Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Adding Algebraic Data Types to a Verification Language

Practical Work

Alessandro Maissen

Tuesday 26th April, 2022

Advisors: Prof. Dr. Peter Müller, Dr. Malte Schwerhoff

Department of Computer Science, ETH Zürich

Abstract

Viper is an intermediate language that powers several front-end verifiers for high-level programming languages. Nowadays, most programming languages support the use of algebraic data types (ADTs). Therefore, this work introduces native support of ADTs in Viper via a plugin. This does not only facilitate the use of ADTs in Viper but also provides a unified solution for front-end verifiers and reduce boilerplate encodings.

Acknowledgements

I want to thank my supervisor Dr. Malte Schwerhoff for his flexibility and constant support in the weekly meetings. Moreover, I would like to thank Prof. Dr. Peter Müller for allowing me to be part of Viper's research group and to work on this project.

Contents

Contents **iii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Goals and outline . . . 2

2 Background **3**
2.1 Algebraic data types . . . 3
2.2 Viper . . . 4
2.2.1 Architecture . . . 4
2.2.2 Language overview . . . 4
2.2.3 Plugin infrastructure . . . 8

3 ADT Plugin for Viper **15**
3.1 Basic syntax . . . 15
3.1.1 Declaration . . . 15
3.1.2 Instantiation . . . 16
3.1.3 Destructors . . . 16
3.1.4 Discriminators . . . 17
3.2 Deriving System . . . 18
3.2.1 Syntax . . . 19
3.2.2 Supported functions . . . 20
3.3 Encoding . . . 21
3.3.1 Basic encoding . . . 21
3.3.2 Contains function . . . 24
3.4 Implementation . . . 25
3.4.1 Codebase and usage . . . 25
3.4.2 AST extension . . . 26
3.4.3 Encoder . . . 27
3.4.4 Tests . . . 28

3.4.5 Performance 28
3.5 Future Work 28

4 Conclusion **31**

A Appendix **33**
A.1 Signatures of new AST nodes 33

Bibliography **35**

Chapter 1

Introduction

1.1 Motivation

Nowadays, program verification plays an increasingly important role, since security and safety critical software is everywhere. This includes technology for self-driving cars, blockchains and smart contracts, systems for aviation and astronautics, payment systems, but also internet protocols providing strong security guaranties.

Consequently, many automated verification tools were developed. However, most program verification tools follow an architecture in which they transform the program to be verified, including its specification, into a much simpler intermediate verification language. Viper [13] is such an intermediate language that currently powers the front-end verifiers Gobra [17], Prusti [1] and Nagini [8] for Go, Rust and Python, respectively. A good intermediate language provides necessary instruments in a way that most, ideally all, features from high-level languages can be encoded. Shared or similar features among high-level languages should not require a complex and boilerplate encoding, which leads us to algebraic data types (ADTs).

ADTs first came up in functional programming languages, but have now also gained popularity in object-oriented programming languages. Concretely, ADTs are supported in more than 30 programming languages, which include Go, Rust, Python, Haskell, Scala, Java, Swift and C++. Hence, existing but also future program verification tools (i.e. front-ends) using Viper as an intermediate language could benefit from a unified solution that is natively part of Viper. A unified solution for ADTs does not only avoid boilerplate encodings but also reduce the points of failure. The reason for this is that the correctness of the encoding does only need to be proved once, and not in all front-ends. However, extending core Viper directly to support ADTs would possibly require adaptions in existing front-ends. Because of that, this work adds ADTs to Viper via plugin, that can be enabled by front-ends if needed.

1.2 Goals and outline

In this work the main goal is to support ADTs in Viper as a built-in type via a Viper plugin by:

- Designing a suitable source syntax, and implementing the necessary parsing, typechecking, etc.
- Designing and implementing an internal representation of ADTs in Viper
- Designing and implementing a desugaring of ADTs into core Viper features and reusing the encoding developed to support ADTs in Gobra [5]
- Designing and implementing an interface to automatically deriving useful functions (e.g. `contains`, `toSet`, `toMultiSet`, `map`) from ADT definitions and implement some of them
- Provide a bunch of suitable unit tests to cover most introduced features for ADTs

The outline of this work is the following. In Chapter 2 we introduce the necessary background about algebraic data types, the Viper intermediate language and its plugin infrastructure used to support ADTs. On the other hand, Chapter 3 presents the ADT plugin in greater detail, while Chapter 4 concludes the work.

Chapter 2

Background

In this chapter we first introduce the concept of algebraic data types (ADTs). Afterwards, we give a brief overview of the Viper framework in general, and discuss relevant features for this work in greater detail.

2.1 Algebraic data types

Algebraic data types were first introduced in an experimental applicative language called Hope [4]. Nowadays, it is a fundamental feature in most functional programming languages but also available in other languages like Scala. However, we introduce algebraic types in terms of Haskell [14] since it comes with very intuitive syntax. For instance, in Haskell, the following code

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

represents an algebraic data type for a tree containing elements of type `a`. It has two data constructors, namely `Leaf` and `Node`. The former takes no arguments and defines a leaf node. On the other side, the data constructor `Node` has three arguments: an instance of type `a` and two instances of type `Tree a` defining the left and right sub-tree, respectively. A concrete instance would look the following.

```
Node 1 (Node 2 Leaf Leaf) (Node 3 Leaf Leaf)
```

Algebraic data-types come with two important properties, which are the *injectivity of constructors* (Definition 2.1) and the *exclusivity of algebraic data types* (Definition 2.1).

Definition 2.1 (Injectivity of constructors) *For an n-ary data constructor C, matching arguments $p_1, p_2, \ldots, p_n$ and $q_1, q_2, \ldots, q_n$ we have that*

$$C(p_1, p_2, \ldots, p_n) = C(q_1, q_2, \ldots, q_n) \implies \forall i \in [1, n] \colon p_i = q_i$$

Definition 2.2 (Exclusivity of algebraic data types) *For an instance d of an algebraic data type D with m data constructors C_i it holds*

$$\exists! i \in [1, m] \colon d = C_i(p_{i_1}, p_{i_2}, \ldots p_{i_n}), \quad \textit{for some } p_{i_1}, p_{i_2}, \ldots p_{i_n}$$

where $\exists!$ is the unique existential quantifier.

Described properties form the basis to define equality of two ADT instances, a notion of destructors (inverse of construction), discriminators and pattern matching, as we will see later in Chapter 3.

2.2 Viper

Viper (Verification Infrastructure for Permission-based Reasoning) [13] is an intermediate language and a set of tools developed at ETH Zurich for program verification. It is implemented in Scala, a modern object-oriented and functional programming language. Compared to similar verification infrastructures such as Boogie [2] and Why3 [3], Viper has strong support for permission logics such as separation logic and implicit dynamic frames, which are well-suited for verifying heap-manipulating programs and thread interactions in concurrent software.

2.2.1 Architecture

As depicted in Figure 2.1, Viper comes with an intermediate language and currently supports two different verification back-ends, one is called Silicon [15] and the other is called Carbon. Both back-ends use the SMT solver Z3 [7] for verification. Silicon applies symbolic execution and directly encodes to SMT code. On the other hand, Carbon involves verification conditions and encodes Viper code to Boogie, which under the hood as well uses Z3.

Viper's architecture easily allows to develop new verification tools (front-ends) for commonly used programming languages. This can be achieved naturally by encoding verification techniques for front-end programming languages into the Viper intermediate language. Some examples are Gobra [17], Prusti [1] and Nagini [8] for Go, Rust and Python, respectively.

2.2.2 Language overview

The Viper intermediate language (internally called Silver) is an imperative language, where its design has been influenced by existing verification

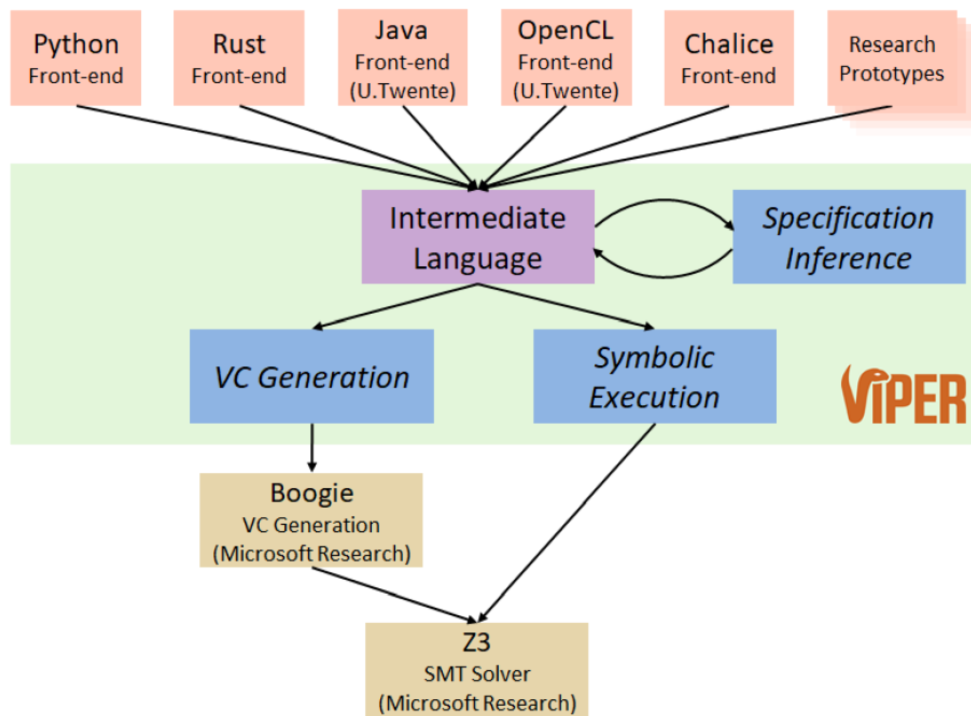

Figure 2.1: Viper verification infrastructure: Front-ends (top layer) use Viper's intermediate language to encode verification problems. Those are then verified with Carbon (VC Generation + Boogie) or Silicon (symbolic execution-based verifier). Ultimately, both backends use the SMT sovler Z3 (bottom layer). The graphic is sourced from the Viper Online Tutorial [9].

languages like Boogie, Dafny [11] and Chalice [12]. A Viper program is structured around different top-level declarations, which includes methods, functions, fields, predicates and domains.

Methods A method, more specifically its body, is a sequence of statements (e.g conditionals, loops, etc.). The signature of a method consist of input arguments, a return value and the method's specification. The latter includes preconditions and postconditions, which can be specified with keywords `requires` and `ensures`, respectively. An example of a method declaration can be found in Listing 2.1 starting on line 7. Method declarations are verified independently of each other, by checking that the method's body fulfills its postcondition under assumptions made by the precondition. On the other hand, the method implementation is unknown for a caller of the method. Hence, the effect of a method is solely observed though its specification.

Functions In contrast to a method, the function's body is an expression, not a sequence of statements. (see Listing 2.1 line 4). In Viper all expressions including functions are pure. Consequently, there cannot be any modifications

to the program state, and therefore function calls can be part of specifications. When a function is called, the result is simply equated with the expression in the function's body. As for methods, one can specify preconditions and postconditions for functions. While preconditions are checked on a function call, postconditions are assumed without checking against the function's body. This has the consequence, that postconditions for functions can easily introduce unsoundness.

```
1  function fac(n: Int): Int
2      requires n >= 0
3  {
4      n == 0 ? 1 : n * fac(n-1)
5  }
6
7  method factorial(n: Int) returns (res:Int)
8      requires n >= 0
9      ensures res == fac(n)
10 {
11     var i: Int
12     i := 0
13     res := 1
14     while (i < n)
15         invariant i >= 0 && i <= n
16         invariant res == fac(i)
17     {
18         i := i + 1
19         res := res * i
20     }
21
22 }
```

Listing 2.1: Example of Viper methods and functions. Both implement the factorial. Observe that in the specification of the method we prove equality of the two implementations.

Fields and predicates Viper supports objects (type `Ref`) on the heap. Fields can be declared as in Listing 2.2 on line 1. Every object has all the declared fields, hence there is no notion of classes in Viper. To access a field or perform a field assignment one needs permissions, which can, for instance, be acquired with `acc(...)`, as in Listing 2.2 on line 5. In Viper there exists several levels of permissions, also called fractional permissions with permission amounts between 0 and 1. In particular, any non-zero permission allows read access, only a permission amount of 1 (`write`) allows write access, whereas 0 (`none`) prohibits any access. Predicates can be used to gather permissions, which allows to represent object-like structures as they are common in object-oriented programming languages. In combination with fractional permission this can for example be used to model read-only objects, as the predicate

`readOnlyAccount(...)` does in Listing 2.2 starting on line 3. Furthermore, predicates can also be recursive so that one can specify permissions over heap structures like lists and trees. However, permissions are not relevant for this work, hence we do not explain this topic any further, instead we refer to the Viper Online Tutorial [9] for a more information.

```
field balance: Int

predicate readOnlyAccount(this:Ref)
{
    acc(this.balance, 1/2)
}

function getBalance(a: Ref): Int
    requires readOnlyAccount(a)
{
    unfolding readOnlyAccount(a) in a.balance
}
```

Listing 2.2: Example of Viper fields and predicates. To access a field in a function one has to acquire permission, which can be done in the precondition. In this example we can acquire the required permission via the predefined predicate `readOnlyAccount(...)`, however it must explicitly be unfolded.

Domains Viper domains allow to define additional types, mathematical functions, and axioms over these functions specifying certain properties. This is one of the most important features for this work, since it can be used to encode ADTs in Viper. The declaration of a domain consists of a name with type parameters and a body in which one can define domain functions and axioms. In contrast to ordinary functions, domain functions are always abstract, i.e they have no function body. Hence, properties of domain functions can only be defined via domain axioms. Listing 2.3 shows a way one could encode a an equivalent to Haskell's `Maybe` data type using the concept of domains.

Domain axioms are often defined using `forall` quantifiers, which need so-called trigger expressions (Listing 2.3 line 8 in curly braces). Triggers are used in the context of e-matching [6], a technique applied in SMT solvers to approach verification of programs with specification containing first-order logic. To put it simple Viper can only use an axiom if it is instantiated with concrete values. Looking at the axiom in Listing 2.3, there are infinitely many ways to instantiate the equality `value == get(Just(value))`. Accordingly, in e-matching triggers are matched against expressions in the program (e.g. using patter matching), and only on a success axioms are instantiated. For instance, if the expression `Just(5)` is present in the program, the axiom

`5 == get(Just(5))` is learned by the SMT solver. However, it is crucial to choose good triggers to guide the SMT solver towards a quick solution. Bad triggers might lead to an infinite loop of instantiations, which is called a matching loop. If this happens, the SMT solver fails the verification with a timeout. Moreover, choosing to restrictive triggers might fail verification as well, since necessary axioms for the proof are not learned by the SMT solver.

```
1  domain Maybe[T] {
2      function Nothing(): Maybe[T]
3      function Just(value: T): Maybe[T]
4
5      function get(m:Maybe[T]): T
6
7      axiom {
8          forall value: T :: {Just(value)}
9              value == get(Just(value))
10     }
11 }
```

Listing 2.3: Example of a Viper domain with domain functions and domain axioms that implement an equivalent to Haskell's `Maybe` data type.

Built-in Types Beside the possibility to introduce user-defined types, Viper offers a bunch of standard types, some of which we have already seen in above code examples. They include

- `Bool` for Boolean values
- `Int` for mathematical (unbounded) integers
- `Ref` for references to objects
- `Perm` for permission amounts
- `Seq[T]`, `Set[T]` and `Multiset[T]` for immutable sequences, sets and multisets with element type `T`

2.2.3 Plugin infrastructure

There are many front-ends that build on top of Viper. Hence, constantly making changes to Viper is undesirable since front-ends need to be adapted too. As a consequence, Viper has a plugin infrastructure, which allows to add new features to Viper without changing the core Viper language.

In a nutshell Viper goes through four main phases, depicted in Figure 2.2. It first parses the input program into the parser abstract syntax tree (PAST). Afterwards the PAST is type-checked, resolved and translated to the internal representation in Viper, namely to the Viper abstract syntax tree (Viper AST).

Then, Viper uses either Silicon or Carbon to retrieve the verification result, and finally reports it. In the next paragraphs we discuss how the plugin infrastructure can be used to hook into the different phases, and further how one can extend the PAST and the Viper AST with new nodes to represent a new Viper feature.

General callbacks As previously said, the Viper plugin infrastructure allows to intercept the described Viper phases by overriding different callbacks. Figure 2.2 lists all the callbacks, whereas important callbacks for this work are:

- `beforeParse(...)` is called after the input file, to be verified, is read but just before parsing has started. This callback can for instance enables to add new custom parsers to extend the syntax of the Viper intermediate language. We will see more details about extending the parser later in a subsequent paragraph.
- `beforeResolve(...)` is called after the PAST is created from the input file but before the PAST is type-checked and resolved. Often it is used to apply transformations on PAST nodes. For a concrete use case consider the following example. First, observe that a function or method application in Viper is syntactically identical. Hence, both are parsed into the same PAST node but later distinguished in the process of resolving. If an other feature with same syntax is introduced, hence also sharing the same PAST node, the plugin infrastructure only allows to alter resolving and type-checking if the new feature is represented with a new node in the PAST. Therefore, this callback can be used explicitly transform a ”wrongly” parsed node. In this work we concretely apply this for ADT constructors.
- `beforeVerify(...)` is called before the AST is passed to the verification back-end (Sillicon or Carbon). Suitable for transforming new AST nodes, introduced by the plugin to represent new features, to ordinary Viper AST nodes, since back-ends only support the latter nodes.

Additional hooks, which are not used in this work but also important, are the following.

- `beforeTranslate(...)` is called before the PAST is translated to the Viper AST.
- `mapVerificationResult(...)` is one of the final hooks of the Viper verification pipeline before the result is returned. It takes the verification result as argument. This is the place one can apply changes to the verification result, and in particular used for error back translation.

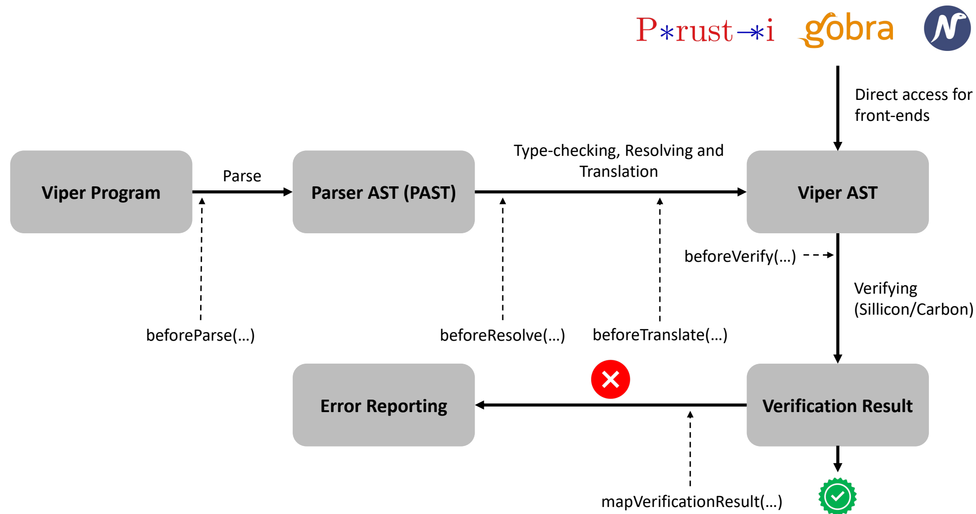

Figure 2.2: Viper plugin infrastructure.

Extending the parser As previously mentioned, the `beforeParse(...)` callback can be used to make changes to the ordinary Viper parser, i.e. to extend the syntax of the Viper intermediate language. Concretely, this can be achieved by creating custom parsers for newly introduced syntax using the library (used by Viper's parsers). They can then be added to the existing parser in Viper by using different hooks provided by the plugin infrastructure. In this works we used the following hooks.

- `addNewDeclAtEnd(...)`† extends the top-level declarations. This can be used to introduce syntax for new functions, new methods or in the case of this work an ADT declaration.
- `addNewKeywords(...)` extends the set of program keyword. In particular, we used this hook to introduce the keywords `adt`, `derives` and `without`, which we will present in Sections 3.1 and 3.2.

For completeness and documentation we discuss the remaining hooks.

- `addNewExpAtEnd(...)`† extends basic expressions.
- `addNewStmtAtEnd(...)`† extends the grammar for statements. An possible application could include a statement for a try-catch block or a statement for pattern matching.
- `addNewPreCondition(...)` allows to add custom precondition expressions.
- `addNewPostCondition(...)` allows to add custom postcondition expressions. This is for instance used by the the termination plugin [16]

to add new syntax for the decrease clause, which is used to check if a function or method terminates.

- `addNewInvariantCondition(...)` allows to add custom specification for loops. Like the previous hook, it is currently used by the termination plugin to a decrease clause.

For presented hooks above with suffix `End` (marked with †), there is a matching hook with the same name, but with suffix `Start`. Namely,

- `addNewDeclAtStart(...)`
- `addNewExpAtStart(...)`
- `addNewStmtAtStart(...)`

To explain the difference, we first need to introduce the either-or operator `|` for parsers. In particular, $\texttt{p}_1$ `|` $\texttt{p}_2$ first tries to apply parser $\texttt{p}_1$, and if that fails it tries parser $\texttt{p}_2$. Then, in the context of the hooks provided by the plugin infrastructure, adding a custom parser $\texttt{p}_c$ to an existing parser `p` at start results in a new parser $\texttt{p}_c$ `|` `p`. On the other hand, adding the custom parser at end leads to a new parser `p` `|` $\texttt{p}_c$. Hence, adding parsers at start overwrite the existing parser, while adding them at end will not affect parsing of ordinary Viper syntax. Consequently, a good practice is to run the Viper test suite with the plugin activated, if new parsers are added at start, to make sure that parsing of ordinary Viper syntax still works correctly.

Extending the PAST When introducing new syntax, one most likely also wants to extend the PAST by creating new PAST nodes. This can be achieved by extending the trait, `PExtender` where one has to implement the following methods

- `getSubnodes()` returns the subnodes of the receiver node. This method is mostly used by Viper's internal framework for traversing and rewriting the PAST, e.g. to collect all top-level declarations during resolving or deleting nodes from the PAST.
- `typecheck(...)` allows to implement custom type-checking and resolving. Note that this method is overloaded and accepts an additional argument to specify an expected type, which in especially should be implemented for new expression nodes (e.g for ADT constructor applications) that can be assigned to a typed local variable.

The following methods are related to the translation of the PAST node to a corresponding AST node. Observe that only a few of those following methods need to be implemented, which depends on the kind of the node (top-level declaration, statement, expression or type).

- `translateMemberSignature(...)` is only necessary to implement for new top-level declarations, that can be accessed globally. For instance in the context of this work, this applies to an ADT constructor declaration that can be called in a method and function body. An application or access to a declared construct (e.g. an ADT constructor application) needs additional information for the process of translation that is held by its declaration. Hence, signatures are translated first, so that circular dependencies do not break the translation process.
- `translateMember(...)` is only necessary to implement for new top-level declarations, that can be accessed globally. Often the signature is already translated by `translateMemberSignature(...)`, so it remains to translate the member's body.
- `translateExp(...)` is only necessary to implement for new expressions, like this work does for ADT constructors.
- `translateType(...)` is only necessary to implement for new types, like we do for the newly introduced built-in type representing ADT's.
- `translateStmt(...)` is only necessary to implement for new statements.

Finally, there is a method `transformExtension(...)`, which is deprecated since node transformations now usually happen through Viper's internal rewriting framework.

Extending the AST Similar as when extending the PAST, there are different traits one can extend to create new AST nodes. This includes

- `ExtensionMember` for top-level declarations
- `ExtensionExp` for expressions
- `ExtensionStmt` for statements
- `ExtensionType` for types

All traits share the method `prettyPrint`, and if implemented, enables to display and print the AST with extensions. Moreover, most traits require to implement the method `extensionSubnodes`. As it was already the case for the PAST, this method is required by Viper's node rewriting strategy framework to traverse the AST. Beside of that, only the trait `ExtensionExp` demands to implement additional methods related to the purity and type of the expression.

Currently, it is not supported to pass an AST with extensions nodes to verification back-ends. Therefore, it is far more important to include the necessary fields and information needed later to encode extensions nodes as

ordinary Viper AST nodes. We will see concrete examples of such fields in Section 3.4.2, where we present the internal representation of ADTs in the AST.

Chapter 3

ADT Plugin for Viper

The chapter first introduces the basic syntax for ADTs in Viper, while afterwards we present a deriving system to automatically derive useful functions for ADTs. Subsequently, the encoding of ADTs as domain functions is presented more generally before we discuss some implementation details.

3.1 Basic syntax

In this section we present the basic syntax for ADTs that is added to Viper with the developed plugin. Furthermore, we discuss alternative designs and their trade-offs.

3.1.1 Declaration

A special keyword `adt` indicates the start of an ADT declaration. The signature of an ADT declaration consists of a unique identifier optionally followed by type parameters in square brackets. Formally, the signature of an ADT declaration is identical to that of a domain, except for the difference in the corresponding keyword. The body of an ADT (delimited by braces) consists of arbitrarily many ADT constructors. As seen in Listing 3.1, constructor declarations have high resemblance to domain function declarations (see Section 2.2.2), except that they do not require a return value. Moreover, constructor declarations do not need to be preceded by a special keyword.

Finally, we require that argument identifiers have to be unique among all constructors within an ADT declaration. This is because argument identifiers implicitly define the identifiers for their corresponding destructors, which are presented in Section 3.1.3. As a concrete example, there cannot be two constructors in an ADT declaration both taking an argument named `value`, otherwise the corresponding destructor is ambiguous.

```
1 adt List[T] {
2     Nil()
3     Cons(value: T, tail: List[T])
4 }
```

Listing 3.1: Example of an ADT declaration. In particular, the ADT defines a list containing elements of type T.

While deciding for the final syntax we considered some alternative ideas. One idea included the keyword `constructor` for an ADT constructor declaration, shown in Listing 3.2. But since only constructors can be part of an ADT's body, introducing such a keyword would only lead to notational overhead. Because of that we choose the former approach shown in Listing 3.1.

```
1 adt List[T] {
2     constructor Nil()
3     constructor Cons(value: T, tail: List[T])
4 }
```

Listing 3.2: Alternative design for ADT declarations (not chosen).

3.1.2 Instantiation

An ADT can be instantiated by calling one of the defined constructors with suitable arguments. Basically, this is the same as calling a regular function or a domain function. Listing 3.3 shows how one can instantiate an ADT of type `List[Int]`, by nesting the constructor `Cons()` several times to form a list, which is then assigned to a local variable.

```
1 var list: List[Int]
2 list := Cons(1, Cons(2, Cons(3, Nil())))
```

Listing 3.3: ADT instantiation of a list with three elements.

3.1.3 Destructors

Until now we have seen how to declare ADTs with their constructors, and how one can instantiate them. However, we do not only want the possibility to construct ADTs, but also destruct them. For instance, we would like to retrieve the value of the first element of our `List` ADT. Accordingly, for each constructors of arity n there are n destructors. Our for destructors is inspired by Dafny [11] and matches the syntax for field accesses. In particular, for each constructor argument P, there is a destructor P that can be applied on an ADT instance, as illustrated in Listing 3.4. Recall from Section 3.1.1 that

identifiers of constructor arguments are unique within one ADT declaration, such that ambiguities across different ADTs can be resolved by inspecting the type of the receiver.

```
1 assert list.value == 1
2 assert list.tail == Cons(2, Cons(3, Nil()))
3 assert list.tail.value == 2
```

Listing 3.4: Syntax for ADT destructors, building on Listing 3.3.

Another approach would be to introduce a function-like destructor `getP(a)` for each argument `P` of a constructor, where `a` is an ADT instance. Destructors are than called in a static manner to avoid name collisions. An example of this syntax is shown in Listing 3.5. The reader may realize that accessing the second element of a list, as in Listing 3.5 on line 3, already results in a quite big notational overhead, which linearly growths with the level of nesting. Furthermore, Viper does currently not have a notion of classes and hence invoking the destructors in a static fashion seems rather odd. Because of these reasons it is not hard to argue for the first approach.

```
1 assert List.getValue(list) == 1
2 assert List.getTail(list) == Cons(2, Cons(3, Nil()))
3 assert List.getValue(List.getTail(list))== 2
```

Listing 3.5: Alternative syntax for ADT destructors, building on Listing 3.3 (not chosen).

3.1.4 Discriminators

In Section 2.1 we learnt about the exclusivity of ADTs. This property allows to introduce the notion of discriminators. Concretely, a discriminator for a constructor `C(...)` returns `true` if and only if the instance of the ADT was constructed using `C(...)`. As previously for the destructors, our first approach matches Dafny's syntax. Namely, for each constructor `C(...)`, there is a discriminator `C?` that can be applied via dereferencing an ADT instance. Listing 3.6 shows the described syntax in an example.

```
1 assert list.Cons? == true
2 assert list.tail.Cons? == true
3 assert list.Nil? == false
```

Listing 3.6: Desired syntax for ADT discriminators (not chosen).

Viper supports ternary operators of the form `b ?` e_1 `:` e_2, where `b` is some condition and e_1, e_2 some expressions. Unfortunately, presented syntax for

discriminators conflict with the syntax of ternary operators. The Viper plugin infrastructure did not allow to resolve this conflict without big changes to existing parses, i.e. to the existing grammar of the language. For this reason, a slight adaptation (see Listing 3.7) of the syntax is necessary. Precisely, for each constructor `C(...)` there is a destructor `isC`. The reader may observe that in contrast to Dafny's approach this does increase the number of identifiers in a given program, and thus increase the changes for name clashes.

```
1 assert list.isCons == true
2 assert list.tail.isCons == true
3 assert list.isNil == false
```

Listing 3.7: Example illustrating the final syntax for ADT discriminators.

3.2 Deriving System

Haskell [14] provides a deriving system to automatically derive commonly used functions. Concretely, this can be achieved by adding a deriving clause (e.g `deriving (...)`) at the end of a data type declaration. Listing 3.8 shows an example where the class `Show` is derived for a data type `Tree a`. Consequently, the implementation of the function `show`, which can be used to display an instance of `Tree a`, is synthesized by the compiler.

```
1 data Tree a = Leaf |
2               Node a (Tree a) (Tree a) deriving (Show)
```

Listing 3.8: Example of a `Tree` that derives the class `Show`.

In Haskell 98 there is a list of derivable standard classes, which include `Eq`, `Ord`, `Enum`, `Ix`, `Bounded`, `Read`, and `Show`, each potentially containing more than one function. Depending on the Haskell compiler, this list might be longer. For instance, the Glasgow Haskell Compiler (GHC) [10] additionally supports derivation of the class `Functor`, which includes a function `fmap`.

Automatically deriving such functions has the advantage that programmers do not have to repeatedly write boilerplate code to support basic operations. Therefore, we aim to provide a similar feature for ADTs in Viper. The Gobra front-end already supports a deriving system for ADTs, which is inspired by Haskell's deriving system. To allow a simple encoding of Gobra's deriving system into Viper's extended intermediate language, our deriving system is influenced by both, Gobra's and Haskell's, solutions.

3.2.1 Syntax

In this subsection we present the syntax that enables to derive useful functions for ADTs. Currently, our deriving system only supports a function `contains`. Hence, we use additional function identifiers, i.e `toSet` and `depth`, to illustrate the syntax in the upcoming examples. These functions are introduced later in Section 3.2.2. However, to understand certain features of the syntax some necessary information about their semantics is explained on the fly.

In a first step, we add an optional `derives`-clause at the end of an ADT declaration, where derivable functions can be specified. Listing 3.9 shows an ADT implementing a `List[T]`, while additionally the functions `contains` and `depth` are automatically derived.

```
1 adt List[T] {
2     Nil()
3     Cons(value: T, tail: List[T])
4
5 } derives { contains depth }
```

Listing 3.9: Basic syntax for deriving useful functions for ADTs in Viper. In this example the functions `contains` and `depth` are automatically derived.

The presented syntax above allows to derive trivial functions. However, we require syntax with more expressiveness to support automatic derivation of functions like `toSet`, which abstracts an ADT into a set. To see why this is the case let us consider the definition of the ADT `SpecialList` in Listing 3.10. When deriving the function `toSet` for `SpecialList` it not clear if the resulting `Set` should contain all values of type `Int` or of type `Bool`. To address this problem, identifiers of derivable functions in the `derives`-clause are followed by an optional type parameter that can be used to resolve potential ambiguities.

```
1 adt SpecialList {
2     Nil()
3     Cons(value1: Int, value2: Bool, tail: SpecialList)
4 } derives { toSet[Int] }
```

Listing 3.10: Abstracting `SpecialList` into a `Set` leads to ambiguities, which can be resolved with type parameters. In this case the derivable function `toSet` collects all integers in the list.

Finally, as in Gobra's deriving system, we introduce the keyword `without`, which allows to add ADT constructor arguments to a blocklist[1] so that they

[1] A blocklist contains a list of items that should be excluded. The counterpart to a blocklist is a clearlist.

are excluded when deriving a function. A concrete example of the syntax is shown in Listing 3.11. The effect of applying blocklisting to a derivable function solely depends on its implementation and accordingly differs from function to function. In the case of the derivable function `contains`, which in the context of trees defines a sub-tree relation, excluding the argument `value` has the effect that `1` is not contained in `Node(1, Leaf(), Leaf())`.

```
1 adt Tree[T] {
2     Leaf()
3     Node(value: T, left: Tree[T], right: Tree[T])
4
5 } derives {
6     contains without value
7     toSet[T] without left, right
8 }
```

Listing 3.11: Extended syntax, which allows fine-grained blocklisting.

The reader may realize that in the presented syntax it is required to specify function identifiers in the `derives`-clause, while in Haskell derivable classes need to be specified. Gobra's approach is similar to Haskell and currently a set of functions, namely `len`, `set` and `mset` can be generated by deriving `Collection`. Due to the lack of a notion of classes in Viper we have decided against this approach. Moreover, requiring to specify functions allow a more fine-grained derivation, i.e a more fine-grained blocklisting. This is for instance in shown in Listing 3.11, where each derivable function has its own blocklist. On the other hand, in Gobra there is only one blocklist for all derived functions.

3.2.2 Supported functions

In the previous section we presented an expressive syntax for the deriving system that allows to support automatic derivation of trivial functions but also more complex functions that require parametrization. Nevertheless, currently the deriving system only supports the function `contains`, while the other functions (i.e. `depth` and `toSet`) were only used to illustrate the syntax of the deriving system. However, additional derivable functions could easily be added.

Contains The derivable function `contains(a: A, b: B): Bool` is a transitive binary relation, which contains the tuple (a, b) if a is contained in b. Note that "contained" in the context of ADTs, means that the instance b is created with a constructor, where a was one of the constructor's arguments. In particular for an ADT that defines a `Tree`, as in Listing 3.11, we have that

- `Leaf()` is contained in `Node(42, Leaf(), Leaf())`
- 42 is contained in `Node(1, Node(42, Leaf(), Leaf()), Leaf())`

Potentially derivable functions in future As already, mentioned one could easily add more derivable functions to the ADT plugin. Concretely, this can be done by firstly extend the parser to accept the identifier of the derivable function, which does not take more than one line of code. In a second step one has to provide the encoding of the derivable function. The scope of adding the encoding strongly depends on the function and its complexity.

Following list present some functions that could be added to the deriving system in future.

- `depth` takes an ADT as argument and returns its maximum level of nesting. For an ADT `List[T]` this corresponds to the length, while for a `Tree[T]` it is simply its depth.
- `toSet[T]` takes an ADT as argument and abstracts it into a set. For example, applying `toSet[T]` on an ADT defining a `List[T]`, returns the elements of type `T` of the list in a set.
- `toMultiSet[T]` is the same as `toSet[T]`, except that we allow a multi-set.
- `map[T]` takes an ADT and a function f as argument. Then, the ADT is recursively transformed, according to the function f. As an example one could increase each element of a `List[Int]` by one, with the function $f(x) = x + 1$. However, since it is currently not possible to pass functions as arguments to functions, this would also require to extend Viper's syntax.

3.3 Encoding

The encoding of ADTs as Viper domains is based on previous work that added ADTs in Gobra [5]. In this section we will first present the basic encoding for general ADTs more formally. To do so, we use the previously introduced syntax for ADTs in Section 3.1. Additionally, we allow identifiers with subscripts, which simplifies generalization. In a second step, we show the encoding for the function `contains` that is part of the deriving system.

3.3.1 Basic encoding

Using the extended syntax with subscripts, a generalized definition of an ADT `A` is shown in Listing 3.12. It has k type parameters $\mathtt{T}_i$ and n constructors $\mathtt{C}_i$ with m_i arguments each. Observe that $\mathtt{S}_{i,j}$ corresponds to the type of argument $\mathtt{p}_{i,j}$ and might be parameterized by $\mathtt{T}_1, \ldots, \mathtt{T}_k$.

```
1  adt A[T₁,T₂,...,Tₖ] {
2
3      C₁(p₁,₁: S₁,₁, p₁,₂: S₁,₂,...,p₁,ₘ₁: S₁,ₘ₁)
4
5      C₂(p₂,₁: S₂,₁, p₂,₂: S₂,₂,...,p₂,ₘ₂: S₂,ₘ₂)
6
7      ⋮            ⋮            ⋮            ⋮
8
9      Cₙ(pₙ,₁: Sₙ,₁, pₙ,₂: Sₙ,₂,...,pₙ,ₘₙ: Sₙ,ₘₙ)
10
11 }
```

Listing 3.12: Shows an ADT A with k type parameters and n constructors. Each constructor C_i takes m_i arguments $p_{i,j}$ of type $S_{i,j}$.

As already mentioned in previous sections, domains are used to encode ADTs in Viper. Accordingly, in a first step a matching domain A is generated for the ADT A with same number of type parameters. Then, each constructor C_i is simply encoded as a domain function C_i that returns a value of type $A[T_1,...,T_k]$. This encoding scheme is shown in Listing 3.13.

```
1  domain A[T₁,...,Tₖ] {
2
3      // Constructors
4      function C₁(p₁,₁: S₁,₁,...,p₁,ₘ₁: S₁,ₘ₁): A[T₁,...,Tₖ]
5
6      function C₂(p₂,₁: S₂,₁,...,p₂,ₘ₂: S₂,ₘ₂): A[T₁,...,Tₖ]
7
8        ⋮                 ⋮                 ⋮                 ⋮
9
10     function Cₙ(pₙ,₁: Sₙ,₁,...,pₙ,ₘₙ: Sₙ,ₘₙ): A[T₁,...,Tₖ]
11
12 }
```

Listing 3.13: Viper domain and encoded constructors for the general ADT definition in Listing 3.12.

Remember that each constructor C_i has exactly m_i destructors. Consequently, for each constructor argument $p_{i,j}$ of type $S_{i,j}$ a domain function $D_{p_{i,j}}$ with an axiom is generated to encode the matching destructor. This function-axiom pair is stated in Listing 3.14. Furthermore, it can easily be shown that the set of axioms for all destructors of one constructor generated according to Listing 3.14 imply injectivity of the constructor as introduced in Definition 2.1. A proof can be found in the work that introduced ADTs in Gobra [5], while the authors furthermore as well evaluate the chosen trigger, namely

$C_i(p_{i,1}, ..., p_{n,m_i})$. Recall our discussion in Section 2.2.2 about triggers, and how they affect the verification process.

```
1  function D_{p_{i,j}}(a: A[T_1,...,T_k]): S_{i,j}
2
3  axiom {
4      forall p_{i,1}: S_{i,1},...,p_{i,m_i}: S_{i,m_i} :: {C_i(p_{i,1},...,p_{n,m_i})}
5          p_{i,j} == D_{p_{i,j}}(C_i(p_{i,1},...,p_{i,m_i}))
6  }
```

Listing 3.14: A destructor $D_{p_{i,j}}$ and the corresponding axiom for a constructor argument $p_{i,j}$. Consequently, for each constructor C_i exactly m_i such function-axiom pairs are generated.

Finally, to support basic functionality of ADTs, it remains to encode the discriminators. This is achieved by adding a so called tag function TG_A for an ADT A that maps instances to integers. Concretely, for two instances a_1 and a_2 of an ADT A instantiated by different constructors, the resulting tag given by the tag function differs as well. In particular, we have $TG_A(a_1)$!= $TG_A(a_2)$. Listing 3.15 gives the definition of the tag function with its axioms for an ADT A.

```
1  // Tag function
2  function TG_A(a: A[T_1,...,T_k]): Int
3
4  // Tag axioms
5  axiom {
6      forall p_{1,1}: S_{1,1},...,p_{1,m_1}: S_{1,m_1} :: {C_1(p_{1,1},...,p_{n,m_1})}
7          TG_A(C_1(p_{1,1},...,p_{1,m_1})) == 1
8  }
9
10     ⋮               ⋮               ⋮               ⋮
11
12 axiom {
13     forall p_{n,1}: S_{n,1},...,p_{n,m_n}: S_{n,m_n} :: {C_n(p_{n,1},...,p_{n,m_n})}
14         TG_A(C_n(p_{n,1},...,p_{n,m_n})) == n
15 }
```

Listing 3.15: The corresponding tag function and its axioms for the ADT A. Accordingly, an instance of an ADT A created with constructor C_i has tag i.

A discriminator for a constructor C_i can then be encoded as $TG_A(a)$ == i, where a is some instance of ADT A.

However, there is still a missing piece. Namely, the exclusivity of ADTs introduced in Definition 2.1. Given the encoding for constructors, destructors

and the tag function, implementing Definition 2.1 as domain axioms is straight-forward, as shown in Listing 3.16.

```
1 axiom {
2     forall a: A[T_1,...,T_k] :: {TG_A(a)}{D_{1,1}(a)}...{D_{n,m_n}(a)}
3         a == C_1(D_{1,1}(a),...,D_{1,m_1}(a)) || ... ||
4         a == C_n(D_{n,1}(a),...,D_{n,m_n}(a))
5 }
```

Listing 3.16: Encoding exclusivity for an ADT.

3.3.2 Contains function

As already mentioned in Section 3.2.2, the derivable function `contains` is a transitive binary relation, which contains the tuple (A, B) if and only if A is contained in B. We encode the `contains` relation as a domain function belonging to a domain named `ContainsDomain` with two type parameters `A` and `B`. (see Listing 3.17)

```
1 domain ContainsDomain[A,B] {
2     function contains(a:A, b:B): Bool
3 }
```

Listing 3.17: The domain for the derivable `contains` function.

Whenever the `contains` function is derived for an ADT, we generate domain axioms that encode the contains relation as defined in Section 3.2.2. In particular, for every constructor $\mathtt{C}_i$, each argument $\mathtt{p}_{i,j}$ is contained in constructor $\mathtt{C}_i$. Listing 3.18 shows the corresponding axiom for constructor $\mathtt{C}_i$. Remember that our deriving system allows blocklisting. Accordingly, if for instance the argument $\mathtt{p}_{i,l}$ is blocklisted, we omit $\mathtt{contains(p}_{i,l}\mathtt{,\ C}_i\mathtt{(p}_{i,1}\mathtt{,...,p}_{i,m_i}\mathtt{))}$ from the axiom.

```
1 axiom {
2     forall p_{i,1}: S_{i,1},...,p_{i,m_i}: S_{i,m_i} :: {C_i(p_{i,1},...,p_{i,m_i})}
3         contains(p_{i,1}, C_i(p_{i,1},...,p_{i,m_i})) && ... &&
4         contains(p_{i,n}, C_i(p_{i,1},...,p_{i,m_i}))
5 }
```

Listing 3.18: The axiom defining the semantics of the contains relation for ADT instances created by constructor $\mathtt{C}_i$.

Finally, it remains to encode transitivity of the contains relation, which turned out to be more involved than encodings we have seen until now, as discussed

in the work on ADTs in Gorba [5]. The authors came up with the following three-step strategy.

1. For each call to `contains`, if the types of its arguments are concrete[2], collect the types as a tuple in a set S
2. Compute the transitive closure S^+ of the tuples in S. In particular, we start by initially setting $S^+ = S$. Then we repeatedly apply the following rule. For two tuples (A, B) and (B, C) in S^+ we add the tuple (A, C) to S^+.
3. For each tuple (A, B) and (C, D) in the set S^+, if $B = C$ generate a domain axiom encoding transitivity as shown in Listing 3.19

```
1 axiom {
2     forall a: A, b: B, c: C ::
3         {contains(a,b)}{contains(b,c)}
4         contains(a,b) && contains(b,c) ==> contains(a,c)
5 }
```

Listing 3.19: The axiom that encodes transitivity of the contains relation for a triple (A, B, C) of concrete types.

3.4 Implementation

In this section we will first reference the codebase, where the ADT plugin is implemented. We proceed by discussing and document the new extension AST nodes that internally represent ADTs. This section is specifically useful for front-end developers, which plan to use the ADT plugin. Afterwards, we shortly list the necessary tasks of the encoder to encode extension AST nodes into ordinary AST nodes. We omit to give more implementation details since they are quite straightforward and hence refer to the codebase for the interested reader. Finally, we discuss the testing infrastructure and performance of the ADT plugin.

3.4.1 Codebase and usage

Currently, the ADT plugin is implemented on a fork of Silver and is publicly available on GitHub[3]. To use the plugin it is additionally required to install either Silicon or Carbon. We suggest to install our forked versions of Silicon[4] and Carbon[5] to also include unit tests. See Section 3.4.4 how to run tests.

[2] A concrete type is a type where all type parameters are resolved.
[3] https://github.com/amaissen/silver
[4] https://github.com/amaissen/silicon
[5] https://github.com/amaissen/carbon

However, in future we plan to add the ADT plugin to Viper's original repositories[6].

Another thing worth mentioning is that the ADT Plugin is currently implemented as a default plugin, which means that the plugin is automatically activated. If this changes in future, one can activate the plugin by passing `--plugin <classPath>` via command line option to Silicon or Carbon, respectively. The current class path for the ADT plugin is the following:

```
viper.silver.plugin.standard.adt.AdtPlugin
```

For general instructions how to install Silver, Silicon or Carbon we refer to their documentation.

3.4.2 AST extension

As discussed in Section 2.2.3, the plugin infrastructure allows to extend the Viper abstract syntax tree (AST). AST nodes are generally defined via Scala case classes with multiple argument lists and extend a base class named `Node`. To support ADTs in Viper via plugin, several new nodes are added to the AST. The list below documents the new AST nodes, where some of them have relevant information in a secondary argument list. We omit the signature of the nodes here for reasons of space but list them in the Appendix (see Appendix A.1).

- `Adt(...)` extends `ExtensionMember`, represents an ADT top-level declaration, and takes the following arguments: a unique name (`String`), a sequence of `AdtConstructor`'s and a sequence of type variables.
- `AdtConstructor()(...)` extends `ExtensionMember`, represents an ADT constructor, and takes the following arguments: a unique name (`String`) and a sequence of local variable declarations defining the constructor's argument list. In the secondary argument list it additionally takes an `AdtType` that defines the return type and a `String` for the corresponding ADT name the constructor belongs to. There exists a matching companion object that should be used to correctly set the arguments in the secondary list by passing the associated `ADT` instance.
- `AdtType(...)` extends `Type` via `ExtensionType` and represents the newly introduced type for ADT's. It requires a name of an ADT as `String` and a `Map` for type variable mapping as arguments. Additionally, it takes a sequence of type variables in the secondary argument list as input. Since the type variables should match the type variables defined in the corresponding `ADT` instance, the associated companion object should be used to set them correctly.

[6] https://github.com/viperproject

- `AdtConstructorApp(...)` extends `Exp` via `ExtensionExp` and represents a constructor application. It requires the name of the constructor (`String`), a sequence of argument expressions (`Exp`) and a `Map` for the type variable mapping. The arguments of the secondary argument list equal the secondary arguments of `AdtConstructor(...)`, which can be set correctly with the corresponding companion object.
- `AdtDestructorApp(...)` extends `Exp` via `ExtensionExp` and represents a destructor application. As argument it takes a `String` which should match a valid destructor identifier, an expression (`Exp`) for the receiver and a `Map` for the type variable mapping. The secondary argument list does not differ from the one in `AdtConstructorApp(...)`
- `AdtDiscriminatorApp(...)` extends `Exp` via `ExtensionExp` and represents a destructor application. It takes exactly the same arguments as `AdtDestructorApp(...)`, except that there is no return type to be set, since it is fixed to `Bool`.

3.4.3 Encoder

The encoder is implemented as a class `AdtEncoder` and applies the encoding of ADTs as presented in Section 3.3. The primary task of the encoder is to transform, more specifically to encode, extension AST nodes into ordinary AST nodes. To do so, the encoder traverses the extended AST using Viper's internal rewriting strategies and

- transforms every `Adt` node to a `Domain` with corresponding `DomainFunc` nodes and `DomainAxiom` nodes according to the encoding for ADTs presented in Section 3.3.
- transforms every `AdtConstructorApp`, `AdtDestructorApp` and `AdtDiscriminatorApp` to the matching `DomainFuncApp`.
- transforms every `AdtType` to the matching `DomainType`.

Moreover, the encoder implements a name manger that keeps track of all currently used identifiers in the program. In particular, it provides a mechanism to generate new unique identifiers. This is very handy to avoiding name clashes when it comes to create new domains and domain functions used to encode constructors, destructors, discriminators and other ADT features.

As already mentioned in Section 2.2.3, the encoding of extension nodes to ordinary AST nodes should happen in the `beforeVerify(...)` hook provided by the plugin infrastructure. In particular, before the AST is passed to a verification backend. Consequently, this is the location where the encoder is invoked.

3.4.4 Tests

The absence of bugs can not be proved with testing, however their number can be reduced. Hence, we implemented a great amount of tests. In particular, we added a new test suite to check ADT features that includes 39 test files, each containing several corner cases. It covers all main stages of the Viper infrastructure, which includes parsing, typechecking/resolving and verification. Furthermore, there are tests that checks the ADT encoding itself and that it does not introduce any unsoundness.

Tests can only be executed with Silicon or Carbon as backend verifier. All available tests including those from core Viper can be executed by calling `sbt test`. However, one can execute the ADT tests only with following command

- for Silicon: `sbt "testOnly viper.silicon.tests.AdtPluginTests"`
- for Carbon: `sbt "testOnly viper.carbon.AdtPluginTests"`

However, as mention in Section 3.4.1 make sure to install the forked versions of Silicon and Carbon, otherwise tests for the ADT plugin are not available.

3.4.5 Performance

The ADT plugin does not perform complex operations that would lead to a significant drop in performance. However, as mentioned in Section 3.4.3, the Viper AST is traversed to transform extension nodes to core Viper nodes, respectively to apply the encoding for ADTs, which is the most costly operation. To implement those traversals and transformations we used Viper's internal framework, so called rewriting strategies that can be applied on an AST. Consequently, overall performance of the ADT Plugin boils down to the performance of this framework and the entire plugin infrastructure. Hence, benchmarking our ADT plugin would most likely not lead to surprising or insightful results, so we omitted a performance analysis. However, far more interesting results could be collected by benchmarking Viper's plugin and rewriting infrastructure but this was out of the scope for this work.

3.5 Future Work

The presented plugin comes with the basic functionality to support use of ADTs in Viper, which includes the possibility of ADT construction, destruction and discrimination. Moreover, it comes with a deriving system that currently supports exactly one function, namely the `contains` function. In Section 3.2.2, we already discussed some useful functions the deriving system could support in future, hence we will not repeat those ideas here again.

However, another feature that is often used together with ADTs is pattern matching. Consider the ADT `Maybe[T]` that represents the data type of same name in Haskell. It has two constructors, namely `Just(value: T)` and `Nothing()`. Then, one might want to implement a method `getOrElse(...)` that takes an instance of `Maybe[Int]` and a default value of type `T` as arguments. If the passed instance of `Maybe[T]` was created by the constructor `Just(value: T)` its value is returned, otherwise the default value is returned. Listing 3.20 gives the concrete implementation of the method `getOrElse(...)` in Viper.

```
1 method getOrElse(m:Maybe[Int], d:Int) returns (res:Int)
2 {
3     if (m.isJust) {
4         res := m.value
5     } else {
6         res := d
7     }
8 }
```

Listing 3.20: A method that returns the value of an instance of `Maybe[Int]`, if it was created with constructor `Just(value: Int)`. Otherwise, the default value `d` is returned.

Introducing a Scala-like `match`-expression for pattern matching would allow to rewrite the method `getOrElse(...)` by conditioning on each constructor with `case`-clauses, as shown in Listing 3.21. The reader may observe that `match`-expression can be rewritten as `if-else` statements, however, for more complicated ADTs this results in big notational overhead. Furthermore, one could extend the plugin to automatically generate exhaustiveness checks for `match`-expression. Hence, supporting this kind of pattern matching would facilitate the use of ADTs in Viper, and could provide a unified solution for front-end verifiers such as Go and Rust.

```
1 method getOrElse(m:Maybe[Int], d:Int) returns (res:Int)
2 {
3     res := m match {
4         case Just(v): v
5         case Nothing: d
6     }
7 }
```

Listing 3.21: The `getOrElse(...)` method implemented with a pattern matching expression instead with an if/else statement. Note that this is hypothetical Viper code, that is not yet supported by the ADT plugin.

Chapter 4

Conclusion

This work presented a plugin for Viper to enable support for ADTs. In particular, we designed a fitting syntax to support basic functionality of ADTs in Viper, which includes constructors, destructors and discriminators, and justified our design decisions.

Using Viper's plugin infrastructure we implemented the corresponding parsing and typechecking for the new ADT features. Additionally, we extended the abstract syntax tree (AST) to internally represent ADTs in Viper. To facilitate the development of front-ends, that would like to support ADTs, we discussed the internal representation of ADTs in more detail and hence it is well documented. Moreover, we formalized the encoding for ADTs, originally elaborated by Gobra's work, and later applied it on the extended AST to encode ADT features to core Viper.

Furthermore, we designed an extendable deriving system to automatically derive useful functions for ADTs with the intention to reduce boilerplate code. Despite, it currently only supports the derivable function `contains`, we described potential functions that could easily be added in the future. We provided a bunch of suitable unit tests that cover every introduced feature for ADTs to ensure that our implementation has a high probability of being correct. Finally, we discussed a pattern matching expression that could be introduced in a future project.

Summing up, the ADT plugin not only facilities the use of ADTs in Viper, but also provides a unified solution for existing and future front-ends, which use Viper as an intermediate language.

Appendix A

Appendix

A.1 Signatures of new AST nodes

In the following section we list the complete signature of the AST nodes related to ADTs in Viper.

ADT declaration

```
Adt(
    name: String, constructors: Seq[AdtConstructor],
    typVars: Seq[TypeVar] = Nil,
    derivingInfo: Map[String, (Option[Type], Set[String])] = Map.empty
)(
    val pos: Position = NoPosition,
    val info: Info = NoInfo,
    val errT: ErrorTrafo = NoTrafos
) extends ExtensionMember {...}
```

ADT constructor

```
AdtConstructor(
    name: String,
    formalArgs: Seq[LocalVarDecl]
)(
    val pos: Position,
    val info: Info, val typ: AdtType,
    val adtName : String,
    val errT: ErrorTrafo
) extends ExtensionMember {...}
```

ADT type

```
AdtType(
```

```
    adtName: String,
    partialTypVarsMap: Map[TypeVar, Type]
)(
    val typeParameters: Seq[TypeVar]
) extends ExtensionType {...}
```

ADT constructor application

```
AdtConstructorApp(
    name: String,
    args: Seq[Exp],
    typVarMap: Map[TypeVar, Type]
)(
    val pos: Position,
    val info: Info,
    override val typ: Type,
    val adtName:String,
    val errT: ErrorTrafo
) extends ExtensionExp {...}
```

ADT destructor application

```
AdtDestructorApp(
    name: String,
    rcv: Exp, typVarMap: Map[TypeVar, Type]
)(
    val pos: Position,
    val info: Info,
    override val typ: Type,
    val adtName:String,
    val errT: ErrorTrafo
) extends ExtensionExp {...}
```

ADT discriminator application

```
AdtDiscriminatorApp(
    name: String,
    rcv: Exp,
    typVarMap: Map[TypeVar, Type]
)(
    val pos: Position,
    val info: Info,
    val adtName:String,
    val errT: ErrorTrafo
) extends ExtensionExp {...}
```

Bibliography

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 3, pages 147:1–147:30. ACM, 2019.

[2] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustanm. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[3] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wroclaw, Poland, 2011.

[4] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An experimental applicative language. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, page 136–143, New York, NY, USA, 1980. Association for Computing Machinery.

[5] Paul Dahlke. *Extending a Go Verifier with Algebraic Data Types*. Bachelor's thesis, ETH Zurich, Zürich, 2021.

[6] Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, pages 183–198, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[8] Marco Eilers and Peter Müller. Nagini: A static verifier for python. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 596–603, Cham, 2018. Springer International Publishing.

[9] Programming Methodology Group. Viper online tutorial. https://viper.ethz.ch/tutorial/. Online; accessed 20th April 2022.

[10] Haskell.org. Glasgow haskell compiler. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/index.html. Online; accessed 20th April 2022.

[11] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[12] Rustan Leino and Peter Müller. A basis for verifying multi-threaded programs. In *ESOP '09 Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, page 378. Springer-Verlag Berlin, Heidelberg, March 2009.

[13] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, page 41–62, Berlin, Heidelberg, 2016. Springer-Verlag.

[14] Simon Peyton Jones. A history of haskell: being lazy with class. In *The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, June 2007.

[15] Malte H. Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. PhD thesis, ETH Zurich, Zürich, 2016.

[16] Fabio Streun. *ETool Support for Termination Proofs*. Bachelor's thesis, ETH Zurich, Zürich, 2019.

[17] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. Gobra: Modular specification and verification of go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 367–379, Cham, 2021. Springer International Publishing.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Adding Algebraic Data Types to a Verification Language

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Maissen	Alessandro

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zurich, 04/26/22

Signature(s)

A. Maissen

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.