Versatile and Flexible Modelling of the RISC-V Instruction Set Architecture*

Sören Tempel[1][0000−0002−3076−893X], Tobias Brandt[3][0000−0002−7041−4319], and Christoph Lüth[1,2][0000−0002−1121−398X]

[1] University of Bremen, 28359 Bremen, Germany tempel@uni-bremen.de
[2] Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), 28359 Bremen, Germany christoph.lueth@dfki.de
[3] tobbra91@gmail.com

Abstract. Formal languages are commonly used to model the semantics of instruction set architectures (*e.g.* ARM). The majority of prior work on these formal languages focuses on concrete instruction execution and validation tasks. We present a novel Haskell-based modelling approach which allows the creation of flexible and versatile architecture models based on free monads and a custom expression language. Contrary to existing work, our approach does not make any assumptions regarding the representation of memory and register values. This way, we can implement non-concrete software analysis techniques (*e.g.* symbolic execution where values are SMT expressions) on top of our model as interpreters for this model. In contrast to prior work, our modelling approach is therefore explicitly focused on the creation of custom ISA interpreters. We employ our outlined approach to create an abstract model and a concrete interpreter for the RISC-V base instruction set. Based on this model, we demonstrate that custom interpreters can be implemented with minimal effort using dynamic information flow tracking as a case study.

1 Introduction and Motivation

An instruction set architecture (ISA) describes the instructions of a processor, its state (number and types of registers), its memory, and more. It is the central interface between hard- and software, and as such of crucial importance; once fixed, it cannot be easily changed anymore. Traditionally, ISAs were specified in natural language, but that has been found lacking in exactness and completeness. Therefore, modelling an ISA, in particular a novel one, with formal languages has become *de rigeur*. Functional languages can be put to good use here: because of their declarative nature, we can formulate the behaviour at an abstract level which at the same time is executable.

Recently, the RISC-V ISA [15,16] has emerged has an attractive alternative to the prevailing industry standards, such as the Intel x86 or ARM architecture.

* Research supported by the German Federal Ministry of Education and Research (BMBF) under grant no. 01IW22002 (ECXL) and grant no. 16ME0127 (Scale4Edge).

It is open source, patent-free, and designed to be scalable from embedded devices to servers. Its open nature has sparked a lot of research activity, in particular many formal models of the ISA, including some in Haskell [3,12,17], or in custom domain-specific languages (DSLs) such as SAIL [1]. An *executable model* of the ISA is a simulator, *i.e.* software which simulates the behaviour of programs as faithful to the hardware as possible. As such, an ISA simulator is essentially an *interpreter* for machine code (*i.e.* software in binary form).

Our contribution as presented here is a highly flexible and versatile model of the RISC-V ISA in Haskell. As opposed to existing models, the interpretation of the ISA can be varied. To this end, we define an embedded domain-specific language (EDSL) via a free monad construction. The idea is that the free monad models the computation given by a sequence of operations from the ISA, where the model of computation (*i.e.* the interpretation) can be varied, from simple state transitions which simulate the ISA faithfully, to sophisticated analyses such as symbolic execution [2] or dynamic information flow tracking [21]. While prior work on formal ISA models focuses largely on validation tasks, our model is specifically centered around the implementation of custom interpreters. To the best of our knowledge, our approach is therefore the first which enables the creation of software analysis tools (e.g. symbolic execution) as interpreters for the formal ISA model. By building these tools on top of an abstract model, we can (1) easily extend the analysis to additional instructions,[4] (2) analyse software written in any programming language that compiles to machine-code for the modelled ISA, and (3) potentially ease proofing the correctness of these analyses tools by leveraging existing proof-assistant definitions for ISA semantics. Our work is motivated by our experience with `riscv-vp` [7], an existing RISC-V simulator written in C++ using SystemC TLM [23]. After having to modify `riscv-vp` repeatedly to allow such analyses [13,25,26], we were looking for a more systematic and structured way to achieve this flexibility. The case study we have conducted with an exemplary implementation of dynamic information flow tracking for RISC-V machine code illustrates the feasibility of our approach for this purpose. Furthermore, performed experiments indicate that interpreters based on our formal model are able to compete with `riscv-vp` in terms of simulation speed.

The remainder of article is structured as follows: we first provide background information on instruction set architectures and the free monad abstraction. We then demonstrate how to model a very simple ISA to motivate our model of the real RISC-V ISA which we present in Sect. 4 and leverage to implement a custom interpreter as a case study. In Sect. 5 we evaluate the performance (*i.e.* simulation speed) of our implementation, and in Sect. 6 we compare our approach to related work. Lastly, we discuss opportunities for future work and provide a conclusion.

[4] This is paramount for modular ISAs (like RISC-V) where different instruction set extensions can be combined, thereby, requiring the analysis tool to support them.

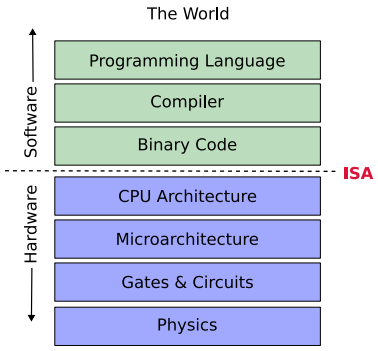

Fig. 1. Relation of the ISA to other software and hardware abstraction layers

2 Preliminaries

In the following, we provide background information on instruction set architectures and the free monad abstraction as a prerequisite for the following sections.

2.1 Instruction Set Architectures

As illustrated in Fig. 1, the ISA is the central interface between the hard- and software and conceptually forms the boundary between the two. In order to allow the software to interact with the hardware, the ISA specifies the instruction encoding for binary code and the semantics of said instructions. Software in high-level programming languages (such as C++ or Haskell) is translated by a compiler to binary code which then uses the instructions of a specific ISA (*e.g.* x86, ARM, or RISC-V). These instructions are then implemented in hardware by the CPU. As software is commonly loaded into memory, the CPU must fetch the next instruction from memory and decode it before executing it. This process is commonly referred to as the *fetch-decode-execute cycle* [20, Sect. 14.3].

Different instruction set architectures exist. In this paper we are generally focusing on load-store architectures where operands to instructions are registers. That is, operations on memory values can only be performed by first loading them into registers, performing an operation on them, and then storing them in memory again [20, Sect. 14.6]. Load-store data processing is widely used by so-called reduced instruction set computer (RISC) architectures (*e.g.* RISC-V) which are focused on simplicity of standardized instructions [20, Sect. 15.4].

An ISA is in essence a low-level imperative programming language with predefined bit-vector data types (words of given length). In order to execute or analyse software in binary form (*e.g.* in a simulator or a dynamic binary software analysis tool), one needs to implement an interpreter for the ISA. Ideally, such an implementation should be flexible in the sense that it can be re-used for different execution and analysis tasks without having to re-implement the entirety of the ISA each time. Prior work on imperative programs has leveraged monadic abstractions for this purpose; more about this in the following section.

2.2 Free Monads

The semantics of imperative programs has many aspects (stateful computations, continuations, exception) each of which can be modelled in Haskell using monads; combining these monads is a notoriously tricky exercise. Early work on interpreting imperative programs used monad transformers for this effect [11], but more recent work uses free monads for better performance and extensibility (see Subsect. 6.2 for a detailed comparison); we sketch the basic concepts here.

A free monad for a type constructor `f` is essentially the closure of `f` under application (it contains arbitrarily many applications of `f`); the appeal is that we can define the monad separately for each constructor of `f`, allowing to write EDSLs in a modular way. The category-theoretic construction of free monads was given by Kelly [8], and first described in the context of functional programming by Swierstra [22]. In its simplest form, the free monad is given as:

```
data Free f a = Pure a | Free (f (Free f a))
```

In our implementation, we use an enhanced version of this concept. Further details on the exact implementation are provided in the appendix (Sect. A).

3 Modelling an ISA

We motivate our approach and its advantages by applying it to a very simple ISA. The ISA implements a 32-bit load-store architecture with five instructions; each of these can be thought of as representing a class of similar instructions in a real ISA:

1. `LOADI` *imm reg*: Load immediate into register *reg*.
2. `ADD` *dst src1 src2*: Add two registers into *dst*.
3. `LW` *dest addr*: Load word from memory at *addr* into register *dest*.
4. `SW` *addr src*: Store word from register *src* into memory at *addr*.
5. `BEQ` *reg1 reg2 off*: Relative branch by *off* if registers *reg1* and *reg2* are equal.

The ISA supports 16 general-purpose registers, word-addressable memory, and a program counter which points to the current instruction in memory. All registers and memory values are 32-bit wide and treated as signed values by all instructions. Instruction fetching and decoding is not discussed. The instruction set is modelled straightforward as a Haskell data type (where `Word` and `Addr` are type synonyms for 32-bit integers):

```
newtype Reg = Reg { reg :: Int } deriving (Ord, Eq)
data INSTR
    = LOADI Word Reg
    | ADD Reg Reg Reg
    | LW Reg Addr
    | SW Addr Reg
    | BEQ Reg Reg Word
```

```
type System = (Registers
               , Mem
               , ProgramCounter)


execute :: INSTR → State System ()
execute i = modify $
  λ(regs, mem, pc) → case i of
    LOADI imm r → (insert r imm regs,
                   mem, nextInstr pc)
    ADD rd rs1 rs2 → let
         v1 = regs ! rs1
         v2 = regs ! rs2
        in  (insert rd (v1+v2) regs,
             mem, nextInstr pc)
    LW r addr → let
         w = mem ! addr
        in  (insert r w regs, mem,
             nextInstr pc)
    SW addr r → let
         v = regs ! r
        in (regs, insert addr v mem,
            nextInstr pc)
    BEQ r1 r2 off → let
         v1 = regs ! r1
         v2 = regs ! r2
         br = if v1 == v2
           then pc+off
           else nextInstr pc
        in (regs, mem, br)
```

Listing 1.1. Concrete Haskell model

```
type System' = (Registers
                , Mem
                , ProgramCounter
                , Int)

execute' :: INSTR → State System' ()
execute' i = modify $
  λ(regs, mem, pc, counter) → case i of
    LOADI imm r → (insert r imm regs, mem,
                   nextInstr pc, counter)
    ADD rd rs1 rs2 → let
         v1 = regs ! rs1
         v2 = regs ! rs2
        in (insert rd (v1+v2) regs,
            mem, nextInstr pc, counter)
    LW r addr → let
         w = mem ! addr
        in (insert r w regs, mem,
            nextInstr pc, succ counter)
    SW addr r → let
         v = regs ! r
        in (regs, insert addr v mem,
            nextInstr pc, succ counter)
    BEQ r1 r2 off → let
         v1 = regs ! r1
         v2 = regs ! r2
         br = if v1 == v2
           then pc+off
           else nextInstr pc
        in (regs, mem, br, counter)
```

Listing 1.2. Memory accesses analysis

3.1 A First Model

The execution model formally describes how instructions are executed. It specifies the system state, and how instructions change the system state (including the control flow).

Listing 1.1 provides a simple Haskell execution model for our exemplary ISA. The architectural state `System`, upon which instructions are executed, is a tuple consisting of two finite maps for the memory and register file as well as a concrete program counter. Instruction execution itself is implemented as a pure function which performs a pattern match on the instruction type and returns a new system state, embedded into a state monad (`State System` α).

Unfortunately, this simple ISA model has several shortcomings. Consider a simple software analysis task for which we want to extend our model to track the number of memory accesses during program execution. For this, we merely need to extend the system state with an access counter, and increment the counter whenever memory access takes place (operations `LW` and `SW`). A possible implementation of this modification is shown in Listing 1.2. Note how, even though our extension to the previous solution did not modify the control flow of the program in any way, we still had to restate the control flow for all supported instructions of our ISA. For our small ISA this inconvenience seem feasible, but considering that a real ISA has often more than 80 instructions, the task of modifying the execution becomes cumbersome and error-prone.

Hence, our aim is to give a modular, abstract representation of ISA semantics, based upon which we can then implement software analysis techniques which require a different kind of interpretation with minimal effort. Such techniques may include symbolic execution [2] or dynamic information flow tracking [21].

3.2 Our Approach

The problem with the previously outlined approach is that the model of the semantics (a state transition given by a state monad) is given in a very concrete and monolithic form: there is no separation between the different aspects of the semantics. However, the semantics of an ISA has several aspects: memory access, register access, arithmetic, and control flow, and most analyses only concern one or two of them (*e.g.* memory access, or arithmetic). Yet, if we want to change the representation of the state, this affects all operations; similarly if we want to reason about *e.g.* integer arithmetic to show absence of integer overflow, we need to re-implement all operations.

Thus, we want to give the semantics of our ISA by combining several constituting parts, which we can change individually. To this end, we define an EDSL which represents the operations of an abstract machine implementing the ISA, *e.g.* loading and storing words into registers, using a free monad as introduced in Subsect. 2.2.

```
data Operations r
    = LoadRegister Reg (Word → r)
    | StoreRegister Reg Word r
    | IncrementPC Word r
    | LoadMem Addr (Word → r)
    | StoreMem Addr Word r
    deriving Functor

loadRegister :: Reg → Free Operations Word
loadRegister r = Free (LoadRegister r Pure)

storeRegister :: Reg → Word → Free Operations ()
storeRegister r w = Free (StoreRegister r w (Pure ()))

incrementPC :: Word → Free Operations ()
incrementPC v = Free (IncrementPC v (Pure ()))

loadMem :: Addr → Free Operations Word
loadMem addr = Free (LoadMem addr Pure)

storeMem :: Addr → Word → Free Operations ()
storeMem addr w = Free (StoreMem addr w (Pure ()))
```

Listing 1.3. EDSL of the machine executing the ISA

The operations comprising the EDSL are given by a parameterized data type `Operations`, see Listing 1.3[5]. The `Operations` data type models the ISA in abstract terms; the free monad `Free Operations` describes combinations of these, which are an abstract representation of the control flow of a (sequence of) ISA operations. This representation is given by a function `controlFlow :: INSTR → Free Operations ()`, which defines the control flow for a given instruction (see Listing 1.4); by composing these we get the control flow for a program (sequence of operations).

```
controlFlow :: INSTR → Free Operations ()
controlFlow = λcase
    LOADI imm r → storeRegister r imm ≫ incrementPC instrSize
    ADD rd r1 r2 → do
        v1 ← loadRegister r1
        v2 ← loadRegister r2
        storeRegister rd (v1+v2)
        incrementPC instrSize
    LW r addr → do
        v ← loadMem addr
        storeRegister r v
        incrementPC instrSize
    SW addr r → do
        v ← loadRegister r
        storeMem addr v
        incrementPC instrSize
    BEQ r1 r2 off → do
        v1 ← loadRegister r1
        v2 ← loadRegister r2
        if v1 == v2 then incrementPC off else incrementPC instrSize
```

Listing 1.4. Interpreting an in ISA instruction in the free monad.

To reconstruct the concrete execution of the ISA instructions from the previous section (Listing 1.1), we need to map the operations in the free monad to concrete monadic effects, in our case in Haskell's pure `State` monad. An example implementation of a function which performs this mapping is provided in Listing 1.5.

```
execute :: State → Free Operations () → State
execute st = flip execState st ∘ iterM go where
    go = λcase
        LoadRegister reg f → gets (λ(rs,_,_) → rs ! reg) ≫= f
        StoreRegister reg w c →
            modify (λ(rs, mem, pc) → (insert reg w rs, mem, pc)) ≫ c
        IncrementPC w c → modify (λ(rs,mem,pc) → (rs,mem,pc+w)) ≫ c
        LoadMem addr f  → gets (λ(_,mem,_) → mem ! addr) ≫= f
        StoreMem addr w c →
            modify (λ(rs,mem,pc) → (rs, insert addr w mem, pc)) ≫ c
```

Listing 1.5. Evaluating the control flow using the `State` monad

[5] For convenience, we add a smart-constructor for each constructor of the data type.

Since we have now separated control flow and semantics of effects, we could also use any other (monadic) effects for the evaluation without changing the control flow. Reconstructing the example from Listing 1.2 just requires adjustments in the semantics without restating the control flow as shown in Listing 1.6 (performed adjustments are highlighted using red text color).

```
execute' :: State" → Free Operations () → State"
execute' st = flip execState st ∘ iterM go where
    go = λcase
        LoadRegister reg f → gets (λ(rs,_,_,_) → rs ! reg) ≫= f
        StoreRegister reg w c → modify
            (λ(rs, mem, pc, counter) →
                (insert reg w rs, mem, pc, counter)) ≫ c
        IncrementPC w c → modify
            (λ(rs, mem, pc, counter) → (rs,mem,pc+w,counter)) ≫ c
        LoadMem addr f  → do
            v ← gets (λ(_,mem,_, counter) → mem ! addr)
            modify (λ(rs,mem,pc,counter) → (rs, mem, pc, succ counter))
            f v
        StoreMem addr w c →
            modify (λ(rs,mem,pc,counter) →
                (rs, insert addr w mem, pc, succ counter)) ≫ c
```

Listing 1.6. Executing and counting memory accesses

While this is a major advantage in terms of reusability, there is still room for improvement. In particular, we are not able to change the semantics of the expression-level calculations an operation performs, since the data type of our EDSL assumes concrete types, which entails they are already evaluated. Hence, we generalize our `Operations` to allow a representation of the evaluation of expressions, much like we did for the instructions (except that the evaluation of expressions is not monadic, hence we do not need a free monad here). For that, we need to introduce a simple expression language, which will replace all the constant values, *e.g.* the constructor `StoreRegister :: Reg → Word → r` becomes `StoreRegister' :: Reg → Expr w → r`, as well as adjust the `Operations` type such that it becomes polymorphic in the word type.

Listing 1.7 shows the changes necessary, *e.g.* the `execute'''` function is now provided with an expression-interpreter `evalE`, which is used to evaluate expressions generated by the control flow. The `Operations` are now polymorphic in the word-type and the semantics of the internal computations can be changed by adjusting `evalE`; this allows our approach to be used to implement various software analysis techniques on the ISA level. In the next section, we will present an application of our approach to the RISC-V ISA, and utilize the resulting RISC-V model to implement one exemplary software analysis technique as a case study.

```
data Expr a = Val a | Add (Expr a) (Expr a) | Eq (Expr a) (Expr a)

data Operations' w r
    = LoadRegister' Reg (Expr w → r)
    | StoreRegister' Reg (Expr w) r
```

```
    | IncrementPC' (Expr w) r
    | LoadMem' Addr (Expr w → r)
    | StoreMem' Addr (Expr w) r

evalE :: Expr Word → Word
evalE = λcase
    Val a → a
    Add e e' → evalE e + evalE e'
    Eq e e' → if evalE e' == evalE e then 1 else 0

execute''' :: (Expr Word → Word) → State''
              → Free (Operations' Word) () → State''
execute''' evalE st = flip execState st ∘ iterM go where
    go = λcase
        LoadRegister' reg f → gets (λ(rs,_,_,_) → Val $ rs ! reg) ≫= f
        StoreRegister' reg w c → modify
            (λ(rs, mem, pc, counter) →
                (insert reg (evalE w) rs, mem, pc, counter)) ≫ c
        IncrementPC' w c → modify
            (λ(rs,mem,pc,counter) → (rs,mem,pc+ evalE w, counter)) ≫ c
        LoadMem' addr f → do
            v ← gets (λ(_,mem,_, counter) → mem ! addr)
            modify (λ(rs,mem,pc,counter) → (rs, mem, pc, succ counter))
            f $ Val v
        StoreMem' addr w c → modify (λ(rs,mem,pc,counter)
            → (rs, insert addr (evalE w) mem, pc, succ counter)) ≫ c
```

Listing 1.7. `Operations` type with simple expression language

4 Modelling the RISC-V ISA

As an application of our approach, we created an abstract model of the RISC-V ISA. RISC-V is an emerging RISC architecture which has recently gained traction in both academia and industry. Contrary to existing ISAs, RISC-V is developed as an open standard free from patents and royalties. It is designed in a modular way: the architecture consists of a base instructions set and optional extensions (*e.g.* for atomic instructions) which can be combined as needed [15].

We refer to our model of the RISC-V architecture as LibRISCV. As the name suggests, LibRISCV is a Haskell library which can be used to implement different interpreters for RISC-V software. As such, the library provides an instantiable framework for versatile interpretation of RISC-V software in binary form. Fig. 2 illustrates how the concepts from Subsect. 3.2 are applied to RISC-V in order to achieve versatile interpretation. The figure will be further described in the following subsections.

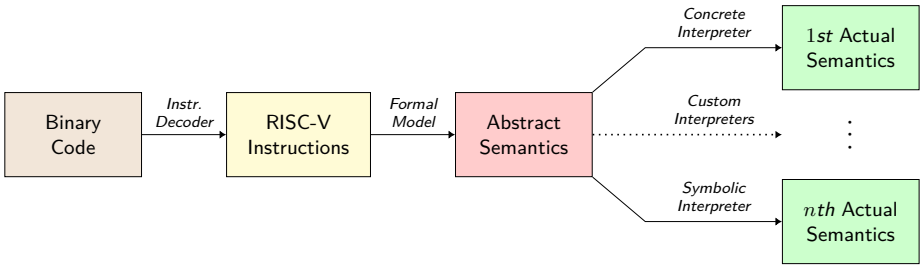

Fig. 2. Application of our ISA modelling approach to the RISC-V architecture

4.1 Instruction Decoder

As depicted in Fig. 2, our RISC-V implementation receives binary code as an input value. This binary code constitutes RISC-V machine code and is converted to an algebraic data structure representing instructions mandated by the RISC-V standard using an instruction decoder. Contrary to imperative programming languages, execution and decoding/parsing is heavily intertwined for machine code. As discussed in Subsect. 2.1, we can only decode the next instruction after finishing execution of the current instruction. For example, when executing a branch instruction the next fetched instruction depends on the result of the branch. We make use of lazy evaluation to model the fetch-decode-execute cycle as part of our control flow description. That is, the fetching of the next instruction is itself—non-strictly—modelled, using free monads as outlined in Subsect. 3.2.

Contrary to existing work, a description of RISC-V instruction decoding is not part of our EDSL. Instead, the LibRISCV instruction decoder is automatically generated from `riscv-opcodes`[6], an existing formal language which describes how binary code is mapped to RISC-V instructions (without modelling instruction semantics). Based on algebraic data types, returned by the instruction decoder, we specify the *abstract semantics* of RISC-V instructions through a formal ISA model described in the following.

4.2 Formal Model

An overview of the ISA model provided by LibRISCV is available in Fig. 3. As illustrated in Fig. 2, the central component of the formal model is the description of the abstract instruction semantics which represents the lazily-generated control flow of the RISC-V ISA operations. As discussed in Subsect. 3.2, we use free monads for this purpose. For the implementation of free monads, we use the `freer-simple` library[7]. The library provides an improved implementation of the free monad approach described in the appendix (Sect. A). Within the abstract description of instruction semantics, all operations on register/memory

[6] `https://github.com/riscv/riscv-opcodes`
[7] `https://hackage.haskell.org/package/freer-simple`

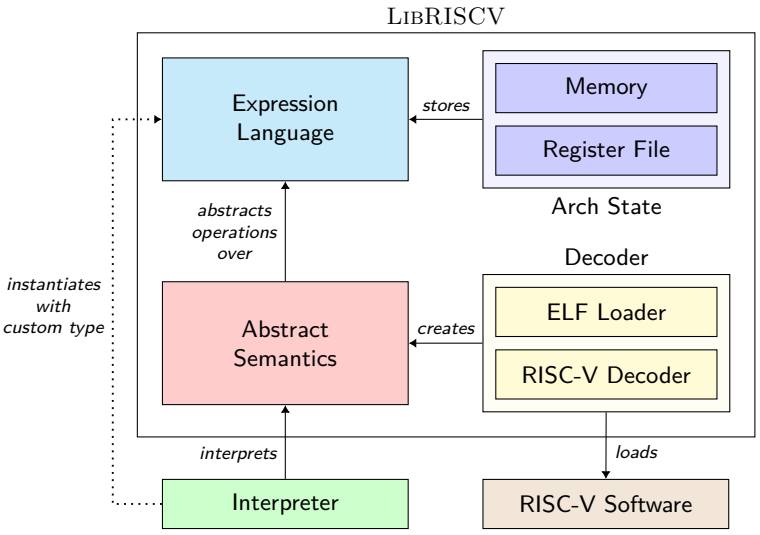

Fig. 3. Overview of the RISC-V ISA model provided by LibRISCV

values are abstracted using a generic expression language. The expression language is implemented as an algebraic data type with an associated evaluation function, as illustrated in Listing 1.7. The algebraic data type, used by the expression language, is parameterized over a custom type. The architectural state (*i.e.* memory and register file) is also parameterized over this type. As shown in Fig. 3, the abstract description of instruction semantics is based on an instruction type which is generated by the aforementioned instruction decoder. The decoder is responsible for loading RISC-V software in the Executable and Linkable Format (ELF) and for decoding/parsing instruction words—contained in the file—according to the RISC-V specification.

Based on the abstract semantics, we can provide different interpreters which implement the *actual semantics* for decoded RISC-V instructions as illustrated in Fig. 2, such as concrete or symbolic execution of modelled instructions. The actual semantics implement the state transition for each modelled instruction while the abstract semantics only describe the control flow. Each interpreter instantiates the expression language with a type. Based on this type, an interpreter for the formal ISA model (*i.e.* the expression language and the free `Operations` monad) needs to be supplied. Presently, LibRISCV provides a formal model for the 32-bit variant of the RISC-V base instruction set (40 instructions). Based on this formal model, we have implemented a concrete interpreter for RISC-V instructions. Both the model and the concrete interpreter are written in roughly 1500 LOC and can be obtained from GitHub[8]. Using the concrete interpreter, we were able to successfully execute and pass the official RISC-V ISA tests for the

[8] `https://github.com/agra-uni-bremen/libriscv`

32-bit base instruction set[9]. These tests include multiple test programs (one for each instruction) which check if the implemented behavior of an instruction conforms to the specification. Passing these tests indicates that our model correctly captures the semantics of the base instruction set. In the following, we illustrate how custom interpreters—beyond the standard concrete interpretation—can be implemented on top of our abstract model, thereby making use of its flexibility.

4.3 Custom Interpreters

Our model of the RISC-V ISA is designed for maximum flexibility and versatility, along the lines sketched in Subsect. 3.2. This allows implementing different interpretations of the ISA on top of our abstract model with minimal effort. Conceptually, each custom interpreter implements actual semantics for the abstract semantics provided by the formal ISA model (see Fig. 2). In order to implement a custom RISC-V interpreter, an evaluator for the expression language and an interpreter for the free `Operations` monad need to be provided. As an example, dynamic information flow tracking [21], where data-flow from input to output is analysed, can be implemented using the following polymorphic data type:

```
data Tainted a = MkTainted Bool a

instance Conversion (Tainted a) a where
    convert (MkTainted _ v) = v
```

The product type `Tainted` tracks whether a value of type `a` is subject to data-flow analysis. Furthermore, a conversion to `Word32` is implemented through an instance-declaration for the `Tainted` type. This conversion is the only class constraint imposed by our abstract model on the type used by the custom interpreter.[10] An evaluator of the expression language for `Tainted Word32` can be implemented as follows:[11]

```
evalE :: Expr (Tainted Word32) → Tainted Word32
evalE (FromImm t)  = t
evalE (FromInt i)  = MkTainted False $ fromIntegral i
evalE (AddU e1 e2) = MkTainted (t1 || t2) $ v1 + v2
    where (MkTainted t1 v1) = evalE e1; (MkTainted t2 v2) = evalE e2
```

The evaluator performs standard concrete integer arithmetic on the `Word32` encapsulated within the `Tainted` type. However, if one of the operands of the arithmetic operations is a tainted value, then the resulting value is also tainted. This enables a simple data-flow analysis for initially tainted values. Based on the evaluation function, an interpretation of the control flow is shown in the following, where $f \rightsquigarrow g$ denotes a natural transformation from f to g (as provided by the `freer-simple` library):

[9] `https://github.com/riscv/riscv-tests`
[10] This constraint is necessary as the instruction decoder operates on `Word32` values.
[11] The `FromImm`, `FromInt`, and `AddU` constructors belong to our expression abstraction.

```
type ArchState = ( REG.RegisterFile IOArray (Tainted Word32)
                 , MEM.Memory IOArray (Tainted Word8))

type IftEnv = (Expr (Tainted Word32) → Tainted Word32, ArchState)

iftBehaviour :: IftEnv → Free Operations (Tainted Word32) ⇝ IO
iftBehaviour (evalE , (regFile, mem)) = λcase
    (ReadRegister idx) → REG.readRegister regFile idx
    (WriteRegister idx reg) → REG.writeRegister regFile idx (evalE reg)
    (LoadWord addr) → MEM.loadWord mem (convert $ evalE addr)
    (StoreWord addr w) → MEM.storeWord mem (convert $ evalE addr)
                                           (evalE w)
```

This function operates on a polymorphic register and memory implementation. Expressions are evaluated using `evalE`, and then written to the register file or memory. When execution terminates, we can inspect each register and memory value to check whether it depends on an initially tainted input value. As shown, the interpreter only implements a subset of the `Operations` monad and the expression language; a complete implementation is provided in the `example/` subdirectory on GitHub. We have already implemented dynamic information flow tracking for RISC-V machine code in prior work based on the `riscv-vp` simulator mentioned in Sect. 1 [13]. For this prior implementation, we had to modify `riscv-vp` extensively to allow for such an analysis to be performed, as it does not separate instruction semantics from instruction execution. In this context, the case study provided here serves to demonstrate that such techniques can be more easily implemented on top of an abstract formal model as custom interpreters for this model.

5 Performance Evaluation

Free monads introduce a well-known performance problem [9, Sect. 2.6] (see Sect. A). As our approach is focused on implementing interpreters, simulation performance is important when executing real-world software. To evaluate simulation speed, we conduct a comparison with existing RISC-V simulators and specifically quantify the impact of the utilized `freer-simple` library on simulation performance. For this purpose, we leverage the existing Embench 1.0 benchmark suite [6]. Embench contains several benchmark applications which perform different computation-intensive tasks (*e.g.* checksum calculation). We compiled all applications for the 32-bit RISC-V base instruction set, executed them with different RISC-V simulators, and measured execution time in seconds. The results are shown in Table 1. All experiments have been conducted on an Intel Xeon Gold 6240 running an Alpine Linux 3.17 Docker image. Artefacts for the performed evaluation are available on Zenodo [24].

For each benchmark application in Table 1, we list the execution time in seconds for different RISC-V simulators. In order to specifically quantify the performance impact of the `freer-simple` library, we use a modified version of LibRISCV as a baseline where we manually removed the dependency on

Table 1. Execution time comparison in seconds with existing RISC-V simulators

Benchmark	baseline	libriscv	forvis	grift	riscv-vp
aha-mont64	21.68	41.32	53.81	351.85	14.15
crc32	8.71	16.61	21.08	148.69	5.75
cubic	28.80	57.99	71.90	614.11	19.20
edn	80.16	160.36	193.93	1 680.24	53.62
huffbench	8.31	15.41	20.18	108.62	5.60
matmult-int	41.71	82.72	96.94	820.24	28.07
minver	13.87	26.93	33.87	272.13	9.16
nbody	24.55	48.85	58.78	529.97	16.50
nettle-aes	8.91	16.19	19.99	118.77	5.93
nettle-sha256	6.94	12.50	15.68	89.82	4.43
nsichneu	4.19	7.63	9.30	59.18	2.79
picojpeg	13.99	26.30	38.66	203.55	9.73
qrduino	11.85	23.33	30.91	200.08	8.52
sglib-combined	7.94	14.33	18.52	106.38	5.24
slre	6.82	12.56	15.88	91.89	4.55
st	16.18	32.48	38.65	344.91	10.94
statemate	1.69	3.26	5.20	23.68	1.39
ud	14.44	27.10	33.47	222.35	9.42
wikisort	7.57	14.49	18.32	136.27	5.09
Geometric mean	12.07	23.02	29.37	197.96	8.16

`freer-simple` and evaluate the ISA directly in Haskell's `IO`-monad. As such, this baseline version is conceptually similar to the primitive model presented in Subsect. 3.1, *i.e.* the interpretation cannot be varied and it unconditionally performs concrete execution of RISC-V instructions. To contextualize the obtained results, we performed further experiments with existing Haskell implementations of the RISC-V ISA, namely Forvis [3] and GRIFT [17]. Contrary to our own work, these implementations do not utilize free monads (see Sect. 6). Lastly, Table 1 also contains evaluation results for the aforementioned `riscv-vp`, which is written in the C++ programming language [7]. To summarize benchmark results, Table 1 provides the geometric mean on a per-simulator basis in the bottom row.

Naturally, the C++ implementation (`riscv-vp`) has the lowest execution time over all benchmark applications. On average, it is roughly three times faster than our own Haskell implementation of the RISC-V ISA (LibRISCV). This is to be expected as, contrary to Haskell, C++ is not garbage collected. Nonetheless, and despite the employment of free monads, LibRISCV is—on average—still faster than Forvis and GRIFT. While LibRISCV and Forvis have similar execution time results, GRIFT is significantly slower even though it is also written in Haskell. We attribute this to the fact that GRIFT employs a bit-vector expression language, as an additional abstraction layer, to perform operations on register/memory values. The performance impact of the free monad abstraction (used in LibRISCV) can be estimated by comparing simulation performance with the

baseline column in Table 1. As discussed above, the baseline column represents execution time for a LibRISCV variant which does not use the `freer-simple` library. The gathered data indicates that LibRISCV is two times slower than the baseline version, confirming that free monads have a significant impact on simulation performance. Nonetheless, LibRISCV is still faster than existing Haskell implementations (Forvis and GRIFT) and approximately only three times slower than a primitive C++ implementation (`riscv-vp`). As such, we believe the induced performance penalty to be acceptable for our use case as the advantages of free monads outweigh this disadvantage by far.

6 Related Work

In the following, we discuss related work on formal ISA semantics, modular interpreters for imperative programming languages, and software analysis tools.

6.1 Formal Specifications

Formal semantics for ISAs is an active research area with a vast body of existing research. Specifically regarding RISC-V, a public review of existing formal specifications has been conducted by the RISC-V foundation in 2019 [14]. From this review, SAIL [1] emerged as the official formal specification for the RISC-V architecture. SAIL is a custom DSL for describing different ISAs and comes with tooling for automatically generating simulators from this description. However, we believe a functional specification in a programming language like Haskell to be more suitable for rapid prototyping of custom interpreters. Similar to our own work, existing work on GRIFT [17], Forvis [3], and riscv-semantics [12] models the RISC-V ISA using a Haskell EDSL. Forvis and riscv-semantics are explicitly designed for readability and thus only use a subset of Haskell. As opposed to our own work, instructions are executed directly and this prior work does not separate the description of instruction semantics from their execution. In this regard, GRIFT is closer to our own work as it uses a bit-vector expression language to provide a separate description of instruction semantics. However, GRIFT's expression language is designed around natural numbers as it focuses on concrete execution. For this reason, it is not possible to represent register/memory values abstractly using GRIFT (*i.e.* not as natural numbers, but for example as SMT expressions). To the best of our knowledge, our formal RISC-V model is the first executable model which focuses specifically on flexibility and thereby enables non-concrete execution of RISC-V instructions.

6.2 Modular Interpreters

Early work on modular interpreters for imperative languages [11] used *monad transformers* to compose the monads used to interpret the imperative features in a modular way. Monad transformers can be thought of as monads with a hole; instead of a monad `m` modelling a feature f (say, stateful computation), we give a

monad transformer `m'` modelling the addition of feature f to an existing monad. This allows us to combine features in a "stack" of monads, and is implemented in Haskell in the `mtl` library[12].

However, this approach has three drawbacks: firstly, the monad transformer already specifies the interaction with the other monad, so the approach is not truly compositional; secondly, it is not truly extensible, as once the monad stack is composed, no more monads can be added (this would result in a new monad stack); and thirdly, there is a severe performance cost for larger monad stacks [9, Sect. 4]. For these reasons, we use free monads with extensible effects which do not suffer from these drawbacks as explained in Sect. A, even though lowering the performance penalty of free monads (*cf.* Sect. 5) is still an open challenge.

Our work is intended as a framework for abstract interpretation on machine code. Leveraging monads for this purpose, it is related to work on monadic abstract interpreters [18]. Besides the use of monad transformers in that work, there is one crucial difference: for software, abstract interpretation techniques extract the control flow graph (CFG) of the program ([18] uses continuation-passing style semantics for this). We model the control flow implicitly using lazy evaluation; the next instruction is only fetched and decoded once it is needed.

6.3 Binary Software Analysis

Due to the utilization of free monads, we believe our RISC-V ISA model to be a versatile tool for implementing dynamic software analysis techniques that operate directly on the machine code level. Prior work has already demonstrated that it is feasible to implement techniques such as symbolic execution [26] or dynamic information flow tracking [13] for RISC-V machine code. However, this prior work does not leverage functional ISA specifications and thus relies on manual modifications of existing interpreters and is not easily applicable to additional RISC-V extensions or other ISAs (ARM, MIPS, ...). For this reason, the majority of existing work on binary software analysis does not operate directly on the machine code level and instead leverages intermediate languages and lifts machine code to these languages [4,5,19].

This prior work therefore operates on a higher abstraction level and can thus not reason about architecture-specific details (*e.g.* instruction clock cycles) during the analysis. By building dynamic software analysis tools on an abstract ISA model, we can bridge the gap between the two approaches; we can operate directly on the machine code level while still making it easy to extend the analysis to additional instructions or architectures. This is especially important for modular ISAs like RISC-V.

7 Discussion and Future Work

So far, we have only applied our approach to the RISC-V architecture. Nonetheless, we believe the concepts described in Subsect. 3.2 to be applicable to other

[12] `https://hackage.haskell.org/package/mtl`

architectures as well. We have, focused on the RISC-V architecture due to its simplicity as we consider the main contribution of this paper to be the implementation of custom interpreters on top of a formal ISA model. A possible direction for future work would be focusing more on modelling aspects by supporting additional RISC-V extensions (especially from the privileged specification [16]), further RISC-V variants (e.g. 64- and 128-bit RISC-V), and maybe even additional ISAs (*e.g.* ARM). Alternatively, it would also be possible to perform further experiments with additional interpreters for our abstract ISA model. We are specifically interested in complementing our own prior work on symbolic execution of RISC-V machine code by implementing it on top of the formal ISA model proposed here, thereby making it easier to extend this prior work to additional RISC-V extension or even additional architectures [26]. More broadly, one end goal of our work in this regard would be facilitating formal ISA models for the implementation of binary software analysis tools along the lines sketched in Subsect. 6.3. Compared to the prevailing prior work on binary software analysis tools—which lifts machine code to an intermediate representation—we believe that building these tools on top of a formal ISA model also allows easier proofs of their correctness. An interesting direction for future work would therefore be investigating the issue of correctness of custom ISA interpreter. As illustrated in Fig. 2, correctness proofs are paramount as we need to ensure that both the abstract and the actual semantics correctly implement the behaviour mandated by the modelled ISA. Considering that our approach is specifically designed to support multiple actual semantics—through custom interpreters—manual validation is infeasible. Instead, it may be possible to leverage existing proof-assistant definitions for ISAs [1] to prove the correctness of created ISA interpreters through computer-aided theorem proving.

8 Conclusion

We have presented a flexible approach for creating functional formal models of instruction set architectures. The functional paradigm gives a natural and concise way to model the instruction format on different levels of abstraction, and the structuring mechanisms allow us to relate these levels. This way, by leveraging free monads, our approach separates instruction semantics from instruction execution. Contrary to prior work, our approach does not make any assumption about the representation of memory/register values. Therefore, it can be used to implement software analysis techniques such as dynamic information flow tracking or symbolic execution; achieving the benefits outlined in Sect. 1.

We have demonstrated our approach by creating an abstract formal model of the RISC-V architecture. Based on this formal RISC-V model, we have created a concrete interpreter—which passes the official RISC-V ISA tests—for the 32-bit base instruction set and a custom interpreter for information flow tracking as a case study. An evaluation conducted with the Embench benchmark suite indicates that our concrete interpreter is faster than prior executable Haskell models of the RISC-V architecture. In future work, we would like to model

additional extensions of the RISC-V architecture, perform further experiments with additional interpreters for our model, and investigate correctness proofs for these interpreters through computer-aided theorem proving. To stimulate further research in this direction, we have released our formal RISC-V model as open source software on GitHub.

References

1. Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K.E., Norton, R.M., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami, N., Sewell, P.: ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290384
2. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3) (May 2018). https://doi.org/10.1145/3182657
3. Bluespec, Inc.: Forvis: A formal RISC-V ISA specification. GitHub, https://github.com/rsnikhil/Forvis_RISCV-ISA-Spec, accessed 2022-12-06
4. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. pp. 463–469. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_37
5. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A platform for in-vivo multi-path analysis of software systems. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. p. 265–278. ASPLOS XVI, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1950365.1950396
6. Free and Open Source Silicon Foundation: Embench: A modern embedded benchmark suite, https://www.embench.org/, accessed 2023-01-24
7. Herdt, V., Große, D., Pieper, P., Drechsler, R.: RISC-V based virtual prototype: An extensible and configurable platform for the system-level. Journal of Systems Architecture **109**, 101756 (2020). https://doi.org/10.1016/j.sysarc.2020.101756
8. Kelly, G.M.: A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves, and so on. Bulletin of the Australian Mathematical Society **22**(1), 1–83 (Aug 1980). https://doi.org/10.1017/S0004972700006353, publisher: Cambridge University Press
9. Kiselyov, O., Ishii, H.: Freer monads, more extensible effects. SIGPLAN Not. **50**(12), 94–105 (Aug 2015). https://doi.org/10.1145/2887747.2804319
10. Kiselyov, O., Sabry, A., Swords, C.: Extensible effects an alternative to monad transformers. In: Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell. vol. 48, pp. 59–70 (Jan 2014). https://doi.org/10.1145/2578854.2503791
11. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 333–343. POPL '95, Association for Computing Machinery, New York, NY, USA (1995). https://doi.org/10.1145/199448.199528
12. Massachusetts Institute of Technology: riscv-semantics. GitHub, https://github.com/mit-plv/riscv-semantics, accessed 2022-12-06
13. Pieper, P., Herdt, V., Große, D., Drechsler, R.: Dynamic information flow tracking for embedded binaries using SystemC-based virtual prototypes. In: 2020 57th ACM/IEEE Design Automation Conference (DAC). pp. 1–6 (2020). https://doi.org/10.1109/DAC18072.2020.9218494

14. RISC-V Foundation: ISA Formal Spec Public Review. GitHub (2019), https://github.com/riscvarchive/ISA_Formal_Spec_Public_Review, accessed 2022-12-06
15. RISC-V Foundation: The RISC-V Instruction Set Manual, Volume I: User-Level ISA (Dec 2019), https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf, Document Version 20191213
16. RISC-V Foundation: The RISC-V Instruction Set Manual, Volume II: Privileged Architecture (Jun 2019), https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf, Document Version 20190608-Priv-MSU-Ratified
17. Selfridge, B.: GRIFT: A richly-typed, deeply-embedded RISC-V semantics written in Haskell. In: SpISA 2019: Workshop on Instruction Set Architecture Specification (Sep 2019), https://www.cl.cam.ac.uk/~jrh13/spisa19/paper_10.pdf
18. Sergey, I., Devriese, D., Might, M., Midtgaard, J., Darais, D., Clarke, D., Piessens, F.: Monadic abstract interpreters. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 399–410. PLDI '13, Association for Computing Machinery, New York, NY, USA (Jun 2013). https://doi.org/10.1145/2491956.2491979
19. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SOK: (state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 138–157 (2016). https://doi.org/10.1109/SP.2016.17
20. Stallings, W.: Computer Organization and Architecture: Designing for Performance. Pearson Education Inc., ninth edn. (2012)
21. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems. p. 85–96. ASPLOS XI, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/1024393.1024404
22. Swierstra, W.: Data types à la carte. Journal of Functional Programming **18**(4), 423–436 (Jul 2008). https://doi.org/10.1017/S0956796808006758
23. System C Standardization Working Group: IEEE Standard for Standard SystemC Language Reference Manual. Tech. rep., IEEE (2012). https://doi.org/10.1109/IEEESTD.2012.6134619
24. Tempel, S., Brandt, T., Lüth, C.: Artifacts for the 2023 trends in functional programming publication: Versatile and flexible modelling of the RISC-V instruction set architecture. Zenodo (Apr 2023). https://doi.org/10.5281/zenodo.7817414
25. Tempel, S., Herdt, V., Drechsler, R.: Automated detection of spatial memory safety violations for constrained devices. In: Proceedings of the 27th Asia and South Pacific Design Automation Conference. ASPDAC '22 (2022). https://doi.org/10.1109/ASP-DAC52403.2022.9712570
26. Tempel, S., Herdt, V., Drechsler, R.: SymEx-VP: an open source virtual prototype for OS-agnostic concolic testing of IoT firmware. Journal of Systems Architecture p. 12 (2022). https://doi.org/10.1016/j.sysarc.2022.102456
27. Wadler, P., Thiemann, P.: The marriage of effects and monads. ACM Transactions on Computational Logic **4**(1), 1–32 (2003). https://doi.org/10.1145/601775.601776

A Free Monads and Extensible Effects

Subsect. 2.2 introduced the *free monad* for a type constructor `f` as the closure of `f` under application, conceptually given as

```
data Free f a = Pure a | Free (f (Free f a))
```

However, this implementation incurs a severe performance penalty. In this appendix, we explain how the library that we use [13] remedies these deficiencies.

The problem is that each application of the functor `f` corresponds to one application of the constructor `Free`, and moreover, when running the computation we need to compose from the inside (from the right), whereas we construct monads from the outside (left):

```
run :: (Monad m, Functor f) ⇒ (f (m a) → m a) → Free f a → m a
run _ (Pure x) = pure x
run p (Free f) = p (fmap (run p) f)
```

This makes the run-time of the simple approach quadratic. Kiselyov et al. [9,10] extended the approach, by representing each application of `f` with a continuation:

```
data Freer f a where
  Pure :: a → Freer f a
  Impure :: f x → (x → Freer a f) → Freer f a
```

This is a generalized algebraic data type (GADT); it is needed here (as opposed to a plain recursive data type) because the argument type `a` changes in the continuation (the first argument of `Impure`)[14]. The resulting free monad can be implemented more efficiently, by concatenating all the continuations in a queue, which gives a linear run-time [9].

The data type `Freer` is not fully *extensible*: once the type variable `f` is fixed, we cannot later extend it (by adding new types for computations). In our case, this means we have to foresee all possible interpretations in our interpretation of the ISA, or alternatively change the implementation. This limitation can be overcome by combining the free monad with *extensible effects* (see also [27]). Effects insert labels, which are evaluated later, for computational features; they can be made extensible by using an open union type (a type `Union r v` with injection function `inj:: t v→ Union r v` and partial projection `prj :: Union r v→Maybe (t v)`) as labelling type. Crucially, this union type can be extended at run-time, so we can add effects to our analysis functions later on as we need them; *e.g.* we can adapt the way in which we evaluate expressions, using symbolic evaluation instead of fixed operations on bit-vectors.

Thus, `freer-simple` uses `Union r v` as the functor type, and a second type `Arrs` encapsulates the type-indexed queues:

```
data Eff r a where
  Pure :: a → Eff r a
  Impure :: Union r x → Arrs r x a → Eff r a
```

This representation is both efficient and extensible, making it flexible as required.

[13] The aforementioned `freer-simple` library.

[14] The same effect can be achieved by an existential type variable for `x`.