This is a research article submitted as a draft paper.

Versatile and Flexible Modelling of the RISC-V ISA*

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). Most 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. 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, so these days 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 the declarative nature, we can formulate the behaviour at an abstract level which at the same time is executable.

Recently, the RISC-V ISA [11] has emerged has an attractive alternative to the prevailing industry standards, such as the Intel x86 or ARM architecture. It is open source, patent-free, and designed to be scalable from embedded devices

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

to servers. Its open nature has sparked a lot of research activity, in particular many formal models of the ISA, including some in Haskell [12,2,8], or in custom functional 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.

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 execution model 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 can be varied, from simple state transitions which simulate the ISA faithfully, to sophisticated analyses such as symbolic execution or dynamic information-flow tracking. To the best of our knowledge, our approach is the first which focuses explicitly on creating software analysis tools as interpreters for the formal ISA model. Our approach is motivated by our experience with a RISC-V hardware platform simulator written in SystemC TLM [15]; after having to modify it repeatedly to allow such analyses, we were looking for a more systematic and structured way to achieve this flexibility.

2 Modelling an ISA

We explain our approach and its advantages with a simple ISA. It implements a 32-bit load/store architecture with five instructions:

1. `LOADI` *imm reg*: Load immediate into register *reg*.
2. `ADD` *dst src1 src2*: Add two registers into *dst*.
3. `LW` *dest mem*: Load word from memory into register *dest*.
4. `SW` *mem src*: Store word from register *src* into memory.
5. `BEQ` *reg1 reg2*: Branch 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
```

2.1 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).

Fig. 1a provides a simple Haskell execution model for our basic 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).

```
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 imm → let
         v1 = regs ! r1
         v2 = regs ! r2
         br = if v1 == v2
           then pc+imm
           else nextInstr pc
        in (regs, mem, br)
```

(a) 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 imm → let
         v1 = regs ! r1
         v2 = regs ! r2
         br = if v1 == v2
           then pc+imm
           else nextInstr pc
        in (regs, mem, br, counter)
```

(b) Counting memory accesses

Fig. 1: Model of the ISA and sample extension implemented on top of it

Unfortunately, this simple ISA model has several shortcomings. Consider a scenario in 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 Fig. 1b. 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 this semantics, based upon which we can then implement software analysis techniques which

require a different kind of interpretation with minimal effort. Such techniques include symbolic execution [4] or dynamic information-flow tracking [9].

2.2 Our Approach

The problem with the approach from above 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*. A free monad for a type constructor `f` is essentially the closure of `f` under application (so it contains arbitrary many applications of `f`); the appeal here is that the free monad generated by different constructors is the combination of the free monads generated by the single constructors, so we can change the constructors separately. The category-theoretic construction of free monads was given by Kelly [5], and first described in the context of functional programming by Swierstra [14], to be later extended by Kiselyov et al. [7,6].

```
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 1.1: EDSL of the machine executing the ISA

The operations comprising the EDSL are given by a parameterized datatype `Operations`, see Listing 1.1[4]. The `Operations` datatype models the ISA in abstract

[4] For convenience, we add a factory function for each constructor of the datatype embedding it into the free monad.

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; 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 4
    ADD rd r1 r2 → do
        v1 ← loadRegister r1
        v2 ← loadRegister r2
        storeRegister rd (v1+v2)
        incrementPC 4
    LW r addr → do
        v ← loadMem addr
        storeRegister r v
        incrementPC 4
    SW addr r → do
        v ← loadRegister r
        storeMem addr v
        incrementPC 4
    BEQ r1 r2 imm → do
        v1 ← loadRegister r1
        v2 ← loadRegister r2
        if v1 == v2 then incrementPC imm else incrementPC 4
```

Listing 1.2: Interpreting an in ISA instruction in the free monad.

To reconstruct the concrete execution of the ISA instructions from the previous section (Fig. 1a), we need to map the operations in the free monad to concrete monadic effects, in our case in Haskell's pure `State` monad.

```
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 1.3: Evaluating the control flow using the `State` monad

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; *e.g.* reconstructing the example from Fig. 1b just requires adjustments in the semantics as in Listing 1.4.

```
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))
```

Listing 1.4: 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 of 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 1.5 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.

```
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
storeMem' addr w = Free (StoreMem' addr w (Pure ()))

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
```

Listing 1.5: `Operations` type with simple expression language

3 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 reduced instruction set computer (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 [11].

Our abstract model of the RISC-V architecture implements the 32-bit variant of the base instruction set (40 instructions). Based on this abstract model, we have implemented a concrete interpreter for RISC-V instructions. Both the model and the concrete interpreter are written in roughly 1000 LOC in Haskell and can be obtained from GitHub[5]. As opposed to our example above, the implementation also includes a decoder for RISC-V instructions and a loader for ELF executables. Using the full interpreter we were able to successfully execute and pass the official RISC-V ISA tests for the 32-bit base instruction set[6]. This 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, making use of its flexibility.

3.1 Implementing Custom Interpreters

Our abstract model of the RISC-V ISA is designed for maximum flexibility and versatility, along the lines sketched in Sect. 2. We provide a polymorphic operation and expression language built on top of the `freer-simple` library[7], which provides an extended implementation of the free monad approach discussed in the previous section. This allows implementing different interpretations of the ISA on top of our abstract model with minimal effort. In order to implement a custom RISC-V interpreter, an evaluator for the expression language and an interpreter for the free instruction monad need to be provided. As an example, dynamic information flow tracking (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. A conversion to `Word32` needs to be implemented to satisfy the only class constraint imposed by our abstract model.[8] An evaluator of the expression language for `Tainted Word32` can be implemented as follows:

```
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
```

[5] Available for review via `http://unihb.eu/libriscv`

[6] `https://github.com/riscv/riscv-tests`

[7] `https://hackage.haskell.org/package/freer-simple`

[8] This constraint is necessary as the instruction decoder operates on `Word32` values.

The evaluator performs standard 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).

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

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

iftBehaviour :: IftEnv → Instruction (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 instruction monad and the expression language; a complete implementation is provided in the `example/` subdirectory on GitHub. Nonetheless, the example serves to demonstrate that software analysis techniques can be implemented easily on top of an abstract ISA model as custom interpreters for this model.

3.2 Related Work

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 itself in 2019 [10]. From this review, SAIL [1] emerged as the official formal specification for the RISC-V architecture. SAIL is a custom functional language for describing different ISAs and comes with tooling for automatically generating emulators from this description. Similar to our own work, existing work on GRIFT [12], Forvis [2], and riscv-semantics [8] 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 bitvector 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 functional model which focuses specifically

on flexibility and thereby enables non-concrete execution of RISC-V instructions on top of the abstract ISA model.

As such, we believe our RISC-V ISA model to be a versatile tool for building dynamic software analysis techniques that operate directly on the machine code level. Prior work has already demonstrated that it is possible to implement techniques such as symbolic execution [16] or dynamic information flow tracking [9] 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 on the machine code level and instead leverages intermediate languages and lifts machine code to these languages [13,3,4]. 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.

4 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. This way, it can be used to implement software analysis techniques such as dynamic information flow tracking or symbolic execution.

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.

In future work, we would like to model additional extensions of the RISC-V architecture and perform further experiments with interpreters for our model, most importantly symbolic execution. It would also be interesting to investigate the issue of correctness of the custom interpreters further, *e.g.* by embedding the interpreters into a theorem prover.

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. Bluespec, Inc.: Forvis: A formal RISC-V ISA specification. GitHub, https://github.com/rsnikhil/Forvis_RISCV-ISA-Spec, accessed 2022-12-06
3. 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)
4. 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
5. 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
6. Kiselyov, O., Ishii, H.: Freer monads, more extensible effects. SIGPLAN Not. **50**(12), 94–105 (Aug 2015). https://doi.org/10.1145/2887747.2804319
7. Kiselyov, O., Sabry, A., Swords, C.: Extensible effects an alternative to monad transformers. vol. 48, pp. 59–70 (Jan 2014). https://doi.org/10.1145/2578854.2503791
8. Massachusetts Institute of Technology: riscv-semantics. GitHub, https://github.com/mit-plv/riscv-semantics, accessed 2022-12-06
9. 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
10. RISC-V Foundation: ISA Formal Spec Public Review. GitHub (2019), https://github.com/riscvarchive/ISA_Formal_Spec_Public_Review, accessed 2022-12-06
11. 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
12. 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
13. 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
14. Swierstra, W.: Data types à la carte. J. Funct. Program. **18**(4), 423–436 (Jul 2008). https://doi.org/10.1017/S0956796808006758
15. 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
16. 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