TU WIEN Informatics

Efficient parsing of OpenVADL

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Technische Informatik

eingereicht von

Michael Nestler
Matrikelnummer 11776863

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 21. Oktober 2024

Michael Nestler	Andreas Krall

TU WIEN Informatics

Efficient parsing of OpenVADL

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Michael Nestler
Registration Number 11776863

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, October 21, 2024

Michael Nestler	Andreas Krall

Erklärung zur Verfassung der Arbeit

Michael Nestler

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich keiner generativer KI-Tools als Hilfsmittel bedient habe.

Wien, 21. Oktober 2024

Michael Nestler

Acknowledgements

The entire VADL team has been a great help during the implementation of this project, advising on key decisions and providing well-founded opinions on technical decisions. I thank them for their welcoming attitude and collegial atmosphere.

In particular, I want to thank my advisor, Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall, for his continued feedback on the progress of the implementation, as well as his expertise in the VADL domain space. Due to his advice, I was able to avoid several pitfalls in the implementation. Additionally, his knowledge of the original VADL implementation proved invaluable in times of profound confusion.

I would also like to thank Florian Freitag, B.Sc., who is working on an adjacent topic in the same project, for the effective collaboration and friendly communication.

Parts of this project were implemented by Andreas Krall and Florian Freitag. Florian Freitag implemented the preliminary expression parsing, while Andreas Krall implemented the expression reordering algorithm.

Kurzfassung

Die Vienna Architecture Description Language (VADL) ist eine Processor Description Language (PDL). Die ursprüngliche Implementierung des VADL-Parsers verwendet *Xtext*, ein Framework, welches auf dem ANTLR Parser-Generator aufbaut und zusätzliche Artefakte wie einen Typchecker und IDE-Werkzeuge generiert. OpenVADL ist eine von Grund auf neue Implementierung von VADL. Wenngleich OpenVADL mit alten VADL-Spezifikationen weitgehend kompatibel bleiben soll, bieten sich durch die Neuimplementierung Gelegenheiten, die Sprache zu erweitern und mitunter andere Richtungen einzuschlagen.

Ein Hauptziel der neuen Implementierung des OpenVADL Parsers war, die Laufzeit mindestens um einen Faktor 5 zu verbessern. Durch die Verwendung von Coco/R als Parser-Generator und eine Reduktion der Verarbeitungspässe konnte eine Laufzeitverbesserung vom Faktor 14, mit der Verwendung von Ahead-of-Time-Kompilierung sogar bis zu 150 erreicht werden.

Abstract

The Vienna Architecture Description Language (VADL) is a processor description language (PDL). The original VADL parser used Xtext, a framework built on top of the ANTLR parser generator, which generates artifacts such as IDE tooling and type checkers in addition to a parser. OpenVADL is a completely new implementation of VADL. While OpenVADL aims to remain largely compatible with specifications written for VADL, the new implementation takes the opportunity to improve some aspects of the language and take a different direction in some aspects.

The main goal of the new OpenVADL parser was an expected runtime improvement factor of 5. By choosing Coco/R as parser generator and reducing the number of passes performed on the abstract syntax tree, we reached a runtime improvement of up to 14, which we furthermore increased to 150 by using Ahead of Time compilation.

Contents

CHAPTER 1

Introduction

The Vienna Architecture Description Language (VADL) is a Processor Description Language (PDL) [HHH+24]. A PDL is a language used to describe instruction set architectures and the structure of processors [MD08]. VADL's goal is to generate compilers, linkers, simulators, assemblers, disassemblers, debuggers, and hardware definitions from a high-level processor specification.

VADL as a language distinguishes between the description of an Instruction Set Architecture (ISA), micro architectures, application binary interfaces (used for generating compilers), and microprocessors (used by the hardware generators and the simulators) [MD08]. This strict separation enables readable specifications and separation of concerns.

While some results have been published [HHH+24, HK23], implementation details and source code are not openly accessible due to having been developed as part of a research cooperation.

The new implementation, named OpenVADL, aims to be largely backwards-compatible, while containing no code of the original implementation as well as making different architectural decisions. In particular, the speed of the VADL parser has hindered its widespread adoption, as large specifications can take dozens of seconds to parse.

By choosing a lightweight parser generator, using fewer processing passes, and employing modern technologies, OpenVADL parses specifications up to 150 times faster than the original VADL implementation. Despite not being feature-complete yet, these developments are a promising beginning of the new implementation.

CHAPTER 2

Background and Related Work

2.1 The original VADL parser

The original implementation of VADL [HHH+24] was built using the *Xtext* framework [xteb, Bet16]. This framework uses ANTLR [PQ95, PF11] as the foundation of its parser, but also generates additional artifacts that allow integration into the Eclipse Integrated Development Environment (IDE) for the specified language. The implementation of the original VADL project was written in *Xtend*, which is a Java dialect and part of the *Xtext* project [xtea].

As part of the parser generation process, *Xtext* generates a model for the parsed text that tightly follows the written grammar rules. VADL refers to this model as the Concrete Syntax Tree (CST) [HHH+24, 4.2 Language Parser]. After parsing a VADL specification, the VADL parser then converts this CST into an Abstract Syntax Tree (AST) that only contains the relevant information about the VADL specification.

Macro expansion and CST validation were implemented as distinct passes that operated on the CST, while module importing, symbol resolution, type inference, recursive call detection, and AST validation were implemented as passes that operated on the AST. In total, a VADL specification went through 22 passes before it was ready to be further used in compiler generation or simulation.

Complex specifications can take dozens of seconds to parse, which makes specifying complex instruction sets with a large amount of macros burdensome.

2.2 Introduction to VADL

A VADL specification consists of definitions and optional import declarations. Typical specifications contain global definitions, followed by instruction set and micro architecture definitions. Listing 2.1 shows the definition of a fictional ISA with two instructions, `ADD` and `SUB`.

```
1  constant RegIndexLen = 4
2  constant RegValueLen = 16
3  constant AddrLen = 32
4  using RegIndex = Bits<RegIndexLen>
5  using RegValue = Bits<RegValueLen>
6  using Address = Bits<AddrLen>
7
8  instruction set architecture ISA = {
9    register file X : RegIndex -> RegValue
10   memory MEM      : Address -> Bits<8>
11
12   format F : Bits<16> =
13   { rs2 : RegIndex
14   , rs1 : RegIndex
15   , rd  : RegIndex
16   , op  : Bits<4>
17   }
18
19   instruction ADD : F = X(rd) := X(rs1) + X(rs2)
20   encoding ADD = { op = 0 }
21
22   instruction SUB : F = X(rd) := X(rs1) - X(rs2)
23   encoding SUB = { op = 1 }
24
25   assembly ADD, SUB = (mnemonic, " ", register(rd), ",", register(rs1)
         , ",", register(rs2))
26 }
```

Listing 2.1: Typical VADL definition

The example in Listing 2.1 starts with constant definitions (`constant`) and type aliases (`using`) that are relevant for the entire specification. Types in VADL consist of a type name, like `Bits`, and an optional bit-width surrounded by angle brackets (< and >).

A register file in VADL is assigned a name (in this case, `X`) as well as an index and an element type. The index type specifies how the register file is indexed, and the element type specifies the type of the registers in the file. Register file indexing in VADL uses a style similar to function calls in other languages (e.g., `X(rd)`). Memory access works similarly, although sometimes authors of specifications may want to access multiple cells of memory at once. In this case, authors can specify a vector size like `MEM<size>(address)`.

A `format` has a name, a type, and multiple constituent fields. By using formats, authors can assign meaningful names to parts of a whole binary value. In this example, a `Bits<16>` value can be partially accessed by the top three `Bits<4>` parts and the trailing `op` bits.

VADL `instructions` have a name, an instruction format type and a behavior. The fields of the format type are available within the instruction behavior, the instruction encoding and the instruction assembly. The examples `ADD` and `SUB` show how the format's register fields can be used to access a register file.

An `encoding` definition instructs VADL on the details of the instruction's binary encoding. Only fields that identify the instruction will be specified in the encoding. The name of the encoding has to correspond to the name of a corresponding `instruction`.

An `assembly` definition defines the way an instruction is converted to readable assembly code. As with the `encoding`, its name has to refer to a corresponding `instruction`. As shown in the example, the assembly of multiple instructions can be defined with a single definition using the `mnemonic` keyword, which stands for the respective instruction name.

The example in Listing 2.1 shows the usage of several syntax types: definitions like `constant AddrLen`, statements like the assignment in the `ADD` instruction definition, expressions like `X(rs1) + X(rs2)`, numeric literals like `32`, string literals like `","`, and encodings like `op = 1 .`

2.2.1 Macros in VADL

The VADL macro system was conceived and implemented by Christoph Hochrainer et al. in [HK23]. The article goes into detail on the many design decisions that went into the original macro system. It also elaborates on literature related to macro systems and the different needs of a PDL's macro system compared to a general purpose programming language's.

OpenVADL uses a slightly extended version of the original VADL implementation's type system (see 4.1 `Defs` as a new syntax type). Figure 2.1 shows the syntax types of OpenVADL as well as their subtype relationships.

Listing 2.2 shows a minimal VADL specification containing one instruction. In this specification, the names of the definitions (`X`, `F` and `ADD`), the names used in the assignment statement (`X`, `rd`, `rs1`, and `rs2`), and the name of the type literals (`Bits`) are identifiers (type `Id`). A statement (`Stat`) can be seen in the behavior of the `ADD` instruction, whereas expressions (`Ex`) can be found in the bit width of the type literals, as well as both sides of the assignment statement (e.g., `X(rd)`, which is an expression of type `CallEx`). In binary expressions, the operator (e.g., `+`) has the type `BinOp`.

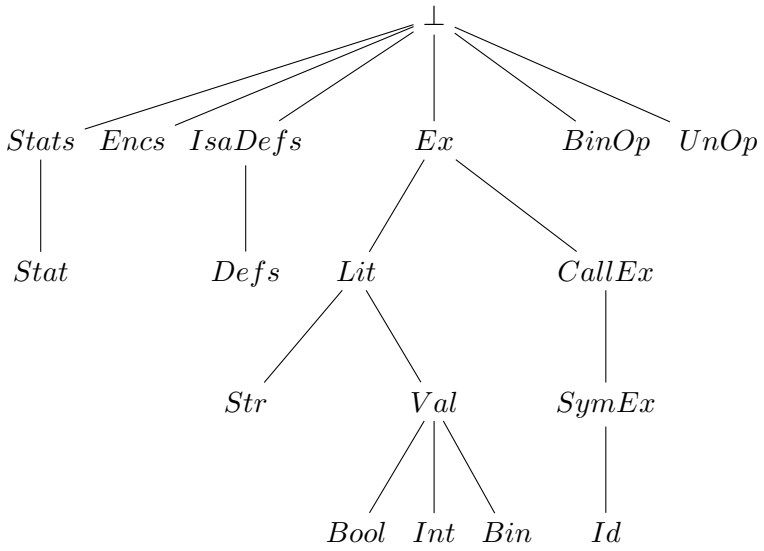

Figure 2.1: Syntax types in the OpenVADL macro system

```
instruction set architecture ISA = {

  register file X : Bits<4> -> Bits<32>

  format F : Bits<16> =
  { rs2 : Bits<4>
  , rs1 : Bits<4>
  , rd  : Bits<4>
  , op  : Bits<4>
  }

  instruction ADD : F = X(rd) := X(rs1) + X(rs2)
}
```

Listing 2.2: Example of a VADL instruction

In VADL's macro system, macro elements can produce language elements of any syntax type defined in Figure 2.1, and can be used wherever the respective syntax type can be used. The following subsections describe the different kinds of macro elements.

2.2.2 Model instances

In VADL, structural code reuse can be implemented by using `models`. Models can be parameterized and produce a node of one of the syntax types described in Figure 2.1. The example in Listing 2.3 shows a model that produces an arbitrary instruction, and two invocations of that model to produce `ADD` and `MUL` instructions.

```
model CreateInstr(instr: Id, fmt: Id, behavior: Stats) : IsaDefs = {
  instruction $instr : $fmt = {
    $behavior
  }
}

$CreateInstr(ADD ; I32 ; X(rd) := X(rs1) + X(rs2))
$CreateInstr(MUL ; I32 ; X(rd) := X(rs1) * X(rs2))
```

Listing 2.3: Instantiating a model

As can be seen in the example, the invocation of a model can specify expressions, statements, definitions, encodings and operators directly in the argument list, separated by semicolons (`;`). Using macros in the argument lists of a model instance is also supported. Only arguments that are of a valid syntax type are accepted, where an argument is considered valid if its syntax type is a subtype of the formal model parameter.

2.2.3 Model parameter placeholders

In the example in Listing 2.3, `$behavior` is used as a placeholder for the argument that the model is actually invoked with. As shown, parameters of a model can be used wherever their syntax type is considered appropriate.

2.2.4 Macro matches

In order to compare a macro value with predefined options, the macro `match` can be used. It will conditionally insert the matching options into the syntax tree. In the example in Listing 2.4, a user can switch between 32 and 64 Bit length by setting the appropriate `Arch` to `x64`. A default case, characterized by the underscore symbol (_), always needs to be provided.

```
constant AddrWidth = match : Ex ($Arch() = x64 => 64; _ => 32)
using Address = Bits<AddrWidth>
memory MEM : Address -> Bits<8>
```

Listing 2.4: Matching on `Id` nodes

2.2.5 Built-in macro functions

To append or prepend string snippets to `Id` values, the macro function `ExtendId` is provided. It will produce an `Id` by concatenating all given arguments, where `Id` arguments are first converted to a string literal (`Str`). This is especially useful if one `model` is used to produce multiple variants, e.g., signed and unsigned operations, or 32 and 64-bit variants.

```
constant ExtendId(A, "2", B, "3") = 1
// equivalent to
constant A2B3 = 1
```

Listing 2.5: Simple `ExtendId` usage

The macro function `IdToStr` converts a VADL `Id` to a corresponding string literal. This is useful in situations like an `assembly` definition, where the value of the instruction's name may be extended by a suffix and then turned into a part of the assembly.

```
model CreateAssembly(instr: Id, suffix: Str): IsaDefs = {
  assembly $instr = (IdToStr(ExtendId($instr, $suffix)), " rd")
}
$CreateAssembly(SET ; "32")
```

Listing 2.6: Using `IdToStr` with `ExtendId`

The outcome of the model invocation in Listing 2.6 is an `assembly` definition for the instruction `SET` with a value of `"SET32 rd"`.

2.2.6 Advanced types in macros

The type hierarchy in Figure 2.1 only covers the basic syntax types in VADL. A specification can also define composite types, which can then be used in the formal parameters of a `model` definition.

One of the available classes of composite types are record types. A record consists of named fields, each with their own syntax type. Records can be nested, although a record name can only be used if it has been defined earlier in the specification. They are defined using the `record` keyword. Record members are accessed using the dot (`.`) operator, and record values are created using parentheses, where record members are separated with a semicolon (`;`) symbol. Listing 2.7 shows an example of the usage of records, where the model `Instr` takes an argument of type `NameFmtAndStats`.

```
record NameFmtAndStats =
{ name: Id
, fmt: Id
, stats: Stats
}

model Instr(data: NameFmtAndStats): IsaDefs = {
  instruction $data.name : $data.fmt = {
    $data.stats
  }
}

$Instr( (ADD ; F_Type ; X(rd) := X(rs1) + X(rs2)) )
```

Listing 2.7: Using `record` types

Another kind of composite type is VADL's `model-type`. It allows using references to `model` definitions as arguments in model instantiations, which allows the definition and usage of higher-order macros. In OpenVADL, a model reference is assignable to a `model-type` parameter if all its parameters are supertypes and the result type are subtypes of the `model-type`'s respective parameters and result type. Listing 2.8 shows a specification wherein a reference to the `FInstrFactory` model is used as an argument for the invocation of another model.

```
model-type InstrFactory = (Id, Id, Stats) -> IsaDefs

model FInstrFactory(name: id, fmt: Id, stats: Stats): IsaDefs = {
  instruction $name : $fmt = {
    $stats
  }
}

model CreateAddInstr(factory: InstrFactory): IsaDefs = {
  $factory(ADD ; F_Type ; X(rd) := X(rs1) + X(rs2))
}

$CreateAddInstr(FInstrFactory)
```

Listing 2.8: Using `model-type` in macros

2.3 Binary expression parsing

Nested binary expressions in the original VADL implementation are always explicitly enclosed by parentheses. In OpenVADL, binary expressions without explicit parentheses are allowed and parsed using a well-defined operator precedence.
Table 2.1 shows the binary operator precedence, weakest-precedence operators first.

Operator	Description
$\|\|$	Logical OR (weakest precedence)
$\&\&$	Logical AND
$\in, \notin, in, !in$	List inclusion operators
$\|$	Binary OR
$\hat{}$	Binary XOR
$\&$	Binary AND
$=, != $	Equality operators
$>=, >, <, <=$	Comparison operators
$>>, <<, <>>, <<>$	Shifts and rotations
$+, -, +\|, -\|$	Additions (optionally saturated)
$*, /, \%, *\#$	Multiplications (strongest precedence)

Table 2.1: Binary operator precedence

2.3.1 Operator precedence in compilers

One approach to implementing operator precedence in compilers is by using dedicated grammar rules. The article [Aas95] lays out a process of transforming precedence grammars (grammars where ambiguous rules are annotated with a numerical precedence) into unambiguous context-free grammars. In addition, the presented algorithm supports languages with postfix and prefix operators in addition to infix operators. The article also provides definitions for precedence correctness of trees and operator covering.

As outlined in [Aas95], this style of precedence implementation has a number of drawbacks. Firstly, it requires intermediate terms whose meaning can be unintuitive to a reader. Secondly, it is not able to handle operators with dynamic, user-defined precedences.

As an alternative, [LdR81] lays out a mechanism of implementing operator precedence by not considering precedence during parsing, instead performing tree operations on the already-parsed expression tree. The main idea is to successively modify parts of the binary expression tree to resolve local precedence mismatches until eventually, the entire tree is well-ordered. A proof of correctness as well as time and space linearity is provided too.

Figure 2.2 shows the expression $2 + 3 * 10 > 5$ parsed as a left-sided binary expression tree, as well as the same tree ordered according to operator precedence.

2.3.2 Precedence in OpenVADL

As binary expressions in VADL can also contain *BinOp* macros as the operator (e.g. `5 $op 2`), binary expression trees can only be reordered when every binary expression in the tree has a non-macro operator. For this reason, using the precedence grammar approach described in [Aas95] proved impossible - instead, binary expressions are parsed as-is as a left-sided binary tree and reordered at a later point as described in [LdR81] .

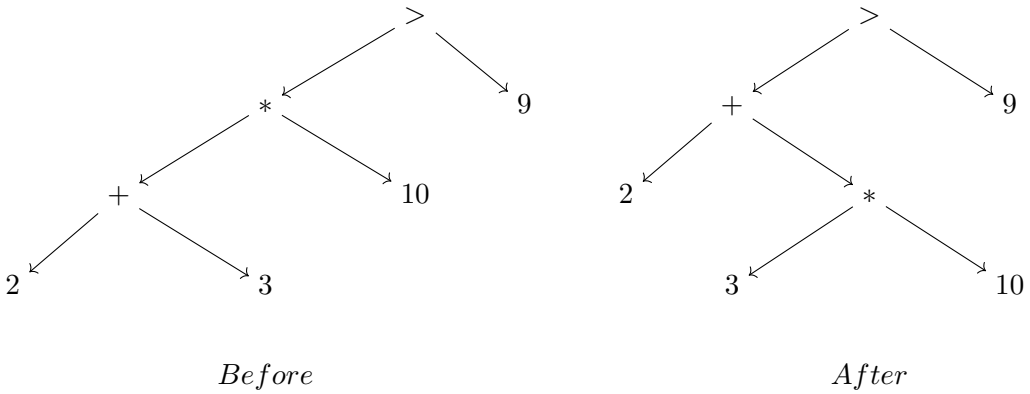

Figure 2.2: A binary expression tree before and after operator reordering

Binary expressions without *BinOp* macros are reordered immediately after they are parsed. Other binary expression trees are marked as *not* reordered, and will be reordered by the macro expansion once all macro elements have been replaced by concrete operators. The expression tree is then marked as successfully reordered to prevent redundant reordering.

2.4 Coco/R and $LL(k)$ grammars

Coco/R is a compiler generator primarily developed at the Johannes Kepler University Linz. It is available for multiple languages, with versions for C#, Java and C++ being the most up-to-date. It generates a recursive descent parser from an attributed grammar [WLM03, Mö91]. Usually, recursive descent parsers are only able to parse languages of grammars in the $LL(1)$ class. $LL(k)$ grammars are grammars that are parsable **l**eft-to-right, with derivations also performed **l**eft-to-right, and with k lookahead tokens. Thus, a $LL(1)$ parser can only parse grammars where a single lookahead token is enough to derive the selected derivation rule unambiguously.

However, despite producing a $LL(1)$ recursive decent parser, Coco/R can parse languages in the $LL(k)$ classes for an arbitrary k by using predicates and allowing token peeking. Not only does this enable Coco/R to parse a greater class of languages, it also allows the grammar author to make their grammar easier to read.

```
1 instruction set architecture ISA = {
2   register ra : Bits<32>
3   register file R : Bits<5> -> Bits<32>
4 }
```

Listing 2.9: Simple ISA definition

Listing 2.9 shows a short, valid VADL specification. A reasonable approach to distinguishing the `register` and the `register file` definitions is by factorization, i.e.

moving the common prefix `register` into a grammar rule and distinguishing the two cases at the point of divergence, as illustrated in the Coco/R example in Listing 2.10.

```
isaDef = registerOrFile.
registerOrFile = "register" ( "file" registerFileDef | registerDef).
registerFileDef = id ":" type "->" type.
registerDef = id ":" type.
```

Listing 2.10: Disambiguation via factorization

Not only does this increase the amount of rules, it also impairs the readability of the grammar specification. Coco/R conflict resolvers can be used to transform this into a readable, unambiguous grammar that uses a more intuitive layout, as shown in Listing 2.11. The corresponding syntactic predicate is shown in Listing 2.12.

```
isaDef = IF (isRegisterFile()) registerFileDef | registerDef.
registerFileDef = "register" "file" id ":" type "->" type.
registerDef = "register" id ":" type.
```

Listing 2.11: Disambiguation via resolvers

```
boolean isRegisterFile() {
  return la.kind == REGISTER && scanner.Peek().kind == FILE;
}
```

Listing 2.12: Syntactic predicate

Another way of using these conflict resolver is to decide between different rules depending on the current state of the parser. The example given in [WLM03, Section 4.2] shows a parser that can parse identifier tokens differently depending on the compiler's semantic state. In OpenVADL, this functionality is used do differentiate macro expressions that may syntactically look identical, but should be parsed differently depending on their type (see 3.6.2 Parsing macro expressions).

Coco/R works with attributed grammars, which allow for passing of semantic information up and down the parse tree. Attributes passed up from lower nodes are called *synthesized*, while attributes passed down from higher nodes are called *inherited* [Knu68]. When targeting Java, Coco/R generates a method for each grammar rule, where *inherited* attributes are implemented as method parameters and the *synthesized* attribute is the method's return value. Both synthesized and inherited attributes are optional in Coco/R [WLM03].

To produce an abstract syntax tree in Coco/R, OpenVADL utilizes attributes and semantic actions. This is in contrast to *Xtext*, which first generates a concrete syntax

tree using the grammar specification which then has to be transformed to an abstract syntax tree. Listing 2.13 shows what building an AST using Coco/R attributes and semantic actions can look like.

```
1 binaryExpr<out BinaryExpr expr> =
2   number<out int left>
3   operator<out Op op>
4   number<out int right> (.expr = new BinaryExpr(left, op, right);.)
5 .
```

Listing 2.13: Attributed grammar example

2.5 GraalVM Native Image

GraalVM is a Java Development Kit (JDK) developed by an open source community led by Oracle. It includes the Graal Compiler, a just-in-time compiler; the Truffle language implementation framework, which is used to build interpreters for other languages in Java; the Polyglot API for hosting other languages in the Java Virtual Machine; and the Native Image technology, which compiles Java programs Ahead Of Time (AOT) and produces a self-contained binary executable [Graa].

GraalVM Native Image performs static analysis to determine which parts of the program are actually reachable and discards unreachable parts. It also tries to perform as much static initialization of the program as possible at build time, including `static` blocks and class loading [Grab]. Finally, it compiles the remaining Java bytecode into native machine code. In this step, classic compiler optimizations like method inlining are performed. Due to its characteristic reduction in program start-up time, the GraalVM Native Image technology is of specific interest to OpenVADL.

Because of GraalVM Native Image's use of static analysis, most Java classes' metadata is discarded and not included in the generated executable. However, certain dynamic Java features cannot be fully analyzed at build time. Examples for these features are the reflection APIs, Dynamic Proxies, Java Native Interface (JNI) and runtime class-loading. In these instances, required class metadata needs to be explicitly included in the native image compilation by providing reachability metadata configuration [Grac]. Some libraries include the required configuration files in their distribution [Net], others generate it during compilation by using annotation processing [Pic], and some rely on the developer to manually configure the required metadata. A community-driven effort to provide a centralized repository of metadata configuration files for third-party libraries exists as well [Grad].

In the usual lifecycle of a Java runtime, the JIT gathers runtime behavior data and uses the collected profiles to optimize the program further. As such, the longer a program runs, the more efficient it will become. As GraalVM Native Image compiles the Java bytecode ahead-of-time, any optimizations will be static for the lifetime of the program.

Instead, developers can opt into Profile Guided Optimization (PGO), where one or more training runs of a native image produce a "profile", which is then used to compile a binary with optimizations derived from the profile. [Grae].

As of writing, GraalVM Native Image supports the Serial and the Epsilon garbage collectors on all supported platforms, as well as the G1 garbage collector on Linux.

CHAPTER 3

Implementation

OpenVADL's parser is implemented with a focus on efficiency, which makes a reduction of the number of syntax tree passes a priority. Where possible, the implementation moves logic from dedicated passes into the syntax parsing process.

3.1 Architecture

Figure 3.1 shows a high-level overview of the compiler architecture.

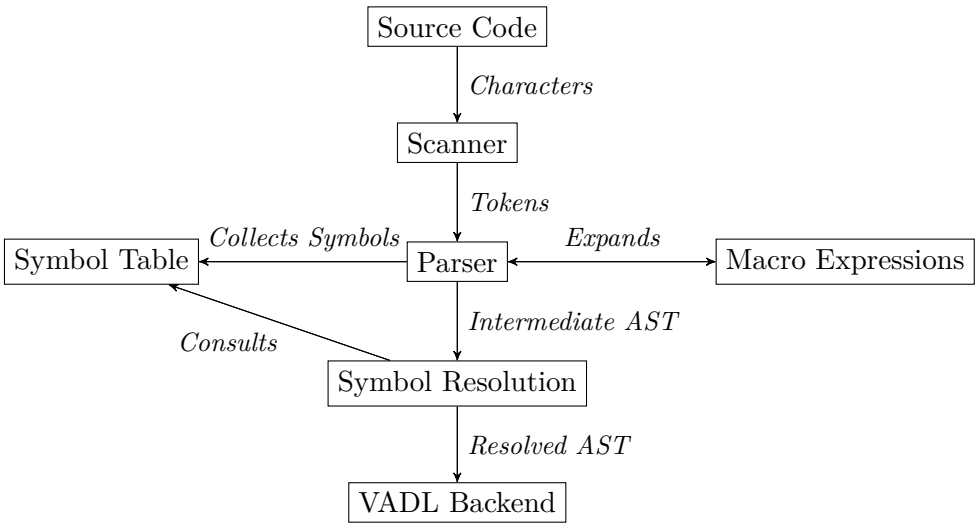

Figure 3.1: OpenVADL compiler architecture

The `Scanner` and the `Parser` are generated by Coco/R. In OpenVADL, the Java variant of Coco/R is used. OpenVADL generates the AST using attributes and semantic

actions (see also 2.4 Coco/R and $LL(k)$ grammars).

During parsing, macro expressions are expanded and symbol definitions are collected. The collected symbol information is then used to resolve the usage of symbols in expressions and type literals. Sections 3.3 Scoping and Symbols and 3.6 Macros describe respective implementation details of these processes.

3.2 Abstract Syntax Tree Modeling

The Abstract Syntax Tree (AST) is modeled using Java classes. The `Node` class is an abstract class that is the superclass of any kind of AST node. Listing 3.1 shows the most important methods of this class. Dedicated subclasses `Definition`, `Statement` and `Expr` are provided for the most common node types. Collections of nodes used in macros are represented via dedicated collection node types. If one of these collection nodes is to be added to the AST, its contained items are added instead.

```
abstract class Node {
  /**
   * Used for syntax type checks in the parser.
   */
  abstract SyntaxType syntaxType();

  /**
   * Stores the corresponding source file location
   * of this node for diagnostics.
   */
  abstract SourceLocation location();

  /**
   * Used to construct a canonical source representation
   * of an AST or single node.
   */
  abstract void prettyPrint(int indent, StringBuilder builder);

  /**
   * Created by the symbol collection, used by symbol resolution.
   */
  SymbolTable symbolTable() { /* ... */ }
}
```

Listing 3.1: Node class for AST modeling

AST elements that can be either a concrete node or a macro invocation are modeled using a `sealed interface`. A `sealed interface` is an `interface` that only permits a specified set of implementations. As an example, Listing 3.2 shows an interface `IsUnOp` which identifies any nodes of syntax type `UnOp`.

```
sealed interface IsUnOp permits UnOp, PlaceholderNode,
    MacroInstanceNode, MacroMatchNode {}
```

Listing 3.2: Interface used to model macro invocation options

In this example, `UnOp` is a concrete `Node` implementation, whereas `PlaceholderNode`, `MacroInstanceNode`, and `MacroMatchNode` are nodes that will be replaced by a `UnOp` node upon macro expansion. The usage of this interface can be seen on `UnaryExpr`, shown in Listing 3.3.

```
class UnaryExpr extends Expr {
  IsUnOp operator;
  Expr operand;
}
```

Listing 3.3: Binary expression model

In the specification in Listing 3.4, the `ADD` instruction of the minimal example in Listing 2.2 is restated. For this instruction definition, the produced AST is shown in Figure 3.2.

```
instruction ADD : F = X(rd) := X(rs1) + X(rs2)
```

Listing 3.4: Minimal VADL instruction

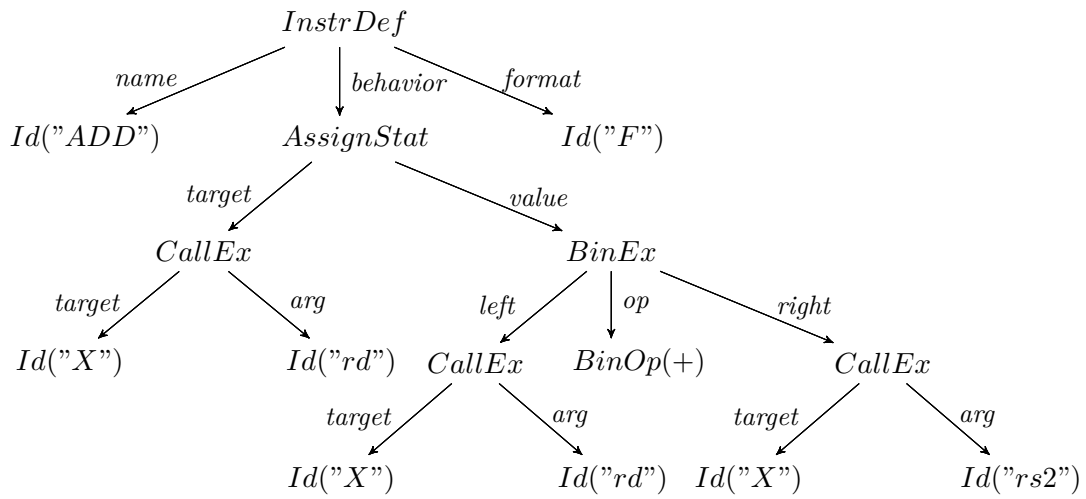

Figure 3.2: Syntax tree of an instruction definition

3.3 Scoping and Symbols

In VADL, symbols are generally visible in the scope that they are defined in, as well as any child scopes. For example, a `register file` defined in an ISA is visible within any definitions within the ISA. Some definitions have additional scoping effects – for example, statements in `instruction` definitions can access symbols defined within the associated `format`.

To manage the definition and resolution of symbols, OpenVADL uses *symbol tables*. A symbol table in OpenVADL contains a list of symbols defined in a scope, as well as an optional reference to the parent scope's symbol table. Every symbol table entry contains a reference to the AST node that originally defined the symbol, called the symbol's `origin`. When resolving a symbol using a symbol table, the symbol is first searched within the table entries, consulting the parent symbol table only if it could not be found therein. A reference to the enclosing scope's symbol table is stored within every AST node that resides in the scope – for example, while an ISA node holds a reference to the root symbol table, its child definitions contain a reference to a child of the root symbol table. Figure 3.3 illustrates this concept, detailing an example ISA specification where every scope's symbol table is shown with its respective declared symbols.

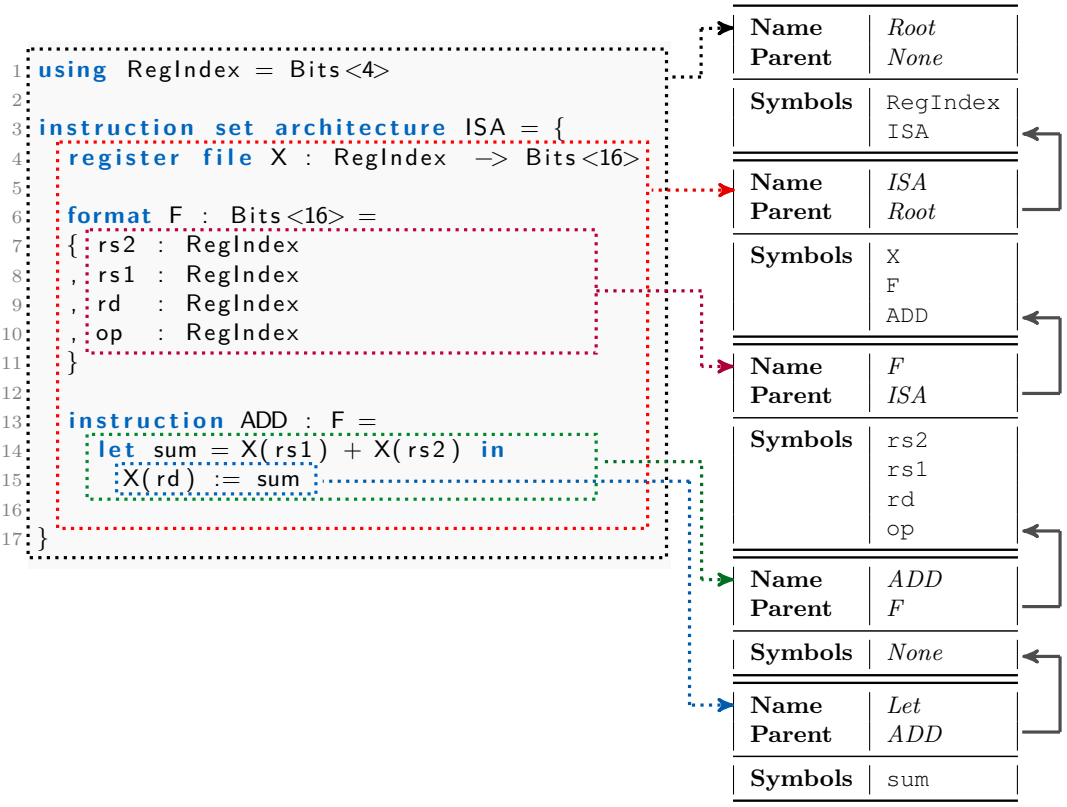

Figure 3.3: Scopes and symbol tables in VADL

The process of assigning AST nodes references to the respective symbol tables and populating the symbol table entries is called **Symbol Collection**. As shown in Figure 3.1, this process occurs during parsing. The result of the symbol collection process is a fully-built tree of symbol tables, with AST nodes referencing symbol tables, and symbol table entries referencing AST nodes.

Some AST nodes, such as binary expression, do not create a new child scope. Therefore, the resulting symbol table tree is of lesser depth and width than the AST. Additionally, since the symbol table tree is based on parent rather than child references, its directionality is reversed as compared to the AST. Figure 3.4 illustrates a simple AST and the corresponding symbol table tree.

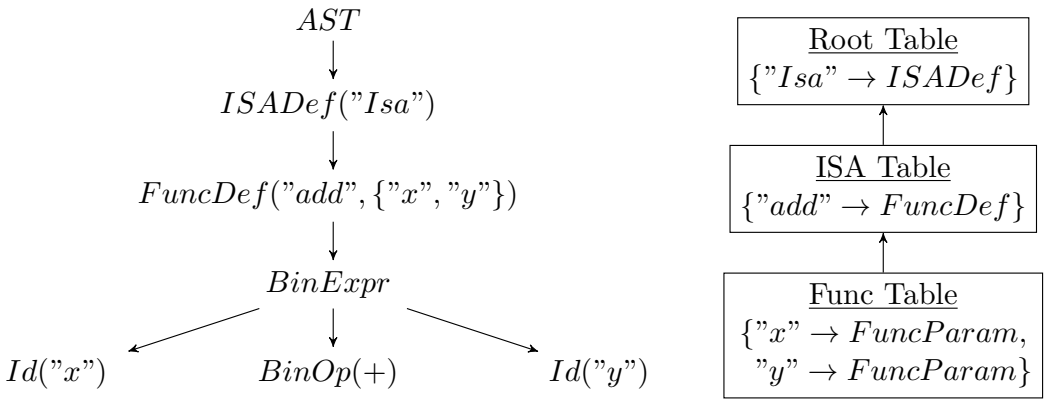

Figure 3.4: An AST with its corresponding symbol tables

3.3.1 The Symbol Resolution Pass

After the parsing process is completed and an AST with the populated symbol tables is created, OpenVADL performs a dedicated AST pass called **Symbol Resolution**. The goal of this pass is to verify that every identifier used in the specification refers to a valid symbol, and to store references to those symbols' `origin` nodes in the AST.

Symbol resolution in OpenVADL is implemented as a recursive process that inspects every node in an AST. For most nodes, symbol resolution consists solely of invoking symbol resolution on its children. For example, an assignment statement's symbol resolution simply invokes the symbol resolution of the target and value expressions. For type literals, the existence of a symbol with the type's name is verified using the node's stored symbol table, and a reference to the type's original definition is stored in the node. For `Id` expressions, if the referenced name can be resolved to a defined symbol, a reference to that symbol's `origin` node is stored within the `Id` node. This allows further processing steps to process the fully-resolved syntax tree without any indirections. Unresolvable symbols lead to an error that reports the symbol name and the corresponding location in the source file.

3.3.2 Macro evaluation

A dedicated symbol table is also used during parsing to store `model`, `model-type` and `record` definitions. This symbol table is consulted when parsing macro expressions, or when a model parameter uses a `record` or `model-type` name (see 2.2.6 Advanced types in macros).

After parsing, this dedicated symbol table is discarded.

3.4 Binary expression ambiguity

Type literals in VADL have the form `Name<sizeExpr>`, where `sizeExpr` can be any numerical expression. However, binary expressions containing the > operator cause an ambiguity between the binary operator and the end of the type literal, as both use the same symbol. Another example of ambiguity in expressions is the `let expr in expr` expression type, because the `in` symbol is a binary operator as well as the terminal of part of the expression.

To resolve these conflicts, OpenVADL restricts the use of binary operators in certain situations. In most situations, all operators are allowed, but the `typeLiteral` rule allows only a subset of operators called `BIN_OPS_EXCEPT_GT`. Similarly, `let` expressions use a subset called `BIN_OPS_EXCEPT_IN`. These restrictions are implemented as *inherited* attributes on the `expression` grammar rule (see 2.4 Coco/R and $LL(k)$ grammars).

If the author of a VADL specification needs to use these binary operators in these situations, it is possible to wrap the expression in parentheses `()`.

3.5 Vector Size Expressions

In VADL expressions, it is possible for memory accesses to specify a vector size for multi-byte memory access.
For example, the expression `MEM<4>(addr) := 7654321` would assign four bytes at the address `addr`.

As these size expressions use the < symbol, distinguishing them from a comparison expression is not trivial. The original VADL implementation uses a syntactic predicate and only interprets the expression as a size expression if a corresponding closing > is found. In OpenVADL, this is implemented using a disambiguation rule `symbolOrBinaryExpression`, which returns a `SymbolExpr` node if a closing > is found and a `BinaryExpr` node otherwise. For situations where no binary expression is appropriate, e.g. in the assignment statement above, an attribute `allowLtOp` is available on the rule to *require* a matching > symbol – or report an error through the parser if not found.

The same ambiguity is also an issue in cast expressions with sizes.
Consider two expressions `X(0) as SInt<5>` and `X(0) as SInt < 5`. The only difference between them is the closing angle bracket (>) of the first expression. To resolve

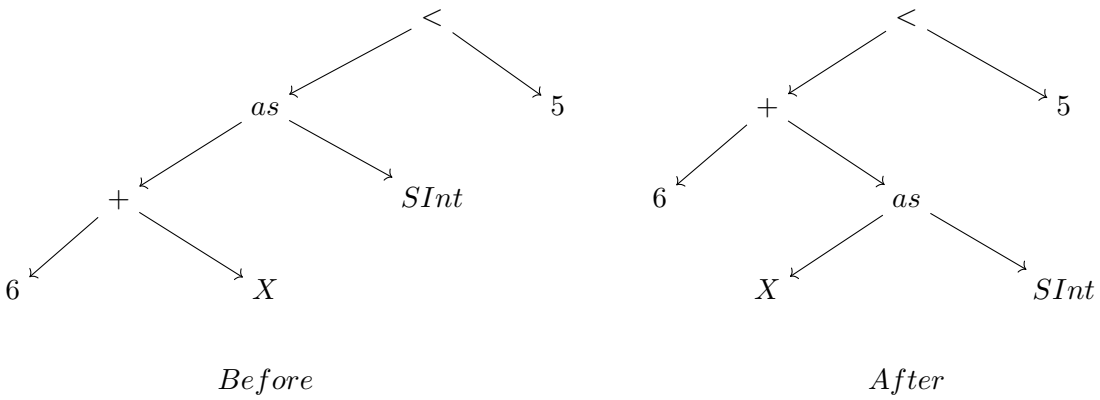

Figure 3.5: A binary expression tree before and after cast reordering

the ambiguity that arises at the opening angle bracket (<), the cast expression parser has a fallback clause to return a `BinaryExpr` if a closing angle bracket (>) is not found. Cast expressions are not implemented as binary operators – instead, their strong precedence is implemented by manually manipulating the last element of a binary expression tree and applying the cast to that element. Figure 3.5 shows an expression tree of the expression `6 + X as SInt < 5`, before and after reordering the cast expressions.

3.6 Macros

3.6.1 Implementing macros in Coco/R

The philosophy of the macro systems and general language rules in OpenVADL are described in 2.2.1 Macros in VADL. Its implementation in Coco/R is non-trivial, and required several design decisions.

Consider the parsing rules of unary expressions, which allow elements of the `UnOp` syntax type as their operator. In the grammar definition, the distinction between concrete elements (e.g., a literal +) and macro elements has two consequences. Firstly, grammar rules that might accept a macro element need an additional alternative in the rule, illustrated in Listing 3.5.

```
1 unaryOperator<out IsUnOp op> (. op = null; .)
2 = SYM_MINUS                  (. op = new UnOp(NEGATIVE); .)
3 | SYM_EXCL                   (. op = new UnOp(LOG_NOT); .)
4 | SYM_TILDE                  (. op = new UnOp(COMPLEMENT); .)
5 | macroNode<out Node n>      (. op = (IsUnOp) n; .)
```

Listing 3.5: Unary operator parsing

Secondly, to avoid macros that are **not** a `UnOp` being parsed using this rule, a check has to be added at the use site, as shown in Listing 3.6. In this listing, `la` is used to refer to

the parser's lookahead token.

```
term<out Expr expr> (. expr = null; .)
= literal<out expr>
| IF(isUnaryOperator(la)
    || isMacroNodeOfType(this, BasicSyntaxType.UN_OP))
  unaryOperator<out IsUnOp op>
  term<out expr>     (. expr = new UnaryExpr(op, expr); .)
| ...
```

Listing 3.6: Unary operator use site

Using the `isMacroNodeOfType` guard, only the proper `UnOp` macro expressions are parsed as unary expressions. The algorithm of this guard is described in detail in 3.6.2 Parsing macro expressions.

The `isUnaryOperator` function compares the lookahead token with the set of valid unary operator tokens. This type of token lookup table check has to occur wherever the `isMacroNodeOfType` guard is used to also allow non-macro syntax to be parsed. Automated tests ensure the parser rules and the respective lookup tables do not diverge.

3.6.2 Parsing macro expressions

Both placeholders (usages of a ‘model‘ parameter within its body) and model instances (invocation of a model) start with a `$` symbol. As Coco/R is a LL(1) parser generator, it needs help to distinguish these different types of macro expressions. This can be especially challenging when parsing `Stats`.

```
1 model OtherStats(): Stats = { /* Snipped */ }
2 model AssignmentTarget(): CallEx = { /* Snipped */ }
3 model LL1ExampleStats(param: Stats, regFileId: Id): Stats = {
4   $OtherStats()              // (1)
5   $AssignmentTarget() := 2   // (2)
6   $param                     // (3)
7   $regFileId(0) := 4         // (4)
8 }
```

Listing 3.7: Different statements with same prefix

In Listing 3.7, the statements (1) through (4) start with a `$` token followed by an identifier token. Three of the four statements also use parentheses, indicating either a model instance or a `CallEx`.

1. The entire line is parsed as a `ModelInstanceStatement`, because `OtherStats` is a model that produces a `Stats` node
2. `$AssignmentTarget()` is recognized as a model instance that produces a `CallEx`, thus parsing the line as an assignment statement
3. `$param` is a parameter of type `Stats`, thus the line is parsed as a `PlaceholderStatement`
4. The `$regFileId` is parsed as a `PlaceholderExpr` of type `Id`, which causes the call argument `(0)` not to be parsed as part of the macro invocation despite looking like a model instantiation

The OpenVADL implementation described in Algorithm 3.1 uses Coco/R’s multi-token lookahead to determine the type of statement that will be parsed.

Algorithm 3.1: Scanning for macro syntax types

```
1  type ← null;
2  while true do
3  |  peekedToken ← Peek();
4  |  if isIdentifier(peekedToken) then
5  |  |  type ← resolveType(type, peekedToken)
6  |  else if isDot(peekedToken) then
7  |  |  if not isRecord(type) then
   |  |  |  // This dot "." belongs to a CallEx, not a record
8  |  |  |  return type
9  |  |  end
10 |  else if isOpenParen(peekedToken) then
11 |  |  if isModelType(type) then
12 |  |  |  return resultType(type)
13 |  |  else
   |  |  |  // This parenthesis "(" belongs to a CallEx, not a
   |  |  |     macro invocation
14 |  |  |  return type
15 |  |  end
16 |  else
   |  |  // Peeked token is not part of the macro expression
17 |  |  return type
18 |  end
19 end
```

These same ambiguities also occur when using macro `match`, thus it also requires a lookahead parsing. However, macro `match` always specifies the produced types, which leads to a much simpler implementation as shown in Algorithm 3.2. Macro `match` is described in detail in 2.2.4 Macro matches.

Algorithm 3.2: Scanning for macro match type

```
1 if isMatch(Peek()) and isColon(Peek()) then
2 |  return parseType(Peek())
3 else
4 |  return error
5 end
```

CHAPTER 4

Changes to the VADL language

The OpenVADL implementation changes some aspects of VADL. Parts of the language have been simplified, while other changes introduce features that simplify authoring specifications.

4.1 `Defs` as a new syntax type

The original VADL implementation supported using `IsaDefs` in macros, but not any non-ISA definitions. This meant that definitions of application binary interfaces and micro processors could not use macros to define constants, formats and type aliases, as these `IsaDefs` definitions were not allowed in non-instruction set definitions.

OpenVADL introduces a new type `Defs`, which describes all definition types that can be used across ISAs, application binary interfaces, micro processors, and micro architectures. It also allows models that produce this type to be defined in the outer-most level of the specification, which was not possible in the original VADL implementation.

Listing 4.1 shows a model that defines a `function` being used by both an ISA and an application binary interface.

```
model CastedOp(func: Id, op: Id, inType: Id, outType: Id): Defs = {
  function $func(left: $inType, right: $inType) -> $outType =
    VADL::$op(left, right) as $outType
}

using SInt16 = SInt<16>
using SInt32 = SInt<32>
using SInt64 = SInt<64>

instruction set architecture ISA = {
  $CastedOp(add1632 ; adds ; SInt16 ; SInt32)
}

application binary interface ABI for ISA = {
  $CastedOp(mul3264 ; muls ; SInt32 ; SInt64)
}
```

Listing 4.1: Functions as a macro

4.2 Flexible macro matches

In VADL, usage of the macro `match` syntax allows an author to conditionally insert syntax elements into the specification (see 2.2.4 Macro matches). In the original VADL implementation, only comparisons with syntax elements of type `Ex` and its subtypes were allowed.

OpenVADL enables authors to use macro `match` conditions of any syntax type. The main motivation behind this change was to allow `BinOp` and `UnOp` comparisons, where certain operators can lead to problematic behavior (e.g. divide-by-null behavior). Listing 4.2 shows a `model` that optionally wraps an operation into a zero check.

A second change of the macro `match` syntax concerns the *condition*, where it is now possible to provide a list of conditions separated by a comma. If any of the conditions match, the rule will be applied. Previously, match rules needed to be duplicated for each alternative condition. Listing 4.2 shows this feature applied to two operators (`/` and `%`).

```
model SafeOp(left: Id, op: BinOp, right: Id): Ex = {
  match : Ex
  ( $op = /, $op = %
      => if $right = 0 then -1 else $left $op $right
  ; _ => $left $op $right
  )
}
```

Listing 4.2: Divide-by-null safeguard

4.3 Record access without parentheses

In the original VADL implementation, certain situations required the access of `record` members to be surrounded by parentheses. This decision was made to disambiguate the record access syntax from `CallEx` occurrences. In OpenVADL, this syntax is no longer necessary, and not supported. Instead, the parser will consult the respective record definition to determine which segments of an apparent `CallEx` are part of a record access, and which are normal fields on a format. Listing 4.3 shows an example of a record access expression in the original VADL implementation (1), contrasted with the OpenVADL implementation (2).

```
record NameAndFmt ( name: Id, fmt: Id )

model OrigVADL(nameAndFmt: NameAndFmt): IsaDefs = {
  instruction $(nameAndFmt.name) : $(nameAndFmt.fmt) = {} // (1)
}

model OpenVADL(nameAndFmt: NameAndFmt): IsaDefs = {
  instruction $nameAndFmt.name : $nameAndFmt.fmt = {}      // (2)
}
```

Listing 4.3: Record access comparison

4.4 Models that produce models

In the macro expansion system of OpenVADL, model instances are expanded immediately at the site they are declared. This allows the usage of a feature not present in the original VADL implementation: Models that produce models.

```
model BinExFactory(binExName: Id, op: BinOp): IsaDefs = {
  model $binExName(left: Ex, right: Ex): Ex = {
    $left $op $right
  }
}
$BinExFactory(Addition ; +)
instruction ADD : RType = X(rd) := $Addition(X(rs1) ; X(rs2))
```

Listing 4.4: A model-producing model

Listing 4.4 shows the model `BinExFactory` which in turn produces a model. Because the `$BinExFactory` instance is evaluated immediately after it is parsed, the produced model `Addition` is known to the parser and can be used in the definition of the `ADD` instruction.

As can be seen in the example above, the inner `model` uses a parameter of the outer `model` called `op`. To facilitate this, the outer `models` arguments are *captured* upon its instantiation. When an inner model uses a placeholder, its captured arguments are consulted if the placeholder expression cannot be resolved using the model's own arguments.

4.5 Type variance in `model-type` parameters

When passing a reference to a `model` as an argument, the original VADL implementation considers the reference type valid only if the referenced `model` has the exact same type for each parameter and as the result type as the used `model-type` definition. OpenVADL relaxes this restriction, allowing the model parameters to be supertypes of the `model-type` parameters and the result type to be a subtype of the `model-type` result. Listing 4.5 shows a reference to `model Constants` being used as an `IsaDefsFactory`. The reference is of a valid type because the result type `Defs` is a subtype of `IsaDefs` and the type `Ex` of parameter `size` is a supertype of `Id` (see Figure 2.1).

```
1  instruction set architecture ISA = {
2    constant Word = 16
3
4    model-type IsaDefsFactory = (Id) -> IsaDefs
5
6    model Constants(size: Ex): Defs = {
7      constant a = $size
8      constant b = $size / 2
9    }
10
11   model BitDefs(factory: IsaDefsFactory, size: Id): IsaDefs = {
12     $factory($size)
13   }
14
15   $BitDefs(Constants ; Word)
16 }
```

Listing 4.5: Valid types in model references

4.6 Binary expressions without parentheses

In the original VADL implementation, chained binary expressions were only possible using parentheses. This also included type casts. As a result, expressions with several operators became cumbersome to use.

OpenVADL implements operator precedence as described in 2.3.2 Precedence in OpenVADL. Listing 4.6 shows a comparison for a shift-by-immediate computation.

Authors may still choose to use parentheses – in the example, the addition should likely remain parenthesized for clarity.

```
1 constant a = (1 as Bits<2>) << ((imm(0) as UInt<2>)+(1 as UInt<2>))
2
3 constant b = 1 as Bits<2> << imm(0) as UInt<2> + 1 as UInt<2>
```

Listing 4.6: Equivalent binary expressions with and without parentheses

4.7 Flexible Imports

In the original VADL implementation, imports had the pattern (in EBNF notation):

```
"import" filename { "::" symbol_path } ["with" string_literal]
```

For example, `import riscv::ISA with "ArchSize=Arch64"` would import a symbol `ISA` from `riscv.vadl` in the current directory and set the model `ArchSize` to `Arch64` during parsing. The imported files *had* to be located in the same directory as the importing specification. Multiple symbols needed multiple `import` declarations.

In OpenVADL, two backwards-compatible extensions were made to this syntax:

- The file name can now be a string literal pointing to any valid path
- Multiple symbols can be imported using a `{}`-wrapped list

To showcase these new features, let's consider the declaration
`import "../isa.vadl"::{RVI, RVIM}`
In this example, two different symbols are imported from a file in the specification's parent directory. Specifying the `.vadl` extension is optional, but complements editor support for file paths.

Another change in OpenVADL concerns the parsing model of imports: In the original VADL implementation, an `import` declaration was evaluated in a dedicated pass, after the parsing and macro expansion had already taken place. This meant that `model` definitions could not be imported across specifications. Instead, OpenVADL evaluates `import` declarations immediately after parsing, making any macro symbols available in the current specification.

An additional side effect of these changes is that imported specifications are parsed and evaluated once per `import` declaration. The original VADL implementation would only evaluate each specification once, and reuse the parsed syntax tree for other import occurrences. In practice, OpenVADL proved fast enough that this did not lead to a noticeable performance degradation.

4.8 Constant expressions in enumerations and groups

In the original VADL implementation, `enumeration` values and literals in `group` definitions only allow the usage of integer literals. In OpenVADL, this restriction is relaxed, and constant expression are usable. Listing 4.7 shows a comparison between enumerations in the original VADL implementation as well as an OpenVADL equivalent.

```
instruction set architecture OrigVADL = {
  enumeration Flags =
  { READ  = 0b001
  , WRITE = 0b010
  , EXEC  = 0b100
  }
}

instruction set architecture OpenVADL = {
  function nthBit(n: Int) -> Bits<3> = 0b001 << n

  enumeration Flags =
  { READ  = nthBit(0)
  , WRITE = nthBit(1)
  , EXEC  = nthBit(2)
  }
}
```

Listing 4.7: VADL enumerations and constant expressions

CHAPTER 5

Performance Evaluation

To evaluate the performance of the OpenVADL parser, a diverse range of VADL specifications were used. Table 5.1 summarizes the characteristics of the specifications. The `.expanded` variants of `miniARMv7` and `TIC64x` are semantically equivalent to their base variants, except every usage of macros has been substituted with the equivalent expanded VADL code.

File	# Lines	# Instructions	# Model Instances
aarch64	2328	799	1362
ARMNeon	2968	140	609
hexagon	1454	204	154
mipsIV	1131	106	163
miniARMv7	1272	8865	5701
miniARMv7.expanded	189555	8865	0
TIC64x	1001	8636	7083
TIC64x.expanded	187022	8636	0

Table 5.1: A summary of the specifications used for performance evaluation

5.1 Single file parsing

The objective of this scenario is to measure the time it takes the parser to parse a single file in a CLI usage scenario. Six of the specifications listed in Table 5.1 are used to compare the performance of the original VADL parser and the OpenVADL parser: `aarch64`, `miniARMv7`, `ARMNeon`, `mipsIV`, `TIC64x` and `hexagon`.

5.1.1 Methodology

To compare results of the OpenVADL parser with the VADL implementation in a single file parsing scenario, four classes of tests were run against the VADL specification files:

1. The old *Xtext*-based implementation using Java 21.0.4 OpenJDK
2. The new Coco/R based implementation using Java 21.0.4 OpenJDK
3. The new Coco/R based implementation using Java 21.0.4 GraalVM
4. The new Coco/R based implementation compiled ahead-of-time with GraalVM 21.0.4 Native Image

Every test file was parsed using the respective command line interface 50 times, and the average execution time (wall clock) was measured.

For the non-native image runs (1, 2 & 3), the same Java Virtual Machine flags were used:

```
-XX:TieredStopAtLevel=1 -Xmx8G
-XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC
```

For the Native Image run (3), the image was compiled using ML-inferred profile guided optimization and the following flags:

```
-O4 --gc=epsilon -march=native
```

In addition, the native image also was assigned a maximum heap size via `-Xmx8G`. Profile-guided optimization was not utilized, as its performance impact would depend on whether its training workload is representative of real-world specifications – using the test files themselves as training data would bias results beyond real-world performance gains. All tests were run on a PC using an Intel i7-7700K with 16 GB of DDR4 2667MHz memory.

As several features of the old VADL parser are not yet implemented, the results of the old VADL parser have been normalized by deducting the duration of the non-implemented passes from the measured run time:

```
AstAnnotationPass, AstPipelineSplittingPass, AstRAIIPass,
AstRecursiveCallDetectionPass, DefaultGrammarInjectionPass,
AstOperationAnnotationPass, AstTypeInferencePass,
AstLoweringPass, AstValidationPass
```

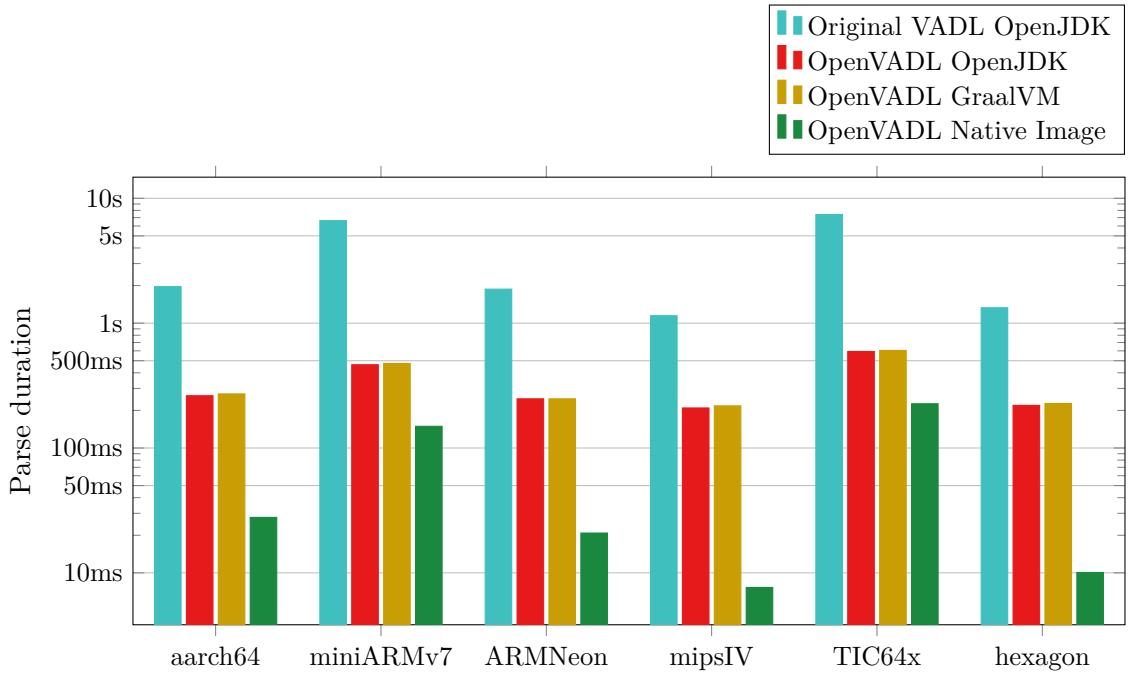

Figure 5.1: Parse durations of various VADL files (logarithmic scale)

5.1.2 Results

As Figure 5.1 shows, by using a faster parser generator and combining it with the ahead-of-time compilation capabilities of GraalVM, a speedup factor of 32 to 150 was achieved for these test files, with simpler files showing larger improvements. Looking towards the future, these initial response times should be more than satisfactory for a responsive language server.

5.2 Performance Impact of Macros

The objective of this scenario is to measure whether, and by how much, using macros has an impact on parse performance compared to manually typing out specifications.

This test uses the `miniARMv7` and `TIC64x` specifications, as well as their respective `.expanded` variants (as described in Table 5.1).

5.2.1 Methodology

This test consists of the same four classes as 5.1 Single file parsing, with the same build time and run time flags.

All four specification files were parsed 50 times, and the average execution time (wall clock) was recorded. Results for the original VADL implementation were normalized as

described in Section 5.1.

5.2.2 Results

Figure 5.2 shows that for `miniARMv7`, both the original VADL implementation and the Native Image build of OpenVADL parse the macro-based specification considerably faster than its expanded version. OpenVADL on OpenJDK and GraalVM only shows a minor difference in runtime.

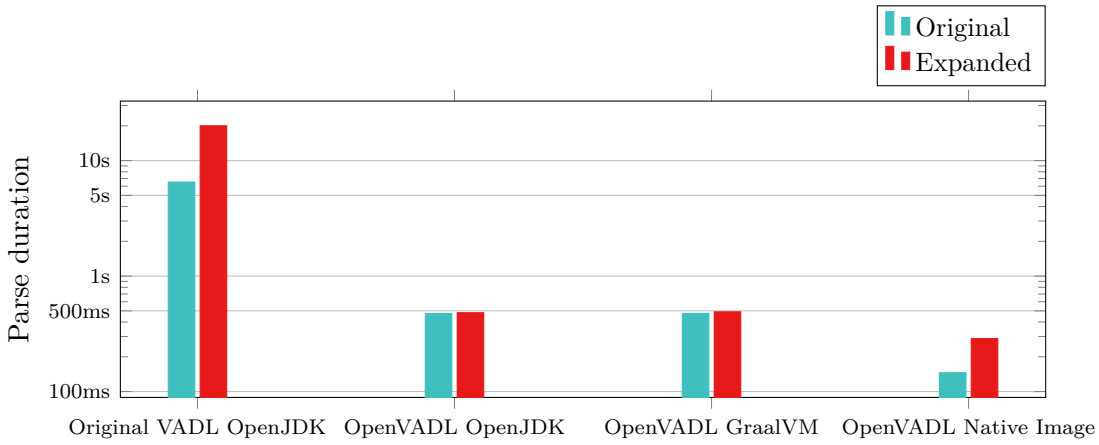

Figure 5.2: Parse durations of original and expanded miniARMv7 (logarithmic scale)

On the other hand, Figure 5.3 shows that OpenVADL on OpenJDK and on GraalVM parse the expanded `TIC64x` specification *faster* than the original, macro-based version. Only the original VADL implementation and OpenVADL built with Native Image are faster parsing the macro-based specification than the expanded variant.

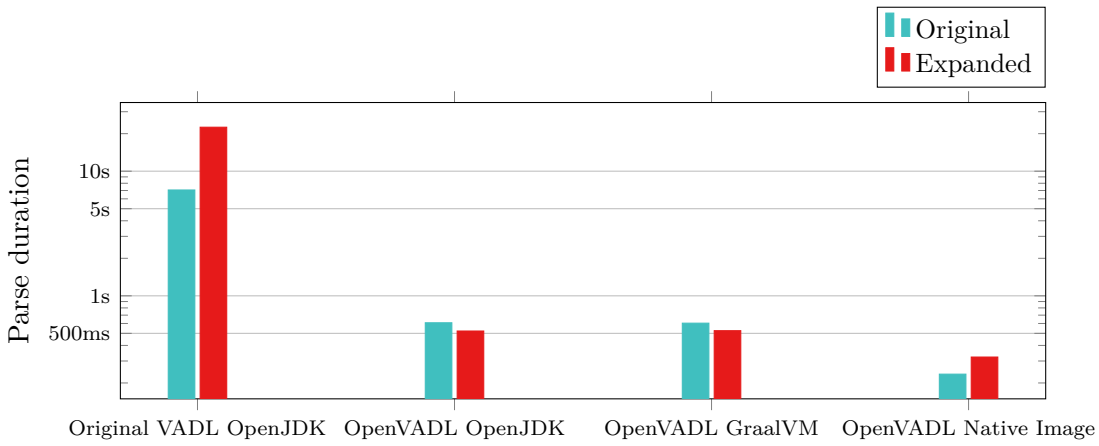

Figure 5.3: Parse durations of original and expanded TIC64x (logarithmic scale)

5.3 Performance as a long-running process

The objective of this scenario is to measure how a theoretical language server built on top of OpenVADL may perform. For this purpose, a varying number of files is parsed and the average time per parse is recorded. Files `miniARMv7` and `miniARMv7.extended` of Table 5.1 are used for this evaluation.

5.3.1 Methodology

Three classes of tests were used to parse the VADL specification files:

1. The new Coco/R based implementation using Java 21.0.4 OpenJDK
2. The new Coco/R based implementation using Java 21.0.4 GraalVM
3. The new Coco/R based implementation compiled ahead-of-time with GraalVM 21.0.4 Native Image

Every test file was parsed using OpenVADL's development-only `-n` command line flag, which repeats the file parsing `n` amount of times. For this evaluation, files are parsed between 1 and 100 times and their average parse time is compared.

For the non-native image runs (1 & 2), the same Java Virtual Machine flags were used:

```
-Xmx8G -XX:+UseG1GC
```

For the Native Image run (3), the image was compiled using the following flags:

```
-O4 --gc=G1 -march=native
```

In addition, the native image also was assigned a maximum heap size via `-Xmx8G`.

5.3.2 Results

As can be seen in Figure 5.4, the advantage of ahead-of-time compilation fades as the virtual machine warms up. As the native image is approximately 25% faster per-parse with larger amounts, it stands to reason that 25% of the native image's run time is its startup logic (scheduling by operating system, binary loading, allocations, initialization).

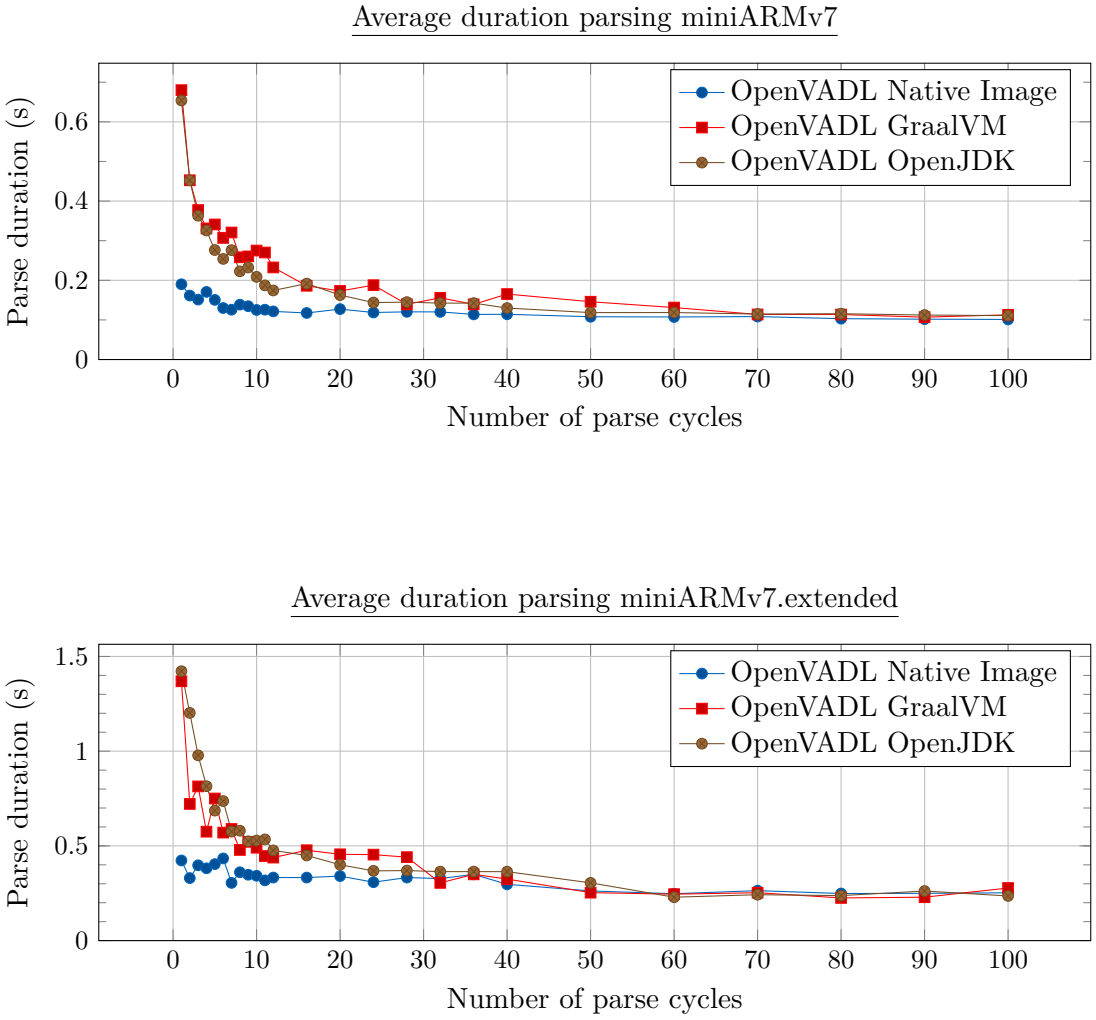

Figure 5.4: Warm-up behavior of the OpenVADL parser

CHAPTER 6

Future Work

The original VADL implementation produced a user manual containing descriptions, specifications, and examples of the VADL language. Parts of this user manual were generated from the *Xtext* specification. To replace this user manual, an OpenVADL documentation needs to be created. As part of this effort, an adapter that converts Coco/R specifications into a documentation format will be necessary.

In the macro system, additional syntax types `AbiDefs`, `MiaDefs`, `MipDefs` analogous to the existing `IsaDefs` may improve the authoring experience. This would improve the reuse of common elements outside the ISA definition.

To further improve performance, changes to the parser and scanner generated by Coco/R may be considered. Currently, OpenVADL uses the default Coco/R file templates. Initial investigation showed that the parser/scanner can be improved by rewriting some parts to allocate fewer objects – for example, every parsed token will always contain the token text as a `String` object. In practice, only identifiers and literals ever access this value, leading to thousands of needlessly allocated objects.

Along the same lines, both the internals of the generated parser and the AST nodes allocate a large amount of objects. Java's Project Valhalla [Pro] aims to provide constructs to the Java language that would discard the identity of `value` objects, allowing flattening in function calls and on the stack. The `Token` and `SourceLocation` classes are prime examples of often-created objects that do not need identity and would benefit from flattening. Once these concepts land in the Java language, an analysis of their performance impact on parsers may reveal a large potential.

To replace the IDE tooling generator provided by the *Xtext* framework, it is planned to implement the language server protocol (LSP) [LSP] to enable IDE integration.

Other parts of the OpenVADL implementation, such as the type system, are currently in development.

CHAPTER 7

Conclusion

OpenVADL significantly improves upon the original VADL implementation regarding performance. Choosing a different parser generator and planning the compiler structure with performance in mind, OpenVADL was able to achieve great performance gains without compromising on readability and maintainability.

In addition to being more efficient, OpenVADL also introduced several useful features to the language. These features will enable our team to more efficiently write VADL specifications for complex instruction sets and microprocessors.

Despite restricting use of dynamic Java features, GraalVM Native Image proved a good fit for a static command-line interface program. Avoiding the use of reflection also helps new developers to rapidly understand the structure and flow of the implementation. Overall, we are delighted with the advantages of GraalVM Native Image and do not see the disadvantages as relevant to OpenVADL.

Although the first step of a new, open implementation of VADL is now completed, significant effort is still necessary to implement the full feature set of VADL.

List of Figures

List of Tables

List of Algorithms

List of Listings

Bibliography

[Aas95] Annika Aasa. Precedences in specifications and implementations of programming languages. *Theoretical Computer Science*, 142(1):3–26, 1995. Selected Papers of the Symposium on Programming Language Implementation and Logic Programming. doi:10.1016/0304-3975(95)90680-J.

[Bet16] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. Packt Publishing, 2nd edition, 2016. URL: https://dl.acm.org/doi/10.5555/3074444.

[Graa] GraalVM – Introduction. https://www.graalvm.org/jdk23/introduction/. Online; accessed 24 Sept 2024.

[Grab] GraalVM – Native Image Basics. https://www.graalvm.org/jdk23/reference-manual/native-image/basics/. Online; accessed 24 Sept 2024.

[Grac] GraalVM – Reachability Metadata. https://www.graalvm.org/jdk21/reference-manual/native-image/metadata. Online; accessed 24 Sept 2024.

[Grad] Community-driven collection of GraalVM reachability metadata for open-source libraries. https://github.com/oracle/graalvm-reachability-metadata. Online; accessed 24 Sept 2024.

[Grae] GraalVM – Profile Guided Optimization. https://www.graalvm.org/jdk23/reference-manual/native-image/optimizations-and-performance/PGO/. Online; accessed 24 Sept 2024.

[HHH+24] Simon Himmelbauer, Christoph Hochrainer, Benedikt Huber, Niklas Mischkulnig, Philipp Paulweber, Tobias Schwarzinger, and Andreas Krall. The Vienna Architecture Description Language, 2024. URL: https://arxiv.org/abs/2402.09087, arXiv:2402.09087.

[HK23] Christoph Hochrainer and Andreas Krall. A pred-LL(*) Parsable Typed Higher-Order Macro System for Architecture Description Languages. In

Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2023, page 29–41, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3624007.3624052.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2:127–145, 1968. doi:10.1007/BF01692511.

[LdR81] Wilf R. LaLonde and Jim des Rivieres. Handling operator precedence in arithmetic expressions with tree transformations. *ACM Trans. Program. Lang. Syst.*, 3(1):83–103, Jan 1981. doi:10.1145/357121.357127.

[LSP] Language Server Protocol Specification https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/. Online; accessed 20 Sept 2024.

[MD08] Prabhat Mishra and Nikil Dutt. Chapter 1 - introduction to architecture description languages. In Prabhat Mishra and Nikil Dutt, editors, *Processor Description Languages*, volume 1 of *Systems on Silicon*, pages 1–12. Morgan Kaufmann, Burlington, 2008. URL: https://www.sciencedirect.com/science/article/pii/B9780123742872500045, doi:10.1016/B978-012374287-2.50004-5.

[Mö91] Hanspeter Mössenböck. A generator for production quality compilers. In *Proceedings of the Third International Workshop on Compiler Compilers*, CC '90, page 42–55, Berlin, Heidelberg, 1991. Springer-Verlag.

[Net] Netty project – an event-driven asynchronous network application framework. https://github.com/netty/netty/blob/bbd3a4a50ed23f2ae6c3d16b17b4423722380296/handler/src/main/resources/META-INF/native-image/io.netty/netty-handler/reflect-config.json. Online; accessed 24 Sept 2024.

[PF11] Terence Parr and Kathleen Fisher. LL(*): the foundation of the ANTLR parser generator. *SIGPLAN Not.*, 46(6):425–436, June 2011. doi:10.1145/1993316.1993548.

[Pic] Picocli Code Generation. https://github.com/remkop/picocli/tree/a856a14636fc22a24db6ab502b71c0439b384716/picocli-codegen. Online; accessed 24 Sept 2024.

[PQ95] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380250705, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380250705, doi:10.1002/spe.4380250705.

[Pro] Project Valhalla https://openjdk.org/projects/valhalla/. Online; accessed 17 Sept 2024.

[WLM03] Albrecht Woess, Markus Löberbauer, and Hanspeter Mössenböck. LL(1) Conflict Resolution in a Recursive Descent Compiler Generator. In *Modular Programming Languages*, pages 192–201, 01 2003. Updated version available at https://ssw.jku.at/Research/Projects/Coco/Doc/ConflictResolvers.pdf. doi:10.1007/978-3-540-45213-3_25.

[xtea] Xtend – Modernized Java. https://eclipse.dev/Xtext/xtend/. Online; accessed 24 Sept 2024.

[xteb] Xtext – Language Engineering Made Easy! https://eclipse.dev/Xtext/. Online; accessed 24 Sept 2024.

Acronyms

AOT Ahead Of Time. 13

API Application Programming Interface. 13

AST Abstract Syntax Tree. 3, 13, 15–19, 37

CST Concrete Syntax Tree. 3

IDE Integrated Development Environment. 3, 37

ISA Instruction Set Architecture. 1, 4, 18, 25

JIT Just In Time. 13

JNI Java Native Interface. 13

PDL Processor Description Language. 1

PGO Profile Guided Optimization. 14

VADL Vienna Architecture Description Language. 1, 3–6, 8–11, 18, 20, 25–35, 37, 39