM backend for the entire core language which is simpler in design than the existing C backend.

Another aspect of COGENT we would also like to improve is the verification framework. For a given COGENT program, the certifying compiler produces a refinement proof relating the generated C code and the Isabelle/HOL shallow embedding. The proofs that are generated with each COGENT compilation are large and take a significant amount of time to check in Isabelle/HOL- we would like to avoid such overhead.

There are two popular approaches to establishing compiler assurance in an interactive theorem prover: directly verifying the compiler, as is the case for CakeML (Kumar et al., 2014) and most of CompCert (Leroy, 2009), or by developing certifying compilers, as is the case for COGENT (Rizkallah et al., 2016) and parts of CompCert (Rideau and Leroy, 2010). Verifying the COGENT compiler would relieve us of the overhead of checking a proof for each compilation, but compiler verification is costly to maintain.

We have formalised a denotational semantics for COGENT using the theory of Interaction Trees (Xia et al., 2020) to represent memory effects for a *deep embedding* of COGENT in the Coq proof assistant (Bertot and Castéran, 2013). Our COGENT semantics can be related to VIR, a deep embedding of the LLVM-IR in Coq with Interaction Tree semantics building upon the proof techniques used for the HELIX to LLVM-IR compiler (Zaliva, 2020).

Using the Interaction Tree semantics for both languages we have embarked on a compiler correctness proof for a substantial subset of our LLVM backend reimplemented in Coq. Using Coq's extraction feature we have generated Haskell code corresponding to this implementation and integrated it with the existing COGENT compiler toolchain. By opting for a compiler correctness proof, this backend can be verified once and forall to ensure the trustworthiness of COGENT to LLVM compilations. This approach alleviates the need for any proof generation or checking at compile time.

Chapter 2

Background

2.1 Cogent

COGENT is a purely functional language intended for systems programming (O'Connor, 2019). It shares a similar syntax to Haskell and ML, but is heavily restricted - functions must be total, and has limited support for recursion, so iteration must be done via the C FFI. Despite these restrictions, COGENT has been used to implement multiple Linux file systems including `ext2` (Amani et al., 2016). In this section, we will briefly outline its syntax and semantics.

2.1.1 Core Language

The COGENT type system supports rank-1 *polymorphism* (O'Connor et al., 2016), and includes records and variants as algebraic data types. The type system is based on *uniqueness types*, similar to *linear types*, with the restriction that under uniqueness, linear variables cannot have multiple references (De Vries et al., 2007). For our purposes, we are only concerned with the desugared, monomorphised type system which is summarised in Figure 2.1.

```
⟨field⟩            ::= ⟨name⟩

⟨ctr⟩              ::= ⟨name⟩

⟨primitive type⟩   ::= Bool | U8 | U16 | U32 | U64

⟨record type⟩      ::= { ⟨field⟩ : ⟨type⟩ , ... }

⟨variant type⟩     ::= < ⟨ctr⟩ ⟨type⟩ | ... >

⟨function type⟩    ::= ⟨type⟩ -> ⟨type⟩

⟨abstract type⟩    ::= ⟨name⟩ (⟨type⟩ ...)

⟨type⟩             ::= ⟨primitive type⟩ | String | ()
                    |  ⟨record type⟩ | # ⟨record type⟩
                    |  ⟨abstract type⟩ | # ⟨abstract type⟩
                    |  ⟨variant type⟩ | ⟨function type⟩
```

Figure 2.1: Simplified COGENT type system

COGENT's primitive types correspond to booleans, 8-, 16-, 32-, and 64-bit integers. Record types are like C struct types, consisting of an ordered list of field names and their types. Variant types are a sum type consisting of a list of constructor names and their argument types. Function types consist of a single argument and return type. Abstract types are given a name and optionally, type parameters. Record and abstract types can be unboxed by a `#` prefix, otherwise they represent a heap allocated value.

The desugared COGENT syntax is laid out in Figure 2.2. There are primitive operators such as `+` or `||`, and literals such as `23` or `True`. Expressions may be variables, literals, binary/unary operators, `let` expressions for binding values, `let!` for read-only bindings, and `if` statements for branching. Additionally, there are `cast` and `promote` expressions for casting expressions. Expressions may also be record literals, variant constructions, or function applications. Member access can be done via `.` or the `take` construct. `put` constructs must be used to safely modify fields within records due to uniqueness types. `case` and `esac` expressions are the desugared syntax for pattern matching for variants. A COGENT program is simply a list of top-level concrete/abstract definitions of functions and types.

```
⟨var⟩            ::= ⟨name⟩

⟨lit⟩            ::= () | True | 23 | 'hello' | ...

⟨binop⟩          ::= + | - | * | / | %
                  |  < | <= | > | >= | == | /= | && | ||
                  |  .&. | .|. | .^. | << | >>

⟨expr⟩           ::= ⟨var⟩ | ⟨lit⟩
                  |  ⟨expr⟩ ⟨binop⟩ ⟨expr⟩
                  |  not ⟨expr⟩ | complement ⟨expr⟩
                  |  let ⟨var⟩ = ⟨expr⟩ in ⟨expr⟩
                  |  let! (⟨var⟩ ...) ⟨var⟩ = ⟨expr⟩ in ⟨expr⟩
                  |  if ⟨expr⟩ then ⟨expr⟩ else ⟨expr⟩
                  |  cast ⟨type⟩ ⟨expr⟩ | promote ⟨type⟩ ⟨expr⟩
                  |  { ⟨field⟩ = ⟨expr⟩ , ... }
                  |  ⟨ctr⟩ ⟨expr⟩
                  |  ⟨expr⟩ ⟨expr⟩
                  |  ⟨expr⟩ . ⟨field⟩
                  |  take ⟨expr⟩ { ⟨field⟩ = ⟨var⟩ } = ⟨expr⟩ in ⟨expr⟩
                  |  ⟨expr⟩ { ⟨field⟩ = ⟨expr⟩ }
                  |  case ⟨expr⟩ of ⟨ctr⟩ ⟨var⟩ -> ⟨expr⟩ else ⟨var⟩ -> ⟨expr⟩
                  |  esac ⟨expr⟩

⟨function def⟩    := ⟨name⟩ : ⟨function type⟩, ⟨name⟩ ⟨var⟩ = ⟨expr⟩
                  |  ⟨name⟩ : ⟨function type⟩

⟨type def⟩        := type ⟨name⟩ = ⟨type⟩ | type ⟨abstract type⟩

⟨definition⟩      := ⟨function def⟩ | ⟨type def⟩

⟨program⟩         := ⟨definition⟩ ...
```

Figure 2.2: COGENT core syntax

Permissions in COGENT are used to denote the restrictions of a type within the uniqueness type system. Using these, we can relax the uniqueness type system for types that need not be linear/unique. The three permissions are:

- D - discardable without being used
- S - shareable to be used multiple times
- E - bound in a `let!` expression, allowing read-only sharing of a linear value

within a restricted scope

Primitive types may have any kind as they are not linear, and are always unboxed (O'Connor, 2019). Modes (`r`, `w`, and `u`) are used to control permissions on record fields and abstract types.

2.1.2 Update Semantics

COGENT has two dynamic semantics: a functional style *value semantics* and an imperative style *update semantics* (O'Connor, 2019). In terms of implementing an LLVM-IR backend, the imperative-style update semantics is of higher relevance. The update semantics of COGENT is defined using the big step semantics relation:

$$\gamma \vdash (\sigma, e) \Downarrow (\sigma', u)$$

The relation holds when, under the variable context γ an expression e evaluates to a value u and the mutable store σ (abstractly representing memory) is mutated to σ'. The value u may be a literal, function value, abstract function, variant, record, abstract value, or a pointer. We use a context γ that maps variables to values, and a mutable store σ that maps pointers to values. This relation is formalised in Isabelle/HOL as an inductive relation `u_sem`. For each way to construct COGENT expressions, there is a corresponding rule to define the update semantics for that expression. For example, the update semantics for `let` expressions is given by the rule:

$$\frac{\gamma \vdash (\sigma, a) \Downarrow (\sigma', a') \qquad (a' :: \gamma) \vdash (\sigma', b) \Downarrow st}{\gamma \vdash (\sigma, \texttt{Let}\ a\ b) \Downarrow st}$$

The left premise states that in the context γ, the memory σ and expression a will evaluate to a new memory σ' and a value a'. The right premise states that in the updated context $(a' :: \gamma)$, the memory σ' and expression b will evaluate to st. If both premises hold, we can conclude that in the context γ the memory σ and expression `Let` a b will evaluate to st.

A second example, the rule for member access for boxed records, can be stated as follows:

$$\frac{\gamma \vdash (\sigma, e) \Downarrow (\sigma', \texttt{UPtr}\ p) \qquad \sigma'(p) = \texttt{URecord}\ r}{\gamma \vdash (\sigma, \texttt{Member}\ e\ f) \Downarrow (\sigma', r\ !\ f)}$$

For this rule, the left premise requires that in the context γ, the memory σ and expression e will evaluate to σ' and some pointer p. Our right premise is simply that the pointer p, when looked up in the memory σ', points to some record containing a list of fields r. Then, if both premises hold, we conclude that under γ, the memory σ and the expression `Member` $e\ f$ will evaluate to σ' and $r\ !\ f$ (the fth field of r).

A full definition of COGENT's update semantics is given in O'Connor (2019).

2.1.3 Certifying Compiler to C

COGENT is accompanied by a certifying compiler to C that generates a formal proof that the output C code is a refinement of a generated Isabelle/HOL embedding of the original COGENT code (Rizkallah et al., 2016). The proof relies on a once and forall language-level proof that the update semantics refines the value semantics as well as per-program translation validation phases proving the C code refines the update semantics and that the value semantics refines the Isabelle/HOL embedding. This means any behaviour of the C program is also a behaviour of the COGENT program (as modelled in HOL). A simplified overview of the COGENT refinement framework is given in Figure 2.3. In particular, this allows convenient equational reasoning on top of an Isabelle/HOL embedding rather than directly about low-level C code.

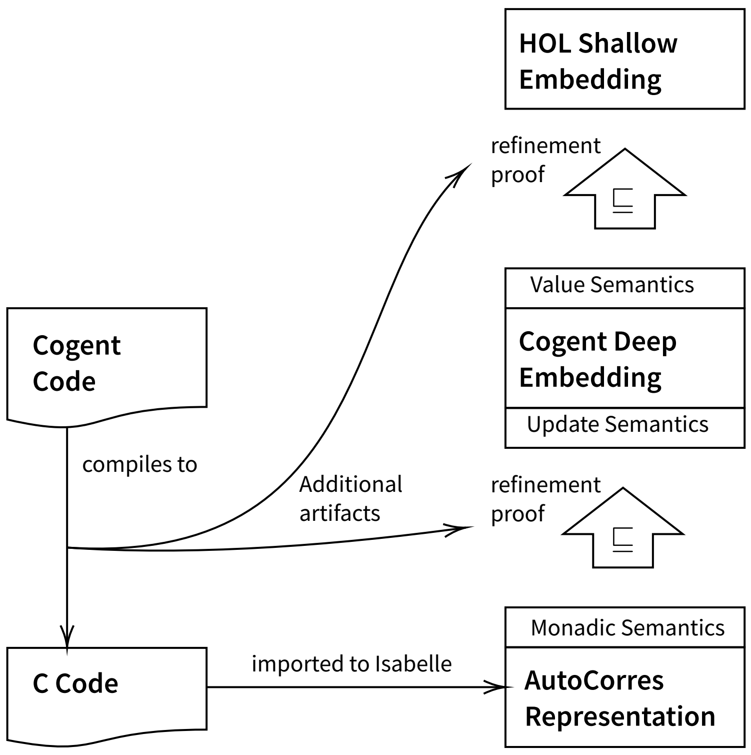

Figure 2.3: The core artifacts produced by the COGENT compiler

2.2 Trusted Compilation

As distinguished by Leroy (2009) there are three widely used methods of establishing trustworthiness for a compiler: *translation validation* (Pnueli et al., 1998) through either a *verified validator* or through *proof-carrying code* (Necula, 1997), and *compiler verification*. All methods aim to assure that the source language semantics is preserved through compilation to the target language. For compiler verification, the compiler is formally verified once and forall, establishing that every compiler output is a refinement of its input. In contrast, translation validation establishes the correctness of a compilation on a per-program basis. So far, *certifying compilers* have produced proofs and can therefore be categorised under the proof-carrying code approach.

2.2.1 Translation Validation

Verified Validators

The method of translation validation does not require verification of the compiler, instead it relies on a validator, which is essentially a boolean-valued function *Validate* : $\mathcal{S} \times \mathcal{T} \to (\textbf{Accept} \mid \textbf{Reject})$, where $\mathcal{S}$ is the set of source code programs and $\mathcal{T}$ is the set of compiled programs in the target language. A validator will accept the tuple (S, T) if T is a valid compilation of S, and reject otherwise. A verified validator is accompanied by a formal proof of this requirement. Using verified validators to check instance correctness is often less work than compiler verification, although implementing a validator and verifying it is not always straightforward.

CompCert is an optimising compiler for Clight, a substantial subset of C (Leroy, 2009). For CompCert, translation validation is used to validate the assembling and linking stages (Leroy et al., 2016), that are not yet verified alongside the semantic preservation steps. The tool, Valex, checks a fully linked, assembled executable for correctness against the internal abstract PowerPC representation of CompCert. CompCert also uses translation validation for certifying its register allocator (Rideau and Leroy, 2010).

Proof-Carrying Code

Proof-carrying code is a variation of translation validation where for each input/output, the compiler produces a proof certifying its compilation. Conceptually, this is similar to using a validator, but in this case, the validator is an entire proof assistant such as Isabelle/HOL or Coq rather than a specialised validator program. Let Π be the domain of formal proofs. Then a certifying compiler can be modelled as a total function *CComp* : $\mathcal{S} \to (\textbf{Ok}\ \mathcal{T} \mid \textbf{Error}) \times \Pi$, and a correct certifying compiler has the property $\mathit{CComp}(S) = (\textbf{Ok}\ C, \pi) \ \Rightarrow\ \pi \models S \sqsubseteq C$. In this case, the only thing that must be trusted is the tool used to check the compiler-generated proof π. This is why the technique was named proof-carrying code by Necula (1997), but π can also be called a *certificate* (Leroy, 2009).

Moreover, this is the approach used by COGENT as a whole, generating a refinement proof of the COGENT to C semantic preservation (Rizkallah et al., 2016). This proof is supported by numerous theorems that relate the pure, functional semantics of the input COGENT code, to an imperative update semantics that can be related to the C produced by the compiler. The refinement proofs are conducted using the Isabelle/HOL proof assistant.

2.2.2 Compiler Verification

By contrast, verification of a compiler entails producing a once and forall compiler correctness proof. A compiler can be modelled as a total function $\mathit{Comp} : \mathcal{S} \rightarrow \textbf{Ok}\ \mathcal{T} \mid \textbf{Error}$ (Leroy, 2009). The compiler correctness proof must demonstrate $\forall S \in \mathcal{S}, T \in \mathcal{T}.\ \mathit{Comp}(S) = \textbf{Ok}\ T \Rightarrow S \sqsubseteq T$. That is, whenever the compiler is successful, the output code should refine the source code. Notably, an always failing compiler $Comp(\cdot) = \textbf{Error}$ would vacuously satisfy this definition of correctness, but be useless in practice.

A downside to a once and forall correctness proof is that whenever the compiler changes, the proof must also be updated, and re-checked. As a result, verifying compilers in this way can be expensive, though this has not discouraged research in this area.

Firstly, CompCert's transformation from the Clight abstract syntax to PowerPC abstract syntax has been implemented and checked in Coq proof assistant (Bertot and Castéran, 2013). By implementing the compiler in Coq, semantic preservation proofs could be performed directly by equational reasoning and other Coq tactics.

Secondly, the CakeML system is a subset of Standard ML with a compiler end-to-end verified in the HOL4 theorem prover (Kumar et al., 2014). The first CakeML compiler implements a read-eval-print loop (REPL) with verified parsing, type inference and compilation with verified x86-64 bytecode. The work on CakeML complements the optimisation-focused efforts of CompCert for imperative programming languages (Kumar et al., 2014). Following this, a new compiler backend for CakeML was developed, mimicking the structure of mainstream compilers, adding FFI support as well as new compilation targets and optimisations (Tan et al., 2016).

2.3 LLVM

The LLVM project comprises an *intermediate representation* (LLVM-IR), several *backends* that transform IR code to various instruction sets, and *frontends* that generate IR from source code languages. To clarify, a COGENT frontend for LLVM is the same as an LLVM backend for COGENT. The IR is an abstract RISC-like instruction set with static types, explicit Control Flow Graphs (CFGs), and explicit dataflow via SSA with an infinite set of registers (Lattner and Adve, 2004).

2.3.1 IR Syntax

We will briefly outline a subset of the syntax of LLVM-IR as a reference for the remainder of this thesis.

An LLVM-IR program consists of multiple modules, each containing function definitions, forward declarations, and global variable definitions (LLVM Project, 2020). From now on we will consider only single LLVM-IR modules on their own. A function definition contains one or more blocks of instructions. LLVM instructions manipulate a set of virtual registers which can be numeric (`%0`) or named (`%a`). Each instruction and register must have an associated type, and strict type rules can be checked for code written in the IR.

LLVM-IR's structural type system is similar to the C type system (LLVM Project, 2020), consisting of base types (integers, floats, etc.) and derived types (vectors, structures, arrays). Integer types are represented by `iN` for a given bit width (up to $2^{23} - 1$) e.g.

`i64` will represent 64-bit integers. Pointer types specify memory locations of a specific type, for example, `i32*` is a pointer to a 32-bit integer. Function types correspond to function signatures, with a return type and parameter types, e.g. `i64 (i32, i32)` is the type of a function that takes in two 32-bit integers and returns a 64-bit integer. Structure types consist of one or more ordered fields, similar to C structs but without field names, for example `{ i32, i8* }` is the structure type containing a 32-bit integer and a pointer to an 8-bit integer. Lastly, opaque types can represent types with an unknown body, e.g. `%Foo = type opaque` forward declares a type `%Foo` with an unspecified body.

Instructions in the IR include arithmetic binary operations, such as `add`, `sub`, `mul`, `udiv`, `urem` and bitwise binary operations `shl`, `lshr`, `and`, `or`, `xor`. There are also instructions for modifying aggregate values: `extractvalue` retrieves a field from a structure and `insertvalue` inserts a value into the specified field of a structure. To work with structures not stored in registers, it may be necessary to allocate stack memory, this can be done via the `alloca` instruction. `load` and `store` retrieve and write the contents of the memory at a specified address. The `getelementptr` allows for address calculation within an aggregate data structure. LLVM-IR also contains many conversion instructions to cast operands, such as `zext` for zero-extending an integer operand to a new bit size, and `bitcast` for performing arbitrary type casts without changing any bits. To compare integer operands, the `icmp` operand is used, which takes a mode such as `eq`, `ne`, `ugt`, `uge`, `ult`, or `ule` to use for the comparison. Lastly, calls to other functions can be made using the `call` instruction.

LLVM-IR blocks conclude with a special type of instruction called a terminator. The `ret` terminator will end control flow for the current function, returning a value or `void`. Another useful terminator is `br` which transfers control flow to another block in the current function. It is possible to branch to different blocks based on the result of a condition by specifying a condition register and two destination blocks, one to take if the condition register contains true and one for false. The `phi` instruction takes a pair of predecessor labels and values to assign based on the path taken to the current block. `phi` instructions may only appear at the beginning of a block, as per the SSA form of LLVM code.

Constants such as integer literals (e.g. `i32 23`) can be used in an LLVM-IR program. The `undef` constant represents an undefined value. Other than constants, previously used registers e.g. `%3` or global variables `@foo` can be used as operands to LLVM instructions.

An example LLVM-IR program is given in Listing 2.1. The program contains a function `@add` which accepts two 32-bit arguments. After adding the arguments, it checks if the result has overflown and sets the first field of the return value appropriately. Conditional branching is used to set the second field only if no overflow has occurred. In the `done` block, the result is chosen via a `phi` instruction and returned from the function.

Listing 2.1: An LLVM-IR program to add unsigned integers and check for overflow

```
define {i1, i32} @add(i32 %a, i32 %b) {
entry:
  %0 = add i32 %a, %b
  %1 = icmp ult i32 %0, %a
  %2 = icmp ult i32 %0, %b
  %3 = or i1 %1, %2
  br i1 %3, label %overflow, label %ok
overflow:
  %4 = insertvalue { i1, i32 } undef, i1 0, 0
  br label %done
ok:
  %5 = insertvalue { i1, i32 } undef, i1 1, 0
  %6 = insertvalue { i1, i32 } %5, i32 %0, 1
  br label %done
done:
  %7 = phi { i1, i32 } [ %4, %overflow ], [ %6, %ok ]
  ret { i1, i32 } %7
}
```

2.3.2 GHC's LLVM Backend

A noteworthy LLVM backend in the area of functional programming languages is that of the Glasgow Haskell Compiler (GHC) (Jones et al., 1993) The GHC LLVM backend was introduced by Terei and Chakravarty (2010), and similar to the case of COGENT, LLVM was chosen as an alternative to C code generation. In the case of GHC, using LLVM removed the dependence on GNU C compiler extensions, improving portability. Moreover, similarly to COGENT, the GHC C backend generates large C files that are difficult to optimise, motivating LLVM as an alternative backend. The GHC pipeline is illustrated in Figure 2.4.

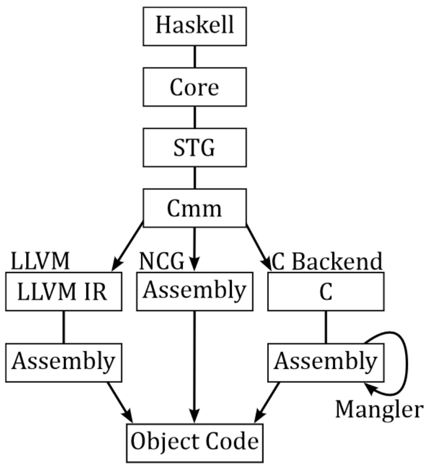

Figure 2.4: GHC backend pipelines (Terei and Chakravarty, 2010)

In particular, the authors mention a lower maintenance effort, as well as the LLVM framework and its community as other advantages gained by replacing the C backend with an LLVM one. The GHC LLVM backend reuses the existing GHC pipeline, which transforms Haskell code to the intermediate imperative language Cmm. For our new COGENT backend, we produce LLVM-IR directly from desugared COGENT code.

2.3.3 LLVM Backend for MLton

Similarly to the case of GHC, LLVM was chosen as an alternative target to the existing C and native code generators for MLton, a Standard ML compiler (Leibig, 2013). Like COGENT, the LLVM backend for MLton supports an FFI to C. As is the case for GHC, the LLVM MLton backend resulted in a simpler implementation than the existing backends. Furthermore, the binaries produced through the LLVM pipeline are comparably smaller than the binaries produced from the C and NCG backends. The MLton pipeline is similar to that of GHC.

2.3.4 Existing Prototype LLVM Backend of Cogent

Before our work, Shang (2020) implemented a prototype LLVM backend as a drop-in replacement for the existing COGENT C backend, for a subset of COGENT. The prototype uses `llvm-hs` (Cowley et al., 2019), an LLVM-IR binding for Haskell, providing the basis for our work. In the prototype, COGENT types are first translated to equivalent types in the IR, and then expressions are compiled to IR expressions via pattern matching.

Though the prototype includes no associated verification, Shang has taken care to restrict the use of LLVM-IR to the subset formalised by the Vellvm project (Shang, 2020, Zhao et al., 2012).

2.3.5 LLVM Formalisations

A *deep embedding* is a representation of a program via an abstract data type modelling the syntax of the language. Conversely, a *shallow embedding* represents a program directly using functions in the logic of the proof assistant (Myreen, 2012).

Isabelle-LLVM

Isabelle-LLVM is a refinement framework to generate verified LLVM-IR from within Isabelle/HOL (Lammich, 2019). It uses a new imperative language shallowly embedded in Isabelle/HOL that translates to actual LLVM code. There is little overhead with the compiled code compared with native unverified C code. Additionally, the project includes a *separation logic*-based verification condition generator.

Vellvm

Alternatively, Vellvm (Zhao et al., 2012) is a framework for reasoning about LLVM-IR programs within the Coq proof assistant. It provides a deep embedding of a core subset of LLVM in Coq. With this, LLVM-IR code can be imported into the proof assistant to be reasoned about. Vellvm gives an operational semantics for the IR and its SSA form. These semantics are parametrised by a memory model.

The features of LLVM that are not modelled by Vellvm are (Zhao et al., 2012):

- Intrinsic functions, used for extending LLVM or representing library functions
- Attributes, for denoting linkage types, calling conventions, and more
- The `invoke` & `unwind` instructions, which are for modelling exception handling
- Variadic functions and vector types

VIR

To avoid confusion between the legacy Vellvm project (Zdancewic et al., 2014) and the latest Vellvm work (Zakowski et al., 2021), we hereafter refer to the most recent Vellvm development as VIR. VIR is a newer Coq formalisation of LLVM-IR using a similar deep embedding to legacy Vellvm. However, it gives a denotational semantics based on the theory of Interaction Trees (Xia et al., 2020), which we will discuss in Chapter 4.1. The VIR semantics for LLVM admit an executable interpreter, allowing for differential testing using the QuickChick (Dénès et al., 2014) property-based testing framework for Coq.

VIR has been used in the verification of the HELIX compiler (Zaliva, 2020) which produces VIR code as part of its LLVM backend. HELIX translates a mathematical formula to LLVM-IR via a series of intermediate languages. A correctness proof for the final compilation step, from FHCOL to LLVM-IR is underway using equational reasoning between an Interaction Tree semantics for FHCOL, and the VIR semantics (Zaliva, 2020). We chose to follow in the footsteps of HELIX with our COGENT to LLVM verification story.

The syntax of VIR is very similar to LLVM-IR (Zakowski et al., 2021). At the top level, a VIR module is an MCFG, a mutually recursive collection of control-flow-graphs (CFGs). Each CFG has a name, a list of arguments, a list of blocks, and a block label to enter from. Blocks consist of an entry label, phi-nodes, a list of instructions, and a terminator. The phi-nodes correspond directly to LLVM-IR `phi` instructions that can occur at the start of a block, allowing for an SSA form. VIR Instructions directly correspond to all the supported LLVM-IR instructions, such as `load` or `add`. A block's terminator is also the same as in LLVM-IR, affecting control flow after a block such as `ret` or `br`.

The VIR representation for the LLVM-IR program in Listing 2.1 as embedded in Coq is given in Listing 2.2. We will discuss the semantics of VIR in Section 4.4.

Listing 2.2: Deep embedding of LLVM-IR program in Coq as a VIR AST

```
Definition prog := [TLE_Definition {|
  df_prototype := {|
    dc_name := (Name "add");
    dc_type := (TYPE_Function (TYPE_Struct [(TYPE_I 1); (TYPE_I 32)]) [(TYPE_I 32); (TYPE_I 32)]);
    dc_param_attrs := ([], []);
    dc_linkage := None; dc_visibility := None; dc_dll_storage := None; dc_cconv := None;
    dc_attrs := []; dc_section := None; dc_align := None; dc_gc := None
  |};
  df_args := [(Name "a"); (Name "b")];
  df_instrs := (
    {|
      blk_id := (Name "entry");
      blk_phis := [];
      blk_code := [
        (IId (Anon 0%Z), (INSTR_Op (
          OP_IBinop (Add false false) (TYPE_I 32)
          (EXP_Ident (ID_Local (Name "a")))
          (EXP_Ident (ID_Local (Name "b"))))));
        (IId (Anon 1%Z), (INSTR_Op (
          OP_ICmp Ult (TYPE_I 32)
          (EXP_Ident (ID_Local (Anon 0%Z)))
          (EXP_Ident (ID_Local (Name "a"))))));
        (IId (Anon 2%Z), (INSTR_Op (
          OP_ICmp Ult (TYPE_I 32)
          (EXP_Ident (ID_Local (Anon 0%Z)))
          (EXP_Ident (ID_Local (Name "b"))))));
        (IId (Anon 3%Z), (INSTR_Op (
          OP_IBinop Or (TYPE_I 1)
          (EXP_Ident (ID_Local (Anon 1%Z)))
          (EXP_Ident (ID_Local (Anon 2%Z))))))
      ];
      blk_term := TERM_Br ((TYPE_I 1), (EXP_Ident (ID_Local (Anon 3%Z)))) (Name "overflow") (Name "ok");
      blk_comments := None
    |},
    [
      {|
        blk_id := (Name "overflow");
        blk_phis := [];
        blk_code := [
          (IId (Anon 4%Z), (INSTR_Op (
            OP_InsertValue ((TYPE_Struct [(TYPE_I 1); (TYPE_I 32)]),EXP_Undef)
            ((TYPE_I 1),(EXP_Integer (0)%Z)) [0%Z]%Z)))
        ];
        blk_term := TERM_Br_1 (Name "done");
        blk_comments := None
      |}; {|
        blk_id := (Name "ok");
        blk_phis := [];
        blk_code := [
          (IId (Anon 5%Z), (INSTR_Op (
            OP_InsertValue ((TYPE_Struct [(TYPE_I 1); (TYPE_I 32)]),EXP_Undef)
            ((TYPE_I 1),(EXP_Integer (1)%Z)) [0%Z]%Z)));
          (IId (Anon 6%Z), (INSTR_Op (
            OP_InsertValue ((TYPE_Struct [(TYPE_I 1); (TYPE_I 32)]),(EXP_Ident (ID_Local (Anon 5%Z))))
            ((TYPE_I 32),(EXP_Ident (ID_Local (Anon 0%Z)))) [1%Z]%Z)))
        ];
        blk_term := TERM_Br_1 (Name "done");
        blk_comments := None
      |}; {|
        blk_id := (Name "done");
        blk_phis := [
          ((Anon 7%Z), Phi (TYPE_Struct [(TYPE_I 1); (TYPE_I 32)]) [
            ((Name "overflow"), (EXP_Ident (ID_Local (Anon 4%Z))));
            ((Name "ok"), (EXP_Ident (ID_Local (Anon 6%Z))))
          ])
        ];
        blk_code := [];
        blk_term := TERM_Ret ((TYPE_Struct [(TYPE_I 1); (TYPE_I 32)]), (EXP_Ident (ID_Local (Anon 7%Z))));
        blk_comments := None
      |}
    ]
  )
|}].
```

Chapter 3

Compiler Implementation

Our first contribution is the construction of an LLVM backend for COGENT. We developed two implementations of the COGENT to LLVM compiler. Both backends transform a desugared, monomorphised COGENT AST into a corresponding LLVM-IR module. The first, building upon the prototype by Shang (Shang, 2020), is a Haskell implementation that covers the entire core COGENT language and allows for interfacing with C (Section 3.1). The second implementation of the COGENT to LLVM compiler is developed in Coq's Gallina language for the purpose of verification (Section 3.2). The Gallina implementation is extracted into Haskell using Coq's code extraction features. It supports many key features of the COGENT core language including arithmetic (except `U16`), record, and variant operators, but does not currently support abstract functions, abstract types, strings, unary operators, or a C FFI.

3.1 Haskell Implementation of the LLVM Backend

The Haskell implementation of the compiler supports all features, types, and expressions in the core COGENT language. The backend has been designed with compatibility with C code in mind, supporting a primitive C foreign function interface can be supported.

3.1.1 Design

The backend is convenient to implement since LLVM's structural type system is similar to the C type system which the existing COGENT compiler targets. Secondly, since COGENT code is strict and functional, we can generate the LLVM-IR SSA form easily by traversing a COGENT AST and emitting instructions for each sub-expression as per the evaluation order defined by COGENT's update semantics. We split the compilation task across various Haskell functions to translate definitions, types, and expressions to their LLVM equivalents, using cases for each piece of the abstract syntax.

The implementation is split across multiple Haskell modules for ease of future maintainability. Appendix A.1 describes the role of each module; this will be useful for future maintainers of the code. The LLVM backend is enabled via the `--llvm` flag of the COGENT compiler, which will emit a `.ll` file for the input program.

The llvm-hs Library

`llvm-hs` provides bindings for the LLVM API and a representation of the LLVM-IR AST, allowing us to implement the LLVM backend in Haskell (Cowley et al., 2019). We utilise the `IRBuilder` module to avoid boilerplate when generating the AST, which consists of various monadic functions for each LLVM `Instruction` such as `add` or `insertValue`. These functions manipulate `Operand`s that correspond to LLVM operands such as constants or references to virtual registers. The `IRBuilder` monad keeps track of operands and blocks for us, and instructions will yield an `Operands` in this monad corresponding to their result. By using do-notation, the Haskell code resembles the structure of the LLVM-IR we wish to represent. For example, in Listing 3.1 we can build the LLVM-IR module from Listing 2.1 using various functions from `llvm-hs`'s `IRBuilder` and `AST` modules.

Listing 3.1: Building an LLVM-IR module using `llvm-hs`

```
example :: Module
example = buildModule "example" $ do
  let rt = StructureType False [i1, i32]
```

```
function "add" [(i32, "a"), (i32, "b")] rt $ \[a, b] -> mdo
  entry <- block `named` "entry"
  r0 <- add a b
  r1 <- icmp ULT r0 a
  r2 <- icmp ULT r0 b
  r3 <- or r1 r2
  condBr r3 overflow ok
  overflow <- block `named` "overflow"
  r4 <- insertValue (ConstantOperand (C.Undef rt))
                    (ConstantOperand (C.Int 1 0)) [0]
  br done
  ok <- block `named` "ok"
  r5 <- insertValue (ConstantOperand (C.Undef rt))
                    (ConstantOperand (C.Int 1 1)) [0]
  r6 <- insertValue r5 r0 [1]
  br done
  done <- block `named` "done"
  r7 <- phi [(r4, overflow), (r6, ok)]
  ret r7
```

Compiler State

`llvm-hs` provides `ModuleBuilderT` and `IRBuilderT`, which are state transformer monads that allow us to easily generate LLVM-IR like in Listing 3.1 without managing our register, block, and definition state (Cowley et al., 2019). We augment the state with our own `Env` state containing the following additional structures:

- `vars` - a De Bruijn indexed list of operands for each in-scope COGENT variable
- `tags` - a global list of variant constructor names
- `typedefs` - an associative map from COGENT type definitions to LLVM types

For any monad wrapping our `Env` state, we can define the `bind` wrapper in Listing 3.2.

Listing 3.2: Helper for binding during compilation (`CodeGen.hs`)

```
bind :: MonadState Env m => Operand -> m a -> m a
bind var action = do
    vars <- gets vars
    modify $ \s -> s {vars = var : vars}
    res <- action
    modify $ \s -> s {vars = vars}
    pure res
```

This function allows us to perform a monadic operation `action` inside a context where an operand `var` has been bound as a new variable, and then restore the variable state after the action is complete and the scope of the variable has ended. This approach hides the underlying variable state management so we can focus on code generation in our expression compilation function.

3.1.2 Types

All core COGENT types are supported by the Haskell implementation of the LLVM backend, including abstract types. This section aims to detail the translation of each COGENT type to its LLVM-IR equivalent. Thanks to LLVM's C-like structural type system, most COGENT types have straightforward translations, examples of each are given in Table 3.1.

Type	**Cogent example**	**LLVM IR**
Integer	`U64`	`i64`
Bool	`Bool`	`i8`
Unit	`()`	`i8`
String	`String`	`i8*`
Record	`#{a: U32, b: U8}`	`{i32, i8}`
Boxed Record	`{p: U16}`	`{i16}*`
Variant	`<Ok U16 | Err>`	`{i32, i16}`
Function	`(U8, U8) -> U8`	`i8 ({i8, i8})*`
Abstract	`Foo`	`%Foo`

Table 3.1: A summary of the backend's type translation

The conversion from core COGENT types to LLVM-IR types is performed by the `toLLVMType` function, which traverses a type and recursively converts it to an `llvm-hs` type. The forthcoming sections examine each type translation case in more detail.

Primitive Types

The LLVM type system provides us with integers of arbitrary size. In Listing 3.3 COGENT's integer types are translated as `iN` by defining the number of bits for each type in a `primIntSizeBits` function. For example, `U8` becomes `i8` and `U64` becomes `i64`. This is consistent with the existing C backend. `Bool` is represented as an `i8` rather than an `i1`. This choice was made to preserve compatibility with the C `bool` type which is defined as an 8-bit type.

Listing 3.3: Conversion from COGENT to LLVM-IR types (`Types.hs`)

```
toLLVMType (TPrim p) =
    pure $ IntegerType $ fromInteger $ primIntSizeBits p
```

Unit Type

LLVM provides a `void` type but it is not a first-class type, it may only be used for function return types, so instead, we must choose an alternate representation for COGENT's unit type. In our implementation below, `()` is translated to an `i8`, but alternate representations are possible, such as `i32`, so long as the type can be represented equivalently in C.

```
toLLVMType TUnit = pure i8
```

String Type

The `String` type of COGENT can be represented as a byte pointer `i8*` inside LLVM-IR via the following case. By choosing this representation, compatibility with C strings comes for free.

```
toLLVMType TString = pure $ ptr i8
```

Record Types

We translate COGENT's records to non-packed structure types, for which padding is inserted between fields according to a data layout. To enable C compatibility, we choose not to define our own layout rules and instead use the default. For each field of the record type, we can translate it to an LLVM-IR type and construct a structure type containing those fields. For example, `#{ a : U32, b : U8 }` becomes `{ i32, i8 }`. In the implementation below, `mapM` performs the traversal over a COGENT record's field types.

```
toLLVMType (TRecord _ ts Unboxed) =
    StructureType False <$> mapM toLLVMType (fieldTypes ts)
```

Boxed records can also be represented natively in LLVM by taking a pointer to the unboxed representation, for instance `{ p: U16 }` gets translated as `{ i16 }*`.

```
toLLVMType (TRecord r ts (Boxed _ _)) =
    ptr <$> toLLVMType (TRecord r ts Unboxed)
```

Variant Types

Variant types are the most complex type to translate to LLVM-IR as the type system does not provide sum types or unions. Instead, we must simulate the variant type as a tagged union, which is a structure type containing a *tag* field and a value field. In our implementation, tags are 32-bit integers that we base on the lexicographic ordering of all constructor names in a COGENT program.

The value field takes on the shape of the *maximal type* for the variant. The maximal type for a variant is the largest type (in terms of memory layout size) of all its different payloads, with earlier types breaking ties. For example, the maximal type of `< A U16 | B U32 >` is `U32`.

For example the variant `< Success U16 | Failure () >` is represented as `{ i32, i16 }`. The first `i32` is for the tag, and the second field is an `i16` since the `U16` argument to `Success` has a larger representation (16 bits) than the `()` argument to `Failure` (8 bits). In the implementation below, we calculate the maximal type with `maxMember` before constructing a structure type containing `i32` and it.

```
toLLVMType (TSum ts) = do
    f <- toLLVMType (maxMember ts)
    pure $ StructureType False [i32, f]
```

Though our representation is efficient, when retrieving the value field, the result must afterwards be casted to the correct inhabitant type (based on the current tag). Conversely, when setting the value field, the payload must be first casted to the maximal type. Our treatment of variant expressions correctly performs both types of casts when necessary.

Function Types

A COGENT function type is translated to a pointer to an LLVM function type using code that follows. For example, `U64 -> U32` becomes `i32 (i64)*`. Currently, all COGENT functions take exactly one argument but were the core language to change we could easily translate argument lists to a list of LLVM types.

```
toLLVMType (TFun t1 t2) = do
    at <- toLLVMType t1
    rt <- toLLVMType t2
    pure $ ptr $ FunctionType rt [at] False
```

Abstract Types

COGENT permits abstract types which may be boxed or unboxed, which in LLVM-IR can be forward declared using `type opaque`. Our implementation emits the required global declaration when an abstract type is encountered in the AST. The names of the opaque types are generated based on their COGENT name, including the instantiation of type parameters for polymorphic abstract types.

```
toLLVMType t@(TCon _ _ Unboxed) = abstractType t
```

The case above handles unboxed abstract types, where `abstractType` is a function defined in Listing 3.4 that interacts with the compiler's `Env` state to retrieve/set a type definition so that duplicate definitions are avoided:

Listing 3.4: Handling abstract type definitions (`Types.hs`)

```haskell
abstractType t = do
    mt <- gets (Map.lookup name . typedefs)
    case mt of
        Just td -> pure td
        Nothing -> do
            td <- typedef (mkName name) Nothing
            modify $ \s -> s {typedefs = Map.insert name td (typedefs s)}
            pure td
    where
        name = nameType t
```

As with records, boxed abstract types can be represented in LLVM by taking a pointer to the unboxed representation of the same type, as shown in Listing 3.5.

Listing 3.5: Conversion from COGENT to LLVM-IR types (`Types.hs`)

```haskell
toLLVMType (TCon tn ts (Boxed _ _)) =
    ptr <$> toLLVMType (TCon tn ts Unboxed)
```

3.1.3 Expressions

All of COGENT's core expression language is supported by the Haskell implementation of the LLVM backend. Despite being quite minimal, the LLVM-IR language is expressive enough to allow us to implement most COGENT expressions in a single instruction. Table 3.2 provides a summary and small examples demonstrating the LLVM-IR instructions each COGENT expression is translated into. The rest of this section provides a detailed explanation of the translation of COGENT expression into corresponding instructions.

To compile a COGENT expression we have the `exprToLLVM` function with the signature given in Listing 3.6.

Expression	**Cogent example**	**LLVM IR**
Unit	`()`	`0`
Integer	`23`	`23`
String	`"Cogent"`	address to const array
Arithmetic ops	`1 + 2`	`add 1, 2`
Logical ops	`True && False`	`and 1, 0`
Comparison ops	`7 > 4`	`icmp ugt 7, 4`
Bitwise operators	`1 << 3`	`shl 1, 3`
Unary ops	`not 23`	`xor -1, 23`
Casting	`upcast 9 : U16`	`zext i8 9 to i16`
Promotion	`promote`	no-op
Variable bindings	`let a in b`	in-scope vars threaded via `Env` state
Branching	`if .. then .. else`	`br` and `phi`
Case analysis	`case .. esac`	`br` and `phi`
Record literals	`{a: 2}`	`insertvalue undef, 2, 0`
Member access	`x.a`	`extractvalue` / `getelementptr`, `load`
Take	`b' {a = x} = b`	`extractvalue` / `getelementptr`, `load`
Put	`x{a = 3}`	`insertvalue` / `getelementptr`, `store`
Con	`Success 2`	`insertvalue`
Functions	`f 3`	`call @f(3)`

Table 3.2: A summary of the backend's expression translation

Listing 3.6: COGENT expression compilation (`Expr.hs`)

```
exprToLLVM :: TypedExpr t v a b -> Codegen Operand
```

Since COGENT is a functional programming language, each expression produces a value. The return value of the `exprToLLVM` function indicates that this is a `Codegen` computation with the result being an `Operand`. Hence when compiling an expression, we must return the operand corresponding to the register (or constant) containing the final result of the expression. The `llvm-hs` library provides utility functions for each LLVM-IR instruction that yield operand results inside our `Codegen` monad.

Literals

To start, the simplest pieces of syntax are COGENT's literals which correspond to constant operands in LLVM-IR. Firstly, the unit literal could be given an arbitrary value but in the following excerpt of our implementation, it is compiled to zero.

```
exprToLLVM (TE _ Unit) = pure $ int8 0
```

Next, integer literals consist of a size (in bits) and an unsigned value.

```
exprToLLVM (TE _ (ILit val sz)) = pure $ constInt (primIntSizeBits sz)
    val
```

The above case uses `constInt`, a helper for generating integer constants defined in Listing 3.7.

Listing 3.7: Constructing `llvm-hs` integer constants

```
constInt :: Integer -> Integer -> Operand
constInt n i =
    ConstantOperand C.Int {integerBits = fromInteger n, integerValue = i}
```

The last kind of literals, string literals, are also constant operands, but their value is the address to a constant array containing the characters of the string. `llvm-hs` provides a `globalStringPtr` helper for generating these arrays, provided we supply it with a fresh name to use for the global variable. Listing 3.8 continues our implementation.

Listing 3.8: COGENT expression compilation (`Expr.hs`)

```
exprToLLVM (TE _ (SLit str)) =
    freshName "str" >>= (fmap ConstantOperand . globalStringPtr str)
```

Binary Operators

Continuing, COGENT binary operator expressions consist of an operator and two sub-expression to apply the operator to. Each operator corresponds to a single LLVM-IR instruction. All COGENT binary operators are supported. In the `llvm-hs` library, operators are represented by binary functions (of the same name) that wrap the result in a monad. The full mapping from COGENT operators to LLVM operators is given in Listing 3.9.

Listing 3.9: Mapping COGENT operators to LLVM-IR (`Expr.hs`)

```haskell
toLLVMOp :: Sy.Op -> (Operand -> Operand -> Codegen Operand)
toLLVMOp Sy.Plus = add
toLLVMOp Sy.Minus = sub
toLLVMOp Sy.Times = mul
toLLVMOp Sy.Divide = udiv
toLLVMOp Sy.Mod = urem
toLLVMOp Sy.And = IR.and
toLLVMOp Sy.Or = IR.or
toLLVMOp Sy.Gt = icmp UGT
toLLVMOp Sy.Lt = icmp ULT
toLLVMOp Sy.Le = icmp ULE
toLLVMOp Sy.Ge = icmp UGE
toLLVMOp Sy.Eq = icmp P.EQ
toLLVMOp Sy.NEq = icmp NE
toLLVMOp Sy.BitAnd = IR.and
toLLVMOp Sy.BitOr = IR.or
toLLVMOp Sy.BitXor = xor
toLLVMOp Sy.LShift = shl
toLLVMOp Sy.RShift = \a b -> emitInstr (typeOf a) $ LShr False a b []
```

The final case is handled manually to set the `exact` flag for `lshr` to `False`, allowing non-zero bits to be shifted out.

With a mapping for each operator, what remains is to provide a translation for the actual `a op b` expression. Our approach is to compile each operand (`a` and `b`), and then emit code for the operator `op` using the above translation. Our chosen representation of booleans requires us to zero-extend boolean results to an `i8` value rather than keeping them as an `i1`. It should be noted that the LLVM optimiser is likely to remove these unnecessary casts, except when a boolean value is directly returned from a function, where the final cast to `i8` will be retained.

Haskell's do-notation is used in Listing 3.10 to implement the above process as an effectful computation inside the `Codegen` monad, in which the final result `res` is the operand corresponding to the result of the binary operation.

Listing 3.10: COGENT expression compilation (`Expr.hs`)

```
exprToLLVM (TE t (Op op [a, b])) = do
    oa <- exprToLLVM a
    ob <- exprToLLVM b
    res <- toLLVMOp op oa ob
    case t of
        -- Coerce boolean results back to a full byte
        TPrim Boolean -> if typeOf res == i8 then pure res
                                              else zext res i8
        _ -> pure res
```

Unary Operators

As well as binary operators, COGENT includes two unary operators, `not` and `complement`. However, LLVM-IR doesn't contain instructions directly implementing these unary operators. Instead, we must emulate each using the existing binary operators. The bitwise `not` operator can be implemented as follows using the `xor` instruction with the `-1` literal which represents all 1s in an unsigned format. This equivalence follows from the definition of both operators.

```
exprToLLVM (TE t (Op Sy.Complement [a])) = do
    oa <- exprToLLVM a
    xor oa (constInt (typeSize t) (-1))
```

For `complement`, the logical not operator, we can check if the boolean argument is equal to the `0` literal using `icmp eq` (`icmp P.EQ` in `llvm-hs`) in the following code. We do this to mimic C semantics, treating all non-zero values as true and only zero as false. As above, we must also remember to convert the `i1` result back to an `i8` using `zext`.

```
exprToLLVM (TE _ (Op Sy.Not [a])) = do
    oa <- exprToLLVM a
    res <- icmp P.EQ oa (int8 0)
    zext res i8
```

Bindings and Variables

Next, we deal with the `let` syntax of COGENT and the resultant bound variables. Recall that inside the `Codegen` monad we maintain a list of in-scope variables during code generation. Hence, for a `let` binding, we can evaluate the bound expression, and then temporarily add it to the front of the variable list when evaluating the body of the binding. Our concise way to do this is given below, using the `bind` helper defined earlier in Listing 3.2.

```
exprToLLVM (TE _ (Let _ val body)) =
    exprToLLVM val >>= flip bind (exprToLLVM body)
```

`let!` bindings have identical dynamic semantics to regular `let` bindings, meaning they can be compiled as their non-exclamatory counterparts - this is handled as follows.

```
exprToLLVM (TE t (LetBang _ a val body)) =
    exprToLLVM (TE t (Let a val body))
```

Casts

The first type of cast permitted in COGENT is an integer `upcast`. We can translate this directly to LLVM-IR's `zext`, which we have already seen used for our boolean conversion, in the below case.

```
exprToLLVM (TE _ (Cast t e)) = do
    v <- exprToLLVM e
    toLLVMType t >>= zext v
```

The other type of cast in COGENT is variant promotion, which is a no-op in the dynamic semantics. This means we can compile the sub-expression directly as shown here.

```
exprToLLVM (TE _ (Promote _ e)) = exprToLLVM e
```

The COGENT language does not allow unsafe casts to any type, but the operation defined in Listing 3.11 is useful when compiling variant expressions. Our unsafe cast is achieved by using the `bitcast` operation on a temporary piece of stack memory.

Listing 3.11: Helper for arbitrary downcasting (`Expr.hs`)

```haskell
castVal :: Operand -> AST.Type -> Codegen Operand
castVal o t = do
    ptr_o <- alloca (typeOf o) Nothing 4
    ptr_t <- bitcast ptr_o (ptr t)
    store ptr_o 1 o
    load ptr_t 1
```

This implementation is unsafe when doing a cast to a larger type since the stack memory is allocated for the current type of `o`. For upcasts, we instead allocate stack memory for the larger type `t` first, shown in Listing 3.12.

Listing 3.12: Helper for arbitrary upcasting (`Expr.hs`)

```haskell
upcastVal :: Operand -> AST.Type -> Codegen Operand
upcastVal o t = do
    ptr_t <- alloca t Nothing 4
    ptr_o <- bitcast ptr_t (ptr (typeOf o))
    store ptr_o 1 o
    load ptr_t 1
```

Records

COGENT allows the construction of record literals by providing an expression for each field. To do this in LLVM-IR we must evaluate the sub-expressions to insert into each field of the record, whilst building up the record using successive `insertvalue` instructions. We can elegantly perform this operation using a monadic fold. To have access to each field's index during compilation, we construct a zipped list containing the record's sub-expressions and their index in the record to fold over.

Listing 3.13: COGENT expression compilation (`Expr.hs`)

```haskell
exprToLLVM (TE t (Struct flds)) = do
    t' <- toLLVMType t
    foldlM
        (\struct (i, v) -> exprToLLVM v
            >>= \value -> insertValue struct value [i])
        (constUndef t')
        [(i, snd fld) | (i, fld) <- zip [0 ..] flds]
```

In Listing 3.13 the `constUndef` helper is used to construct a constant operand of the LLVM-IR `undef` value. This undefined value can be used to represent an empty struct of the desired type which is our starting point before inserting each field.

The next record operation in COGENT is the ability to access record members. For unboxed records, this is doable with a single `extractvalue` instruction, but for boxed records, we must first calculate the correct memory offset with `getelementptr` and then `load` the value from memory. Our compilation case looks deceptively simple:

```
exprToLLVM (TE _ (Member recd fld)) = snd <$> loadMember recd fld
```

The `loadMember` helper defined in Listing 3.14 is doing the heavy lifting for this case.

Listing 3.14: Helper for member operations (`Expr.hs`)

```
loadMember :: TypedExpr t v a b -> Int -> Codegen (Operand, Operand)
loadMember recd fld = do
    recv <- exprToLLVM recd
    fldv <-
        if isUnboxed (exprType recd)
            then extractValue recv [toEnum fld]
            else gep recv [int32 0, int32 (toEnum fld)]
                >>= \fldp ->load fldp 0
    pure (recv, fldv)
```

`loadMember` not only computes an operand corresponding to the accessed field, but it also yields the operand corresponding to the whole record expression. This decision allows us to implement the `take` expression with ease. In addition to loading the member, this COGENT expression binds both the field and record as variables. In Listing 3.15 we demonstrate the use the `bind` helper function from earlier to help with this case.

Listing 3.15: COGENT expression compilation (`Expr.hs`)

```
exprToLLVM (TE _ (Take _ recd fld body)) = do
    (recv, fldv) <- loadMember recd fld
    foldr bind (exprToLLVM body) [recv, fldv]
```

The final COGENT expression involving records is the `put` expression which corresponds to field insertion. First, we need to compile the expression to be inserted, then for unboxed records, we produce a single `insertvalue` instruction, and for boxed records, we need to do a `getelementptr` before then using `store` to modify the field in memory. The Haskell implementation which follows for this case resembles the structure of the `loadMember` helper above.

```haskell
exprToLLVM (TE _ (Put recd fld val)) = do
    recv <- exprToLLVM recd
    v <- exprToLLVM val
    if isUnboxed (exprType recd)
        then insertValue recv v [toEnum fld]
        else do
            fldp <- gep recv [int32 0, int32 (toEnum fld)]
            store fldp 0 v
            pure recv
```

Variants

Next, we consider the COGENT expression for constructing a variant literal. This case is similar to a `put` expression, except we must also insert a value corresponding to the tag for the chosen variant constructor in the first field before inserting the argument into the second field. In the following case, `insertvalue` is used to set both fields in the structure. For our LLVM-IR program to be type-correct, we need to cast the payload to the maximal type of the variant before inserting it into the field. However, for variant constructors with no parameter, which are desugared to constructors accepting a unit, we can skip casting and inserting altogether since setting the tag field is sufficient in this case.

```haskell
exprToLLVM (TE t (Con tag e (TSum ts))) = do
    tagv <- tagIndex tag
    t' <- toLLVMType t
    tagged <- insertValue (constUndef t') tagv [0]
    v <- exprToLLVM e
    case e of
        -- Don't bother doing anything for constructors with no arguments
        TE TUnit _ -> pure tagged
        _ -> do
            casted <- toLLVMType (maxMember ts) >>= upcastVal v
            insertValue tagged casted [1]
```

Cases

Since LLVM-IR does not support a form of `if` or `switch` statements like in C, conditional expressions must be implemented via explicit branching between blocks. Recall that the `phi` instruction of LLVM-IR allows a program to assign a value based on the predecessor block. Hence to implement COGENT's `if c then t else f` expressions we can jump to one of two new blocks containing compiled code for each subexpression `t` and `f` depending on whether the condition `c` is true or false. Then, at the end of each block, we can jump to a third block containing a `phi` instruction which will recover the final result from either branch, depending on the execution path taken.

Moreover, some complexity is introduced by the fact that the compilations for the `if` statement's sub-expressions may themselves contain further branching. Hence, we must inspect the current block label before terminating each block, otherwise, we may refer to the wrong predecessor block in our `phi` instruction. Haskell's recursive do-notation (named mdo-notation after its `mdo` syntax) allows us to create bindings that are visible to prior instructions in the `mdo` group, provided our monad belongs to the `MonadFix` class which defines a fixed-point operator (our `Codegen` does thanks to `llvm-hs`). This mdo-notation simplifies code generation involving branches as it allows us to 'time travel' (Fancher, 2017) and lazily emit branch instructions before we have named the blocks to jump to. For the condition, we can check the truthiness of a `i8` boolean by checking that it is not equal to zero, thus permitting any non-zero value as truthy, as in C. Lastly, in the `phi` node we provide a value to return depending on which predecessor branch was used. Line 142-144 in the following listing compute the condition, 145-148 handle the true case, 149-152 handle the false case, and 153-154 recover the result from either branch.

```
141 exprToLLVM (TE _ (If cd tb fb)) = mdo
142     v <- exprToLLVM cd
143     cond <- icmp NE v (int8 0)
144     condBr cond brTrue brFalse
145     brTrue <- block `named` "if.true"
146     valTrue <- exprToLLVM tb
147     brTrue' <- currentBlock
148     br brExit
```

```
    brFalse <- block `named` "if.false"
    valFalse <- exprToLLVM fb
    brFalse' <- currentBlock
    br brExit
    brExit <- block `named` "if.done"
    phi [(valTrue, brTrue'), (valFalse, brFalse')]
```

Furthermore, the other form of conditional branching in COGENT is variant case analysis via pattern matching. The upstream compilation desugars COGENT's case expressions into simpler binary `Case` expressions. The only difference between these expressions and the `if` expressions we just dealt with is that instead of evaluating a boolean expression, we must retrieve the current tag for the variant and see if matches the current case, and secondly, in line with COGENT's semantics, we must bind variables in each branch. If the variant matches, we bind a casted version of the value inhabiting the variant. If it does not match, we bind the variant itself. As used previously, the `bind` helper allows us to temporarily modify the variable state for the compiler.

```
116     exprToLLVM (TE _ (Case e@(TE rt _) tag (_, _, tb) (_, _, fb))) = mdo
117     variant <- exprToLLVM e
118     tagv <- extractValue variant [0]
119     cond <- tagIndex tag >>= icmp P.EQ tagv
120     condBr cond brMatch brNotMatch
121     brMatch <- block `named` "case.true"
122     v <- extractValue variant [1]
123     casted <- toLLVMType (tagType rt tag) >>= castVal v
124     valTrue <- bind casted $ exprToLLVM tb
125     brMatch' <- currentBlock
126     br brExit
127     brNotMatch <- block `named` "case.false"
128     valFalse <- bind variant $ exprToLLVM fb
129     brNotMatch' <- currentBlock
130     br brExit
131     brExit <- block `named` "case.done"
132     phi [(valTrue, brMatch'), (valFalse, brNotMatch')]
```

Note that lines 120-121, 125-127, and 129-132 in the above code are semantically equivalent to their corresponding lines in the compilation of `if` expressions.

A final operation for variants, the `esac` expression, arises at the end of variant case analysis. It indicates a guaranteed tag match when all alternatives have been checked. For this expression, all we must do is extract the value from the variant's second field and cast it to the expected type. Since the COGENT AST is type-correct, this cast is safe.

```
exprToLLVM (TE t (Esac e)) = do
    variant <- exprToLLVM e
    v <- extractValue variant [1]
    toLLVMType t >>= castVal v
```

Functions

Function expressions in COGENT are just references to top-level functions and can be compiled directly to constant operands containing a global reference to that function. The name of the function is used when constructing the global reference. To ensure this function expression refers to the LLVM-compatible wrapper/implementation of the function, `.llvm` is appended to the name of all functions which are safely callable from LLVM-IR code, and the definitions without `.llvm` appended are reserved for being called from C. Our implementation below firstly converts the COGENT function type to an LLVM-IR type, before constructing a constant operand containing the global function reference.

```
exprToLLVM (TE t (Fun f _ _ _)) = do
    ft <- toLLVMType t
    pure $ ConstantOperand $ C.GlobalReference ft
        ((mkName . (++ ".llvm") . toCName . unCoreFunName) f)
```

To compile a function application, we need to first compile the argument expression, the function expression, and then produce an LLVM-IR `call` instruction to call the function with that argument, shown in the following case. Since the COGENT program is type-correct, we need not worry about the type of function or argument, but even if it were somehow wrong, the LLVM compiler would report the error during linking.

```
exprToLLVM (TE _ (App f a)) = do
    arg <- exprToLLVM a
    fun <- exprToLLVM f
    call fun [(arg, [])]
```

3.1.4 Program Definitions

COGENT functions are not first-class citizens, instead, they are given top-level definitions in a program (like in C and LLVM-IR). Also at the top level resides abstract function declarations, and type definitions. Function definitions correspond directly to LLVM function definitions (`define` in LLVM-IR). Abstract function declarations can be also defined in LLVM-IR using `declare` and are given no body. Abstract type definitions correspond to LLVM-IR's forward type declarations (`type opaque`) which are emitted when their usages occur in the AST. Lastly, COGENT's non-abstract type definitions, such as named aliases for record and variant types, are removed in an earlier stage of compilation and require no corresponding LLVM-IR type definitions.

For example, an abstract type definition such as `type Heap` would be compiled to `%Heap = type opaque` in LLVM-IR. On the other hand, the abstract function definition `exit : U16 -> ()` is compiled to `declare i8 @exit(i16 %a_0)`.

3.1.5 C Foreign Function Interface

Since COGENT is such a restricted language, it is difficult to program entirely within it and many functions are instead implemented in C. Another reason to support compatibility is that COGENT systems need to interact with surrounding systems that are typically implemented in C. By allowing for compatibility with external C code, our LLVM backend becomes significantly more useful. However, even though the Clang compiler will happily compile C code to LLVM-IR and link it with ours, we must take care in ensuring the generated IR is compatible with the LLVM-IR our COGENT backend generates.

In most simple cases, this compatibility comes for free by virtue of our chosen representations of primitive types - these are directly compatible with C code. Our representation of unboxed aggregate types such as records and variants should also be equivalent, however, this is not the case.

A Motivating Example

To understand the complexity, consider the following functions implemented for five C structs of increasing size:

```c
typedef struct {int f1; } s1;
typedef struct {int f1; int f2; } s2;
typedef struct {int f1; int f2; int f3; } s3;
typedef struct {int f1; int f2; int f3; int f4; } s4;
typedef struct {int f1; int f2; int f3; int f4; int f5; } s5;

s1 id1(s1 x) { return x; }
s2 id2(s2 x) { return x; }
s3 id3(s3 x) { return x; }
s4 id4(s4 x) { return x; }
s5 id5(s5 x) { return x; }
```

If we concern ourselves with the LLVM-IR function declarations generated by `clang x86-64` for these functions, the argument and return types do not correspond to the canonical LLVM translations for the C struct types.

```llvm
%struct.s5 = type { i32, i32, i32, i32, i32 }

define i32 @id1(i32 %0) { ... }
define i64 @id2(i64 %0) { ... }
define { i64, i32 } @id3(i64 %0, i32 %1) { ... }
define { i64, i64 } @id4(i64 %0, i64 %1) { ... }
define void @id5(%struct.s5* noalias sret(%struct.s5) %0, %struct.s5*
    byval(%struct.s5) %1) { ...}
```

The compiler has instead attempted to pack aggregate type arguments into one or two arguments during the translation to LLVM-IR. It would rather pass `s1` as an `i32` rather than a `{ i32 }`. Moreover, it is very happy to pass `s2`, a struct with two 32-bit fields, as a single 64-bit argument. When it is impossible to fit the aggregate type into two arguments, a stack pointer is used instead, such as for `s5`. Similarly, for return types, when the 2x64-bit limit is reached, the aggregate value is returned by populating a pointer passed into the function as an additional argument. Problematically, the C compiler performs this optimisation manually during the conversion to LLVM-IR, so we cannot rely on the LLVM compiler to do the same for us. Instead, we are at the mercy of the compiler's implementation of the platform ABI, which specifies how argument and return value marshalling is performed, and we have no choice but to try and match it in our LLVM backend.

Therefore, if we are to generate LLVM-IR from our COGENT backend which can be called from compiled C code, we must design glue code generators to wrap our concrete functions. Furthermore, if we would like to call external C code in our generated LLVM-IR, we need to generate similar wrappers for abstract functions.

Aggregate Layout Rules

Before proceeding with the details of the wrapper generator implementations, it is worth demystifying the aggregate layout rules that the compiler is using. Unfortunately, these rules differ by compiler and architecture, and since developing a platform-independent FFI was beyond the scope of this thesis, we will concern ourselves with the particular rules of `x86-64` only.

To convert a COGENT type into a compatible layout we need to start with how the type should be laid out in memory according to C's standard struct alignment rules. In Listing 3.16 we define an abstract *C layout* with respect to the C type system with pointers, immediate values, structs and unions.

Listing 3.16: Definition of possible abstract C layouts (`Types.hs`)

```haskell
data CLayout t b = Ptr (Core.Type t b)
                 | Im Size
                 | St [CLayout t b]
                 | Un [CLayout t b]
```

In Listing 3.17, given a COGENT type, we compute a `CLayout` for that type:

Listing 3.17: Conversion from COGENT types to C layouts (`Types.hs`)

```haskell
typeLayout :: Core.Type t b -> CLayout t b
typeLayout (TPrim p) = Im (primIntSizeBits p)
typeLayout TUnit = Im 8
typeLayout (TRecord _ ts Unboxed) = St (typeLayout <$> fieldTypes ts)
typeLayout (TSum ts) = St [Im 32, Un (typeLayout <$> fieldTypes ts)]
typeLayout t = Ptr t
```

Next, in Listing 3.18 we define a concrete *memory layout* that includes padding, values, pointers, and invalid memory.

Listing 3.18: Definition of fields in a memory layout (`Types.hs`)

```haskell
data Field t b = Padding
               | Value
               | Pointer (Core.Type t b)
               | Invalid
type MemLayout t b = [(Field t b, Size)]

-- Convert a C-like layout to a memory layout including padding
flatLayout :: CLayout t b -> (Size, MemLayout t b)
```

We can convert our abstract `CLayout` to a tuple of its total size and corresponding `MemLayout` by a ‘flattening’ function. Listing 3.19 shows the simple cases for this function. Pointers are easy to store in memory, they are a single field occupying the system’s `pointerSizeBits`, which is 64 bits for `x86-64`. Immediate values are also straightforward, occupying whatever number of bits they require for their type. Unions are more complex; we need to calculate the flattened layouts of each possible field in the union and take whatever layout has the largest overall size.

Listing 3.19: Conversion from C layouts to memory layouts (`Types.hs`)

```haskell
flatLayout (Ptr t) = (pointerSizeBits, [(Pointer t, pointerSizeBits)])
flatLayout (Im i) = (i, [(Value, i)])
flatLayout (Un ts) = maximumBy (compare `on` fst)
                               (reverse (flatLayout <$> ts))
```

Lastly, the most complex case is structs which include padding between fields to align fields based on their type. Thankfully, we can take a `CLayout` and determine its desired alignment quite easily using the code in 3.20.

Listing 3.20: Calculating alignments for a C layout (`Types.hs`)

```haskell
typeAlignment :: CLayout t b -> Size
typeAlignment (Ptr _) = pointerSizeBits
typeAlignment (Im i) = min i pointerSizeBits
typeAlignment (St ts) = maximum (typeAlignment <$> ts)
typeAlignment (Un ts) = maximum (typeAlignment <$> ts)
```

Using this alignment function, we can compute a flattened memory layout for structs by calculating the alignment for each field and including padding between fields, as necessary. The Haskell code implementing this case is given in full in Listing 3.21

Listing 3.21: Conversion from C layouts to memory layouts (`Types.hs`)

```haskell
flatLayout (St ts) = foldl flatLayout' (0, []) ts
    where
        flatLayout' :: (Size, MemLayout t b) -> CLayout t b -> (Size,
            MemLayout t b)
        flatLayout' (offset, layout) t =
            let layout' = flatLayout t
                alignment = typeAlignment t
                offset' = roundUp offset alignment
                padding = offset' - offset
             in ( offset' + fst layout'
                , layout ++ [(Padding, padding) | padding > 0] ++ snd
                    layout'
                )
```

For a concrete memory layout, we can look up a particular offset and see what inhabits that space using the function in Listing 3.22. If the offset does not line up exactly, we can return the `Invalid` field which is reserved for this purpose.

Listing 3.22: Looking up a field in a memory layout (`Types.hs`)

```haskell
memLookup :: Size -> MemLayout t b -> Field t b
memLookup _ [] = Invalid
memLookup offset ((m, s) : ms)
    | offset == 0 = m
    | offset < 0 = Invalid
    | otherwise = memLookup (offset - s) ms
```

The final piece of the puzzle is to then take one of these flattened memory layouts and convert it to an LLVM-IR layout that is compatible with `clang x86-64`. As we saw illustrated in the C example, there are three different virtual *register layouts* for an aggregate defined as a data type in Listing 3.23: passing it as a single virtual register, two virtual registers, or by reference.

Listing 3.23: Definition of register layouts (`CCompat.hs`)

```haskell
data RegLayout = One AST.Type
               | Two AST.Type AST.Type
               | Ref
```

We start by converting a memory layout into a single register argument in Listing 3.24 using a wrapper around the `memLookup` which additionally takes a register size. If our memory lookup gives us a pointer, then it will occupy a single register, otherwise, we use the provided register size.

Listing 3.24: Converting a memory layout to an LLVM-IR type (`CCompat.hs`)

```haskell
toReg :: Size -> MemLayout t b -> Integer -> LLVM AST.Type
toReg offset layout regSize = case memLookup offset layout of
    Pointer pt -> toLLVMType pt
    _ -> pure $ IntegerType $ fromInteger regSize
```

Putting this all together is the `regLayout` function in Listing 3.25 which, given a COGENT type, produces a suitable register layout to pass that type as. If the computed size in memory of the type is less than what would fit in one register (64 bits), or it is a primitive type, then the correct choice is to pass this type via a single virtual register. If the type cannot fit in one register but could fit in two, we split the memory layout into two 64-bit sections and compute suitable LLVM types for each half. Lastly, if the type exceeds the two-register limit, it should be passed by reference.

Listing 3.25: Choosing a register layout for a COGENT type (`CCompat.hs`)

```haskell
regLayout :: Core.Type t b -> LLVM RegLayout
regLayout t
    | size <= p || isPrim t = One <$> toReg 0 layout size
    | size <= 2 * p = liftM2 Two (toReg 0 layout p) (toReg p layout (size
        - p))
    | otherwise = pure Ref
    where
        p = pointerSizeBits
        (size, layout) = (flatLayout . typeLayout) t
```

This approach is not directly applicable to other architectures such as ARM which for example will pass structs as array arguments instead. It will also need adaptation to support Dargent, which is COGENT's own data layout specification mechanism. However, it is still suitable for use in the generation of glue code which can be used to test the LLVM backend of COGENT with C code compiled with `clang x86-64`. Using the above layout rules, we can generate glue code to wrap COGENT and C functions that manipulate aggregate types.

Wrappers for Cogent Functions

The LLVM backend for COGENT produces wrappers that allow COGENT functions to be called from C code. It is only necessary to create wrappers for functions that accept or return variants or unboxed records, as every other type fits into a single argument or return virtual register. The anatomy of a COGENT wrapper is as follows:

1. If the COGENT function requires argument type marshalling:
    - If the argument was passed by reference, load it from memory
    - Else if a single argument was passed, cast it to the expected type
    - Otherwise, two arguments were passed, store them in a structure type and then cast the entire structure to the expected type
2. Call the inner COGENT function
3. If the COGENT function requires return type marshalling:
    - If the return value should be passed by reference, store it in the correct memory location, and then return `void`
    - Otherwise, the return value should be by value, cast it to the expected type and return it

We use a convention that COGENT functions generated by the LLVM backend have the suffix `.llvm` appended to their function name to denote that they are safe to be called from other LLVM functions but not from C code. The generated wrapper does not include the suffix - it is the function expected to be called by external C functions.

Wrappers for Abstract C Functions

We also need to create 'un-wrappers' that do the same kind of marshalling before a COGENT function calls an external C function. The anatomy of a C wrapper is the inverse of a COGENT wrapper:

1. If the C function requires argument type marshalling:
    - If the argument should be passed by reference, allocate stack memory for it, and store the argument
    - Else if a single argument should be passed, cast the argument to the expected type
    - Otherwise, two arguments should be passed, cast the argument to a structure type containing a field for each argument type, and then extract the two fields
2. If the C function's return value is passed by reference:
    - Allocate stack memory for the return value and pass this pointer as an additional argument
3. Call the inner C function
4. If the C function requires return type marshalling:
    - If the return value was passed by reference, load the value from that address
    - Otherwise, the return value was by value, cast it to the expected type and return it

The generated wrappers for external C functions have the suffix `.llvm` appended to their function name, representing they can be called from COGENT functions which have been compiled to LLVM-IR.

Header File Generation

Ultimately, to link an LLVM-IR COGENT program with C code we need to provide header files containing function prototypes and type definitions. It is a simple task to produce these during compilation if we are not too concerned about their readability.

Our generated header files will contain the following sections:

- A header guard to prevent multiple inclusion of this header
- An import for the common COGENT definitions
- An `enum` containing each variant tag name in the correct order
- A list of type definitions for each structurally different type in the COGENT program
- A list of convenient type aliases for each argument and return type
- A list of function prototypes for each function in the COGENT program

Firstly, the header guard is trivial to generate using an `#ifndef` macro based on the COGENT program's file name. Next, the common COGENT definitions can also be included via a standard `#include` statement in the generated header file. Our definitions, shown in Listing 3.26 give aliases that allow the programmer to refer to COGENT's primitive types rather than C types, as well as enums for boolean and unit values.

Listing 3.26: Common FFI definitions (`cogent-llvm-defns.h`)

```c
typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned int u32;
typedef unsigned long long u64;
typedef u8 bool_t;
typedef u8 unit_t;

enum { false, true };
enum { unit };
```

In order to generate an `enum` for variant tags, we need to walk the AST and collect all the unique tag names mentioned in the COGENT program's types. This can be done by the `collectTags` function in Listing 3.27.

Listing 3.27: Collection of all tags mentioned in a program (`Types.hs`)

```haskell
collectTags :: Core.Definition TypedExpr VarName VarName -> Set TagName
collectTags (FunDef _ _ _ _ t rt _) =
    collectTags' t `union` collectTags' rt
collectTags (AbsDecl _ _ _ _ t rt) =
    collectTags' t `union` collectTags' rt
collectTags (TypeDef _ _ mt) = maybe empty collectTags' mt

collectTags' :: Core.Type t b -> Set TagName
collectTags' (TRecord _ ts _) = unions $ collectTags' <$> fieldTypes ts
collectTags' (TSum ts) = fromList (fst <$> ts) `union` unions (
    collectTags' <$> fieldTypes ts)
collectTags' (TFun t1 t2) = collectTags' t1 `union` collectTags' t2
collectTags' (TCon _ ts _) = unions $ collectTags' <$> ts
collectTags' _ = empty
```

Once this has been done, the list of all tag names can be retrieved from the set as an ascending order list using Haskell's `elems` function. With this list, it is simple to `intercalate` them into an `enum` definition for inclusion in the header.

Next, we would like to collect all type definitions, aliases, and function prototypes for the COGENT program. It is possible to use the state monad to collect all these in various associative lists. The chosen approach to represent the collected definitions via strings drastically simplifies the implementation given in Listing 3.28.

Listing 3.28: Header generation state (CCompat.hs)

```
type CType = String
type CIdent = String
data TypeDef = Fn CType CType | Ty CType deriving (Eq)

data HGen = HGen
    { typeDefs :: [(TypeDef, CIdent)]
    , typeAliases :: [(CIdent, CIdent)]
    , funProtos :: [(CType, CType, CIdent)]
    }
```

At this point, it is important to highlight the distinction made here between type aliases and type definitions, even though both are compiled to `typedef` declarations in C. A *type definition* is a structurally unique definition of a COGENT aggregate type (record, variant, or function) in C. Meanwhile, a *type alias* is another name given to a COGENT type, which may be an aggregate or primitive type.

For each definition in the COGENT program, we will emit some C definitions using the function in Listing 3.29. COGENT function definitions will have C prototypes associated with them, and COGENT's type definitions will translate to C type aliases.

Listing 3.29: C definition generation (CCompat.hs)

```
define :: Core.Definition TypedExpr VarName VarName -> State HGen ()
define (FunDef _ name _ _ t rt _) = toCProto name t rt
define (AbsDecl _ name _ _ t rt) = toCProto name t rt
define (TypeDef name _ (Just t)) =
    toCType t >>= typeAlias (toCName name) . filter (/= '*')
define _ = pure ()
```

In Listing 3.30 we dig deeper into the `toCProto` function - to create a C function prototype we must convert the argument and return types of the function and then emit a suitable function prototype. In addition, argument and return type aliases are emitted as it is often nicer to use these aliases when writing C code. For higher-order functions, aliases are generated as normal, but these aliases will refer to additional type definitions for the function-type argument and/or result.

Listing 3.30: Function prototype generation (`CCompat.hs`)

```haskell
toCProto :: FunName -> Core.Type t b -> Core.Type t b -> State HGen ()
toCProto name t rt = do
    arg <- toCType t
    ret <- toCType rt
    let cName = toCName name
    modify $ \s -> s {funProtos = (arg, ret, cName) : funProtos s}
    typeAlias (cName ++ "_arg") arg
    typeAlias (cName ++ "_ret") ret
```

The `toCType` function, used above, converts a COGENT type to a C type name. When converting aggregate types, a new type definition is created only if the desired aggregate type has not yet been defined. If the type already exists it is retrieved from the `HGen` state, otherwise a new `typedef` is added with a fresh identifier. Whilst deterministic, these identifier names are unstable and will likely change when the COGENT program is modified, so in most cases, it is less painful to use the argument and return type aliases.

Finally, after running the `define` computation over each definition in the COGENT program, we can retrieve all the necessary type definitions, aliases, and function prototypes from the `HGen` state and print them in the header file.

3.1.6 Example

Cogent Program

The following COGENT program in Listing 3.31 is similar to the LLVM-IR example introduced in Listing 2.1. It adds two `U32` values which are passed to the function as a tuple. After adding the arguments, it checks if the result has overflown, and returns an appropriate variant value, either an `Error` or the result inside a `Success`.

Listing 3.31: A simple COGENT program (`adder.cogent`)

```
add : (U32, U32) -> < Success U32 | Error >
add (x, y) =
    let sum = x + y
    in if (sum < x) || (sum < y)
        then Error
        else Success sum
```

LLVM-IR Translation

The only definition in this program to compile is the `add` function. Listing 3.33 shows the output of our compiler next to the COGENT input. The function has been spaced out in Listing 3.32 so that its sub-expressions roughly correspond to the generated LLVM-IR on each line. In the output, the tuple argument type `(U32, U32)` is compiled to a structure type `{ i32, i32 }`. Lines 4 and 5 use `extractvalue` to unpack the two fields in this structure. COGENT addition translates directly to an `add` instruction in LLVM on line 6. The two logical comparisons are performed with `icmp ult`. Because our representation of booleans is `i8` rather than `i1`, lines 8 and 10 zero extend the results of each comparison. The COGENT or expression is implemented via LLVM-IR's `or`, but again we cast the result to `i8`. To check the boolean condition, the result is compared to `0` on line 12. In the true branch, the tag field of the variant is set to `0` using `insertvalue` (line 15) and we jump to the done branch, in the false branch the tag field is set as `1` (line 18). To store the `sum` inside the constructed variant we first cast it to the maximal type for the variant over lines 19-22. Since `i32` is already the maximal type for the variant, this is a no-op and will be optimised out. Lastly, in the done branch, the `phi` on line 26 picks the value to return based on whether the true or false path was taken.

Listing 3.32: Input

```
add : (U32, U32) ->
  < Success U32 | Error >
add (
     x,
     y) =
  let sum = x + y
  in if (sum < x)

  || (sum < y)




    then
        Error

    else
        Success




          sum
```

Listing 3.33: Output (adder.ll)

```
define { i32, i32 } @add.llvm({ i32, i32 } %a_0) {

entry_0:
  %0 = extractvalue { i32, i32 } %a_0, 0
  %1 = extractvalue { i32, i32 } %a_0, 1
  %2 = add i32 %0, %1
  %3 = icmp ult i32 %2, %0
  %4 = zext i1 %3 to i8
  %5 = icmp ult i32 %2, %1
  %6 = zext i1 %5 to i8
  %7 = or i8 %4, %6
  %8 = icmp ne i8 %7, 0
  br i1 %8, label %if.true_0, label %if.false_0
if.true_0:                                ; preds = %entry_0
  %9 = insertvalue { i32, i32 } undef, i32 0, 0
  br label %if.done_0
if.false_0:                               ; preds = %entry_0
  %10 = insertvalue { i32, i32 } undef, i32 1, 0
  %11 = alloca i32
  %12 = bitcast i32* %11 to i32*
  store i32 %2, i32* %12, align 1
  %13 = load i32, i32* %11, align 1
  %14 = insertvalue { i32, i32 } %10, i32 %13, 1
  br label %if.done_0
if.done_0:                                ; preds = %if.false_0, %if.true_0
  %15 = phi { i32, i32 } [%9, %if.true_0], [%14, %if.false_0]
  ret { i32, i32 } %15
}
```

LLVM-IR Optimisation

As mentioned, the various casting instructions used in the generated code are inconsequential for this program, so the LLVM optimiser happily removes them for us. Listing 3.34 shows the optimised version of `adder.ll` using `opt -O1`.

Listing 3.34: Optimised LLVM IR for adder

```
define { i32, i32 } @add.llvm({ i32, i32 } %a_0) {
  %0 = extractvalue { i32, i32 } %a_0, 0
  %1 = extractvalue { i32, i32 } %a_0, 1
  %2 = add i32 %0, %1
  %3 = icmp ult i32 %2, %0
  %4 = icmp ult i32 %2, %1
  %5 = or i1 %3, %4
  %6 = insertvalue { i32, i32 } { i32 1, i32 undef }, i32 %2, 1
  %7 = select i1 %5, { i32, i32 } { i32 0, i32 undef }, { i32, i32 } %6
  ret { i32, i32 } %7
}
```

As well as taking care of the redundant `zext`s and `bitcast`, the optimiser has replaced our block-based handling of the COGENT `if` statement with a single `select` instruction. Not all conditional expressions can be handled in such a way, which is why we generate block-based control flow and leave it up to the LLVM optimiser to discover these optimisations for us.

This example provides a good intuition of our implementation but fails to demonstrate many of the complex cases handled by our LLVM backend. Appendix A.2 contains a detailed explanation of compilation for a more comprehensive example involving abstract functions and types, as well as the C FFI, which are omitted by our introductory example.

3.2 Coq Implementation of the LLVM Backend

The Coq implementation of the COGENT LLVM backend began as part of work to develop a separate validator for the Haskell implementation, called a *checker* (see Chapter 5.3 for more detail on this approach). It was observed that for simple programs, a checker could simply compile the program and verify the output is consistent with the provided output. Therefore, to build a checker in Coq we would start by reimplementing the compiler for an increasing subset of COGENT. Currently, the Coq implementation supports almost all features of the Haskell implementation. The decision to proceed with a compiler implementation was motivated by the promising progress of the HELIX compiler's verification, which targets LLVM using VIR (Zaliva, 2020). Since the mapping from COGENT types and expressions to LLVM has been explained in depth in the previous section, this section will focus exclusively on the design and differences in the Coq implementation of the backend.

3.2.1 Design

The Coq implementation takes as input a simplified COGENT AST and produces a VIR AST as output. The COGENT AST is based on the existing deep embedding in Isabelle/HOL. Since we do not have an `llvm-hs` equivalent for Coq we must generate the VIR AST manually. Our implementation borrows the `cerr` monad from the HELIX compiler which threads an `IRState` through our compiler (Zaliva, 2020) containing counters for blocks, locals, and void instructions, though it is much more primitive than the `IRBuilder` state of `llvm-hs`. Additionally, we maintain a list of VIR expressions (registers or constants) corresponding to in-scope variables. To compile a COGENT program, each function definition is compiled independently. Functions each correspond to a `TLE_Definition` in VIR and their bodies consist of the CFG of blocks resulting from compiling the COGENT function body to a list of VIR blocks and then finally we append a block to return the final result of the function.

Reusable VIR Codegen and Theorems from Helix

Much of the infrastructure of the FHCOL to VIR compiler developed as part of HELIX is not specific to the language and can be useful when implementing any VIR back-end. Some of these common functions, types, and lemmas have been collected as a library ('HelixLib') which assists with our implementation of the VIR backend in Coq. `cerr` and `IRState` both form part of the `Codegen.v` module which also includes helper functions to add variables, local registers, and blocks.

Destination-Passing Style

To construct a compiler where the compilation of expressions can be easily broken down into a compilation of sub-expressions independent of the block state, we use a destination-passing style (DPS) like HELIX (Zaliva, 2020). DPS requires that for each expression to compile, we provide a destination block ID to jump to after the code corresponding to the expression has been generated. The output of our expression compilation function is a VIR typed expression corresponding to the result of the expression, a block ID to enter from, and a list of blocks that compute the expression. Listing 3.35 gives the Coq definition for our expression compilation function.

Listing 3.35: Expression compilation function

```
Fixpoint compile_expr (e : expr) (next_bid : block_id)
                         : cerr (texp typ * block_id * list (block typ))
```

DPS works best when we can compile expressions in reverse order since we need to use the entry block ID of the second sub-expression as the destination block when compiling the first sub-expression. This proves to be problematic for expressions such as `Let e b` where the compilation of `b` depends on `e`. To solve this problem, intermediate *linking blocks* are created. A linking block contains no regular instructions, only a terminator that jumps immediately to another block. The notation `b1` $\rightsquigarrow$ `b2` defines a linking block with ID `b1` that immediately jumps to block ID `b2`. Therefore, to compile the `let` expression (with `next_bid` as our final destination):

1. We first reserve a block ID `let_bid`
2. Then, compile `e` using `let_bid` as the destination, yielding a result `e'`, entry `e_bid`, and code `e_blks`
3. Push `e'` to the variable context to represent `e` bound
4. Compile `b` using `next_bid` as the destination, yielding a result `b'`, entry `b_bid`, and code `b_blks`
5. Pop `e'` from the variable context
6. Create a linking block `let_bid` $\rightsquigarrow$ `b_bid`
7. The final result of this compilation should be `b'`, the entry point will be `e_bid`, and our code is the code for both sub-expressions as well as the linking block

Coq code implementing the above case of `compile_expr` is given in Listing 3.36.

Listing 3.36: Compiling `let` (`Compiler.v`)

```
| Let e b =>
  let_bid <- incBlockNamed "Let" ;;
  '(e', e_bid, e_blks) <- compile_expr e let_bid ;;
  addVars [e'] ;;
  '(b', b_bid, b_blks) <- compile_expr b next_bid ;;
  dropVars 1 ;;
  ret (b', e_bid, e_blks ++ let_bid ~> b_bid ++ b_blks)
```

In VIR the order of blocks does not matter, but the convention used in our implementation is to construct the list of blocks such that jumps to earlier blocks are avoided.

3.2.2 Extraction

Coq's extraction feature allows us to generate Haskell code corresponding to the definitions which implement the compiler (Bertot and Castéran, 2013). The extracted Haskell module contains definitions for the COGENT AST, the VIR AST, and the compilation functions. This module can be integrated into the existing COGENT compiler code as an alternative to the LLVM backend implemented directly in Haskell.

To make use of the extracted compilation function, we first need to convert the original Haskell representation for a COGENT AST to the simplified representation that was defined for the Coq implementation. The conversion is straightforward by simple recursion, traversing and converting each part of the old AST and omitting information not required in the new AST.

Likewise, to render the resulting LLVM module for the COGENT program, we take the VIR output of the extracted compiler and transform this into an `llvm-hs` AST. This conversion only needs to support the subset of VIR that the compiler produces, though many extra features of VIR are already supported in our conversion function so the compiler can be extended in the future.

Using both conversion functions, the compiler's new flag `--llvm-coq` invokes the extracted implementation to compile a COGENT program to an LLVM module.

3.2.3 Differences from the Haskell Implementation

There are a few key differences in the Coq implementation of the LLVM backend for COGENT. Currently, variants are stored less efficiently, and function calls are inlined in order to simplify verification. Furthermore, there are various features of the COGENT language that are not yet supported. The implementation supports COGENT's arithmetic (except `U16`), record, and variant operators, but does not currently support abstract functions, abstract types, `String`, unary operators, or a C FFI. Lastly, the destination-passing style results in LLVM-IR output that differs from the Haskell implementation by including unnecessary extra blocks in each function, however, this can be optimised out.

Variants

Unlike their compact union-based representation in the Haskell implementation of the backend, the Coq implementation's treatment of COGENT's variant types is similar to the existing C backend. Each variant corresponds to a structure type containing a tag field and then a field for each possible constructor rather than a tag and a single field. When building a variant, the tag is set and the corresponding field is set, with all other values left undefined.

For example, the variant `<Foo U32 | Bar U8 >` is represented as a `{ i32, i8, i32 }` (`i8` comes before `i32` due to lexicographic ordering of constructor names), rather than `{ i32, i32 }` as it would be using the Haskell implementation of the LLVM backend.

This avoids the need to port the complicated type-size calculations from the Haskell implementation that were used to determine the maximal type for a variant. It also avoids casting variant values since each field is already of the correct LLVM-IR type for that alternative. With a bit more time, it would be possible to use the more efficient representation if desired.

Function Calls

The next difference from Haskell is that COGENT function calls have been inlined in the Coq implementation to match the semantics defined for them in Section 4.2. Section 4.3 presents an extension to the semantics that would encourage us to compile our function calls to proper LLVM function calls as in the Haskell implementation. The following snapshot of the COGENT repository contains an older implementation that did not inline function calls. The current implementation translates function calls into code similar to `let` bindings.

Unsupported Features

As mentioned, abstract types are not currently supported by this compiler. This is because they are not yet supported by our denotational semantics of COGENT, nor are opaque types yet formalised by VIR. String literals are also not supported but could easily be supported in a similar manner to the Haskell implementation. COGENT's `U16` type is unsupported as the VIR formalisation does not currently support `i16`, but all other integer types are still supported. Unary operators are unsupported but could be supported using the same implementation as the Haskell version. Lastly, since abstract functions are not supported, wrapper functions for the C FFI are not generated by the Coq implementation of the compiler.

Code Style

One minor difference in code style is that the generated register names differ from the Haskell implementation, due to the way the `Codegen.v` functions generate fresh names. This behaviour could be adjusted to match register names from Haskell if required.

The main difference in the style of the LLVM-IR, alluded to previously, is the addition of many redundant linking blocks due to the DPS style of code generation. For example, the simple COGENT adder program from Listing 3.31 produces the following LLVM-IR after compilation:

Listing 3.37: Coq implementation output (`adder.ll`)

```
define { i32, i8, i32 } @add({ i32, i32 } %a0) {
Field2:
  %l0 = extractvalue { i32, i32 } %a0, 0
  br label %Take1
Take1:
  br label %Field4
Field4:
  %l1 = extractvalue { i32, i32 } %a0, 1
  br label %Take3
Take3:
  br label %Prim6
Prim6:
  %l2 = add i32 %l0, %l1
  br label %Let5
Let5:
  br label %Prim8
Prim8:
  %l3 = icmp ult i32 %l2, %l0
  br label %Let7
Let7:
  br label %Prim10
Prim10:
  %l4 = icmp ult i32 %l2, %l1
  br label %Let9
Let9:
  br label %Prim12
Prim12:
  %l5 = or i1 %l3, %l4
  br label %Let11
Let11:
  br label %If13
If13:
  br i1 %l5, label %Let15, label %Con20
Let15:
  br label %Con17
Con17:
  %l6 = insertvalue { i32, i8, i32 } undef, i32 1, 0
  %l7 = insertvalue { i32, i8, i32 } %l6, i8 0, 1
  br label %Let16
Let16:
  br label %Then_Post14
Con20:
  %l8 = insertvalue { i32, i8, i32 } undef, i32 2, 0
  %l9 = insertvalue { i32, i8, i32 } %l8, i32 %l2, 2
  br label %Let19
Let19:
  br label %Else_Post18
Then_Post14:
  br label %If_Post21
Else_Post18:
  br label %If_Post21
If_Post21:
  %l10 = phi { i32, i8, i32 } [ %l7, %Then_Post14 ], [ %l9, %Else_Post18 ]
  br label %Return0
Return0:
  ret { i32, i8, i32 } %l10
}
```

As we can see, in Listing 3.37 there are many instances of `br` instructions terminating a block that lead directly onto the next block. If that next block has no other predecessors, the two blocks can be fused via *block fusion*. Block fusion does not affect the semantics of the LLVM-IR code but can reduce its length. If we apply block fusion to this LLVM-IR code, the result is much shorter.

Listing 3.38: `adder.ll` after block fusion

```
define { i32, i8, i32 } @add({ i32, i32 } %a0) {
Field2:
  %l0 = extractvalue { i32, i32 } %a0, 0
  %l1 = extractvalue { i32, i32 } %a0, 1
  %l2 = add i32 %l0, %l1
  %l3 = icmp ult i32 %l2, %l0
  %l4 = icmp ult i32 %l2, %l1
  %l5 = or i1 %l3, %l4
  br i1 %l5, label %Let15, label %Con20
Let15:
  %l6 = insertvalue { i32, i8, i32 } undef, i32 1, 0
  %l7 = insertvalue { i32, i8, i32 } %l6, i8 0, 1
  br label %If_Post21
Con20:
  %l8 = insertvalue { i32, i8, i32 } undef, i32 2, 0
  %l9 = insertvalue { i32, i8, i32 } %l8, i32 %l2, 2
  br label %If_Post21
If_Post21:
  %l10 = phi { i32, i8, i32 } [ %l7, %Let15 ], [ %l9, %Con20 ]
  ret { i32, i8, i32 } %l10
}
```

The LLVM optimiser will happily perform block fusion for us as part of more general CFG optimisations. Comparing Listing 3.38 to the output of the Haskell implementation in Listing 3.33, as expected the generated IR differs in three aspects only: the naming of blocks/registers, our representation of booleans, and our representation of variants.

3.2.4 Comparison with the HELIX Compiler

The main source of inspiration for the current Coq implementation of the COGENT compiler is the FHCOL to LLVM-IR compiler. FHCOL consists of various operators (instructions), including assignment, mapping, binary operators, loops, and sequential composition (Zaliva, 2020). These operators manipulate various expression types: integers (`NExpr`), pointers (`PExpr`), memory blocks (`MExpr`), and scalars (`AExpr`).

Our treatment of COGENT expressions is a hybrid approach between the compilation of NExprs and the compilation of operators. In the HELIX compiler, `NExpr`s are translated to a list of VIR instructions and a result. A list of instructions is used rather than a list of blocks since `NExpr`s do not require code that uses branching. Furthermore, the HELIX compiler produces a list of blocks when compiling an FHCOL operator, but for these, it does not produce a result as FHCOL operators have side effects only. For COGENT expressions we must output blocks for expressiveness, as well as a result in order to implement the functional value semantics of the language. Our top-level function compilation is done similarly to how a top-level FHCOL operator is compiled to LLVM-IR.

Our compilation of each COGENT expression differs drastically from the HELIX compiler. COGENT does not support the imperative (loop, alloc, seq, etc.) operators of FHCOL, our only similar pieces of syntax are at the expression level for arithmetic expressions and such. Our linking blocks are unnecessary when compiling FHCOL expressions since they result in instructions rather than whole blocks. We cannot compile COGENT expressions to a list of instructions since there is no way to implement `if` and `case` expressions without branching, requiring us to use linking blocks to maintain a DPS style. Meanwhile, at the FHCOL operator level, sub-expressions/operators can be compiled in reverse order without forward dependencies, for example, FHCOL's sequential composition `DSHSeq f g` differs from our treatment of `Let e b` expressions since the compilation of `g` has no dependency on the result of `f`.

3.3 Future Compiler Work

A direct next step in terms of compiler development is to extend the Coq implementation to cover all of the features supported by the Haskell implementation. It is cumbersome to maintain two compiler implementations, so it would make sense to prioritise development on the Coq implementation going forward.

There is also work to be done in terms of packaging the common code generation helpers from HELIX and extending this into a reusable `llvm-hs`-like VIR compiler framework for Coq. The higher level of abstraction, perhaps via custom `Notation` and similar monads to `llvm-hs`, would reduce boilerplate when generating VIR/LLVM-IR, and would ease implementing an LLVM backend in Coq.

Another area to be explored is the C FFI support offered by the LLVM backend. The approach used in the Haskell implementation only supports `x86-64` so far and the approach may not even be sustainable for more complicated register marshalling rules. It should be easy to add support for `x86-32` to the current implementation but supporting `armv7` will require more changes. Alternatively, it may be possible to define a custom ABI for use in Clang to forbid it from modifying C function signatures during LLVM-IR compilation, so that we do not have to support each architecture's ABI. A different solution for the C FFI, prototyped by Shang (Shang, 2020), would be to define our own interface for calling COGENT code from C and vice versa, such as by passing all aggregate arguments and return types as pointers.

Additionally, some quality-of-life features could be added to the LLVM backend for COGENT, such as LLVM-IR annotations (via metadata) that could enable source-debugging or more optimisations. Also, antiquoted C, which allows embedding COGENT syntax inside a C source file, is not yet supported by the LLVM backend's C FFI, it would be worthwhile integrating this feature to improve the experience for integrating a COGENT-LLVM program with handwritten C code. Lastly, there is now a myriad of COGENT language extensions that could be supported by the LLVM backend as well, for example native array types, Dargent for describing data layouts (O'Connor et al., 2018, Kerr, 2020), or recursive types (Murray, 2019, Qiu, 2020) and functions.

Chapter 4

Formalisation

In this chapter, we introduce a denotational semantics for COGENT using the Interaction Trees (ITrees) library for the Coq proof assistant. Firstly, the necessary background for ITrees will be introduced, before proceeding to outline an ITree-based denotational semantics of COGENT. Lastly, VIR, an ITree semantics for LLVM, will be presented as this is a crucial part of the verification strategy for COGENT's LLVM backend.

4.1 Introduction to Interaction Trees

ITrees are a co-inductive data structure to represent program behaviours via *events* and their continuations (Xia et al., 2020). The events of an ITree can be interpreted by *handlers* to give the semantics for each type of event. ITrees may be related to each other and are considered equivalent by weak bisimulation. The ITree library for Coq provides an interface to easily construct ITrees, and allows clients to reason about ITrees without dealing with the co-inductive implementation.

An ITree consists of three types of nodes: `Ret r` nodes are terminating computations of a particular value `r`, `Tau t` nodes are internal computation steps with successor t, and Vis e k models an external computation consisting of a visible effect e and the continuation k that receives the answer of type A following the external event (Xia et al., 2020). Listing 4.1 defines ITrees as a co-inductive type in Coq.

Listing 4.1: ITree definition in Coq (Xia et al., 2020)

```
CoInductive itree (E : Type → Type) (R : Type) : Type :=
| Ret (r : R)
| Tau (t : itree E R)
| Vis {A : Type} (e : E A) (k : A → itree E R).
```

The ITree library provides an implementation of bind (and its x <- e ;; k notation) and ret for the data structure, making it possible to construct ITrees using do-notation. Additionally, the trigger function provides a clean way to dispatch an external event and yield the result (Xia et al., 2020).

Events are inductive types indexed by the answer type for each of their various *interactions*. Each constructor for an event has arguments corresponding to the data sent externally for that interaction. For example, an IO event type could consist of two interactions, Input and Output. A Coq definition for this event is given in Listing 4.2.

Listing 4.2: Example event definition (Xia et al., 2020)

```
Inductive IO : Type → Type :=
| Input : IO nat
| Output : nat → IO unit.
```

For this event, the Input interaction requires no data but provides a nat answer, and the Output interaction accepts a nat yet provides a useless (unit) answer.

Handlers for a single event E are given the notation E ~> X which corresponds to the type forall X, E X -> M X, mapping interactions in E to computations in some monad M. The library provides an interp function which takes an E handler and applies it to any E events in a given ITree (Xia et al., 2020). The ITree library also formalises an algebra of events, defining E +' F as the disjoint union of events E and F. We can combine handlers for single events into a handler for the union using the case_ helper.

One included event type from the ITree library that we will make use of is the `exceptE` event with a single `Throw` interaction which can be used to throw exceptions parametrised by a given type. The `throw` function produces a visible `Throw` action specialised to any return type.

4.2 Interaction Tree Update Semantics for Cogent

The motivation behind defining a denotational semantics for COGENT using ITrees is that it can be easily related to VIR's ITree semantics (Zakowski et al., 2021). The goal of our new semantics is to closely match the existing update semantics for COGENT, whilst also making use of ITree events and handlers.

This section first explains the approach that was taken to develop the denotational semantics by referring to the existing update semantics. Next, we proceed to define the various components that comprise the ITree semantics for COGENT, starting with values and events, the denotation function, and finally handlers and interpreters for the events.

4.2.1 Methodology

COGENT's big-step update semantics, which was briefly introduced in Section 2.1.2, is represented as a relation `u_sem` formalised in Isabelle/HOL (O'Connor et al., 2016). Each rule for this relation handles an inductive or base case for COGENT's expression syntax. These rules were directly used to create a case in our denotation function for the ITree semantics.

Our denotation function returns an ITree for the universe of COGENT events. We use this universe of events to trigger memory interactions when required rather than by explicitly passing around a mutable store. Thus, we separate the pure aspects of COGENT's semantics from the effectful interpretation of the memory interactions. The monadic structure of ITrees has been used to implement the denotation function as a function `denote_expr` which binds ITrees for a sub-expression and returns an appropriate ITree that denotes the entire COGENT expression.

Example: From a big-step semantics to a denotational semantics

The process of transforming a case of the semantic relation to a case for the denotation function is best illustrated via an example. Recall the `u_sem_let` rule which defines the big-step update semantics for COGENT `let` expressions:

$$\frac{\gamma \vdash (\sigma, a) \Downarrow (\sigma', a') \qquad (a' :: \gamma) \vdash (\sigma', b) \Downarrow st}{\gamma \vdash (\sigma, \texttt{Let}\ a\ b) \Downarrow st}$$

There are three key things to notice in this example. Firstly, the 'flow' of memory from σ to σ' dictates the order in which the sub-expressions of `Let` a b should be executed. Since the right premise requires the final memory state of the left premise, we must evaluate a before b. Second, the context for evaluating b depends on the output from the first premise. Finally, note that the output st of the conclusion is equal to the output from the right premise.

This corresponds to the following case of the `denote_expr` which gives a denotation for COGENT expressions:

```
| Let a b ⇒
    a' <- denote_expr γ a ;;
    denote_expr (a' :: γ) b
```

Based on the inference rule, we first denote the sub-expression `a`. Then, the sub-expression `b` is denoted using the updated context `(a' :: γ)`. Finally, the result of denoting this expression is implicitly returned. There is no explicit passing of updated memory states like σ, instead our denotation function returns an ITree containing `Vis` nodes for memory events that are hidden via the ITree monad. The `let` syntax doesn't trigger any memory events, but its sub-expressions may, and their resulting ITrees are bound together in this case without having to manually thread the memory state ourselves.

4.2.2 Semantics

Now we are well equipped to cover the complete ITree semantics for COGENT. Our semantics consists of the following components: a definition of COGENT's values, the external events we will use, the denotation of COGENT syntax into an ITree, and finally an interpretation of COGENT ITrees via handlers for each event type.

Values

This representation of COGENT values in Listing 4.3 is based on the existing Isabelle/HOL update semantics values (O'Connor, 2019), however, it currently omits abstract functions and types, and for address types, the VIR memory model is used (Zakowski et al., 2021).

Listing 4.3: Update values for COGENT (`Denotation.v`)

```
Inductive uval : Set :=
| UPrim (l : lit)
| URecord (us : list (uval * repr))
| USum (t : name) (u : uval) (rs : list (name * repr))
| UUnit
| UPtr (a : addr) (r : repr).
```

This inductive type represents every possible result of a fully evaluated COGENT expression. `repr` is a representation type identical to the existing Isabelle/HOL formalisation which abstractly describes the memory representation for aggregates and pointers.

In Listing 4.4 the variable context for COGENT is represented via a Coq list of values.

Listing 4.4: Variable context representation (`Denotation.v`)

```
Definition ctx : Type := list uval.
```

Events

The main difference between this new ITree semantics of COGENT and the existing Isabelle/HOL update semantics is that memory interactions are dispatched via ITree events inside our denotation function rather than by explicitly passing a memory store in the semantics relation. Our memory event `MemE` defined in Listing 4.5 consists of two interactions, loading and storing from memory, which both require an address parameter. The `LoadMem` interaction returns an optional value depending on what is contained at that address, and the `StoreMem` takes a value to store at the provided address, but yields only a `unit` result, meaning we assume all writes to memory are successful.

Listing 4.5: COGENT memory event (`Denotation.v`)

```
Variant MemE : Type → Type :=
| LoadMem (a : addr) : MemE (option uval)
| StoreMem (a : addr) (u : uval) : MemE unit.
```

Since our denotation function must be total, we also require a `FailE` to throw in absurd cases, such as if we were to denote a program that is not type-correct. In Listing 4.6 we use the included `exceptE` event from the ITree library to throw error messages in such cases.

Listing 4.6: COGENT failure event (`Denotation.v`)

```
Definition FailE := exceptE string.
```

Additionally, we introduce a notion of interpretation levels for COGENT at which different event families have been interpreted away. As shown in Listing 4.7, at level 0 both memory and failure events are present, but at level 1 only failure events remain. If more events are added to COGENT's ITree semantics, we can add further interpretation levels as necessary.

Listing 4.7: COGENT interpretation levels (`Denotation.v`)

```
Definition CogentL0 := MemE +' FailE.

Definition CogentL1 := FailE.
```

Denotation

The expression level denotation function for COGENT is of the form:

Listing 4.8: Denotation of COGENT expressions (`Denotation.v`)

```
Fixpoint denote_expr (γ : ctx) (e : expr) {struct e}
                    : itree CogentL0 uval :=
    match e with
    (* cases *)
    end.
```

The type signature of this denotation function means that for a given context and a COGENT expression, we construct an ITree containing `CogentL0` events that results in a `uval`.

The simple cases (`Unit` and `Lit`) are denoted by ITrees immediately returning the corresponding `uval`.

```
| Unit  ⇒ ret UUnit
| Lit l ⇒ ret (UPrim l)
| Var i ⇒ match nth_error γ i with
          | Some v ⇒ ret v
          | None ⇒ throw "unknown variable"
          end
```

`Var` is complicated by its use of the `nth_error` function which will return `None` if the requested variable is not in the context `γ` - in this case, we trigger a `FailE` event using the `throw` function, otherwise, we `ret` the retrieved value.

As seen in Section 4.2.1, to denote a `Let a b` expression the binding `a` must be denoted before the body `b` can be denoted with the new binding added to the variable context.

```
| Let a b ⇒
    a' <- denote_expr γ a ;;
    denote_expr (a' :: γ) b
```

Here, the `a <- f ;; b` notation has been used to bind the ITree for `a` to the ITree for `b`, resulting in an ITree to denote the whole `Let a b` expression.

We can denote primitive operations (such as arithmetic expressions) in a similar way. The arguments to the expression are denoted first and then the operator is denoted:

```
| Prim p xs ⇒
   xs' <- map_monad (denote_expr γ) xs ;;
   denote_prim p xs'
```

In the above implementation, map_monad is used to denote each argument in the list xs and bind the resulting ITrees together in sequence. denote_prim is a helper function to take a COGENT operator and a list of uvals and perform a concrete arithmetic or logical operation on those values. For example, the Plus operator corresponds to an addition operation modelled for the desired number of bits.

For If x t e expressions we must denote the condition x before then denoting either the t or e sub-expression depending on the result from the ITree of x:

```
| If x t e ⇒
   x' <- denote_expr γ x ;;
   match x' with
   | UPrim (LBool b) ⇒ denote_expr γ (if b then t else e)
   | _ ⇒ throw "expression is not a boolean"
   end
```

We resolve the absurd case, where the result from the ITree of x is not boolean, by triggering a FailE event via throw.

Cast τ e expressions are denoted by first denoting an ITree for e, and then casting the value that results to type τ.

```
| Cast τ e ⇒
   e' <- denote_expr γ e ;;
   match e' with
   | UPrim l ⇒ option_throw (cast_to τ l) UPrim "invalid cast"
   | _ ⇒ throw "invalid cast"
   end
```

We make use of a few helper functions here: option_throw x f m attempts to apply a function f to an optional value x and throws the message m if x is None. In the case of casting, cast_to τ l is a function that returns None if and only if the cast is invalid, and otherwise, a casted literal, which we wrap inside UPrim.

Struct literals correspond to a URecord containing a uval for each field.

```
| Struct ts xs ⇒
   vs <- map_monad (denote_expr γ) xs ;;
   ret (URecord (combine vs (map type_repr ts)))
```

Each field in xs is denoted and the resulting ITrees are bound together, as with Prim. Finally, we build a URecord containing the list of uvals and their type representations.

For memory-based expressions, we handle both unboxed and boxed representations within the same case. The Member and Take operations have similar denotations that make use of a denote_member helper that returns a uval for the record and a uval for the field.

```
| Member e f ⇒ snd <$> denote_member γ e f
| Take x f e ⇒
   '(r, m) <- denote_member γ x f ;;
   denote_expr (m :: r :: γ) e
```

For Member we only care about the second uval, which is the requested field. However, for Take x f e the sub-expression e must be denoted in a new context where the uval corresponding to the record and member are now bound. Now we can look to the implementation of the denote_member function in Listing 4.9.

Listing 4.9: Helper for member denotation (Denotation.v)

```
let denote_member (γ : ctx) (e : expr) (f : nat)
                : itree CogentL0 (uval * uval) :=
   r <- denote_expr γ e ;;
   m <- match r with
   | URecord fs ⇒ access_member fs f
   | UPtr p r ⇒
       m <- trigger (LoadMem p) ;;
       match m with
       | Some (URecord fs) ⇒ access_member fs f
       | _ ⇒ throw "invalid memory access"
       end
   | _ ⇒ throw "expression is not a record"
   end ;;
   ret (r, m)
```

Like the expression denotation function, this denotation function returns a CogentL0

ITree, but its result type is a tuple of uval, one for the record and one for the member. First, the record is denoted, and then depending on the form of the result, either unboxed or boxed semantics are used. In the unboxed URecord fs case, we can simply access the member from the list of fields fs. In the boxed UPtr p r case we first trigger a memory event, the LoadMem interaction, to retrieve data at the address p. If the result is a URecord fs we can access the member like in the unboxed case, otherwise we trigger an exception. Finally, the pair of uvals can be returned from the computation.

In Listing 4.10 we provide the denotation for Put x f e which has a similar structure to denote_member, dealing with both unboxed and boxed cases.

Listing 4.10: Denotation of COGENT expressions (Denotation.v)

```
| Put x f e ⇒
    x' <- denote_expr γ x ;;
    e' <- denote_expr γ e ;;
    match x' with
    | URecord fs ⇒
        rep <- option_throw (nth_error fs f) snd "invalid member access" ;;
        ret (URecord (list_upd fs f (e', rep)))
    | UPtr p r ⇒
        m <- trigger (LoadMem p) ;;
        match m with
        | Some (URecord fs) ⇒
            rep <- option_throw (nth_error fs f) snd "invalid member access" ;;
            trigger (StoreMem p (URecord (list_upd fs f (e', rep)))) ;;
            ret (UPtr p r)
        | _ ⇒ throw "invalid memory access"
        end
    | _ ⇒ throw "expression is not a record"
    end
```

Before insertion, we need to denote both the record x and e, the value to insert. In the unboxed URecord fs case, insertion is done via using the list_upd function on fs with the new uval for e. In the boxed case, the record must first be loaded with a LoadMem memory interaction, and then after the field is updated, it is stored by triggering a StoreMem interaction using the new uval, then the pointer uval is returned.

Constructing a variant literal is denoted as follows:

```
| Con ts t x ⇒
    x' <- denote_expr γ x ;;
    ret (USum t x' (map (fun '(n, t, _) ⇒ (n, type_repr t)) ts))
```

The argument `x` is denoted and then some massaging is done to include the correct type representation list for the variant `uval`.

To promote a variant, we do not need to do anything at all, as it is a no-op in the dynamic semantics of COGENT.

```
| Promote t e ⇒ denote_expr γ e
```

`Esac` and `Case` have a similar structure, both will denote the sub-expression `x` and proceed only if it is a variant `uval`.

```
| Esac _ x ⇒
    x' <- denote_expr γ x ;;
    match x' with
    | USum t v rs ⇒ ret v
    | _ ⇒ throw "expression is not a variant"
    end
| Case _ x t m n ⇒
    x' <- denote_expr γ x ;;
    match x' with
    | USum t' v rs ⇒
        if t =? t'
          then denote_expr (v :: γ) m
          else denote_expr (x' :: γ) n
    | _ ⇒ throw "expression is not a variant"
    end
```

For `Esac` if the value is a variant `USum t v rs` its inhabitant `v` can be returned directly. With `Case`, the sub-expression `m` is denoted if the tag matches and `n` is instead denoted if the tag does not match the variant. In both cases, the context is augmented to bind the inhabitant or the entire variant respectively.

Lastly, function application denotation is done similarly to that of `let` bindings. For simplicity, we handle the case where a function value is applied directly rather than adding a function `uval` to wrap un-applied functions inside.

```
| App (Fun b) a ⇒
    a' <- denote_expr γ a ;;
    denote_expr [a'] b
| Fun _ ⇒ throw "naked function"
| App _ _ ⇒ throw "expression is not a function"
```

The denotation for `App (Fun b) a` differs from `Let a b` as instead of adding the result from the ITree denoting `a` to the current context, a new context containing only the argument's `uval` is used to evaluate the function body `b`. The remaining absurd cases are handled as done previously, using `throw`.

To use this expression denotation function to denote an entire COGENT program, in Listing 4.11 we define types for function and program denotation:

Listing 4.11: COGENT program denotation (`Denotation.v`)

```
Definition function_denotation := uval → itree CogentL0 uval.

Definition denote_function (b : expr) : function_denotation :=
    fun a ⇒ denote_expr [a] b.

Definition program_denotation := alist name function_denotation.

Definition denote_program (p : cogent_prog) : program_denotation :=
    map (fun '(FunDef n t rt b) ⇒ (n, denote_function b)) p.
```

By representing function denotations as functions returning an ITree - these are known as KTrees (Xia et al., 2020), we can denote a function via a lambda that creates a context containing the provided argument for denoting the function body. Therefore, a program's denotation can be represented by a mapping (implemented as an `alist`) from each top-level function name to its denotation.

Handlers and Interpretation

Now that we have defined a denotation function for COGENT programs into an ITree containing uninterpreted `MemE` and `FailE` events, to complete the definition of the semantics we must provide handlers and interpreters for these events.

We can interpret memory events by modelling memory as an associative map from addresses to `uval`s in Listing 4.12:

Listing 4.12: Concrete implementation of memory (`Denotation.v`)

```
    Definition memory := alist addr uval.
    Definition empty_memory : memory := empty.
```

In Listing 4.13 our handler for `MemE` events maps load and store interactions to operations in the `stateT` monad (wrapping an ITree), using `memory` as our state.

Listing 4.13: Memory handler (`Denotation.v`)

```
Definition handle_mem {E} : MemE ~> stateT memory (itree E) :=
    fun _ e σ ⇒
        match e with
        | LoadMem a ⇒ ret (σ, alist_find a σ)
        | StoreMem a u ⇒ ret (alist_add a u σ, tt)
        end.
```

If the event `e` is a `LoadMem a` interaction, the state `σ` is left unchanged, and the answer we provide is the result of searching for the address `a`. Otherwise, for a `StoreMem a u` interaction, the state is replaced by the result of adding the value `u` at address `a`, and the result is the unit literal `tt`.

We can use several helper functions provided by the ITree library to construct an interpreter in Listing 4.14 from `CogentL0` to `CogentL1` that handles `MemE` events and leaves `FailE` events uninterpreted.

Listing 4.14: Memory interpeter (`Denotation.v`)

```
Definition interp_mem : itree CogentL0 ~> stateT memory (itree CogentL1) :=
    interp_state (case_ handle_mem pure_state).
```

By separating the pure denotational semantics from the effectful interpretation of ITree events, we are free to replace our memory model in the future without adjusting the denotational semantics. Next in Listing 4.15, to handle `FailE` events, we can use the `failT` monad to wrap an ITree.

Listing 4.15: Failure handler (`Denotation.v`)

```
Definition handle_failure : FailE ~> failT (itree void1) :=
    fun _ '(Throw m) ⇒ ret None.
```

Whenever we see a `Throw m` event we can simply 'terminate' the entire computation via `None`. The ITree library provides `interp_fail` for us, which takes in a handler and create an interpreter for our failure events. Putting both interpretation layers together in Listing 4.16, we can interpret all events in a `CogentL0` ITree.

Listing 4.16: Combined interpreter (`Denotation.v`)

```
Definition interp_expr {E T} (l0 : itree CogentL0 T) (mem : memory)
                           : failT (itree E) (memory * T) :=
    let l1 := interp_mem l0 mem in
    let l2 := interp_fail handle_failure l1 in
    translate inject_signature l2.
```

The final line allows this interpreter to return a result in an arbitrary event universe `E` using the `inject_signature` function from HELIX.

To interpret an entire COGENT program, we need to specify the entry function and argument to interpret with.

Listing 4.17: Program interpreter (`Denotation.v`)

```
Definition interp_cogent {E} (p : program_denotation) (fn : name)
                             (a : uval) (mem : memory)
                           : failT (itree E) (memory * uval) :=
    match alist_find fn p with
    | Some f ⇒ interp_expr (f a) mem
    | None ⇒ ret None
    end.
```

Listing 4.17 gives the top-level interpretation of a COGENT program denotation. If the semantics are defined, the result will be `Some (m, r)`, a pair of the final memory state `m` and the function's return value `r`, otherwise, the result will be `None`.

Lastly, we define a top-level runner function for COGENT in Listing 4.18. `run_cogent` combines both the denotation of a program and the interpretation of the resulting ITree starting from a particular memory state.

Listing 4.18: Program runner (`Denotation.v`)

```
Definition run_cogent (p : cogent_prog) (fn : name) (a : uval)
                    : failT (itree void1) (memory * uval) :=
    interp_cogent (denote_program p) fn a empty_memory.
```

4.2.3 An Interpreter for Free

The ITrees library functions are executable via code extraction (Xia et al., 2020), meaning that we can extract the Coq definitions for the ITree semantics for COGENT to Haskell code similar to how the compiler implementation was extracted in Section 3.2.2. This allows us to develop an interpreter in Haskell for COGENT using the semantics developed in the previous section with very little additional effort.

After we have extracted the interpreter code to Haskell, the resulting Haskell module contains the `run_cogent` function from Listing 4.18. To reach the result we need to step through the ITree, ignoring `Tau` transitions, until a `Ret r` is reached. This will contain the result `r` which is a tuple containing the expression's value and final memory state of the program after interpretation. If during our steps we encounter a `Vis` node, which represents an uninterpreted event, we can abort the interpreter since this should not happen. In the case of the current COGENT semantics, all events will already have been dealt with via our memory and failure handlers so this case will not occur in practice. However, it could be valuable to defer the interpretation of some new events, such as debugging or FFI calls, so they can be concretely interpreted via Haskell code instead. Listing 4.19 defines `step`, based upon the OCaml ITree interpreter from VIR (Zakowski et al., 2021) which implements the ITree evaluation sequence described above.

Listing 4.19: Stepping through an ITree (`WithCoq.hs`)

```
step :: Itree () (Maybe t) → Maybe t
step t = case observe t of
    TauF t → step t
    RetF r → r
    VisF e k → error "uninterpreted call"
```

Our Haskell driver code also defines `interpWithCoq` which converts the Haskell COGENT AST to the extracted AST before passing it to `run_cogent`, and finally `step`s through it to produce a result.

Listing 4.20: Top-level interpreter (WithCoq.hs)

```haskell
interpWithCoq :: [C.Definition TypedExpr VarName VarName] → IO ()
interpWithCoq monoed =
    case step (run_cogent (convCogentAST monoed) "main" UUnit) of
        Just (mem, res) → do
            putStrLn ("Memory: " ++ show mem)
            putStrLn ("Result: " ++ show res)
        Nothing → hPutStrLn stderr "Invalid semantics"
```

The implementation in Listing 4.20 starts from an empty memory state, execution entry is via the `main` function, and its parameter is `()`, but there is no technical limitation preventing us from allowing the user to specify these three things. Our new compiler flag `--coq-interp` performs this ITree interpretation for the `main` function of a specified COGENT program.

4.3 Future Formalisation Work

In this section, we highlight two interesting future areas of work that build upon the ITree semantics for COGENT: a refinement proof between our denotational ITree semantics and the big-step update semantics, and ITree semantics for recursive and abstract functions.

Relation to Existing Semantics

For us to achieve end-to-end assurance for any proofs involving the new ITree semantics for COGENT, we would have to prove that the new semantics are a refinement of the existing update semantics. Such a proof would involve showing that for any COGENT program that evaluates in the ITree semantics, the same COGENT program evaluates to the same result and memory state using the update semantics.

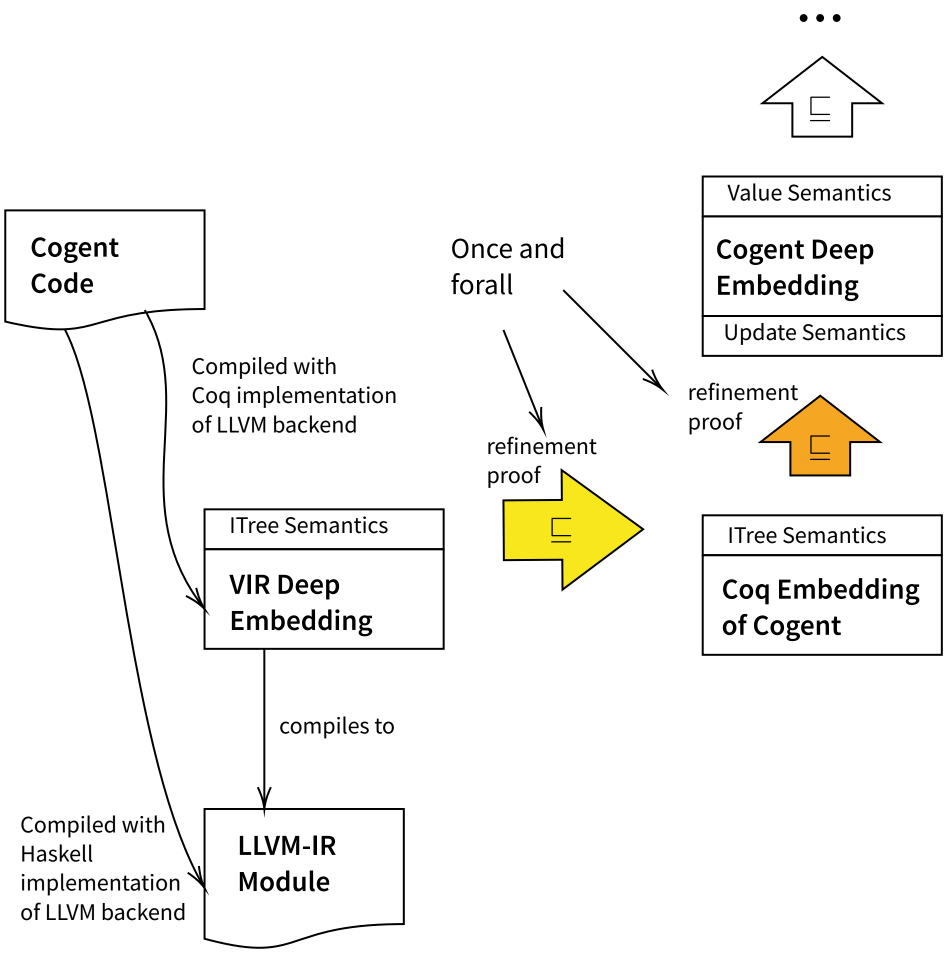

Figure 4.1: COGENT to LLVM-IR end-to-end verification

This refinement proof is currently difficult as the update semantics of COGENT is formalised in Isabelle/HOL and the ITree semantics is formalised in Coq. To proceed with the proof, either semantics would need to be ported to the other proof assistant, neither being a trivial task. There are also a few differences between the COGENT deep embedding in Isabelle/HOL and the deep embedding implemented in Coq for this project, namely:

- A different treatment of functions and addresses
- The representation of literals as arbitrary-length integers
- Differing implementations for the n-bit operations on these integers

- Inclusion of variant type information for `Esac` and `Case`

The Coq deep embedding of COGENT will likely need to be adjusted to match the Isabelle/HOL deep embedding for this refinement proof to go smoothly. This proof corresponds to the orange refinement step in Figure 4.1 below. Chapter 5 introduces our initial work for the yellow refinement proof between the ITree semantics of VIR (see Section 4.4) and COGENT.

Semantics for Recursion and Abstract Functions

It is possible to extend COGENT's ITree semantics to allow for (mutually) recursive function calls, as well as external calls to abstract functions, by adding new events and handlers. These events were not integrated with the current formalisation as they complicate verification but are included here as an example of how ITree constructs can be used to expand our new semantics for COGENT.

First, functions and abstract functions can be given `uval` representations, like in the Isabelle/HOL update semantics.

Listing 4.21: Function values

```
Inductive uval : Set :=
  ...
  | UFunction (fn : name)
  | UAFunction (fn : name).
```

Our new function call event would be defined as in Listing 4.22.

Listing 4.22: Function call events

```
Variant CallE : Type → Type :=
| Call (f : uval) (a : uval) : CallE uval.
| AbsCall (f : uval) (a : uval) : CallE uval.
```

The two interactions represent internal and external function calls separately. Both accept a function value `f`, an argument value `a`, and yield a result.

A handler for function calls can lookup functions in a program by their name and call the corresponding ITree function, and abstract functions could be looked up similarly. An example Coq definition for this handler is given in Listing 4.23.

Listing 4.23: Handling function calls

```
Definition program_denotation := alist name function_denotation.
Definition abstract_functions := alist name function_denotation.

Definition handle_call (p : program_denotation) (abs : abstract_functions)
                    : CallE ~> itree CogentE :=
    fun _ e ⇒
        match e with
        | Call (UFunction fn) a ⇒
            match alist_find fn p with
            | Some f' ⇒ f' a
            | None ⇒ throw ("unknown function " ++ fn)
            end
        | AbsCall (UAFunction fn) a ⇒
            match alist_find fn abs with
            | Some f' ⇒ f' a
            | None ⇒ throw ("unknown abstract function " ++ fn)
            end
        | _ ⇒ throw "expression is not a function"
        end.
```

Since function calls may be recursive, in Listing 4.24 we use the `mrec` combinator provided by the ITrees library (Xia et al., 2020) to apply the above handler to an entire program denotation, starting from a specified entry function.

Listing 4.24: Interpreter for programs with function calls

```
Definition interp_call (p : program_denotation) (abs : abstract_functions)
                       (entry_f : uval) (entry_a : uval) : itree CogentL1 uval :=
    mrec (handle_call p abs) (Call entry_f entry_a).
```

To implement `abstract_functions`, an ITree denotation function would be required for each abstract function. It is quite easy to model a pure COGENT function as an ITree using the code of Listing 4.25.

Listing 4.25: Defining abstract functions using Coq functions

```
Definition cogent_fun : Type := uval → uval.

Definition model_cogent_fun (cf : cogent_fun) : function_denotation :=
    fun a ⇒ ret (cf a).

Definition abstract_id := model_cogent_fun (fun a ⇒ a).
```

In this example, we model the identity function, but any `uval -> uval` function that is implementable inside Coq would work here. Hence there is the possibility to model abstract functions via shallowly embedded Coq functions.

It may also be desirable to model abstract functions directly via ITree implementations. For example, to model an abstract function that interacts with memory, or to allow abstract functions to call back into COGENT functions or other abstract functions. This level of expressiveness can be achieved by implementing these computations directly as ITrees. For example, in Listing 4.26 we model a function that calls back into a COGENT function called `do` with a decremented `U32` argument.

Listing 4.26: Defining abstract functions using ITrees

```
Definition abstract_dec : function_denotation :=
    fun a ⇒
        match a with
        | UPrim (LU32 x) ⇒
            trigger (Call (UFunction "do") (UPrim (LU32 (x - 1))))
        | _ ⇒ throw "type error"
        end.
```

The existing definitions make writing these functions cumbersome, but it would be possible to include various operators and functions that make writing `uval` functions easier.

A final idea is that we could even defer the handling of abstract function calls until inside the extracted interpreter. This would require representing `Call` and `AbsCall` as different events, with `Call` events interpreted but `AbsCall` events left unhandled in the final interpretation layer. This would allow abstract functions to be implemented in Haskell rather than in Coq. However, it is unclear how a correctness proof would proceed with uninterpreted effects remaining in the COGENT ITree.

4.4 VIR: Interaction Tree Semantics for LLVM

VIR formalises a substantial subset of the LLVM IR using ITrees and event handlers (Zakowski et al., 2021).

Dynamic Values

VIR's semantics manipulates a set of dynamic values, those of interest to us are 1-, 8-, 32-, and 64-bit integers (16-bit integers are unsupported). Memory addresses are represented by an abstract type, and structure types are also given dynamic values in VIR (Zakowski et al., 2021).

Events

VIR programs trigger a variety of events (Zakowski et al., 2021):

- Global state events for reading and writing to global variables
- Local state events for reading and writing to local variables
- Local stack events for handling environments during function calls
- Memory events including frame manipulation, loads and stores, allocation, pointer computation, and pointer casts
- Internal function call events for calling to other VIR functions
- External call events for calling into to non-VIR code
- Intrinsic call events to use LLVM Intrinsic functions
- Non-determinism events to model non-deterministic features of LLVM-IR
- Undefined behaviour events to model LLVM-IR's undefined behaviour
- Failure events for dynamic errors

Program Denotation

VIR programs are represented as ITrees via various levels of denotation functions (Zakowski et al., 2021). The top-level denotation function transforms an MCFG into an ITree containing VIR events. The denotation of expressions may trigger global or local read events when global and local variables are mentioned. Simple instructions are denoted based on their operation and then a local write is triggered to store the result. Memory instructions will trigger additional memory or undefined behaviour events. Calling a function will trigger a function call event or an intrinsic call event depending on the identity of the function called. The denotation of a terminator involves returning either a value or a block label to jump to. A list of blocks is considered an open CFG, which can be denoted by providing the entry block label and the predecessor block label. The `iter` ITree combinator is used to loop through the control flow as it jumps between these blocks until an external jump or `ret` terminator is encountered. Phi-nodes are also denoted at this stage in parallel using the predecessor block label provided. A closed CFG can be denoted by creating an open CFG of its blocks, providing the entry block id, and ensuring that at the end of its control flow a value is returned rather than a block label. Finally, an MCFG can be denoted by using the `mrec` combinator to mutually resolve internal function calls, and dispatch external function call events when an unknown function is encountered.

Interpretation

The events in a VIR ITree are handled at various levels of interpretation (Zakowski et al., 2021). Level 0 is the denotation of VIR into an ITree that contains all VIR events. At Level 1 intrinsics have been interpreted, at level 2 global environment events are interpreted, at level 3 local environments are interpreted, then the memory model is interpreted at level 4. Following this, the remaining undefined behaviour and nondeterministic events can be interpreted propositionally (as a set of possible behaviours) or as interpreted in a way suitable for execution by picking a behaviour.

Chapter 5

Compiler Verification

The final contribution for this thesis is the development of definitions and lemmas that can ultimately be used to formally verify the LLVM backed of COGENT in Coq. We will start with an overview of the approach and then study the current formulations of the invariants and relations that will be used throughout the proof. This chapter concludes with a summary of proof progress so far, which currently only covers the base cases of the expression correctness induction proof. We have started with the easier features of COGENT, whereas the more involved aspects such as variants and their operations are beyond the scope of the current verification efforts. Our partial work on the simpler inductive cases, including `Let` and `Member` is promising but also incomplete.

5.1 Approach

Our approach follows the style of the proofs of the HELIX compiler, as it is currently the only suitable example of a verified compiler to VIR (Zaliva, 2020). Since the Coq implementation of the COGENT compiler shares a similar structure to HELIX's FHCOL to VIR compiler, we have identified various lemmas and definitions that are useful for our formal verification.

5.1.1 Compiler Correctness

Our top-level theorem for the COGENT compiler is stated in Listing 5.1.

Listing 5.1: Top-level compiler correctness theorem

```
Theorem compiler_correct :
  forall (c : cogent_prog) (ll : llvm_prog),
    compile_cogent c ≡ inr ll →
    eutt R (semantics_cogent c) (semantics_llvm ll).
```

The theorem `compiler_correct` states that for any COGENT program `c` that successfully compiles to an LLVM program `ll`, the resulting ITrees of the semantics of both programs should be equivalent. Currently, we adopt the HELIX approach for top-level correctness, where `semantics_cogent` and `semantics_llvm` interpret a COGENT and VIR program from a fixed entry point starting with empty memory, but it can be generalised to start from any related pair of states and entry points. The notion of equivalence used here is *equivalence up-to Tau* (`eutt`) as it involves ignoring `Tau` nodes when comparing the two ITrees. The relation `R`, which remains to be defined, relates COGENT and VIR states. One known component of `R` will relate the memory state of COGENT to VIR using the memory relation outlined in Section 5.1.3.

A consequence of this top-level theorem is that an always-failing compiler would be considered to be correct. To combat this, it would serve us well to prove that the compiler will be successful whenever the input program is well-formed. In order to define the notion of well-formedness, it may be necessary to formalise COGENT's type system in Coq, which is well beyond the scope of this thesis.

To prove the top-level correctness theorem we first need to prove that single COGENT expressions compile to suitably equivalent VIR blocks. Before we can state the expression correctness theorem, various definitions that must be introduced.

5.1.2 Value Relation

To relate the semantic values of COGENT and VIR we define a value conversion from COGENT `uval`s to VIR `uvalue`s, defined in Table 5.1. Each case is straightforward except the variant case has not yet been defined. Pointers are converted trivially thanks to our use of the VIR memory model for our addresses.

Cogent `uval`	VIR `uvalue`
`UPrim (LBool b)`	`UVALUE_I1 (Int1.repr (if b then 1 else 0))`
`UPrim (LU8 w)`	`UVALUE_I8 (Int8.repr w)`
`UPrim (LU32 w)`	`UVALUE_I32 (Int32.repr w)`
`UPrim (LU64 w)`	`UVALUE_I64 (Int64.repr w)`
`UUnit`	`UVALUE_I8 (Int8.repr 0)`
`UPtr a r`	`UVALUE_Addr a`
`URecord us`	`UVALUE_Struct ((convert_uval . fst) <$> us)`

Table 5.1: Value equivalence table (`convert_uval`)

By implementing this the Coq function `convert_uval`, we can define a relation `correct_result` between COGENT values and VIR values in their respective semantics. This relation, given in Listing 5.3, is more complicated as our COGENT value may need to be related to a constant or a local/global register in VIR. Inspired by HELIX, we can handle both cases at the same time by evaluating an intermediate VIR value against its current context (Zaliva, 2020). The VIR context consists of memory (`memV`), a local context (`l`) and a global context (`g`) (Zakowski et al., 2021). To evaluate the intermediate result `i = (im_type, im_val)` we use the expression in Listing 5.2.

Listing 5.2: Evaluating a single expression in a VIR context

```
interp_cfg (
  translate exp_to_instr (
    denote_exp
      (Some (typ_to_dtyp [] (fst i)))
      (convert_typ [] (snd i))))
  g l memV
```

If the resulting ITree is equivalent to a single `Ret` node containing the same state and a VIR value obtained by apply `convert_uval` to our COGENT value, then the result can be considered correct. We can build the ITree as `Ret (memV, (l, (g, convert_uval u)))` and compare using the ≈ ITree equivalence operator (Xia et al., 2020).

Additionally, we can strengthen this condition: instead of considering only the current local context (`l`), like HELIX, we require that this equivalence holds in any local context `l'` sufficiently similar to `l` (Zaliva, 2020). We consider a context to be sufficiently similar if it does not modify locals (line 5) or variables (line 6) that were touched between the initial and final compiler states `s1` and `s2`. To implement these restrictions the HELIX definitions of `local_scope_preserved` and `Gamma_preserved` are used. Listing 5.3 contains our final value relation that relates a COGENT uval to a VIR intermediate `i` using a VIR context `(memV, (l, g))` and compiler states `s1` and `s2`.

Listing 5.3: Result checking

```
 1  Definition correct_result (γ : ctx) (s1 s2 : IRState) (u : uval) (i : im)
 2                              : config_cfg → Prop :=
 3    fun '(memV, (l, g)) ⇒
 4      forall l',
 5        local_scope_preserved s1 s2 l l' →
 6        Gamma_preserved γ s1 l l' →
 7        interp_cfg
 8          (translate exp_to_instr (
 9            denote_exp (Some (typ_to_dtyp [] (fst i))) (convert_typ [] (snd
                  i))))
10          g l' memV ≈
11        Ret (memV, (l', (g, convert_uval u))).
```

5.1.3 Relating Memory

Now that we can relate individual values, we once again look to HELIX to gain inspiration for our COGENT-VIR memory relation. For every in-scope COGENT variable, we would like the corresponding value in the compiler state to be correct. Additionally, if any of our in-scope variables are a pointer, we want to compare the COGENT and VIR memory states. Our memory invariant in Listing 5.4 handles both these cases.

Listing 5.4: Relating memory

```
 1 Definition memory_invariant (γ : ctx) (s : IRState) : Rel_cfg :=
 2   fun (memC : config_cogent) '(memV, (l, g)) ⇒
 3     forall (n : nat) (i : im) (u : uval),
 4       nth_error γ n ≡ Some u →
 5       nth_error Γ( s) n ≡ Some i →
 6       correct_result γ s s u i (memV, (l, g)) ∧
 7       (forall a r, u ≡ UPtr a r → exists v,
 8         alist_find a memC ≡ Some v ∧
 9         forall i, get_array_cell memV a i (convert_repr r)≡
10                     inr (convert_uval v)).
```

We use `alist_find` on line 8 to look up the pointer's address `a` inside the COGENT memory `memC` to retrieve the stored value `v`. Because we use the same addresses as VIR, we can check that same address inside the VIR memory model (accessed using `get_array_cell`) to see if it contains an equivalent representation. As part of the VIR memory lookup, we need to convert the COGENT representation type `r` to a corresponding VIR `dtyp` using a function defined in Table 5.2.

Cogent `repr`	VIR `dtyp`
`RPrim Bool`	`DTYPE_I 1`
`RPrim (Num U8)`	`DTYPE_I 8`
`RPrim (Num U32)`	`DTYPE_I 32`
`RPrim (Num U64))`	`DTYPE_I 64`
`UUnit`	`DTYPE_I 8`
`UPtr a r`	`DTYPE_Pointer`
`RRecord rs`	`DTYPE_Struct (map convert_repr rs)`

Table 5.2: Representation type equivalence table (`convert_repr`)

5.1.4 Invariants

We are now equipped to define a state invariant `state_invariant` over compiler and COGENT/VIR memory states. Currently, our state invariant in Listing 5.5 checks that the compiler state is well-formed and the memory is correctly related as per the `memory_invariant` covered in the previous section.

Listing 5.5: State invariant

```
Record state_invariant (γ : ctx) (s : IRState) (cogent : config_cogent) (
    vir : config_cfg) : Prop :=
{ mem_is_inv : memory_invariant γ s cogent vir
; IRState_is_WF : WF_IRState γ s
}.
```

Based on the HELIX state invariant, we have identified further invariants that may need to be added to the state invariant to prove the inductive cases of our expression correctness proof, namely: the memory at addresses corresponding to pointers has been allocated, and each register variable in the compiler state is bound (Zaliva, 2020).

5.1.5 Expression Correctness

To prove the top-level correctness theorem, we have first started at the expression level. Our expression compilation correctness lemma is based on HELIX's `genNExpr_correct` lemma. It states the correctness of expression compilation for any initial compiler, COGENT, and VIR states that are related by the state invariant.

Listing 5.6: Expression correctness lemma

```
Lemma compile_expr_correct :
  forall (e : expr) (γ : ctx) (s1 s2 : IRState) (v : im)
         (next_bid entry_bid prev_bid : block_id) (blks : ocfg typ)
         (memC : memoryC)
         (g : global_env) (l : local_env) (memV : memoryV),
    compile_expr e next_bid s1 ≡ inr (s2, (v, entry_bid, blks)) →
    no_failure (interp_expr (E := E_cfg) (denote_expr γ e) memC) →
    bid_bound s1 next_bid →
    state_invariant γ s1 memC (memV, (l, g)) →
    eutt
      (succ_cfg (compile_expr_post v γ s1 s2 next_bid l))
      (interp_expr (denote_expr γ e) memC)
      (interp_cfg
        (denote_ocfg (convert_typ [] blks) (prev_bid, entry_bid))
        g l memV).
```

Let us take a moment to break down the lemma `compile_expr_correct` in Listing 5.6. It states that for any COGENT expression `e` evaluated in a context `γ`, compiled in a compiler state s1 using a block ID next_bid to jump to, if the state after compilation is s2, and the compiler produces a list of blocks blks that implement the Cogent expression when started from entry_bid (line 6), and the semantics of the expression e do not fail for a Cogent memory state memC (line 7), and next_bid was bound already in s1 (line 8), and our state invariant holds for the context, initial compiler state, memory state, and VIR state (which is a tuple of memory, local, and global contexts) (line 9), then the ITree denoting the Cogent expression (line 12) should be equivalent up-to Tau to the ITree denoting the blocks generated by the compiler (line 13-15). We use compile_expr_post to create a relation (line 11) to use for weak bisimulation. This relation is essentially a combination of HELIX's genIR_post and genNExpr_post.

Listing 5.7: Relating Cogent and VIR states

```
1 Definition compile_expr_post (i : im) (γ : ctx) (s1 s2 : IRState)
2                              (to : block_id) (l : local_env)
3                            : Rel_cfg_T uval ocfg_res :=
4   lift_Rel_cfg (state_invariant γ s2) ⊠
5   correct_result_T γ s1 s2 i ⊠
6   branches to ⊠
7   (fun _ '(_, (l', _)) ⇒ local_scope_modif s1 s2 l l').
```

Listing 5.7 states our post-condition for expression compilation: the compiler's state_invariant holds at the end of compilation (line 4), the Cogent result corresponds to the VIR result i (line 5), the VIR code concludes with a jump to the to block (line 6), and finally that all modified variables in the new VIR local environment l' were bound between s1 and s2 (line 7). The state invariant enforces our memory invariant for the variables in the context γ.

5.2 Summary of Progress

A custom inductive principle for Cogent expressions has been defined to allow for an inductive proof of compile_expr_correct in Coq. The base cases have been proven correct with ease. Of recursive cases, Let, Prim, App, and Member have all been attempted but not completed, due to a lack of time and inexperience using Coq.

5.2.1 Base Cases

The base cases `Unit` and `Lit` are simple because they do not generate any VIR blocks, and so their semantics corresponds to an ITree containing a single `Ret` node. There are two key lemmas in use for the base cases. Firstly, we use the `denote_ocfg_unfold_not_in` lemma from VIR which reduces the semantics for a list of blocks to a single `Ret` node if the entry block is not among the list of blocks. The `eutt_Ret` lemma from the ITrees library can then be used to relate our COGENT ITree consisting of a `Ret` to the VIR ITree which now also contains just a `Ret`.

For `Var` we must additionally use the invariant that the in-scope variables in the COGENT context are correctly related to the VIR registers in the compiler state, which is part of our memory invariant, to ensure that the retrieved variable has the correct value. We discharge the absurd case of an out-of-scope variable reference (which could not occur in a type-correct COGENT program) by the 'no failure' assumption for expression compilation.

5.2.2 Let Bindings and Function Application

`Let` and `App` have similar semantics and their proofs proceed similarly, but neither are complete. The proof strategy so far has been to apply a similar approach to the HELIX proof case for sequential composition. Our additional challenge is that the context must be updated before evaluating the body of a function or let binding. Some subgoals have yet to be completed, and most of these fall into two categories: facts about block ID uniqueness, and progression lemmas for the state invariant and premises of the expression correctness statement. When the compiler and semantics are adjusted to use proper function calls as opposed to inlining, the function application case will need to be revisited.

5.2.3 Primitive Operations

Primitive operations do not have the challenge of changing contexts but do involve more blocks than a `let` binding since all the sub-expressions must be linked together before emitting the code for the operation. Since the semantics of COGENT's operators has been defined to match the VIR semantics for the operators used, once that step is reached it should be trivial to relate them. The difficulty comes from successfully splitting the compiled VIR blocks into distinct sections where the inductive hypotheses for each operand can be applied. The main thing preventing us from doing this is the burden of ensuring that the block IDs do not allow for control-flow re-entrance from one block to another. Since the COGENT expressions compile to VIR code that never makes backwards jumps, this should be easier to guarantee than in HELIX.

5.2.4 Member Access

Member access is the only case attempted so far that contains memory effects. We have chosen to interpret the COGENT and VIR ITrees at a level where both language's memory events have been interpreted into a state monad. Our memory relation defines a relationship between those states. The proof proceeds similarly to the other inductive cases where the compilation of the expression is split into the compilation of various sub-expressions. For the block containing the member access instruction(s), the proof splits into two cases, one for the boxed case and one for the unboxed case, since boxed records are accessed via two pointer instructions and unboxed records use a single instruction. We currently lack a VIR lemma to work with these instructions nicely, but we could define it ourselves. Once this case is completed, it should be split off into a lemma that can be used along with the techniques from `let` bindings to prove correctness for `Take` expressions as well.

5.2.5 Remaining Cases

Of the remaining expression cases not yet considered, `Cast` should be the easiest to prove correctness for. This is because it only contains one sub-expression and the control flow of the generated VIR is linear. `If` and `Case` should prove quite difficult due to their branching control flow. The strategy would be to break the list of resulting blocks into independent sub-lists which can be discharged via the inductive hypothesis for each sub-expression. `Put` should proceed similarly to `Member` and `Take` but with memory mutation - we must prove only memory at the relevant address is modified. Variant operations should be left for last as the representation of variants could be updated to match the compact representation of the Haskell implementation. In this case, we would require lemmas that ensure the casts involved are safe and reasonable.

5.3 Alternative Approach

Before finishing this chapter, it is worth briefly discussing an alternative approach to verification considered during this work. If compiler verification proves to be too hard, it may be possible to use a new translation validation approach for establishing compiler correctness that combines benefits from existing approaches. This approach, called *certifying compilers with verified checkers*, is inspired by *certifying algorithms* with *verified checkers* (Alkassar et al., 2014). Instead of producing a formal correctness proof for the compiler, our compiler could produce a witness certifying its compilation. This witness can be used by a separate, simpler program known as a checker to validate the compiler's input against its output for a given compilation. We consider a checker correct if given a correct witness, it accepts correct computations and rejects wrong ones. Thus, a formally verified checker would establish the trustworthiness of the accompanying certifying compiler.

5.3.1 Certifying Compilers with Verified Checkers

The deficiency of our chosen approach is that full verification is difficult and costly to maintain, but we still wish to improve the existing translation validation approach used by COGENT which burdens us with a proof that must be checked for every single compilation. An ideal hybrid approach would require a small proof that need not be updated or validated as often as either of the above approaches.

The inspiration for this hybrid approach is known as *certifying computations and checkers* (Alkassar et al., 2014). The certifying IO program, in our case a compiler, is complemented by a checker which is similar to a validator. Unlike validators, a checker requires a witness to check computations. A certifying program outputs a witness $w \in \mathcal{W}$ in addition to its normal output to be given to the checker, rather than producing an entire formal correctness proof. A checker for a certifying program from X to Y is a simple program, modelled as total boolean-valued function $C : X \times Y \times \mathcal{W} \to$ (**Accept** | **Reject**) that verifies that $y \in Y$ is a correct output for $x \in X$ using the witness w. Figure 5.1 illustrates the use of a checker with a certifying program.

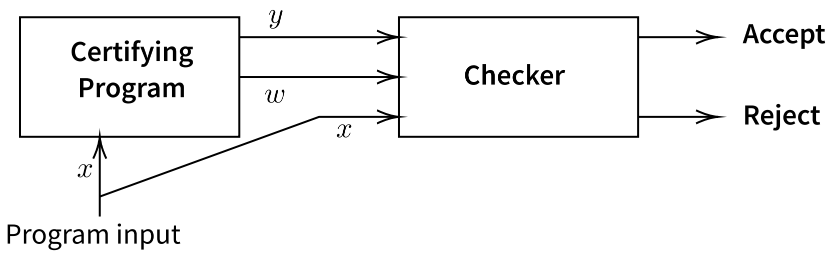

Figure 5.1: Using a checker to certify an IO program

Like a validator, by itself the guarantees of a checker are not as strong as a verified program (Leroy, 2009) - the checker could have a bug, so it is preferable to verify the checker's implementation. Intuitively, a checker algorithm must be at most as complex as the original program's implementation, as it could simply re-compute the solution to check it. The checker approach has been historically used in the context of algorithms, in particular, to verify graph algorithms including connectedness and planarity checking. Noschinski et al. (2014) found that verifying checkers for these algorithms was easier than verifying the original algorithm implementations in C, demonstrating the feasibility of the checker approach.

Using a checker would give us our desired properties of less involved, easier verification if we were to apply it to a compiler. A checker, as described above, takes in a triple of the input, output, and a witness. For a compiler, the input x and output y are the source code (COGENT) and target representation (LLVM-IR / VIR) respectively, and our witness w would take the form of 'hints' emitted by the certifying compiler.

A checker has to terminate and must only reject if the provided triple is invalid, unlike validators which may not terminate for all input tuples or may incorrectly reject validate compilations. The first, weaker notion of correctness for a verified checker is to cover the **Accept** case only. That is, partial correctness for a checker will be established if it can be shown that whenever the checker accepts, the input/output pair is correct, using the given witness (Alkassar et al., 2011). A stronger notion of correctness is to additionally show that the checker only rejects when the input/output is incorrect for the witness, that is the certifying program has given us the wrong output, or the wrong witness (Alkassar et al., 2014). This stronger definition also requires that the checker terminates for each possible input.

5.3.2 Checkers and Interaction Trees

It may be possible to construct a checker using the ITree semantics of COGENT and VIR. ITrees can represent impure and non-terminating programs, but currently, COGENT programs are guaranteed to terminate unless abstract functions call back into COGENT code. Thus, the ITree corresponding to the compiler input as well as the ITree corresponding to its output should be terminating. This should allow one to check compilations by simply interpreting the corresponding ITrees of the input and output programs.

In this case, the witness for the checker could take the form of an ITree itself, or some additional hints required by the checker to interpret the COGENT program correctly. Another way to think about this idea is that the checker would be an algorithmic implementation of the notion of equivalence up-to `Tau` for finite ITrees, essentially checking the compiler by weak bisimulation between the input and output. The correctness of such a checker seems to follow directly from its definition, but more investigation into the approach is required as this feels too good to be true.

Chapter 6

Evaluation

Before concluding, we will briefly evaluate the implementation and formalisation regarding the two main aims of the thesis: to develop an alternative backend that is simpler than the C backend of COGENT, and to explore a different verification approach for the new LLVM backend that improves upon the existing translation validation approach of COGENT's C backend.

6.1 Comparison with Existing Cogent to C Backend

Firstly, the implementation of the LLVM-IR backend for COGENT is conceptually simpler than the existing C backend. Our Haskell implementation is smaller than the C backend despite supporting the same features, and the code it produces is still concise, despite targeting a lower-level language.

6.1.1 Implementation

The Haskell implementation for COGENT's LLVM backend is less than half the size of the C backend. Our implementation consists of 700 source lines of code (SLOC) of which 254 lines are used to implement the FFI compatibility with C. Meanwhile, the

C backend consists of 1758 SLOC of which 510 are for representing and rendering the C AST. Focusing solely on the remaining lines of code that implement the translation of COGENT syntax, our implementation of 446 SLOC is 36% the size of the 1248 SLOC used to implement the C backend. In the LLVM backend, we reap the benefits of the `llvm-hs` library which helps abstract from the low-level details of code generation. Thus, our implementation focuses solely on the translation from COGENT expressions to LLVM instructions. Meanwhile, the C backend defines its own code generation state monad as well as an AST for the C language which must then be rendered as C code.

Given that LLVM-IR is a language designed as a compiler target for the C/C++ compiler (Lattner and Adve, 2004), it is no surprise that our COGENT expressions and types map to LLVM-IR instructions and types just as well as they do to C. In addition, it is much easier to generate the IR syntax than it is to generate code for a comparatively higher-level programming language like C. These are the two main factors that have allowed us to implement a smaller LLVM backend than COGENT's C backend.

The Coq implementation of the LLVM backend is also similarly small. It consists of 375 lines of Coq code, plus 189 lines borrowed from HELIX for code generation helpers. Comparing the actual implementation in Coq to the 415 SLOC required in Haskell to support the same subset of COGENT, we find the Coq implementation is a bit longer (but still less than the C backend) since we cannot take advantage of the higher level of abstraction provided by the `llvm-hs` library and instead have to generate AST fragments directly.

6.1.2 Output

A more rigorous investigation into the performance of our new LLVM backend is required, but preliminary findings indicate that the LLVM backend for COGENT generates efficient, clean code, especially when compared to the current C backend.

Since C is a higher-level language than LLVM-IR, we would not expect our LLVM backend to produce a smaller IR output than the C backend's output. This is true, for instance, our 30-line bag example in Appendix A.2 compiles to 110 lines of C code, yet our backend produces 130 lines of LLVM-IR, plus 70 additional lines of glue code for FFI support to C.

An interesting comparison can be made between the IR generated by Clang from our C backend's output and what we generate natively with our LLVM backend. Without optimisation, the C backend's corresponding IR is 360 lines long, much longer than our natively produced LLVM-IR, even including FFI glue code. This appears to indicate that we are producing less redundant code by default when compared to the C backend for COGENT. If we optimise both LLVM-IR files with `opt -O1` our code shrinks to 155 including glue code, but the corresponding IR obtained from Clang is still approximately twice the length, at 307 lines. At `opt -O3` the optimiser begins to inline functions, especially our FFI wrappers, to prioritise performance rather than brevity, so this is no longer a meaningful comparison.

Once the LLVM backend for COGENT supports linking with antiquoted C programs, we will be able to properly assess the performance and binary sizes of programs compiled with our backend using an existing suite of system code implemented in COGENT and C.

6.2 Comparison of Formalisation and Verification

Secondly, our ITree formalisation of COGENT's semantics pave the way for a simpler verification story for the compiler, but reaching a full correctness proof may be quite difficult.

6.2.1 Formalisation

Our ITree formalisation has multiple advantages over COGENT's existing update semantics. Firstly, our semantics is executable via Coq's code extraction, unlike the update semantics relation formalised in Isabelle/HOL (Xia et al., 2020). Secondly, our modular definition of COGENT's events and their handlers allows for the semantics to be extended more easily than the existing update semantics relation. The ITree treatment of events provides flexibility without sacrificing expressiveness since ITrees can represent diverging computations, which would allow us to accurately model abstract function interactions via proper function calls (Xia et al., 2020).

6.2.2 Verification

It is difficult to evaluate the verification aspect of this thesis, as the necessary proof work is far from complete. At this early stage, the proofs seem challenging to complete even with HELIX as an example to follow. Since ITrees and VIR are not yet mature in terms of documentation and examples, there is little to guide us through the remaining verification work other than the design of HELIX and its proofs. If verification of the LLVM backend can one day be completed, we would no longer need to check an Isabelle/HOL proof for each COGENT compilation, instead, our once and forall compiler correctness proof would ensure the LLVM produced by our new backend is correct.

Chapter 7

Conclusion

Over the course of this thesis, we explored not one, but two LLVM backends for the COGENT language. The first backend, implemented in Haskell, demonstrated the elegance of LLVM-IR as a target language for the COGENT compiler. Since this backend supports the full COGENT language, including a C Foreign Function Interface, it can already be used for real systems programming. Our reimplementation of the COGENT to LLVM backend within Coq lays the groundwork for full compiler verification. This furthers COGENT's goal to reduce the cost of verification.

Further work on the compiler is required to the remaining features supported by the Haskell implementation over to the Coq implementation of the backed. Moreover, there is more to be done in improving the code quality of the Coq implementation to that achieved in the Haskell implementation. Additionally, there are language extensions to COGENT that could eventually be implemented in the LLVM backend, perhaps more elegantly than in the existing C backend.

To verify the Coq implementation, we introduced a denotational semantics for COGENT using Interaction Trees. The semantics is designed to be easily relatable to both the existing update semantics of COGENT, and the ITree semantics of LLVM-IR, via the VIR embedding in Coq. Our semantics is easy to extend as the COGENT language grows, allowing visible effects to be modelled using events. From COGENT's

ITree semantics, we have extracted an executable interpreter in Haskell which precisely implements the formal semantics.

It remains to be shown that our new ITree semantics agree with the existing update semantics for COGENT. Such a proof is necessary to establish an end-to-end chain of correctness for COGENT to LLVM compilations. Another future area worth investigating is the representation of abstract functions using ITrees in Coq to give an interpretation for abstract function calls in the COGENT ITree semantics. COGENT and VIR's ITree semantics could potentially be used to design a checker for either LLVM backend, which would be simpler to implement and formally verify.

Within the Coq theorem prover, we have formalised many of the relations, invariants, and lemmas necessary for a compiler correctness proof of the backend, using the ITree semantics. While the overall scope of this work exceeded our initial plan, the compiler correctness proof is far from complete. Nevertheless, this thesis represents significant progress towards a verified compiler from COGENT to LLVM.

Bibliography

Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, and Christine Rizkallah. Verification of certifying computations. In *International Conference on Computer Aided Verification*, pages 67–82. Springer, 2011.

Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, and Christine Rizkallah. A framework for the verification of certifying computations. *Journal of Automated Reasoning*, 52 (3):241–273, 2014.

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–188, Atlanta, GA, USA, April 2016. doi: 10.1145/2872362.2872404.

Andrew W Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.

Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

Anthony Cowley, Stephen Diehl, Moritz Kiefer, and Benjamin S. Scarlet. General purpose LLVM bindings, 2019. URL `https://hackage.haskell.org/package/llvm-hs`.

Edsko De Vries, Rinus Plasmeijer, and David M Abrahamson. Uniqueness typing simplified. In *Symposium on Implementation and Application of Functional Languages*, pages 201–218. Springer, 2007.

Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Quickchick: Property-based testing for coq. 2014.

Will Fancher. Monadfix is time travel, Aug 2017. URL `https://elvishjerricco.github.io/2017/08/22/monadfix-is-time-travel.html`.

SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993.

Luka Kerr. *Extending a Purely Functional Language to Enable Low-Level Systems Programming*. PhD thesis, The University of New South Wales, 2020.

Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. *ACM SIGPLAN Notices*, 49(1):179–191, 2014.

Peter Lammich. Generating Verified LLVM from Isabelle/HOL. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, pages 22:1–22:19, 2019.

Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

Brian Andrew Leibig. An LLVM Back-end for MLton. *Department of Computer Science, B. Thomas Golisano College of Computing and Information Sciences, Tech. Rep*, 2013.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert-a formally verified optimizing compiler. 2016.

LLVM Project. LLVM language reference manual - type system, 2020. URL `https://llvm.org/docs/LangRef.html#type-system`.

Emmet Murray. *Recursive Types For Cogent*. PhD thesis, The University of New South Wales, 2019.

Magnus O Myreen. Functional programs: conversions between deep and shallow embeddings. In *International Conference on Interactive Theorem Proving*, pages 412–417. Springer, 2012.

George C Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, 1997.

Tobias Nipkow and Gerwin Klein. *Concrete semantics: with Isabelle/HOL*. Springer, 2014.

Lars Noschinski, Christine Rizkallah, and Kurt Mehlhorn. Verification of certifying computations through autocorres and simpl. In *NASA Formal Methods Symposium*, pages 46–61. Springer, 2014.

Liam O'Connor. *Type Systems for Systems Types*. PhD thesis, UNSW Sydney, Computer Science & Engineering, 2019.

Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. In *International Conference on Functional Programming*, Nara, Japan, September 2016.

Liam O'Connor, Zilin Chen, Partha Susarla, Christine Rizkallah, Gerwin Klein, and Gabriele Keller. Bringing effortless refinement of data layouts to Cogent. In *International Symposium on Leveraging Applications of Formal Methods*, pages 134–149. Springer, 2018.

Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166. Springer, 1998.

Lucy Qiu. *Termination Checker for Recursive Types in Cogent*. PhD thesis, The University of New South Wales, 2020.

Silvain Rideau and Xavier Leroy. Validating register allocation and spilling. In *International Conference on Compiler Construction*, pages 224–243. Springer, 2010.

Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. A framework for the automatic formal verification of refinement from Cogent to C. In *International Conference on Interactive Theorem Proving*, Nancy, France, August 2016.

Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, 1988.

Zhanlin Shang. An LLVM backend of the Cogent compiler. Project report, UNSW Sydney, Computer Science & Engineering, May 2020.

Yong Kiam Tan, Magnus O Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 60–73, 2016.

David A Terei and Manuel MT Chakravarty. An LLVM backend for GHC. In *Proceedings of the third ACM Haskell symposium on Haskell*, pages 109–120, 2010.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in Coq. In *Proceedings of the 47th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'20). ACM, New York, NY, USA*, 2020.

Yannick Zakowski et al. The Vellvm (Verified LLVM) coq development, 2021. URL `https://github.com/vellvm/vellvm/`.

Vadim Zaliva. *HELIX: From Math to Verified Code*. PhD thesis, Carnegie Mellon University, 2020.

Steve Zdancewic et al. Vellvm-legacy, 2014. URL `https://github.com/vellvm/vellvm-legacy`.

Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 427–440, 2012.

Appendices

A.1 Overview of Haskell Modules

At the time of writing the Haskell implementation can be located at `cogent/src/Cogent/LLVM/` in the COGENT repository.

A.1.1 Overrides.hs

The `llvm-hs` library provides utility functions for most LLVM-IR generation use-cases, however, in this module we reimplement `function` and `extern` to allow parameter attributes to be provided. This change is necessary as in our C FFI glue code we make use of the `sret` attribute for parameters used to return a value from a function, and `byval` to indicate where pointer parameters should be passed by value.

A.1.2 Compile.hs

This module provides the external interface for the LLVM backend. It exports a `toLLVM` function which can be used by the compiler to take a list of top-level COGENT definitions, compile them to an AST, and output the resultant LLVM module to a file. We use `llvm-hs`'s `buildModuleT` to convert our LLVM-IR AST to a module, and then `moduleLLVMAssembly` to produce the textual representation of the LLVM assembly.

A.1.3 CodeGen.hs

In this module, we define a code generation monad `Codegen` using `ModuleBuilderT` and `IRBuilderT`, which are state transformer monads that allow us to easily generate LLVM-IR without having to maintain our own register, block, and definition state. We define the `Env` state described in Section 3.1.1 in this file as well.

A.1.4 Expr.hs

This module converts COGENT expressions into LLVM-IR instructions, using the `Codegen` monad to hide block and register management. The file is barely 200 lines as most expressions correspond to a single LLVM instruction which can be constructed quite simply. Each case is examined in detail in Section 3.1.3.

A.1.5 Types.hs

In this module lies the function to convert COGENT types to LLVM-IR types. As above, each case is examined in detail in Section 3.1.2. Also included in this file are various utilities relating to types:

- `fieldTypes` - converts COGENT types to string representations, used for naming instantiations of parametric types
- `maxMember` - get the maximal type out of a variant type's payloads
- `tagType` - look up a variant constructor's payload type
- `collectTags` - collect all the tag names for a definition or type
- `abstractType` - get or emit an abstract type definition

A.1.6 CCompat.hs

This module forms the first part of this development's C FFI implementation. It attempts to create wrapper functions for generated LLVM code whose signatures should match those of equivalent C functions compiled with Clang. The need for this glue code arises from the argument and return type marshalling performed by C compilers, such as converting a structure argument type to multiple arguments, or a structure return type to a pointer return during the translation to LLVM-IR.

Both `wrapLLVM` and `wrapC` are provided; the former produces a function to wrap a generated LLVM-IR function so it can be called correctly from C, and the latter exists to wrap calls to abstract C functions so they can be called correctly inside generated LLVM-IR code. A deep dive into the type coercion can be found in Section 3.1.5.

A.1.7 CHeader.hs

To link generated LLVM-IR code with C code, we must provide a header file containing the declarations of each generated function and type that the C programmer may wish to refer to. This module performs basic header file generation including function prototypes and type aliases for all the aggregate types in the file. This approach is examined in more detail in Section 3.1.5. The approach used is quite naive and in the future it would be better to integrate the existing header file generation which is designed for use with antiquoted C, which allows for embedding COGENT syntax inside C.

A.1.8 CoqGen.hs

This file is not used by the Haskell implementation of the backend. It contains code to generate Coq definitions corresponding to a Coq AST. The flag `--coq-gen` can be used to emit these definitions for an input COGENT file. The purpose of this was to assist early development of the Coq implementation which did not rely on extraction.

A.1.9 WithCoq.hs

This file is used to interface with code extracted from the Coq implementation of the backend, as well as the ITree interpreter for COGENT. The top-level `compileWithCoq` and `interpWithCoq` functions are used to interface with the backend and interpreter respectively. `convCogentAST` recursively traverses a core COGENT definition and converts it to the extracted deep embedding from Coq. Similarly, `convVellvmAST` converts the VIR AST into an `llvm-hs` AST so that it can be rendered as an LLVM module.

A.2 Comprehensive Compiler Example

A.2.1 Cogent Program

The following COGENT program in Listing A.1 calculates the average of a `List` of integers inserted into a heap-allocated `Bag`. The bag is represented as a record containing the number of items and the total sum inserted. We delegate memory operations (`newBag` and `freeBag`) to be implemented in C using the abstract `Heap` type to represent memory. We also assume the existence of a `reduce` function over `List` which can be implemented also in C.

Listing A.1: An more complex COGENT program (`bag.cogent`)

```
type Heap
type Bag = { count : U32, sum : U32 }
newBag : Heap → < Failure Heap | Success (Bag, Heap) >
freeBag : (Heap, Bag) → Heap

addToBag : (U32, Bag) → Bag
addToBag (x, b { count = c, sum = s }) = b { count = c + 1, sum = s + x }

averageBag : Bag! → < EmptyBag | Success U32 >
averageBag b = if b.count == 0 then EmptyBag else Success (b.sum / b.count)

type List a
reduce : all (a, b). ((List a)!, (a!, b) → b, b) → b

average : (Heap, (List U32)!) → (Heap, U32)
average (h, ls) =
    newBag h
    | Success (bag, h') →
        let bag' = reduce (ls, addToBag, bag)
        in averageBag bag' !bag'
            | Success n → (freeBag (h', bag'), n)
            | EmptyBag  → (freeBag (h', bag'), 0)
    | Failure h' → (h', 0)
```

A.2.2 LLVM-IR Translation

To begin with, we will ignore the wrapper functions generated for the C FFI, they will be examined separately.

Our COGENT program (on the left) starts by defining an abstract type `Heap` and an alias for the `Bag` record type. The LLVM backend produces the Listing A.3.

Listing A.2: Input (`bag.cogent`)

```
type Heap
type Bag = { count : U32, sum : U32 }
```

Listing A.3: Output (`bag.ll`)

```
%Heap = type opaque
```

The abstract type gets compiled to an opaque type definition in LLVM-IR. However, our `Bag` definition vanishes. This is because, by the time the AST reaches our new backend, aliases like this have already been desugared. This could be solved by holding onto these type definitions and providing them to the LLVM-backend, but it is not necessary for code generation, instead, all occurrences of `Bag` are compiled to `{ i32, i32 }*`.

Next we define two abstract memory operations for the `Bag` and `Heap` types.

```
newBag : Heap →
  <Failure Heap | Success (Bag, Heap)>
freeBag : (Heap, Bag) → Heap
```

```
declare void @newBag({ i32, { { i32, i32 }*,
  %Heap* } }* noalias sret, %Heap*)
declare %Heap* @freeBag(%Heap*, { i32, i32 }*)
```

In our declaration of the abstract function `newBag`, we can see that the variant type `< Failure Heap | Success (Bag, Heap) >` has been compiled to `{ i32, { { i32, i32 }*, %Heap* } }` in which the first field is the variant's tag, the second field is the tuple of a `Bag` pointer and `Heap` pointer. The tuple has been chosen as it represents the maximal argument type for the variant. Since the total size of this structure including padding exceeds 128-bits, it has become a return parameter in this function declaration. Meanwhile, the `Heap` argument of `newBag` is simply translated to a pointer to the opaque `%Heap` type. The translation for `freeBag`'s return type is the same as `newBag`'s argument. Lastly, the tuple argument to `freeBag` has been split into two arguments as they are both pointers that will happily fit in a 64-bit register each.

Let us now consider the compilation of the `addToBag` function, which is implemented entirely in COGENT.

```
addToBag : (U32, Bag)      define { i32, i32 }* @addToBag.llvm({ i32, { i32, i32 }* } %a_0){
        → Bag
addToBag (                 entry_0:
  x,                         %0 = extractvalue { i32, { i32, i32 }* } %a_0, 0
  b {                        %1 = extractvalue { i32, { i32, i32 }* } %a_0, 1
    count = c,               %2 = getelementptr { i32, i32 }, { i32, i32 }* %1, i32 0, i32 0
                             %3 = load i32, i32* %2
    sum = s                  %4 = getelementptr { i32, i32 }, { i32, i32 }* %1, i32 0, i32 1
                             %5 = load i32, i32* %4
  }) = b {
    count = c + 1,           %6 = add i32 %3, 1
                             %7 = getelementptr { i32, i32 }, { i32, i32 }* %1, i32 0, i32 0
                             store i32 %6, i32* %7
    sum = s + x              %8 = add i32 %5, %0
                             %9 = getelementptr { i32, i32 }, { i32, i32 }* %1, i32 0, i32 1
                             store i32 %8, i32* %9
  }                          ret { i32, i32 }* %1
                           }
```

First, we observe that the tuple argument `(U32, Bag)` has been desugared to an unboxed record type and compiled to `{ i32, { i32, i32 }* }`. The argument destructuring for `x` and `b` corresponds to two `extractvalue` instructions, one for each field in the structure corresponding to the tuple argument. The number to add is stored in `%0` and the bag address is stored in `%1`. Accessing `count` and `sum` inside the (boxed) bag `b` requires calculating a pointer with `getelementptr` and then `load`ing the field. To insert the updated `count` we must first increment `c`, this is done by using the `add` instruction with the `1` literal. Next, we calculate the address for `count` using `getelementptr` again (this is the same address that was stored in `%2`, the optimiser will realise this) and `store` the updated value. Updating `sum` is done similarly, but we add `%0` instead. Finally, in order to return the updated bag `b`, in LLVM we can `ret` the `%1` register, which contains the pointer to the bag that we have been updating.

Another COGENT function that we can examine is the `averageBag` function.

```
averageBag : Bag! →
  < EmptyBag
  | Success U32 >
averageBag b =
  if b.count == 0






  then
    EmptyBag

  else
    Success (
      b.sum
      /
      b.count







    )
```

```
define { i32, i32 } @averageBag.llvm({ i32, i32 }* %a_0) {


entry_0:
  %0 = getelementptr { i32, i32 }, { i32, i32 }* %a_0, i32 0, i32 0
  %1 = load i32, i32* %0
  %2 = icmp eq i32 %1, 0
  %3 = zext i1 %2 to i8
  %4 = icmp ne i8 %3, 0
  br i1 %4, label %if.true_0, label %if.false_0
if.true_0:                                         ; preds = %entry_0
  %5 = insertvalue { i32, i32 } undef, i32 0, 0
  br label %if.done_0
if.false_0:                                        ; preds = %entry_0

  %6 = getelementptr { i32, i32 }, { i32, i32 }* %a_0, i32 0, i32 1
  %7 = load i32, i32* %6
  %8 = getelementptr { i32, i32 }, { i32, i32 }* %a_0, i32 0, i32 0
  %9 = load i32, i32* %8
  %10 = udiv i32 %7, %9
  %11 = insertvalue { i32, i32 } undef, i32 2, 0
  %12 = alloca i32
  %13 = bitcast i32* %12 to i32*
  store i32 %10, i32* %13, align 1
  %14 = load i32, i32* %12, align 1
  %15 = insertvalue { i32, i32 } %11, i32 %14, 1
  br label %if.done_0
if.done_0:                                         ; preds = %if.
     false_0, %if.true_0
  %16 = phi { i32, i32 } [ %5, %if.true_0 ], [ %15, %if.false_0 ]
  ret { i32, i32 } %16
}
```

The return type of this function is a variant where the maximal type is a `U32`, compiling to a return type of `{ i32, i32 }`, where the first field is the tag and the second is the value. The argument is a pointer to the bag structure we've seen before. This function uses an `if` expression, which gets compiled to four blocks: `entry_0` is the function entry where the condition `b.count == 0` is tested, `if.true_0` is the branch taken if the condition is true, `if.false_0` is the branch taken when it is false, and `if.done_0` is a block that both execution paths return to, and it is here that `phi` is used to recover the result based on the branch taken. In `entry_0` we first extract the `count` field using `getelementptr` and `load`. The condition is checked using `icmp eq` with the `0` literal. Due to our boolean representation, the result of the condition is casted to an `i8` before being checked against `0` again - the optimiser will clean this up for us. `br` is used to jump to either block depending on the condition. In the true block, all we need to do is construct an `EmptyBag` in `%5` by setting the tag value using `insertvalue`, and then jump to the done branch. The false block is more complicated as we must first retrieve the bag's sum and count using `getelementptr` and `load` so they can be divided with `udiv`. Next, the tag field is set with `insertvalue`. Before inserting the value field, we do a cast so we can be sure to insert a value of the correct type. However, in this case, the value we wish to insert is already of the maximal type, so the `bitcast` is a no-op. The optimiser will realise this as well. The final variant we have constructed in this branch is in register `%15`. Once both branches return to the done block, the generated `phi` and `ret` instructions ensure that if we came from the true branch we would like to return `%5` and from the false branch, we will return `%15`.

Let us take a brief break from these large blocks of code and inspect the translation of the polymorphic abstract type `List` and the `reduce` function.

```
type List a

reduce : all (a, b).
  (
    (List a)!,
    (a!, b) → b,
    b
  ) → b
```

```
%"List U32" = type opaque

declare { i32, i32 }* @reduce_0(
  {
    %"List U32"*,
    { i32, i32 }* ({ i32, { i32, i32 }* })*,
    { i32, i32 }*
  }* byval)
```

Since LLVM-IR does not support polymorphic types, we solve this by generating a type definition for each instantiation of an abstract type. In this program, `List` is only ever instantiated with `U32` so we only need to define an `opaque` type for that instantiation. Similarly, our polymorphic `reduce` function is only ever applied with the type arguments `U32` and `Bag`, so the function declaration that gets generated has the type arguments replaced by the LLVM translations of those types. The second tuple argument for `reduce` is a function type, which gets translated to the an LLVM function type `{ i32, i32 }* ({ i32, { i32, i32 }* })*` after the instantiation.

The last function to be compiled is `average` which makes calls to all the previous declared functions.

```
 69  average : (Heap, (List U32)!)  define { %Heap*, i32 } @average.llvm({ %Heap*, %"List U32"* } %a_0) {
 70        → (Heap, U32)
 71  average (                      entry_0:
 72      h,                           %0 = extractvalue { %Heap*, %"List U32"* } %a_0, 0
 73      ls) =                        %1 = extractvalue { %Heap*, %"List U32"* } %a_0, 1
 74    newBag h                       %2 = call { i32, { { i32, i32 }*, %Heap* } } @newBag.llvm(%Heap* %0)
 75                                   %3 = extractvalue { i32, { { i32, i32 }*, %Heap* } } %2, 0
 76                                   %4 = icmp eq i32 %3, 2
 77                                   br i1 %4, label %case.true_0, label %case.false_1
 78    | Success (                  case.true_0:                              ; preds = %entry_0
 79                                   %5 = extractvalue { i32, { { i32, i32 }*, %Heap* } } %2, 1
 80                                   %6 = alloca { { i32, i32 }*, %Heap* }, align 4
 81                                   %7 = bitcast { { i32, i32 }*, %Heap* }* %6 to { { i32, i32 }*, %Heap* }*
 82                                   store { { i32, i32 }*, %Heap* } %5, { { i32, i32 }*, %Heap* }* %6, align 1
 83                                   %8 = load { { i32, i32 }*, %Heap* }, { { i32, i32 }*, %Heap* }* %7, align 1
 84        bag,                       %9 = extractvalue { { i32, i32 }*, %Heap* } %8, 0
 85        h') →                      %10 = extractvalue { { i32, i32 }*, %Heap* } %8, 1
 86      let bag' = reduce (
 87        ls,                        %11 = insertvalue { %"List U32"*, { i32, i32 }* ({ i32, { i32, i32 }* })*, {
 88                                         i32, i32 }* } undef, %"List U32"* %1, 0
 89        addToBag,                  %12 = insertvalue { %"List U32"*, { i32, i32 }* ({ i32, { i32, i32 }* })*, {
 90                                         i32, i32 }* } %11, { i32, i32 }* ({ i32, { i32, i32 }* })* @addToBag.
 91                                         llvm, 1
 92        bag                        %13 = insertvalue { %"List U32"*, { i32, i32 }* ({ i32, { i32, i32 }* })*, {
 93                                         i32, i32 }* } %12, { i32, i32 }* %9, 2
 94      )                            %14 = call { i32, i32 }* @reduce_0.llvm({ %"List U32"*, { i32, i32 }* ({ i32,
 95      in                                 { i32, i32 }* })*, { i32, i32 }* } %13)
 96      averageBag bag' !bag'        %15 = call { i32, i32 } @averageBag.llvm({ i32, i32 }* %14)
 97                                   %16 = extractvalue { i32, i32 } %15, 0
 98                                   %17 = icmp eq i32 %16, 2
 99                                   br i1 %17, label %case.true_1, label %case.false_0
100        | Success                case.true_1:                              ; preds = %case.true_0
101            n → (                  %18 = extractvalue { i32, i32 } %15, 1
102                                   %19 = alloca i32, align 4
103                                   %20 = bitcast i32* %19 to i32*
104                                   store i32 %18, i32* %19, align 1
105                                   %21 = load i32, i32* %20, align 1
106            freeBag (
107              h',                  %22 = insertvalue { %Heap*, { i32, i32 }* } undef, %Heap* %10, 0
108              bag'                 %23 = insertvalue { %Heap*, { i32, i32 }* } %22, { i32, i32 }* %14, 1
109            )                      %24 = call %Heap* @freeBag.llvm({ %Heap*, { i32, i32 }* } %23)
110            ,                      %25 = insertvalue { %Heap*, i32 } undef, %Heap* %24, 0
111            n)                     %26 = insertvalue { %Heap*, i32 } %25, i32 %21, 1
112                                   br label %case.done_0
113        | EmptyBag → (           case.false_0:                             ; preds = %case.true_0
114                                   %27 = extractvalue { i32, i32 } %15, 1
115                                   %28 = alloca i32, align 4
116                                   %29 = bitcast i32* %28 to i8*
117                                   store i32 %27, i32* %28, align 1
118                                   %30 = load i8, i8* %29, align 1
119          freeBag (
120            h',                    %31 = insertvalue { %Heap*, { i32, i32 }* } undef, %Heap* %10, 0
121            bag'                   %32 = insertvalue { %Heap*, { i32, i32 }* } %31, { i32, i32 }* %14, 1
122          )                        %33 = call %Heap* @freeBag.llvm({ %Heap*, { i32, i32 }* } %32)
123          ,                        %34 = insertvalue { %Heap*, i32 } undef, %Heap* %33, 0
124          0)                       %35 = insertvalue { %Heap*, i32 } %34, i32 0, 1
125                                   br label %case.done_0
126                                 case.done_0:                              ; preds = %case.false_0, %case.true_1
127                                   %36 = phi { %Heap*, i32 } [ %26, %case.true_1 ], [ %35, %case.false_0 ]
128                                   br label %case.done_1
129    | Failure                    case.false_1:                             ; preds = %entry_0
130        h' → (                     %37 = extractvalue { i32, { { i32, i32 }*, %Heap* } } %2, 1
131                                   %38 = alloca { { i32, i32 }*, %Heap* }, align 4
132                                   %39 = bitcast { { i32, i32 }*, %Heap* }* %38 to %Heap**
133                                   store { { i32, i32 }*, %Heap* } %37, { { i32, i32 }*, %Heap* }* %38, align 1
134                                   %40 = load %Heap*, %Heap** %39, align 1
135          h',                      %41 = insertvalue { %Heap*, i32 } undef, %Heap* %40, 0
136          0)                       %42 = insertvalue { %Heap*, i32 } %41, i32 0, 1
137                                   br label %case.done_1
138                                 case.done_1:                              ; preds = %case.false_1, %case.done_0
139                                   %43 = phi { %Heap*, i32 } [ %36, %case.done_0 ], [ %42, %case.false_1 ]
140                                   ret { %Heap*, i32 } %43
141                                 }
```

In the above listing the COGENT function calls (on lines 74, 94, 96, 109, and 122) have been translated to `call` instructions on the LLVM side. For the functions requiring a tuple argument, we must build up a struct literal using successive `insertvalue` instructions before calling the function. Moreover, the nested case-matching in the COGENT function has been translated to branching between 6 blocks in the LLVM-IR function.

A.2.3 LLVM-IR Optimisation

Our compiler generates many redundant statements that are taken care of by the LLVM optimiser (`opt`).

Here is an optimised version of the previous function `average.llvm` using `opt -O1`.

```
define { %Heap*, i32 } @average.llvm({ %Heap*, %"List U32"* } %a_0) local_unnamed_addr {
  %0 = extractvalue { %Heap*, %"List U32"* } %a_0, 0
  %1 = call { i32, { { i32, i32 }*, %Heap* } } @newBag.llvm(%Heap* %0)
  %2 = extractvalue { i32, { { i32, i32 }*, %Heap* } } %1, 0
  %3 = icmp eq i32 %2, 2
  %4 = extractvalue { i32, { { i32, i32 }*, %Heap* } } %1, 1
  br i1 %3, label %case.true_0, label %case.false_1
case.true_0:                                          ; preds = %entry_0
  %5 = extractvalue { %Heap*, %"List U32"* } %a_0, 1
  %.fca.0.extract = extractvalue { { i32, i32 }*, %Heap* } %4, 0
  %.fca.1.extract = extractvalue { { i32, i32 }*, %Heap* } %4, 1
  %6 = insertvalue { %"List U32"*, { i32, i32 }* ({ i32, { i32, i32 }* })*, { i32, i32 }* } undef, %"List U32"* %5, 0
  %7 = insertvalue { %"List U32"*, { i32, i32 }* ({ i32, { i32, i32 }* })*, { i32, i32 }* } %6, { i32, i32 }* ({ i32, { i32
        , i32 }* })* @addToBag.llvm, 1
  %8 = insertvalue { %"List U32"*, { i32, i32 }* ({ i32, { i32, i32 }* })*, { i32, i32 }* } %7, { i32, i32 }* %.fca.0.
        extract, 2
  %9 = call { i32, i32 }* @reduce_0.llvm({ %"List U32"*, { i32, i32 }* ({ i32, { i32, i32 }* })*, { i32, i32 }* } %8)
  %10 = call { i32, i32 } @averageBag.llvm({ i32, i32 }* %9)
  %11 = extractvalue { i32, i32 } %10, 0
  %12 = icmp eq i32 %11, 2
  br i1 %12, label %case.true_1, label %case.false_0
case.true_1:                                          ; preds = %case.true_0
  %13 = extractvalue { i32, i32 } %10, 1
  %14 = insertvalue { %Heap*, { i32, i32 }* } undef, %Heap* %.fca.1.extract, 0
  %15 = insertvalue { %Heap*, { i32, i32 }* } %14, { i32, i32 }* %9, 1
  %16 = call %Heap* @freeBag.llvm({ %Heap*, { i32, i32 }* } %15)
  br label %case.done_1
case.false_0:                                         ; preds = %case.true_0
  %17 = insertvalue { %Heap*, { i32, i32 }* } undef, %Heap* %.fca.1.extract, 0
  %18 = insertvalue { %Heap*, { i32, i32 }* } %17, { i32, i32 }* %9, 1
  %19 = call %Heap* @freeBag.llvm({ %Heap*, { i32, i32 }* } %18)
  br label %case.done_1
case.false_1:                                         ; preds = %entry_0
  %.fca.0.extract3 = extractvalue { { i32, i32 }*, %Heap* } %4, 0
  %20 = bitcast { i32, i32 }* %.fca.0.extract3 to %Heap*
  br label %case.done_1
case.done_1:                                          ; preds = %case.true_1, %case.false_0, %case.false_1
  %.sink8 = phi %Heap* [ %16, %case.true_1 ], [ %19, %case.false_0 ], [ %20, %case.false_1 ]
  %.sink7 = phi i32 [ %13, %case.true_1 ], [ 0, %case.false_0 ], [ 0, %case.false_1 ]
  %21 = insertvalue { %Heap*, i32 } undef, %Heap* %.sink8, 0
  %22 = insertvalue { %Heap*, i32 } %21, i32 %.sink7, 1
  ret { %Heap*, i32 } %22
}
```

Instead of the original 55 instructions, the optimised LLVM-IR has 34 instructions.

The redundant operations from our generated code have been removed entirely. If we optimise further it will begin to inline functions as well. `opt` reduces the function `addToBag.llvm` from 13 to 11 instructions, and `averageBag.llvm` from 22 to 11.

A.2.4 C FFI Wrappers

Each COGENT function in the generated LLVM also has a wrapper generated for it, and each abstract function declaration in the LLVM also has an 'un-wrapper' generated:

- `newBag.llvm`, the un-wrapper for `newBag`, changes the return argument into an actual return value
- `freeBag.llvm`, the un-wrapper for `freeBag` changes the two arguments into a single structure argument
- `addToBag`, the wrapper for `addToBag.llvm`, changes the structure argument into two separate arguments
- `averageBag`, the wrapper for `averageBag.llvm`, changes the structure return value into a single return value
- `reduce_0.llvm`, the un-wrapper for `reduce_0`, changes the argument passed by reference into an argument passed by value
- `average`, the wrapper for `average.llvm`, changes the structure argument into two separate arguments

When we run the LLVM optimiser over these wrapper functions at a higher level, such as `opt -O3`, their calls to the inner function will almost definitely be inlined, meaning the glue code doesn't introduce any unnecessary jumps.

A.2.5 Generated Header File

Listing A.4 contains the header file generated alongside the LLVM-IR. Lines 1, 2, and 39 act as a header guard, and line 4 includes the common COGENT definitions.

Listing A.4: Generated header file (`bag.h`)

```
1  #ifndef BAG_H__
2  #define BAG_H__
3
4  #include <cogent-llvm-defns.h>
```

Next, line 6 declares an `enum` containing the tags for all variants, and lines 8-17 declare various aggregate types used in the program.

```
6  typedef enum { EmptyBag, Failure, Success } tag_t;
7
8  typedef struct { u32 count; u32 sum; } t1;
9  typedef struct { u32 p1; t1* p2; } t2;
10 typedef t1*(*f3)(t2);
11 typedef struct { List_u32* p1; f3 p2; t1* p3; } t4;
12 typedef struct { Heap* p1; t1* p2; } t5;
13 typedef struct { t1* p1; Heap* p2; } t6;
14 typedef struct { tag_t tag; union { Heap* Failure; t6 Success; } val; } t7;
15 typedef struct { tag_t tag; union { unit_t EmptyBag; u32 Success; } val; } t8;
16 typedef struct { Heap* p1; List_u32* p2; } t9;
17 typedef struct { Heap* p1; u32 p2; } t10;
```

Lines 18-29 declare type aliases for each return and argument type, and line 30 declares a type alias for the `Bag` type.

```
18 typedef t10 average_ret;
19 typedef t9 average_arg;
20 typedef t8 averageBag_ret;
21 typedef t1* averageBag_arg;
22 typedef t1* addToBag_ret;
23 typedef t2 addToBag_arg;
24 typedef t7 newBag_ret;
25 typedef Heap* newBag_arg;
26 typedef Heap* freeBag_ret;
27 typedef t5 freeBag_arg;
28 typedef t1* reduce_0_ret;
29 typedef t4 reduce_0_arg;
30 typedef t1 Bag;
```

Lastly, lines 32-37 declare prototypes for each function in the program.

```
32 t10 average(t9);
33 t8 averageBag(t1*);
34 t1* addToBag(t2);
```

```c
t7 newBag(Heap*);
Heap* freeBag(t5);
t1* reduce_0(t4);

#endif
```

A.2.6 Programming with C

We can finally write some driver code in C that implements the abstract types, functions, and tests the COGENT program. We can first provide a concrete implementation of the `List` data structure using a linked list. In this case, we must provide an implementation specifically for `List U32`. `reduce` is implemented for the data type via simple linked list traversal, and we can make use of the type aliases for its argument and return types.

Listing A.5: Driver code to use our compiled functions (`main.c`)

```c
#include <cogent-llvm-defns.h>

struct List_u32 {
    struct List_u32 *next;
    u32 data;
};

typedef struct List_u32 List_u32;

reduce_0_ret reduce_0(reduce_0_arg args) {
    List_u32 *list = args.p1;
    t1 *acc = args.p3;

    t2 fargs;
    while (list) {
        fargs.p1 = list->data;
        fargs.p2 = acc;
        acc = (args.p2)(fargs);
        list = list->next;
    }

    return acc;
}
```

Next, we must implement the `Heap` and its `Bag` operations. The heap is just a void pointer to be passed around, and the operations use `malloc` and `free` internally.

```c
typedef void *Heap;

#include "bag.h"
#include <stdlib.h>

newBag_ret newBag(newBag_arg heap) {
    Bag *bag = malloc(sizeof(*bag));
    if (!bag)
        return (newBag_ret){Failure, .val.Failure = heap};

    bag->count = 0;
    bag->sum = 0;

    return (newBag_ret){Success, .val.Success = {bag, heap}};
}

freeBag_ret freeBag(freeBag_arg args) {
    free(args.p2);
    return args.p1;
}
```

A simple interactive program to test the `average` function might look like:

```c
int main() {
    List_u32 *list = NULL;
    u32 x = 0;

    while (1) {
        printf("Enter a number: ");
        if (scanf("%d", &x) < 0)
            break;

        List_u32 *cell = malloc(sizeof(*cell));
        cell->data = x;
        cell->next = list;
        list = cell;
    }

    average_ret avg_ret = average((average_arg){NULL, list});
    printf("The average is: %d\n", avg_ret.p2);

    return 0;
}
```

In our program, we refer to the type aliases from the generated header file (`average_arg`, `average_ret`). By using antiquoted C we could instead embed COGENT syntax directly in our C code, but this is not yet supported by the LLVM backend, as we use currently use a different method to generate type names in the header file.

To link this C code with the COGENT program using the LLVM compiler we can `llvm-as` and `clang` as follows:

```
cogent --llvm bag.cogent
llvm-as bag.ll -o bag.bc
clang -I../include main.c bag.bc -o main
```

At the time of writing this example can be located at `cogent/llvm/examples` in the COGENT repository, plus a few others.