Vellvm: Formalizing the Informal LLVM
(Experience Report)

Calvin Beck[1](✉), Hanxi Chen[1], and Steve Zdancewic[1]

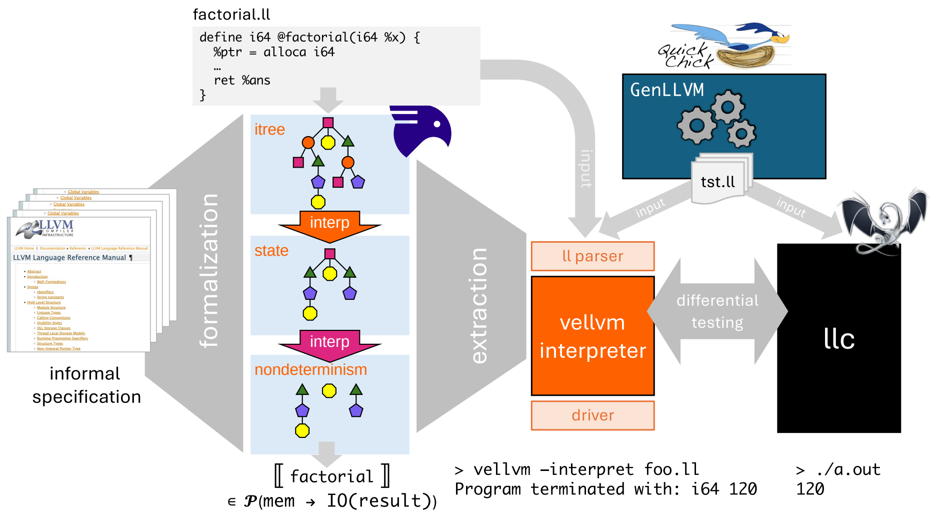
University of Pennsylvania, Philadelphia PA 19104, USA
`{hobbes,hanxic}@seas.upenn.edu` `stevez@cis.upenn.edu`

Abstract. This report presents our methodology for and experience with formalizing a specification of LLVM IR in the Verified LLVM Project (Vellvm). Vellvm provides a specification for a large, practical subset of LLVM IR in the Rocq Proof Assistant in the support of verified compilers and program transformations. Program transformations often rely on the subtle details, and as a result Vellvm's semantics are quite comprehensive: for instance we provide a sophisticated low-level memory model to support low-level operations, such as casts between pointers and integers, and justify optimizations in their presence. Our approach implements the semantics via *monadic interpreters*, which rely on a coinductively-defined data structure called *ITrees*. Crucially, this methodology supports the extraction of an executable interpreter, proved to refine the specification. We use the reference interpreter to validate the accuracy of the formalization, employing random differential testing between Clang and Vellvm implemented in our own LLVM IR program generator (GenLLVM), as well as via state-of-the-art C compiler testing frameworks (CSmith and YARPGen). Such testing has found bugs in both the Vellvm semantics and Clang. We believe that tools from the Vellvm project can be useful for other LLVM IR-related projects, and that the overall methodology applies to other formal verification efforts.

Keywords: Rocq · LLVM · Semantics.

1 Formalizing LLVM IR

Vellvm [3, 4, 26, 28, 29] is an extensive undertaking to formalize a large subset of LLVM IR in the Rocq proof assistant. LLVM IR [11, 21] itself is an intermediate language that is widely used as a common target for front-end compilers for a variety of languages, like C and Rust, that share the LLVM infrastructure for performing optimizations and generating lower-level target code. Since LLVM IR is the bedrock of many languages, formalizing it has great significance. The Vellvm project is an effort to 1) provide a robust and detailed specification for the LLVM IR language, 2) ensure that this specification aligns with real world implementations via a differential testing approach, and 3) reason about LLVM IR programs and verify the correctness of LLVM optimization passes.



```
define i64 @factorial(i64 %x) {
  %ptr = alloca i64
  …
  ret %ans
}
```

```
> vellvm -interpret foo.ll
Program terminated with: i64 120
```

```
> ./a.out
120
```

Fig. 1: The Vellvm Methodology: Formalize, extract, and test against `llc`.

Formalizing LLVM IR is not straightforward for a variety of reasons: 1) LLVM IR is a well-established language, but has been developed with only an informal specification in mind 2) LLVM is a rapidly evolving project, making it a moving target for formalization efforts, 3) LLVM IR is a large language, and the scale is non-trivial for formalization efforts, and 4) LLVM IR contains complex low-level features—including many intentionally *undefined behaviors* (UB) as well as *nondeterministic* operations—that are difficult to formalize, but vital to justifying important program transformations. Ultimately, LLVM IR is a complex language and a moving target, which complicates verification efforts.

2 Methodology

To tackle this challenge, we have adopted a specific methodology for formalizing LLVM IR at scale, as shown in Figure 1.[1] At a high level, the methodology involves three parts: 1) defining the semantics for LLVM IR in Rocq that allows reasoning about programs and optimization passes, 2) extracting from that specification an interpreter that agrees with these semantics, and 3) using randomized differential testing of this interpreter against existing LLVM implementations.

While similar approaches have been taken in the past (see Section 3), we believe that the realization in Vellvm of this methodology offers an appealing combination of properties that can applicable to other formal verification efforts. Moreover, the Vellvm artifact itself can be useful for testing systems that involve LLVM IR's semantics by demonstrating (an) exact behavior of a given LLVM IR program. The remainder of this section describes the components of our methodology.

[1] Though, this methodology is certainly applicable to languages beyond LLVM IR.

2.1 Monadic Interpreters and ITrees

The Vellvm semantics is defined via *monadic interpreters* [13]. Using this approach we can express much of the semantics in precisely the same way that one would typically write an interpreter using a functional programming language, and the modular event structure allows us to reason about the different effectful components of the language separately. These monadic interpreters (in Figure 1, they are represented as the stack of "interp" arrows) are operate on the coinductively-defined data structures called *ITrees* [23, 25].

Interaction Trees. Interaction Trees (ITrees) abstract the complex, effectful portions of LLVM IR into a variety of opaque *events*. These separate events can be reasoned about individually, without worrying about the particular details of other, separate program components. This allows us to reason about complex features of the language (such as the memory model) separately, and alternative implementations can be provided for these events. ITrees provide a framework for reasoning about program behaviors in a compositional and equational fashion, greatly simplifying proofs about coinductive objects—used to represent nonterminating computations—within Rocq. For instance, ITrees come with a notion of equivalence by means of a weak bisimulation relation called "equivalence up to tau" (`eutt`), which is useful for defining behavioral equivalence of ITree-abstracted programs, as well as for rewriting ITree expressions.

Once LLVM IR programs are denoted into ITrees, which capture the control flow of the programs, the stateful components of the semantics are implemented modularly in terms of these abstract events. The interpreter stack for Vellvm contains a number of event types, such as: events for handling local variables, global variables, and stack frames; memory events (loads, stores, allocations, frees); evaluating non-deterministic values (pick events); UB; errors; and out-of-memory exceptions. Figure 1 symbolizes the different types of events using different shapes of nodes in the trees. An advantage of ITrees is that all of these events can be left opaque: we can reason about the equivalence of LLVM IR programs up to a weak bisimulation of these visible events, and any properties proven on opaque events should hold regardless of how these events are implemented. This design enables *modular* reasoning: if, for instance, a program transformation affects only local variables, its correctness proof can ignore all of the details about memory, and the validity of the program transformation will hold regardless of how memory is implemented.

Event Handlers and the Vellvm Memory Model. Defining the semantics for opaque events involves writing *event handlers*, which interpret the events using other primitives (in Figure 1, these are the arrows that "remove" events of a specific shape from the tree). We define a stack of handlers for each of the events in our LLVM IR semantics. For LLVM IR, the most complex handler is the one that implements the *memory model*. Our memory model is based on a two-phase semantics [4] that separates infinitary memory from finite behavior. This model provides a number of practical benefits, including the ability to an accurately describe the behaviors of trickier operations such as pointer-to-integer casts, as well as the ability to justify certain optimizations even in the presence of low-level memory operations.

The memory model itself implements stack/heap allocations, loads, stores, frees, and stack push/pop operations. These operations are defined byte-wise and are used compositionally by the memory model handler uses to implement aggregate data operations for LLVM IR's structured values. The memory event interface gives an appropriate boundary between Vellvm's instruction semantics and its memory model, allowing us to substitute different memory models into the semantics if desired.

Memory events also mark the first place in the semantics where we introduce *nondeterminism*, since allocations inherently yield nondeterministic locations in memory. As a consequence, we provide *two* sets of memory handlers, one for the *propositional* specification, yielding a Rocq predicate that represents the set of all possible behaviors for memory events, and an *executable* one, which implements a deterministic algorithm for allocations. We have proven that the executable implementation's behaviors abide by the specification.

2.2 Extractable Interpreters

An advantage of using ITrees to define our specifications is that they can be extracted into executable interpreters (in contrast to other techniques, such as step relations). The verified executable interpreter provided by Vellvm uses the same top-level ITree representation as the specification. The only difference between the semantics and the executable interpreter is in the implementation of handlers for non-deterministic events (the executable version of our semantics has to pick a specific implementation for allocations, for instance); all of the handlers for deterministic events can be shared between the interpreter and semantics. Because of this, we obtain an executable interpreter which is verified to abide by the specification of our language for very little effort.

With a bit of engineering to implement an LLVM IR parser and a small driver (in Figure 1, these are the components surrounding the vellvm interpreter box), we obtain an executable version of Vellvm's LLVM IR semantics that is compatible with LLVM's `.ll` file format. Moreover, Vellvm's coverage of the IR is substantial—the only major features missing are those having to do with concurrency and exception handling. The Vellvm interpreter, overall, has delivered satisfactory performance, allowing us to run reasonably large programs relatively quickly: in one of our experiments, Vellvm executed over 2.5 million non-trivial instructions in 2m03s. Some of the interpretation overhead could potentially be eliminated by further optimization and/or using more efficient data structures, but the performance has been adequate for our experiments.

2.3 Random Differential Testing

Having an executable interpreter opens the door to testing, which is the last important component of our methodology. In addition to a number of manually written unit tests, as well as a test suite derived from the Alive2 [16] project, we also utilize the property-based testing framework QuickChick [20] to generate LLVM IR programs for *differential* testing in which we compare our verified executable interpreter against Clang's `llc` as shown on the right-hand side of Figure 1. Using this approach, we have been able to find bugs in both Clang and Vellvm [1], particularly around tricky edge cases such as data types of size 0, or `structs` and packed arrays with odd-sized elements.

Generating random programs to differentially test compiler implementations is not a new technique: CSmith [24] and its successors YARPGen (both versions 1 and 2) [14, 15] are capable of generating expressive C programs targeting arithmetic operations and control-flow, respectively. But because these tools generate C programs, they only *indirectly* exercise LLVM IR features: generated test programs must first be compiled from C to LLVM IR. As a result, their outputs (post-compilation) follow certain code patterns and only utilize subsets of instructions which are chosen by the code generation strategy. As an example, Clang-19 favors `getelementptr` over struct operations like `extractvalue` when compiling C into LLVM IR, reducing coverage of the latter.

To address this deficiency in coverage, we created the main artifact of our differential testing framework: GenLLVM, a tool for generating well-formed, UB-free LLVM IR programs. Unlike the aforementioned tools, GenLLVM focuses on testing LLVM IR semantics directly. GenLLVM itself is built via *monadic* functional programming on top of QuickChick's "generator" monad. This structure threads through a random seed and a desired size for test case generation, providing a convenient domain-specific language for randomly sampling values to build up complex structured data—in our case, LLVM programs. GenLLVM exercises a large subset of Vellvm's instructions, including arithmetic operations, pointer manipulation, vector manipulation, aggregate type manipulation, and conversion operations (e.g., `inttoptr`). It also generates complex control-flows using recursion, loops, and calls.

Generating interesting, valid lower-level target programs is challenging: we need the generated programs to be *deterministic*, *type-safe*, *UB-free*, and *diverse* in instruction usage and code patterns. These properties allow us to meaningfully compare executions between Clang and Vellvm, and differences in the execution between these implementations will be highly indicative of a bug in one of them. To give an example of the concerns that arise during instruction generation, consider LLVM's `inttoptr` instruction. According to LLVM's Language Reference [22], `inttoptr "takes an integer value to cast, and a type to cast it to, which must be a pointer type."`[2] However, simply following this type specification to generate `inttoptr` operations in test programs can easily lead to UB: consider a simple LLVM snippet `inttoptr i32 123 to i8*`, which simply calls `inttoptr` on the 32-bit integer `123`; a subsequent `load` from the resulting pointer accesses unallocated memory. To address this challenge while also ensuring rich test coverage of programs containing `inttoptr`, GenLLVM uses a *context-aware metadata* system, heavily inspired by entity-component systems

[2] The Language Reference itself notes that `inttoptr` is "really dangerous".

such as Ecstasy [17, 18]. Using our metadata system we are able to dynamically track the types of variables, as well as other information, such as whether an integer value was derived from a `ptrtoint` cast—information that we use to only generate `inttoptr` casts from integers that point to valid addresses [1]. Other state of the art generators like CSmith and YARPgen don't generate these kinds of cast instructions, so we're able to exercise more of the semantics of LLVM.

This differential testing approach allows us to discover bugs in both Vellvm and, more importantly, Clang. One of the newly found bugs [2], recognized by Clang developers, can be triggered by a simple LLVM instruction `extractvalue <[3 x i8], i8> %v, 1`. This snippet effectively loads a byte from an aligned, packed struct that starts with an odd-sized vector. Clang's implementation of the address calculation rounds up the size of the vector to the next alignment boundary, in this case 4 instead of 3, independently of whether the struct is packed. Such compiler bugs lead to hard-to-find errors when correct source programs are transformed into subtly wrong assembly code.

3 Discussion: Challenges and Related Work

Our methodology allows us to successfully define a formal semantics for a large subset of LLVM IR in Rocq and to validate the correctness of the executable interpreter with respect to the specification. We have used the Vellvm semantics to verify some program transformations [4] and to prove compiler correctness results [27]. However, some challenges to using this methodology remain.

Challenges. The biggest concern with defining a semantics using monadic interpreters and ITrees in this style, and more generally with using an interactive theorem prover like Rocq, is the significant manual proof effort involved. Work on Vellvm requires a large degree of Rocq expertise and often takes significant time and engineering effort.

Another challenge arises in handling LLVM IR's inherent nondeterminism. Our strategy is to interpret nondeterministic events (such as allocations) into a *nondeterministic set*, using Rocq's `Prop` type, in which all possible results are considered. This has complicated some proofs in the later stages of the interpretation pipeline, and also causes a divergence in the implementation of the semantics and interpreter. Additionally, concurrency (a feature we have not yet tackled) similarly requires nondeterminism to model and thus presents similar challenges. Recent advances in *Choice Trees* [5, 6], which natively handle nondeterministic choices, may ameliorate these problems.

Related Work. A number of projects aim to provide specifications for both languages and hardware, each with their own approaches. Notably, our work builds upon the ideas established in the ITrees paper [23], which provides a case study on a simple assembly language. We have extended the techniques from this paper to deal with the more complicated semantics of LLVM IR (for instance, by introducing nondeterminism).

Cerberus [19] uses an elaboration approach to provide a model for C; language primitives in C are implemented using a simpler core language with a straightforward semantics. K-LLVM [12] has a somewhat similar approach, using an abstract state machine as the basis for the behavior of LLVM IR instructions.

The Fiat [8] library demonstrates a means of refining a specification into efficient executable functional programs, which has some similarities to our approach of extracting an executable interpreter using ITrees. Kami [7] uses similar approaches to bring modular reasoning to hardware verification efforts in Rocq. Hardware modules can be verified separately from each other under Kami.

FORMED (Formal Methods Engineering Desktop) DSL [9] is a project building on Unified Modeling Language (UML) with formal methods. Using an automated theorem prover ACL2s, the authors use a similar methodology in modeling DSLs in UML and subsequently generating executable applications in ACL2s code for formal verification and unit testing, as explained in their paper. Vellvm differs from the FORMED project in a several ways: 1) we define LLVM IR semantics via ITrees and reason about the correctness modularly, all in Rocq, 2) while both use unit testing, we exploit random differential testing, which allows us to find bugs in not only our formalization, but also real-world, large-scale implementations, and 3) The formal semantics is defined via monadic interpreters, which can then be turned into an executable version via Rocq's extraction mechanism. The correspondence between the executable interpreter and the formal semantics is formally proved in Rocq, giving us high assurance that the executable meets the specification.

Our QuickChick generators for LLVM IR programs have largely been inspired by similar compiler testing projects, such as CSmith [24] and YARPGen [15]. Alive2 [16] uses a translation-validation approach and SMT solvers in order to validate program transformations on LLVM IR code. QuickChick has also used to test other languages like SCILLA [10], an ML-style functional language for implementing smart contracts.

4 Conclusion

The Vellvm project provides a means for reasoning about LLVM IR programs and optimization passes within the Rocq proof assistant. Using interaction trees we are able to define a modular semantics from which we can derive a verified executable interpreter with relatively little effort. We have also validated our semantics against existing implementations of LLVM using a differential testing approach using our own test case generator, GenLLVM. We're able to generate a wider range of LLVM instructions when compared to existing generators which target higher level languages, and using this approach we have been able to discover bugs in both our own implementation and the Clang compiler. Vellvm is a large project, and we believe that the battle-tested methodology presented in this paper is useful for other formalization efforts.

Bibliography

[1] A selection of bugs discovered in clang / vellvm with genllvm (2024), https://github.com/vellvm/vellvm/issues?q=is%3Aissue+label%3Aqc-discovered-bug+

[2] Extractvalue with packed struct with vector off-by-one error (2025), https://github.com/llvm/llvm-project/issues/124061

[3] Beck, C., Yoon, I., Chen, H., Zakowski, Y., Zdancewic, S.: A two-phase infinite/finite low-level memory model (Jun 2024). https://doi.org/10.5281/zenodo.12518800, https://doi.org/10.5281/zenodo.12518800

[4] Beck, C., Yoon, I., Chen, H., Zakowski, Y., Zdancewic, S.: A two-phase infinite/finite low-level memory model: Reconciling integer–pointer casts, finite space, and undef at the llvm ir level of abstraction. Proc. ACM Program. Lang. **8**(ICFP) (Aug 2024). https://doi.org/10.1145/3674652, https://doi.org/10.1145/3674652

[5] Chappe, N., He, P., Henrio, L., Zakowski, Y., Zdancewic, S.: Choice trees: Representing nondeterministic, recursive, and impure programs in coq. Proc. ACM Program. Lang. **7**(POPL) (Jan 2023). https://doi.org/10.1145/3571254, https://doi.org/10.1145/3571254

[6] Chappe, N., Henrio, L., Zakowski, Y.: Monadic interpreters for concurrent memory models: Executable semantics of a concurrent subset of llvm ir. CPP 2025, Association for Computing Machinery (2025), https://perso.ens-lyon.fr/nicolas.chappe/muvellvm-concurrency-draft.pdf

[7] Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A., Arvind: Kami: a platform for high-level parametric hardware specification and its modular verification. Proc. ACM Program. Lang. **1**(ICFP) (Aug 2017). https://doi.org/10.1145/3110268, https://doi.org/10.1145/3110268

[8] Delaware, B., Pit-Claudel, C., Gross, J., Chlipala, A.: Fiat: Deductive synthesis of abstract data types in a proof assistant. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 689–700. POPL '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2676726.2677006, https://doi.org/10.1145/2676726.2677006

[9] Eakman, G., Reubenstein, H., Hawkins, T., Jain, M., Manolios, P.: Practical formal verification of domain-specific language applications. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NASA Formal Methods. pp. 443–449. Springer International Publishing, Cham (2015)

[10] Hoang, T., Trunov, A., Lampropoulos, L., Sergey, I.: Random testing of a higher-order blockchain language (experience report). Proc. ACM Program. Lang. **6**(ICFP) (Aug 2022). https://doi.org/10.1145/3547653, https://doi.org/10.1145/3547653

[11] Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International

Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California (Mar 2004)

[12] Li, L., Gunter, E.: K-llvm: A relatively complete semantics of llvm ir. In: 34rd European Conference on Object-Oriented Programming, ECOOP 2020, Berlin, Germany (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.7

[13] 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, https://doi.org/10.1145/199448.199528

[14] Livinskii, V., Babokin, D., Regehr, J.: Random testing for C and C++ compilers with YARPGen. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 1–25 (Nov 2020). https://doi.org/10.1145/3428264, https://dl.acm.org/doi/10.1145/3428264

[15] Livinskii, V., Babokin, D., Regehr, J.: Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. Proceedings of the ACM on Programming Languages **7**(PLDI), 1826–1847 (Jun 2023). https://doi.org/10.1145/3591295, https://dl.acm.org/doi/10.1145/3591295

[16] Lopes, N.P., Lee, J., Hur, C.K., Liu, Z., Regehr, J.: Alive2: Bounded translation validation for llvm. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 65–79. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454030, https://doi.org/10.1145/3453483.3454030

[17] Maguire, S.: ecstasy: A ghc.generics based entity component system. (2018), https://hackage.haskell.org/package/ecstasy

[18] Maguire, S.: Why take ecstasy (Jan 2018), https://reasonablypolymorphic.com/blog/why-take-ecstasy/index.html

[19] Memarian, K., Matthiesen, J., Lingard, J., Nienhuis, K., Chisnall, D., Watson, R.N.M., Sewell, P.: Into the depths of c: Elaborating the de facto standards. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 1–15. PLDI '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2908080.2908081, https://doi.org/10.1145/2908080.2908081

[20] Paraskevopoulou, Z., HriŢcu, C., Dénès, M., Lampropoulos, L., Pierce, B.C.: Foundational property-based testing. In: Urban, C., Zhang, X. (eds.) Interactive Theorem Proving. pp. 325–343. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_22, https://doi.org/10.1007/978-3-319-22102-1_22

[21] llvm-admin team: LLVM (2024), https://llvm.org/

[22] llvm-admin team: LLVM Language Reference Manual (2024), https://llvm.org/docs/LangRef.html, documentation

[23] Xia, L.y., Zakowski, Y., He, P., Hur, C.K., Malecha, G., Pierce, B.C.,

Zdancewic, S.: Interaction trees: representing recursive and impure programs in coq. Proc. ACM Program. Lang. **4**(POPL) (dec 2019). https://doi.org/10.1145/3371119, https://doi.org/10.1145/3371119

[24] Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 283–294. PLDI '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1993498.1993532, https://doi.org/10.1145/1993498.1993532

[25] Zakowski, Y., Beck, C., Yoon, I., Zaichuk, I., Zaliva, V., Zdancewic, S.: Modular, compositional, and executable formal semantics for llvm ir. Proc. ACM Program. Lang. **5**(ICFP) (aug 2021). https://doi.org/10.1145/3473572, https://doi.org/10.1145/3473572

[26] Zakowski, Y., Beck, C., Yoon, I., Zaichuk, I., Zaliva, V., Zdancewic, S.: Modular, compositional, and executable formal semantics for llvm ir. Proceedings of the ACM on Programming Languages **5**(ICFP) (2021)

[27] Zaliva, V., Zaichuk, I., Franchetti, F.: Verified translation between purely functional and imperative domain specific languages in helix. In: Software Verification: 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20–21, 2020, Revised Selected Papers. p. 33–49. Springer-Verlag, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-63618-0_3, https://doi.org/10.1007/978-3-030-63618-0_3

[28] Zdancewic, S., et al.: Vellvm (2024), https://github.com/vellvm/vellvm

[29] Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formal verification of ssa-based optimizations for llvm. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 175–186. PLDI '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2491956.2462164, https://doi.org/10.1145/2491956.2462164