VAST is a Program Analysis–Focused Compiler

**Fine-Grained Steps**
Various abstraction levels are useful for different program analyses. VAST enables viewing source code at various stages of translation from AST to LLVM IR. Each step of the LLVM code generation process is modeled as a distinct pass or a dialect.

**Provenance**
To link results of low-level analysis to the source code or to incorporate high-level structural insights into low-level analysis, it is essential for VAST dialects to maintain bidirectional provenance information across representations.

**Program Abstractions**
Not all information is necessary for specific analyses. A different source view can yield more precise results and simplify analysis design. For that, VAST supports user-defined program abstractions (dialects) compatible with the rest.

---

**Compilation**
VAST is designed as a Clang driver wrapper. It translates a new AST containing LLVM IR generation, guided by two key principles:

1. **Fine-grained code representation:** VAST representations may differ from the LLVM code generator and distinct MLIR passes and dialects, such as type desugaring or ABI translation. This approach not only facilitates easier code modification but also allows stepping at the most suitable representation for program analysis.

2. **Consciousness of intermediate steps:** This allows independent code generation to be exchangeable, forming the basis of cross-layer high-level low-level code representations.

VAST offers both specialized VAST dialects and standard MLIR dialects, following the generation of LLVM IR. Future developments aim to incorporate Confluent as a target dialect.

---

**Applications**
VAST keeps snapshots of intermediate MLIR modules, also called the Tower of IRs. These allow us to perform analysis on the most suitable level and make it easier to link analysis results back to the user.

---

**MLIR Dialects**

**High-level dialect** resembles a Clang AST–like dialect and serves as the starting point for VAST, retaining as much information as possible for later stages of code generation.

**Built-in dialect** specifies Clang’s C/C++ built-in operations and types.

**ABI dialect** describes the mapping between high-level types and the LLVM ABI.

**Low-level dialect** is akin to an LLVM dialect, designed to be compatible with high-level structured control flow and high-level types.

**Core dialect** defines generic interfaces and types used in VAST, like symbols, functions, or scopes.

**Meta dialect** enables operations to be tagged with user-defined locations (IDs) and connects operations across the layers of the Tower of IRs.

**Analyses dialects** represent programs in simplified representation for specific scenarios like points-to analysis, deserialization, invariants, etc.

**Future dialects** are planned to incorporate more specific information, such as concurrency, libc, library-specific abstractions, lifetimes, etc.

---

**Tower of IRs**
Each operation in the Tower of IRs employs the MLIR library framework to handle grouping layer: adapted here through various illustrative programs only with just a few examples.

---

**Decompilation**
Decompilation becomes simpler with VAST, as it offers IRs for granular steps. We can progressively elevate control flow and type information to more expressive dialects.

---

**Lower high-level dialect**

**Simplify**
MLIR provides tooling to create IRs for transpilation to languages like Rust or newer C++ versions. VAST can serve as a generator for the most suitable IR for this task.

---

**Codegen through Clang AST**

<table>
<thead>
<tr>
<th>Model IR</th>
<th>Transpiled source</th>
<th>Transpiled IR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA).

The views, opinions, and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Explore on Compiler Explorer