

Midair: An Intermediate Representation for Multi-purpose Program Analysis

Maxim Menshikov(✉)

Saint Petersburg State University, 7-9, Universitetskaya nab.,
St. Petersburg 199034, Russian Federation
info@menshikov.org

Abstract. The static analysis field had grown enough to be used not only for finding casual defects. In practice, it may be used to enforce the coding style and flag undesired syntax constructs, find logical mistakes, prove that the program satisfies its specification, apply domain-specific checks, or even verify cross-program compatibility. Those are all valid use cases that are required to be handled by the static analyzer, and the intermediate representation (IR) affects how can it be done.

A typical compiler or analyzer uses a number of IRs, each of them helps with a specific problem. For our static analyzer project, we found that existing IRs partially do not match our requirements, which led to the creation of Midair—an IR for multi-purpose program analysis. It is positioned right between IRs created primarily for the compilation (like LLVM, MLIR, GIMPLE) and verification IRs (such as Boogie) with the hope that it would be both close to a low level and suitable for verification while applying to practical analysis tools. The IR consists of 4 layers, allowing for a transparent transformation between forms saving time and space. A flexible type system supporting non-machinery types and an ability to augment the representation with external metadata provided by solvers had been added. The application of Midair to our analysis framework uncovered advantages and non-critical issues, which are planned to be worked around.

Keywords: Intermediate representation · Verification · Program analysis · Static analysis · Framework

1 Introduction

Decades ago intermediate representation became an industry-recognized way to separate concerns between parsing, analysis, optimization and code generation passes, yet to unify their input/output formats. Building efficient language-agnostic middle steps depends on it. Before LLVM [3], IRs were mostly considered an internal part of programming tools, unavailable from the outside. LLVM had presented its language and opened the framework for external tools. Many successful analyzers have built upon this universal, but a compiler-specific foundation.

© Springer Nature Switzerland AG 2020
O. Gervasi et al. (Eds.): ICCSA 2020, LNCS 12254, pp. 544–559, 2020.
https://doi.org/10.1007/978-3-030-58817-5_40

When the author was about to design an extensible static analysis framework [19], it was found that such universal IRs require a lot of extensions to be used efficiently. IR has to be wrapped to control flow graph nodes, own command wrappers; type system still has to be abstracted away for efficient conversion to solver formats (e.g. SMT-LIB for satisfiability modulo theories [6]), so it is essentially *separate* architecture and infrastructure side-by-side with the one provided by IR. Low-level IR adds overhead for most analyses, and high-level IR does not represent low-level primitives. What if they are combined and the redundant infrastructure is eliminated? Our hypothesis is that middle-level IR will still handle static analysis efficiently. That was the reason behind designing a new IR called Midair. Existing IRs are reviewed in Sect. 2, our requirements are shown in Sect. 3, and with all this insight our design solutions are judged in Sect. 4. A few notes on the used type system are in Sect. 5, a serialization mechanism is described in Sect. 6. The final evaluation results are presented in Sect. 7.

The project's *goal* is *to create an efficient middle-level intermediate representation, allowing for different analysis methods.* At least Abstract Syntax Tree analysis, control and data flow analyses, model checking and abstract interpretation are possible in that method and the author believes more analyses are necessarily suitable if the corresponding form is supported. At the moment of writing, there is no intention to support other purposes like compilation, code generation, etc. Therefore, these cases are not taken into account.

Novelty. The created IR has four representations with a simple "analyze & separate" transformation strategy, reducing the CPU and memory footprint between transformations. The built-in support for databases allows efficient resource loading. The IR suggests but does not enforce the cooperation between analysis passes by augmenting the IR with their results. To the knowledge of the author, no IR provides such features.

2 Related Work

Many sources [5,9] claim the following *classic* IR types. **Parse tree and Abstract Syntax Tree (AST)** serve the purpose of maintaining precise source mapping. In compilers and analyzers, this is the first structure that is often not considered an IR. In source-to-source systems, it is the first and the last representation. The AST and parse tree nodes might have duplicating children. **Directed Acyclic Graphs (DAG)** avoid the duplication introduced by AST. The DAG nodes might contain multi-parent children, so this IR is by definition smaller than AST. **The Control Flow Graph (CFG)** groups branch-free code. Such a technique helps determine control flow properties and is generally more applicable to other kinds of analysis.

There are five main linear forms. One and two address codes are now used too rarely, and **three-address code**, in which every command consists of two operands and the result, is still the foundation for many IRs. The **stack-machine code** is based on the concept of the stack: every parameter is pushed to

stack or popped from it. This representation is very compact and still very popular. The **Static Single Assignment** form is different from the three-address code in the sense that variables are assigned only once.

Listed IRs are used in most projects, but complicated uses are all based on forks of these types. **SIMPLE** [12]—one of the first structured intermediate representations, employed in McCAT compiler. Effectively, it is not only SIMPLE but also FIRST and LAST representations, named by their actual appearance time during compilation.

GENERIC, **GIMPLE** [20], **RTL** (Registry Transfer Language) [2] are all different level intermediate representations used in GCC. GENERIC is a way to represent entire functions in trees. GIMPLE is influenced by SIMPLE, it is a simplified representation compared to GENERIC. RTL is a very low-level language limited to machine types. These IRs played a significant role in advancing compiler technologies, however, they are not widely used outside GCC community.

The **SUIF** [25] kernel features an intermediate representation, which is primarily used for optimization. This IR is of mixed-level: low-level operations are wrapped by high-level constructs, e.g. loops. That is ideologically close to our implementation. The **Soot** [24] started as an interprocedural Java bytecode analysis framework. It has four intermediate representations: Baf (a bytecode representation without complications), Jimple (3-address representation of byte code), Shimple (Single Static Assignment form of Jimple), and Grimp (unstructured representation of Java code). The difference compared to Midair is a focus on optimization, even though performing analysis is still possible. Also, Midair has more freely interpretable semantics with the intention of aggregating the results of different analyses. The **Byte Code Engineering Library (BCEL)** [1] is a byte code manipulation foundation for many language tools like FindBugs [13], AspectJ [14], etc.

Low Level Virtual Machine (LLVM) IR [3] is the language provided by LLVM. It is meant to be close to assembly and is created as the one IR for all analysis, compilation purposes. It features Static Single Assignment form, machine types, metadata support. The LLVM gradually changed the paradigm regarding the external usage of IRs. It is currently an industrial standard with many successful applications. **Multi-Level Intermediate Representation (MLIR)** [15] is a new language suggested by C. Lattner et al. It is built on top of the *dialect* concept. In some sense, it aims to be a superset of LLVM IR: the latter is considered just one of the possible MLIR dialects. For example, another existing dialect—**tf** from TensorFlow, a machine learning framework—adds tensors as the first-class types. The main focus is at more high-level optimization, i.e. let the compiler know some facts about the program that may help improve the output assembly.

C intermediate language (CIL) [21] is created specifically for program analysis and transformations. It comes with its parser and is said to support the most C features. It is not in active development at the moment of writing. **SAIL** [10]—static analysis intermediate language, suggested by I. Dillig,

T. Dillig and A. Aiken. It features high-level IR, which is close to Abstract Syntax Tree, and a low one, which is essentially a Control Flow Graph.

Boogie [16] is an intermediate verification language for other software verification tools to base on. Created by Microsoft Research, it provides a very comprehensive view of program analysis, combining both mathematics and programming foundations. Based on Racket, **Rosette** [23] is directed to be a programming language rather than IR. It provides a sufficient number of supported theories and data types, allowing for verification.

GraalVM [8] is a remarkable virtual machine for executing polyglot applications written in JVM-based, LLVM-based (e.g. via Sulong [22]) and other languages. It features the language implementation framework called Truffle [26], which provides means for creating AST-based interpreters. The Truffle AST partially covers the language-agnosticism required by our project, however, the described project requires additional flexibility. After all, GraalVM can be potentially used at the parsing stage as prerequisites before generating the Midair IR.

Also, it is worth mentioning **REIL** [11], a framework for static analysis of disassembled code, which is beyond the scope of this paper. Java [27] byte code is probably the most used IR in the world, and Microsoft's counterpart, MSIL or CIL [4], is the byte code for .NET technology. They both are based upon the concept of the stack machine code.

3 Requirements

We had the following set of requirements for the IR:

1. **Three-address code without Static Single Assignment (SSA) enforcement**.
   The SSA form is leaning towards code generation. It is still perfectly valid—not to say desirable—representation for analysis as well, yet a number of use cases imply more clear relations between variable versions:
   (a) Symbolic execution often does not require variable versions.
   (b) Detecting possible values for enumeration type variables.
   (c) The mapping between the source and the IR.
   Thus, the Midair does not enforce the single static assignment usage. LLVM and MLIR are both completely SSA, Boogie is on the other hand completely non-SSA.
2. **Ability to reuse Abstract Syntax Tree or Directed Acyclic Graph** for performance reasons.
   The performance is a major concern for any analysis, especially those associated with C-like languages which headers might inflate the global scope with thousands of unused objects. If it is possible to reuse AST/DAG, then a significant part of CPU cycles might be saved.
   LLVM language is completely unrelated to Clang or any other language AST, as well as Boogie is, so the mapping needs to be formed from scratch.

3. **Awareness of local and remote objects, possible reliance on the database**.
   Objects have different scopes, might be obtained by different computational nodes. Some objects might be located remotely, so language should not enforce objects to be defined in place.
   This requirement indirectly gives additional memory footprint reduction: e.g. C/C++ headers can have many possibly redundant cross-references. With the databases in mind, the analyzer might unload rarely used objects.
4. **The arbitrary type system, an ability to reuse short-lived interpretations**.
   Each module using IR should contribute to the deep understanding of code, however, it is not trivial if command system enforces types, thus the types should only be suggested. Modules should be able to use the contributions of each other to a possible extent.
   Of course, other IRs support omitting specific instructions, but for languages like LLVM IR it is not natural as only full program is supposed to be run. Recovering dependencies between over-simplified instructions might be a completely separate task.

While MLIR seems to have similar goals and even spelling, we believe it still mainly fitting for code generation. Unlike CIL, Midair was supposed to advance further by adopting a three-address code. Unlike Boogie and Rosette, it was supposed to leave mapping to the source (although it is harder in VM IR that we'll outline later). Based on these conclusions and existing IR overview, our intention was to set Midair right in the middle between verification and compilation to get the compilers' machine knowledge level, not sacrificing analysis quality.

The ultimate requirement is to have an architecture not narrowed to the specific analysis method. Compilation, optimization and code generation are not required at the current stage, so IR is not designed for them.

4 IR Overview

Midair consists of four representations (Fig. 1), ranging from DAG to designated virtual machine commands.

4.1 Plain Directed Acyclic Graph

First, we attacked defined problems by creating unified expression architecture [19]. The main idea is old: all language parsers create a unified representation (that is called *generalized syntax tree* in our implementation). Specific objects like structure declarations do not even have a direct equivalent in GST, they are saved directly to the type database. This GST consists of many C-like language constructs, the concepts of *statements*, *expressions*, and the notion of *resources*, as the objects including variables, functions, and implicit model objects.

The DAG helps find trivial issues with duplicate operands, unnecessary assignments, detects coding style violations.

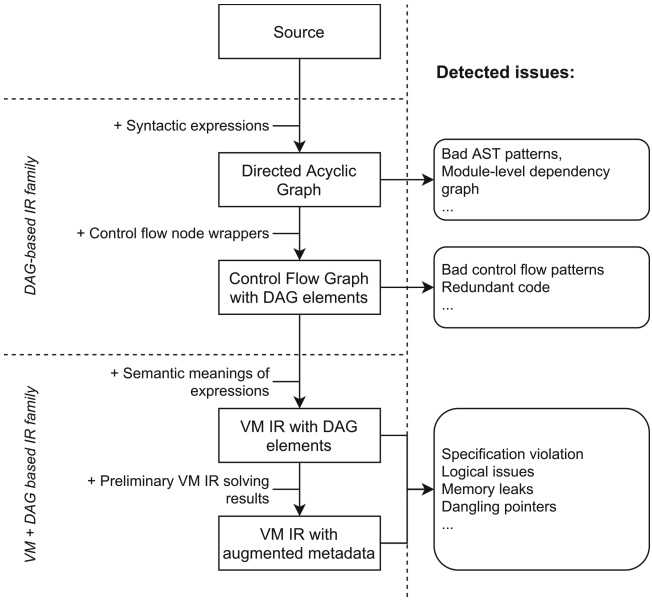


Fig. 1. Midair overview

4.2 Control Flow Graph with DAG Elements

The second step resembles the *Control Flow Graph* (CFG) construction. The DAG nodes are combined not only by the property of being branch-free but *any* property that may be considered worth differentiating by the analyzer. It is necessarily a syntax context, optionally a lockset, etc.

This representation is useful for the control flow checks, redundant code detection.

Transition to CFG. The transition from plain DAG is semantic-driven. The syntax traverse pass forms graph with linear, branch-free statements grouped within nodes. When a single statement is about to be added to some *active* graph node, it is verified against a predefined set of properties forming a *context*. If it operates in a different context, then it is saved to a separate CFG node. In this procedure, it is assumed that expressions are moved as is. The trivial scheme is shown in Fig. 2. This transition is reversible.

4.3 Virtual Machine Instructions

Our project uses a virtual machine [17] for performing data flow, model checking and abstract interpretation.

This level's intermediate representation is the only language for the virtual machine. It adds an understanding of what's happening within the DAG. All

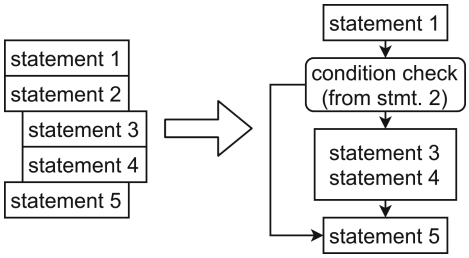


Fig. 2. Morphing between DAG and Control Flow Graph

the assignments, invocations, model checks are stated explicitly, so further verification passes do not have to look these facts up again.

Simplification Degrees. Two distinct simplification degrees exist in IRs. First, no simplification, just leaving DAG as it is in the source code. Second is oversimplifying: every operation essentially becomes an assignment in a three-address code.

We use a hybrid approach: all assigning, type-changing and some other operations are turned to assignments. All other operations remain as complex as they are in DAG. This has clear benefits for trivial operations. Approaching $x = a + b + c + d$ as the sequence of 4 loads and three sums is more computationally and memory expensive than just one command, yet reduced variance of commands is likely to benefit race condition analyses. If needed, a specific analysis pass may break the command to more instructions. Thus, with this approach, we are combining reduced search breadth with operations detailed enough for specific analysis type.

In our implementation, this process assumes *morphing*. Expressions do not get removed but are rather changed in place. This reduces the number of allocations required for analysis and positively improves performance.

Transition to VM Instructions. The transition from CFG to VM codes requires the knowledge of operational semantics of the target language. We will not cover it, but would rather show examples of conversion in Table 1.

if, **do**, **for**, **while** and other constructions are almost universally supported in imperative languages, so they form a basis of our language. This transition is the first to introduce the SSA form, and it is irreversible (Fig. 3).

Overview of VM Commands. The commands have the following syntax: **[implicit]** [**dep**] [**property**] *command* where **implicit** is set on commands created without clear user's request, **dep** signifies dependent commands made by diagnostic models, **property** determines the kind of information which can be retrieved by investigating the command, e.g. *safety*, *liveness*, etc. There are several big categories of *commands*:

1. Analysis-related commands.
   - **constraint** *expression* asserts an expression.

Table 1. The partial mapping between language constructs and VM IR

C construct	*IR equivalent*
type x	declare x:type
type f(args)	function f (args) → type end
if (cond)	branch *cond followup-kind* end branch -or- constraint *cond* -or- check *cond*
for (init; cond; inc)	assign *init* branch *cond* loop follows *inc* end branch
do; while (cond)	branch *cond* loop post end branch
while (cond)	branch *cond* loop end branch

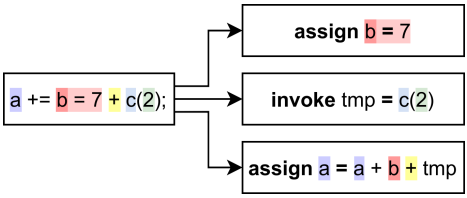


Fig. 3. Morphing between CFG/DAG and VM IR

- **check** *expression* indicates an always false expression that is still evaluated by the final computer/virtual machine.
- **annotate** (*annotation*) designates an ACSL [7] annotation for the next instruction.

2. Control flow altering commands.
   - **enter** [→ *var*] and **exit** signify block start and end, that defines variable visibility scopes.
   - **branch** *cond* [**loop**] [**post**] [*followup*]
     where *followup* can be *or continue* (signifies branch with else clause), *fallthrough* (signifies branch that must be exited, or it will enter the next clause), or *follows inc* (for incrementors)
   - **end branch** *[obj1, ..., objN]* ends the branch and signifies variables changed in a branch, which is immensely useful for variable elimination.

- **invoke** [*variable* =] *function*(*arg1*, ..., *argN*) calls the *function* and saves result to *variable*. The effect differs for intra- or interprocedural analyses.
- **return** *expression* acts as an assignment for the variable stated in **enter**. It is mapped to the **result** ACSL annotation variable.

3. Variable manipulation commands:
   - **declare** *resource* [= *expression*] adds a resource with a given ID to the current visibility list.
   - **load** *resource* prefetches a resource with a given ID, useful in case the variable is physically located on some other computational node.
   - **init** *variable* := *expression* sets the temporary variable to a specific value. It can be safely omitted if the command is processed on the computational node where simplification had been done as temporary variables are bound to values at that time.
   - **assign** *resource* = *expression* sets a resource to a given value.
4. Internal commands:
   - **system** *internal-expression* applies internal data to control or data flow. For example, we call it for replacing user-defined assert functions.
   - **augment** *name*: (*data*) adds an external object to the flow. This process is discussed in the next section.

4.4 Virtual Machine Instructions with Augmented Metadata

The last step feeds VM commands to passes each producing *augmentations* in their formats. For example, in our static analyzer project, we have SMT and Abstract Interpretation solver passes. SMT builds formulas in CVC4 terms and AI builds abstract domains approximating variables. The discovered facts are augmented back to IR (Fig. 4). Augmentation itself could not be helpful if passes did not have a possibility to reuse each other's results. For example, SMT solver may find abstract interpretation' insight about loop important for building better SMT models. A specific implementation might run passes again and again until all needed bits are retrieved, also it decides whether this data is permanent, i.e. saved to the index, or regenerated on demand.

4.5 An Example of a Function Computing Factorial

An example of a simple recursive function computing factorial is listed below. For CVC4 augmentation, we use a slightly modified syntax since the original one is verbose. For variables, we use the following naming schema: **NAME_N**, where NAME is the variable's name, and N is a revision number.

```
function factorial(n:int) -> int
enter -> res
  constraint n >= 0;
  branch n == 0 or continue
    | augment cvc4-sat-result:
      (indeterminate)
```

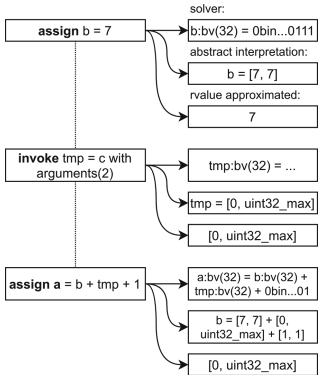


Fig. 4. IR with augmented solver's expressions

```
  return 1
    | augment cvc4:
       (res_1 = 0bin0...01)
    | augment ai:
       (n == 0 => res_1 = 1)
    | augment ai-function:
       (n == 0 => return 1)
    | augment ai-approx:
       (n == 0 => (res_1 = 1))
end branch (res)
implicit branch (n != 0)
  | augment cvc4-sat-result:
    (indeterminate)
  invoke tmp = factorial(n - 1);
    | augment cvc4: (tmp = (<omitted
      due to size constraints>))
    | augment ai: (tmp =
      factorial(n - 1))
    | augment ai-approx:
      (tmp = [n - 1, inf])
  return n * tmp
    | augment cvc4:
      (res_2 = BVMUL(32, n, tmp))
    | augment ai:
      (n != 0 => res_2 = n * tmp)
    | augment ai-function:
      (n != 0 => return n * tmp)
    | augment ai-approx:
      (n != 0 => (res_2 = [n - 1, inf]
        * n = [n(n-1), inf]))
```

```
  end branch
  | augment cvc4-incompatible-branches:
    ((n == 0bin0...0) XOR
     NOT(n == 0bin0...0))
  | augment cvc4:
    ((n == 0bin0...0 =>
       (res_3 = res_1)) AND
     (NOT(n == 0bin0...0) =>
       (res_3 = res_2)))
exit
  | augment meta: (recursive,
    decreasing n, stops at n = 0)
  | augment precond: (n >= 0)
  | augment returns:
    (((n == 0) => return 1) XOR
      (n != 0) => return n *
        factorial(n - 1))
```

5 Type System

The type system [19] is made from scratch to allow for completely virtual types, which are implemented as *unbounded* machine types. It provides a list of integral machinery types of different endianness (e.g. **uint32_le**, **float80_be**, **float128_le**, etc) with unambiguous patterns for cross-conversion, a set of environments for well-known compilers, primarily GCC and Clang, and common CPU types including x86, x86_64, MIPS, private virtual processors. Every expression has either fixed or a dynamic type. The type is fixed on explicit or implicit casts, references, literals. A dynamic type is never saved and only inferred using an internal inference mechanism taking language semantics into account.

The type system provides API for all IR users. Type inference function is of the most common use, the second most used function is a type trait retrieval function, and the third is type conversion. With this API, all IR types get matching capabilities regarding handling complex types and type conversions.

A completely independent type system has the following benefits. First, it adds fine-grained control of the supported features irrespective of CPU types, used compiler or language semantics. Second, different level IRs can save space by unifying the type system across them. Third, there is a better chance of implementing support for conceptual languages, which is our goal in the long run.

6 Serialization

In industry, it is expected that IR can be serialized and deserialized. Midair supports two representations: textual and JSON. The textual representation is

useful for debugging, and that's what is presented in the paper. The deserialization from text is not that efficient as from JSON and it we tend to avoid in real code. JSON representation is suitable for parsing by external tools, for selected NoSQL databases like MongoDB. This is the primary use case, its scheme is presented in paper [18]. The binary serialization is being developed in the moment of writing for more performant indexing.

Why is serialization important? We pursue a few goals. First, program state space may be large, and it is highly probable that it will not fit the RAM. We unload global resources as soon as possible to avoid memory issues. Second, incremental analysis requires saving knowledge about the program to disk. The analyzer uses indices to find all missing yet required semantics and then loads it from disk. Third, we have a work-in-progress mechanism for searching within the state space using program query language, and compared to incremental analysis, it requires not just required semantics, but potentially all the semantics of a program.

The last important point is serializing target-independent binaries. When binary serialization is finished, it would be possible to pack IR to binary for distribution among cluster nodes. However, the input program target independence is not ensured at the moment of writing, thus after the IR is generated, it is not possible to relocate the program representation to another CPU architecture. It might be an interesting challenge for future versions.

7 Evaluation

Comparing IRs from a *practical* perspective is often impractical. The reason is that implementation drives the results, they are barely comparable. The efforts to build two distinct implementations—yet sufficiently optimized to ensure fair comparison—are too high. Instead, we prepared a study of practical implications of decisions made in Midair, and how they impact the real-world use in our extensible static analyzer [19].

7.1 Decisions and Their Impact

There were a few critical decisions made to comply with the requirements.

First, the arbitrary type system. The API supports type and transformation definitions, type inference, however, the actual type handling is on IR users. In result, the code directly related to type support makes 9.9% of CVC4-based analysis pass code base and 1% for abstract interpretation implementation. This difference is caused by the fact that Abstract Interpretation types map to internal type system more naturally.

Second, the static single assignment form choice. The abstract interpreter is based on a non-SSA version of IR, and CVC4 pass uses SSA. This difference requires efforts to bridge the gap between versioned and unversioned variables, allowing for the cooperation. In the project, the problem was fixed by essentially

doing all computations in SSA form but removing versions when needed. That consideration reduced the need to save non-SSA IR.

Third, low-footprint morphing between IRs. In the testing, it gives significant savings. It was measured that IR preparation takes 10% of execution time while converting to SMT and verifying it requires up to 70% of the time.

Fourth, the database support. In practice, saving all objects to MongoDB increases execution time by the factor of 2–2.5, 1.5–2 for internal binary cache facility. However, when applied only to the global scope in headers, the increase for both facilities is only 1.1–1.2. The memory savings are more significant: it may cut memory usage by the factor of 2–3. The actual value depends on the data structure.

7.2 IR Properties

A few properties of the IR were evaluated:

- *Simple translation.* Considering that Midair is imperative, the conversion from LLVM and other "low-level" imperative IRs is trivial, however, not advantageous. The lack of knowledge of input program, e.g. loops, decreases the details the interpreter of Midair might take from it. So, at the moment of writing, the authors believe that evaluating the bidirectional translation requires further examination.
- *Hardware neutrality.* No types are pre-defined: they have to be supported by an interpreter. Thus, the language is completely hardware-neutral. To prove that statement, the authors apply it to several CPUs, including x86, ARM, MIPS (both big-endian and little-endian) and one private virtual machine.
- *Extensibility.* The language has a few "backdoors" for new functionality: first, a strict expression system is not provided. Second, the commands do not have a predefined storage format. With that, it is trivial to change commands and expressions when needed.
- *Semantic gap between source languages and the IR* is not big if only program semantics is considered. The transformation routine implies desugaring of syntax, so the issues arising from the syntax itself are not seen on Midair level.

8 Discussion

The main trade-off comes from a single command size. VM IR is not compact in the sense that it is not possible to use several bytes per command like it is in RISC assembly. A big command means fewer commands and less decoding, resulting in a higher level of abstraction. Also, IR is not serializable to just one representation. The advantage here is that more disk space is left for routine semantics. The disadvantage is that local and temporary variables consume the global symbol table. However, no changes are planned at the moment.

The other issue is a number of IRs to translate between. For C, it is Clang AST, GST or DAG, CFG with DAG elements, VM IR and augmented VM

IR. Even for intended cheap moving, running translation is costly. Our solution is that AST, GST, and CFG with DAG elements are not preserved as they are not required later. VM IR is what saved globally, and solver augmentation is volatile. The latter comes from the fact that not all deductions might be permanent. It has a performance effect since SMT conversion is by far the most expensive operation according to profiling data (40+% of Callgrind samples are taken when converting to SMT). This issue is being worked on. That profiling data also shows that translation from source to Clang AST and then to VM IR takes at most 5% of the time (this timing represents a subset of work done for IR preparation and transformations), which is acceptable.

We believe the development direction going forward would be to further discriminate between semantical actions (represented by complete VM IR commands) and syntactic constructions (represented by expressions). At the moment of writing, a big part of the syntax is passed to solvers as is, of course, not without extensive help for types and control flow management provided by the framework.

The foremost goal for the whole intermediate representation is to pursue more analysis features. For example, fusing variables in Static Single Assignment form is currently left entirely to the code, it is not a clearly defined command. There is no detection of loop transformations or even clear indication that the loop's body has independent iterations. It will all simplify solver passes, but we have to collect more experience to make such a generalization smooth.

The development started when MLIR did not exist. What can be interesting is representing Midair as MLIR dialect, making performant IR-to-native code compilation for faster analysis.

9 Conclusion

We presented Midair, a multi-purpose intermediate representation for program analysis. Its position among other IR was shown. We demonstrated how four representations of the IR materialize and transform: from Directed Acyclic Graph (DAG) to Control Flow Graph with DAG elements, and then to virtual machine (VM) IR and VM IR with augmented metadata. The important property of our IR is *morphing*: basic expressions are passed from DAG to CFG and VM IR as is without major modifications, saving time and RAM/disk space. The latter encourages cooperation between analysis passes. Another property is an ability to serialize any level IR to disk: thus it is possible to save intermediate results and continue later, allowing for incremental analysis. For VM IR, we presented a set of commands related to analysis, control flow and variable manipulation, and internal needs. The IR has its strong points in comparison to well-known IRs and we certainly look forward to its further development.

References

1. Apache Commons BCEL. https://commons.apache.org/proper/commons-bcel/
2. GNU Compiler Collection (GCC) internals: RTL. https://gcc.gnu.org/onlinedocs/gccint/RTL.html
3. LLVM Language Reference. https://llvm.org/docs/LangRef.html
4. Standard ECMA-335 - Common Language Infrastructure (CLI). http://www.ecma-international.org/publications/standards/Ecma-335.htm
5. Aho, A.: Compilers: Principles, Techniques, and Tools. Always Learning. Pearson, London (2014)
6. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 305–343. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11
7. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI C specification language (2008)
8. Bonetta, D.: GraalVM: metaprogramming inside a polyglot system (invited talk). In: Proceedings of the 3rd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection, pp. 3–4 (2018)
9. Cooper, K., Torczon, L.: Engineering a Compiler. Elsevier Science, Amsterdam (2011)
10. Dillig, I., Dillig, T., Aiken, A.: SAIL: static analysis intermediate language with a two-level representation. Technical report (2009)
11. Dullien, T., Porst, S.: REIL: a platform-independent intermediate representation of disassembled code for static code analysis (2009)
12. Hendren, L., Donawa, C., Emami, M., Gao, G., Sridharan, B.: Designing the McCAT compiler based on a family of structured intermediate representations. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua, D. (eds.) LCPC 1992. LNCS, vol. 757, pp. 406–420. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-57502-2_61
13. Hovemeyer, D., Pugh, W.: Finding bugs is easy. ACM SIGPLAN Notices **39**(12), 92–106 (2004)
14. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45337-7_18
15. Lattner, C., Pienaar, J.: MLIR Primer: A Compiler Infrastructure for the End of Moore's Law (2019)
16. Leino, K.R.M.: This is Boogie 2. Manuscr. KRML **178**(131), 9 (2008)
17. Menshchikov, M.: Scalable semantic virtual machine framework for language-agnostic static analysis. In: Distributed Computing and Grid-technologies in Science and Education, pp. 213–217 (2018)
18. Menshikov, M.: An approach to storing program semantics in static program analysis. In: Proceedings of The L Annual International Conference on Control Processes and Stability (CPS 2019), pp. 313–320 (2019)
19. Menshikov, M.: Equid—a static analysis framework for industrial applications. In: Misra, S., et al. (eds.) ICCSA 2019. LNCS, vol. 11619, pp. 677–692. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24289-3_50

20. Merrill, J.: GENERIC and GIMPLE: a new tree representation for entire functions. In: Proceedings of the 2003 GCC Developers' Summit, pp. 171–179. Citeseer (2003)
21. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45937-5_16
22. Rigger, M., Grimmer, M., Wimmer, C., Würthinger, T., Mössenböck, H.: Bringing low-level languages to the JVM: efficient execution of LLVM IR on Truffle. In: Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages, pp. 6–15 (2016)
23. Torlak, E., Bodik, R.: Growing solver-aided languages with Rosette. In: Proceedings of the 2013 ACM International Symposium on New ideas, New Paradigms, and Reflections on Programming & Software, pp. 135–152 (2013)
24. Vallée-Rai, R. Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON 1999, p. 13. IBM Press (1999)
25. Wilson, R.P., et al.: SUIF: an infrastructure for research on parallelizing and optimizing compilers. ACM SIGPLAN Notices **29**(12), 31–37 (1994)
26. Wimmer, C., Würthinger, T.: Truffle: a self-optimizing runtime system. In: Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, pp. 13–14 (2012)
27. Yellin, F., Lindholm, T.: The Java virtual machine specification (1996)