Trees that Grow in the Paragon Compiler

A Step Towards Modularity

Bachelor of Science Thesis in Computer Science and Engineering

John Andersson, Anders Berggren Sjöblom,
Anders Bäckelie, Johannes Ljung Ekeroth,
Lukas Skystedt, Lina Terner

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Bachelor of Science Thesis DATX02-20-30

Trees that Grow in the Paragon Compiler

A Step Towards Modularity

John Andersson
Anders Berggren Sjöblom
Anders Bäckelie
Johannes Ljung Ekeroth
Lukas Skystedt
Lina Terner

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF
GOTHENBURG

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Trees that Grow in the Paragon Compiler
A Step Towards Modularity
John Andersson, Anders Berggren Sjöblom,
Anders Bäckelie, Johannes Ljung Ekeroth,
Lukas Skystedt, Lina Terner

Supervisor: Niklas Broberg, Department of Computer Science and Engineering
Examiner: David Sands, Department of Computer Science and Engineering

Bachelor of Science Thesis DATX02-20-30
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Abstract

Paragon is a programming language that extends Java with statically checked information flow control policies. Paragon's compiler, which is written in Haskell, has a large type checker. Its current implementation is monolithic, making the compiler challenging to develop. Paragon's authors, Broberg et al., have proposed to partition the type checker into five phases, and this project is a step towards such an implementation.

We identify the representation of Paragon's abstract syntax tree as an essential design aspect and emphasize extensibility to conform to the phases' varying requirements. Based on a programming idiom, Trees that Grow, by Najd and Jones, we implement an extensible abstract syntax tree in Paragon's compiler. We observe that our implementation introduces a substantial amount of boilerplate code. To alleviate the issue, we employ several methods for generic programming. We conclude that our AST implementation is extensible but complex.

Sammandrag

Paragon är ett programmeringsspråk som utökar Java med statiskt verifierade regler för informationsflöden. Således har Paragons kompilator, som är skriven i Haskell, en stor komponent som utför typkontroll. Den komponenten är monolitisk vilket gör det svårt att utveckla kompilatorn. Paragon's utvecklare, Broberg m.fl. har föreslagit att typkontrollen kan delas upp i fem faser, och det här projektet är ett steg mot en sådan implementation.

Vi identifierar att representationen av Paragons abstrakta syntaxträd är en viktig designaspekt och betonar utökningsbarhet för att kunna anpassa det efter fasernas varierande krav. Baserat på ett programmeringsidiom, *Trees that Grow*, av Najd och Jones, implementerar vi ett utökningsbart abstrakt syntaxträd i Paragons kompilator. Vi observerar att vår implementation medför en omfattande mängd repetativ kod. För att minska problemets omfattning tillämpar vi flera metoder för generisk programmering. Vi konstaterar att vår implementation är utökningsbar men komplex.

We would like to thank our supervisor, Niklas Broberg, for his frequent and constructive feedback, and for his constant encouragement.

Contents

1

Introduction

Software systems frequently need to deal with security concerns, including *information flows.* These concerns can often be expressed as *policies* that describe under what circumstances information may flow between different parts of a system. Some examples of possible policies include:

- Only users who are friends may see each other's pictures on a social media platform.
- Documents should not leave an organization without being approved by a manager.
- Text from an input form on a website may not end up in a database without first being sanitized (e.g., to prevent XSS attacks and SQL injection).

An approach to ensure that a particular software implementation adheres to a specified set of information flow policies is to use a programming language in which these policies may be expressed and thus checked by a compiler or runtime system. One such language is the Java-based language Paragon, developed by Broberg et al. [1], the compiler of which is the topic of this project.

Paragon is not the first programming language that can express information flow concerns. The language authors describe Paragon as a third-generation information flow language, distinguishing it from earlier languages based on simpler models. First-generation information flow control is based on military-style classification. Information is classified in a strict hierarchy of security levels where information may flow up in the hierarchy but never down. This kind of policy is sometimes called "Denning style confidentiality policies", and *FlowCAML* [2] is an example of a language that implements this idea. Second-generation information flow control extends the strict Denning-style policies with the concept of declassification. Some scenarios require deliberate leaking of information to lower clearance levels under certain conditions. The programming language *Jif* extends Java with information flow control that supports this notion [3].

Second-generation languages provide declassification as an explicit feature. Paragon, however, offers more expressive policies which can, for example, express fine-grained declassification. Paragon builds on earlier work on Paralocks [4], a language which, in turn, builds on Flow Locks [5] by the same authors. At the heart of Paragon's type system are *actors*, *locks*, and *policies*. Actors are object references that represent entities with information-flow concerns. Locks are predicates used to model the state of the system with regards to information flow during runtime. Finally, policies are labels on fields and variables which express how information is allowed to flow within the program. Paragon primarily enforces its policies using static analysis at compile-time. However, the state of locks, policies, and actors cannot be known until runtime, and consequently, Paragon includes a library encoding some of its constructs as plain Java objects.

1.1 The Original Paragon Compiler

The original implementation of the Paragon compiler was written in Haskell. An important dependency was the *Flow Locks framework*, also developed by Broberg [6]. The framework is an implementation of Flow Locks [5], a language for information flow policies. Paragon instantiates the framework to model its information flow control mechanisms.

Prior to this project, the compiler was last updated in November of 2014 [7]. It was not very modular and, in particular, performed type checking monolithically, making it hard to update and add features. Furthermore, the compiler suffered from other problems:

- Parts of it had outdated documentation or lacked it completely.
- Internally, it had circular dependencies.
- The associated test suites did not work with modern versions of Cabal.[1]

Finally, Haskell and The Glasgow Haskell Compiler (GHC) had been updated in non-backward-compatible ways. Therefore, modern versions of GHC were unable

[1]Cabal is the build tool used for Paragon's compiler [8].

to compile the Paragon compiler, making it difficult to continue its development.

1.2 Purpose and Scope

Initially, the purpose of this project was to rework the Paragon compiler so that it would become easier to add features and make changes to it in the future. We aimed to do so by updating, restructuring, and modularizing it. Broberg et al. have suggested that the compiler's type checker can be partitioned into five phases [1], which was our intended method for increasing modularity.

Early in the process, we identified the representation of the abstract syntax tree (AST) as a fundamental design aspect, both in preparation for the modularization and the resulting compiler's extensibility. Therefore, we shifted our focus to the data structure that represents the AST, and in particular, to an implementation using a programming paradigm called *Trees that Grow* [9].

Although we compare our AST implementation based on Trees that Grow to other possible implementations, we have not implemented any other approach in Paragon's compiler.

There are several resources on the Paragon language available on Paragon's research page[2]; hence, we do not provide any in-depth explanation of the language or its type system.

1.3 Structure of the Report

This report covers a sequence of problems and possible solutions that build on each other. Therefore, we have organized the report by topic and continuously introduce relevant theory, present methods, and discuss results relevant to each topic.

We begin, in chapter 2, by discussing modularization of the Paragon compiler through an increase in the number of compilation phases. Then, we suggest what language constructs future developers should prioritize, and we also examine circular internal dependencies in the original compiler, together with an attempt at removing them. The chapter is primarily aimed at future developers of the compiler, but it also motivates the need for an extensible AST, which is the topic of chapter 3. In chapter 3, we demonstrate the problems of AST representation in

[2]`http://www.cse.chalmers.se/research/group/paragon/`

Haskell and its relation to the expression problem. We then describe how we use the programming idiom *Trees that Grow* to implement an extensible AST in the compiler. Our adoption of the idiom increased the amount of boilerplate code, and we, therefore, explore methods for reducing it. We conclude with a summary of the project's results and some final thoughts in chapter 4.

2

Modularization

As described in section 1.2, the initial goal was to modularize the compiler. We have done some work towards this goal, and in this chapter, we present central information that could be useful for future work on modularizing the compiler. Primarily, modularization would consist of dividing the type checker into five phases, outlined in section 2.1. We aimed only to implement a limited subset of the Paragon language, further discussed in section 2.2. The original compiler contained circular dependencies, which we explore in section 2.3.

2.1 Phases

The main task in modularizing the compiler is to separate different aspects of compilation by increasing the number of compilation phases. The original compiler consisted of six phases: lexing, parsing, name resolution, type checking, translating to Java-compatible AST, and Java source-file generation.

Broberg et al. propose to divide the type-checking phase into five distinct phases, each handling different tasks related to type checking or information-flow analysis [1]. Each phase would perform different operations, depending on information computed in previous phases. We dedicate chapter 3 to how to store the computed information in the AST. Sections 2.1.1–2.1.5 summarize the five phases and give examples of what kind of operations and checks they perform. Our description should be viewed as an example-focused complement to the descriptions given by Broberg et al. [10].

2.1.1 Type Checking

The first phase performs type checking that corresponds to that of Java. It includes checking that types are compatible in assignments and method calls,[3] that arguments to `if` statements are of type `boolean`, and that thrown exceptions match method signatures. The Paragon-specific information-flow analysis requires type information; therefore, we cannot wait for Java's compiler to perform type checking after Paragon's compiler has generated the Java files. The Paragon-specific checks that should be performed in this phase are:

- The types of arguments to locks must be consistent with the lock's declaration. In the `open` expression in the example below, we need to check that the type of `variable` is compatible with `MyType` and that `MyLock` is indeed a `lock` (as declared on the line above).

```
lock MyLock(MyType);
open MyLock( variable );
```

- Policy expressions must be of type `policy`. In the code below, `variable` must be a `policy` and not some other type. Note that policy expressions are not limited to previously declared policies, but also include policy-literals declared on-the-fly (e.g. `{:}`), meet and join of policy expressions (`*` and `+`), and methods generating policies.

```
public ?variable void myMethod(){}
```

- Runtime exceptions must be properly handled. In the code below, the parameter, `s`, could be `null`, which would cause a `NullPointerException` to be thrown at runtime. Thrown exceptions can cause information flows since they can reveal information about variables, possibly violating policies. Thus, the compiler should reject the code unless uncaught runtime exceptions are declared in the method signature.[4]

```
int myLength(String s) {
  return s.length()
}
```

[3] We write "compatible" to include subtype relations and automatic type conversions like promotions of numeric types and auto-boxing.

[4] Note however that the original compiler, incorrectly, accepts this code snippet.

2.1.2 Policy Type Evaluation

After the first phase, Java-like type checking is completed, and Paragon-specific information-flow analysis remains. The second phase should translate syntactic representations of policies, locks, and actors (object references) to semantic values.

- In finding semantic representations for actors, aliases must be identified. That is, object references can sometimes be known to refer to the same object, in which case they have the same actor identity, such as `a` and `b` in the following example.

  ```
  final Object a = new Object();
  final Object b = a;
  ```

- Sometimes the value of a field containing a policy cannot be determined until runtime; in such cases, lower and upper bounds should be calculated. An example of this is when a policy is assigned within an *if-else* block, shown below. The policy denoted by `C` then has a semantic representation expressed as bounds obtained from the values of `A` and `B`.

  ```
  policy A...
  policy B...
  policy C = A + B
  if(myBool)
    C=A;
  else
    C=B;
  ```

2.1.3 Lock State Evaluation

At each program point, the locks that can statically be known to be open should be calculated. There are four main ways we can know that a lock is open at a certain point, and should be contained in the set of open locks:

- The method `Open` is called with a lock as a parameter. The lock is open after that point until it might be closed by a method or statement.

- After method calls to methods declared with `+MyLock`,[5] the declared locks are open.

- Locks can be queried by using them as Boolean values. In flow-control structures such as if-statements, we know that a queried lock is either open or

[5]Note that `+` is an overloaded operator since it also denotes the meet of two policy expressions.

closed in the associated branches, giving more precise approximations of the lock state at the respective program points. For example:

```
if(MyLock) {
  // In this branch the lock is open
} else {
  // In this branch the lock is closed
}
```

The lock state is modeled as the set of locks guaranteed to be open. Thus, the information that a lock is closed in a branch is superfluous.

Propagation of query information must also happen for loops (`for`, `while`, `do-while`), and ternary expressions.[6]

```
for(; MyLock; /*lock is open here*/){
  //lock is open here
}
```

- Using ~`MyLock` when declaring a method. By using this syntax, the method demands that `MyLock` is open when calling the method.

Wherever a direct flow takes place, that program point is, to be able to generate policy constraints, annotated with the current lock state, i.e., the set of open locks. The language constructs in which direct flows can take place are assignments, `return` statements, and method invocations.

2.1.4 Policy Constraint Generation

The second to last phase generates constraints, relating policies to each other, given the lock state calculated, and annotated at relevant program points in the previous phase. Constraints can be represented as $p \sqsubseteq_{LS} q$, where p and q are policies and LS is the lock state, which means that policy p has to be less restrictive than policy q.

More generally, when assigning an expression to some variable, the policy of the expression needs to be less restrictive than that of the variable. Therefore, in the following assignment, the constraint $pol1 \sqsubseteq_{LS} pol2$ should be generated.

```
?pol1 int x;
?pol2 int y = x;
```

[6]Note that for `for`-loops, the information needs to be distributed to both the loop body and the third field in the loop head.

Constraint generation occurs not only for policies in direct assignments, but also for policies as write effect on methods, and indirect flows such as if-statements:

- When an assignment to a variable occurs inside a method with write effects, the method policy needs to be less restrictive than the variable's policy. In the following example, the constraint $pol4 \sqsubseteq_{LS} pol3$, should, therefore, be generated.

```
?pol3 int x;

!pol4 void m() {
  x = 3;
}
```

- When assignments occur inside a branch of an if-statement whose condition depends on some variable with a policy, that policy must be less restrictive than the policy in the assignment. The example below generates the constraint $pol5 \sqsubseteq_{LS} pol6$

```
?pol5 boolean x;
?pol6 boolean y;

public void m() {
  if (x) {
    y = true;
  }
}
```

Note that variables without explicitly declared policies have inferred policies that also generate constraints.

2.1.5 Policy Constraint Solving

The last phase tries to find a solution to the constraints generated in the previous phase. Solving the constraints means finding an assignment of policies satisfying all the constraints. It is sufficient to determine that there exists a possible valid assignment of policies, rather than finding that set of policies, to satisfy the constraints.

An example of an assignment which would generate a constraint that is not solvable is:

```
?{:} int x;
?{Object x:} int y = x;
```

Here `x` has the policy *top*, which is the most restrictive policy, while `y` has the least restrictive policy, *bottom*. The generated constraint would state that the policy of `x` has to be less restrictive than `y`, which it is not.

The Flow Locks framework handles the solving of the constraints and provides a function, `solve`, that the Paragon compiler can, after instantiating the framework, use.

2.1.6 Storing Phase-Specific Information

As previously mentioned, we can notice from the phase descriptions that phases carry out computations based on information from previous phases. Some of this information is tightly coupled with specific AST nodes. For example, lock states vary between program points and are therefore well suited to be stored in the AST. The variation in what information needs to be stored is the primary reason why the AST needs to look different in different phases, something we return to in chapter 3.

2.2 Language Constructs to Prioritize

The essential features that Paragon offers, beyond Java, pertain to information flow control. Considering the phases specified in sections 2.1.1–2.1.5, there are certain language constructs which are of more interest than others, in terms of how they utilize Paragon's policy system. Many of these occur in *assignments*, which we recommend as the first construct to implement. Assignments incorporate policies, actors, and locks, and checking them involves all the compiler phases. When assignments are implemented, most of the core infrastructure should be in place to implement, e.g., methods. We want to point out that conditionals such as *if-else* statements require special logic for the propagation of lock states and is, therefore, a key feature to implement. Lastly, exceptions, and "pseudo-exceptions" (`break` and `continue`), also require special logic.

We suggest that these language features should be implemented before, e.g., `do-while` loops, since the latter add very little new compiler logic and also do not contribute much to the expressiveness of the language.

Our recommendations are limited to the language constructs we have considered and are, thus, incomplete. However, they may form a basis for future developers to build on.

2.3 Internal Dependencies

A secondary problem of modularity, which may be of interest to future developers, pertains to the compiler's internal module dependencies. An advantage of modular design is that one module can be changed or completely replaced with only a small impact on other modules, presupposing a clear separation between the modules. This section analyzes the internal module dependencies of the compiler.

In the original compiler, the modules dealing with type checking depend on each other in such a way that there is little separation or abstraction. We show direct dependencies – imports – between the type checker's modules in figure 2.1 (with abbreviated names). An arrow from one module to another indicates that the first module imports the other module. For simplicity, we omit transitive imports from the figure; specifically, if a module `A` imports a module `B` and `B` imports a module `C`, no arrow is drawn from `A` to `C` even if `A` imports `C`. The same dependency graph without these simplifications is given in appendix A.

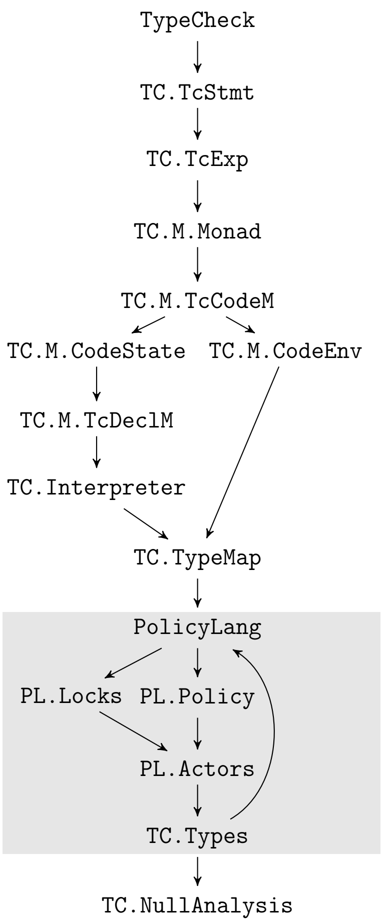

Figure 2.1: Internal module dependencies of the type checker with abbreviated names: `TC` for `TypeCheck`, `PL` for `PolicyLang`, `M` for `Monad`.

The first thing to notice is that there are *circular dependencies*. That is, some modules depend on themselves by transitivity, e.g., `PolicyLang` → `PL.Locks` → `PL.Actors` → `TC.Types` → `PolicyLang` (see the gray selection in figure 2.1). Circular dependencies not only indicate a lack of separation of concerns, but it is also a technical issue because GHC (the Haskell compiler used in this project) is unable to resolve these dependencies on its own and requires '`.hs-boot`' files for dependency resolution [11].

The second thing to notice is that the dependency hierarchy does not correspond to the Haskell module structure (identified by the module names). For example, `TC.M.CodeEnv` imports `TC.TypeMap` which is higher up in the module tree, which indicates a lack of abstraction layers. This is further confirmed by inspecting the complete import graph (appendix A), which shows that there are imports from every level in the dependency graph to almost every layer below it.

2.3.1 An Attempt at Resolving Circular Dependencies

To make the separation of modules clearer, one may desire to remove the circular dependencies between modules of the type checker (e.g., `TC.Types`) and modules involved in the instantiation of the Flow Locks framework (e.g., `PolicyLang`), or equivalent modules in a modularized version of the compiler.

The issue is exemplified by the data types `ActorSetRep` from a sub-module of `PolicyLang` and `TcRefType` from `TC.Types`. Naturally, the type checker depends on types from `PolicyLang` (which instantiates the Flow Locks framework). However, `ActorSetRep` contains a field of type `TcRefType` (used as the type of actors), creating a circular dependency. `TcRefType` is also mutually dependent on other types in `TC.Types` and cannot be moved to a separate module without introducing new circular dependencies.

Parameterizing `ActorSetRep` by the type of actors, removing `TcRefType` from its definition, can remove the latter dependency. The data type can then be used by monomorphizing it with `TcRefType` as the type argument, for example, by declaring a type synonym outside of `PolicyLang`.

The problem with this solution is that the newly introduced type variable must be propagated to every other data type referring to it, making it cumbersome. Also, the type becomes polymorphic even though its semantics in the compiler is as a concrete type. Furthermore, the purpose of `PolicyLang` is to instantiate the Flow Locks framework, and by parameterizing its data types with the type of locks, we move some of the framework instantiation to modules in the type checker.

After implementing the parametrization, we strongly doubt that it improves the code, even when introducing type synonyms for the monomorphized type. We suggest that the issue may, instead, be resolved by identifying another type for actors on which both the type checker and framework instantiation can depend. There are a few similar (and related) occurrences of circular dependencies, and we believe these could be handled the same way.

3

Representing the Abstract Syntax Tree

One of the harder technical challenges when implementing a compiler is how to represent a program's abstract syntax tree (AST) during compilation. As outlined in section 2.1, we want the AST to contain different information in different compilation phases. To deal with this, we explore extensible ASTs, a case of what Philip Wadler calls the expression problem [12]. This chapter aims to explain why AST representation is an issue, motivate how it relates to the Paragon compiler, and show how we apply the Trees that Grow (TTG) idiom to solve it.

3.1 The Expression Problem

Research about extensible data types is an active topic in computer science. In this domain, the expression problem is the problem of how to implement a data type specified by cases such that new cases and functions over the data type can be added in a way that preserves static type safety.

To understand the expression problem better, consider the data type for Boolean expressions containing variables, negation, and conjunction, as well as a pretty-printing function that operates on the data type.

```
data Expr = Var Char
          | Neg Expr
          | And Expr Expr
pretty :: Expr -> String
pretty (Var c)       = c:""
pretty (Neg e@And{}) = "¬(" ++ pretty e ++ ")"
pretty (Neg e)       = "¬" ++ pretty e
pretty (And e1 e2)   = pretty e1 ++ " ∧ " ++ pretty e2
```

To add a new function that operates on `Expr`, for example, a function that retrieves all the variables in an expression, we simply need to implement it by pattern matching on the existing data constructors in the `Expr` data type.

```
variables :: Expr -> [Char]
variables (Var c)      = [c]
variables (Neg e)      = vars e
variables (And e1 e2) = vars e1 ++ vars e2
```

Rather effortlessly, we were able to add the `variables` function without affecting the other function (`pretty`). But what happens if we change the data type itself? Perhaps we would like to include a representation of Boolean values for use during evaluation. We give a possible implementation in figure 3.1. When adding one constructor, we had to alter pre-existing code by adding a case for the new constructor to every function, requiring it to be recompiled. This is the expression problem as it pertains to Haskell: how can we add new cases to the data type, without having to recompile existing code?

```
data Expr =
          ⋮
          | Val Bool

pretty :: Expr -> String
⋮
pretty (Val b) = show b

variables :: Expr -> [Char]
⋮
variables (Val _) = []
```

Figure 3.1: Example of adding a constructor to the data type `Expr`.

3.2 The Expression Problem in the Paragon Compiler

In the compiler, we need a representation of the AST, and we have different requirements for it in different compiler phases. Specifically, we want to add additional information to the constructors (decoration fields) that vary between phases. Also, we want to extend the data type with a different set of constructors for different phases. This is where the expression problem comes in. We cannot cleanly express such a data type in Haskell. Two workarounds which lie on opposite ends of the spectrum of type specificity versus data type reuse are:

1. Define a new AST for every phase, which includes only the fields and constructors necessary for the specific phase.

2. Use a single set of data types that include all fields and constructors used in at least one compiler phase.

Both approaches have significant shortcomings, namely tremendous code duplication and bloated code with poor type safety, respectively.

The original Paragon compiler used an approach that lies in between the two extremes. It used parameterized algebraic data types for each syntactic category,[7] whose data constructors represent different productions. The parameter was used to decorate the AST with different types in different phases. For example, the parser produced an AST decorated with source positions (locations in the Paragon source files where constructs were parsed) while the type checker decorated the AST with type information. We provide an extract from the part of the AST that represents expressions in figure 3.2.

```
data Exp a
    = Lit       a (Literal a)
    | BinOp     a (Exp a)       (Op a) (Exp a)
    | PolicyExp a (PolicyExp a)
    ⋮
```

Figure 3.2: Implementation of expressions in the AST, using parametric polymorphism.

We identify three major problems with using parametric polymorphism:

1. Every data constructor has to be decorated with the same type.

2. Data constructors cannot be added or removed between phases.

[7]Syntactic categories roughly correspond to language constructs.

3. It is only possible to have one decoration type (per type parameter).

The first and second problems both occur clearly in the Paragon compiler, and the third relates to how easy it would be to update the compiler in the future. Specifically:

1. After type checking, only some syntactic categories should be decorated with type information, while the others should be decorated with source positions.

2. The compiler performs a null analysis (to find variables that might be null, see section 2.1.1), and related information needs to be added as decorations to the AST. However, the locations where the information is needed is not in one-to-one correspondence with any set of constructors. Thus, new constructors are desirable.

3. When adding new decorations, every occurrence of AST types needs a new type parameter.

The original compiler used dummy default values to handle the first problem, duplicated parts of the AST to deal with the second, and nested multiple pieces of information into a single type to manage the third. We did not find these solutions satisfactory since they sacrifice type safety, and duplicating the AST is cumbersome, both to do and to maintain. Increasing the number of compiler phases also amplifies the severity of the problems.

Najd and Jones [9] discuss the decoration problem in more detail and propose a programming idiom called *Trees that Grow* (TTG), which aims to solve it. Our new implementation of the AST uses a variation on their idiom. The remainder of this chapter discusses two main topics: firstly, how we adapted the AST to use the TTG idiom; secondly, some approaches for reducing the amount of boilerplate caused by the adoption of the idiom.

3.3 Adopting Trees that Grow

This section outlines the conversion of the Paragon compiler's AST from being decorated using parametric polymorphism to using TTG. The point of TTG is to allow the AST to be extended, both by adding fields to existing data constructors and by adding new data constructors to existing types. It does so using type families, available as a GHC language extension (`TypeFamilies`) [13]. As an example, adopting the TTG idiom transforms the data type for expressions given in figure 3.2 to the one given in figure 3.3.

```
data Exp ξ
  = Lit         (XLit ξ)        (Literal ξ)
  | BinOp       (XBinOp ξ)      (Exp ξ)       (Op ξ) (Exp ξ)
  | PolicyExp   (XPolicyExp ξ)  (PolicyExp ξ)
  ⋮
  | ExpExt (XExpExt ξ)

type family XLit ξ
type family XBinOp ξ
type family XPolicyExp ξ
type family XExpExt ξ
```

Figure 3.3: The code in figure 3.2 transformed using the TTG idiom.

In TTG terminology:

- ξ is the *extension descriptor* — the index of the type families used for the extensions. We use uninhabited (except for by `bottom`) types for the specific extension descriptors, one for each phase (e.g., `data TC` for the type checking extension descriptor). It is important to note that the extension descriptor is *not* the type of the decoration as it would be when using parametric polymorphism.

- `XLit`, `XBinOp`, et cetera, are type families used for the *extension fields*. We prefix the name of the type families with "X" to make them easy to distinguish from other identifiers.

- `ExpExt` is an *extension constructor*.

By instantiating the type families to different types in different phases, we can have different decorations for different data constructors and add data constructors when needed.

In many cases, phases do not use some of the extension fields and constructors, in which case we set them to the type synonyms `NoFieldExt` and `NoConExt`, respectively.

```
type NoFieldExt = ()
type NoConExt   = Data.Void.Void

data UD -- Extension descriptor for 'undecorated'
type instance XLit    UD = NoFieldExt
type instance XExpExt UD = NoConExt
```

We use the unit type, `()`, for the field extensions since it always allows us to access a value of the type, namely the type's only value, also named `()`. By setting the field in the extension constructor to `void`, it becomes impossible to build values using it (other than by using `undefined` and related values), effectively disabling it. These operations correspond to multiplying the algebraic data type by the multiplicative unit and adding the additive unit to it, respectively. It is not strictly necessary to give any type family instance at all for unused extension constructors since omitting it would also give an empty type. However, it becomes a bit clearer when written out explicitly. Also, it ensures that nobody gives it an instance elsewhere [9].

Najd and Jones also describe the use of pattern synonyms to, for example, reduce syntactic noise by hiding unused extension fields [9]. For example, a synonym for the undecorated constructor `Lit` would be: `pattern UdLit l = Lit () l`. Moreover, we can use pattern synonyms to give the appearance of constructors having multiple extension fields when we actually decorate them with a single product type. For example, we can use a pair (tuple) as product type:

```
pattern SynSomeConstr ext1 ext2 = SomeConstr (ext1, ext2)
```

Similarly, we can extend the data type with multiple constructors by setting the extension constructor's type family to a sum type[8] and declaring multiple pattern synonyms that differentiate on that sum type. For example, to extend the data type for expressions (figure 3.3) with two constructors, `ExpI` and `ExpS`, in a phase with extension descriptor `X`:

```
data Extension = Ext1 Int | Ext2 String
type instance ExpExt X = Extension

pattern ExpI i = ExpExt (Ext1 i)
pattern ExpS s = ExpExt (Ext2 s)
```

[8]Sum types, also known as tagged unions, are types that represents a choice, represented in Haskell by having multiple constructors for a single data type.

3.3.1 Converting Existing Code to Use TTG

This section outlines some problems that occurred when converting code that uses parametric polymorphism for AST decorations to use the TTG idiom. Section 3.3.2 then demonstrates aspects of the AST conversion using the parser in the Paragon compiler.

Quantity-wise, name changes are the dominant difference. After instantiating the AST appropriately for the relevant phase, all occurrences of AST types need to be changed, from having the decoration types as the parameter to having the extension descriptor as the parameter. Furthermore, some data constructors may need to be replaced with their corresponding pattern synonyms. In Haskell, it is common to have types and data constructors with the same name. Our experience is that automatic text substitution is, therefore, ill-suited for performing the name conversions. However, it might be possible to use type-driven automatic refactoring (such as the Haskell Language Server [14]).

Name changes are straightforward when the AST type constructors are applied to form concrete types, but can become tricky for polymorphic functions. In particular, we are often unable to interact with extension fields polymorphically, which, in our experience, boil down to two kinds of functions.

The first kind of function transforms decorations, giving the type `ast a -> ast b` when using parametric polymorphism. It is frequently captured by the use of the method `fmap` from the `Functor` type class.

```
fmap :: (a -> b) -> ast a -> ast b
```

When `ast` is a `Functor`, the method is parametrically polymorphic in `a` and `b`. Intuitively, it allows us to transform all occurrences of decorations of type `a` to decorations of type `b`, preserving the structure of the AST. Our new AST cannot be a `Functor` since the decoration types may differ between constructors and do not match the type parameter (the extension descriptor). We discuss this issue further in section 3.4.

The second kind of function previously had the signature `ast a -> a`, and extracted the decoration from an AST type. Originally, these functions were captured by a custom type class, and we have not examined possible replacements.

In the context of modularization, we believe that, compared to continuing using parametric polymorphism, the required code changes are far more extensive under the TTG idiom. With the increase in the number of compilation phases, we would need to perform a greater number of AST traversals, likely increasing the need for generic functions, as discussed in section 3.4.

3.3.2 Converting the Parser

When changing the AST to use the TTG idiom, it became incompatible with the existing parser due to mismatching types. Originally, the polymorphic AST types were applied to the type `SourcePos`, which was used for the decorations. For example, the root of the AST was declared as follows:

```
data CompilationUnit a = CompilationUnit a ...
```

The parser thus returned a value of type `CompilationUnit SourcePos`. However, in the TTG implementation, the parameter is the extension descriptor, `PA` for the parser phase. The declaration looks like this:

```
data CompilationUnit ξ
  = CompilationUnit    (XCompilationUnit    ξ) ...
  | CompilationUnitExt (XCompilationUnitExt ξ) ...

type family XCompilationUnit    ξ
type family XCompilationUnitExt ξ
```

Accordingly, we had to update the parser to return a value of type `CompilationUnit PA`. We still want the source positions to be present, which we accomplish with the appropriate type family instances[9]:

```
data PA -- Extension descriptor
type instance XCompilationUnit PA = SourcePos
```

The result is that only the type signatures differ, not the values. For the type `CompilationUnit`, no extension constructor is needed. Thus, we set the type family instance to `Void`.[10]

```
type instance XCompilationUnitExt = Void
```

Because we use a single decoration type, we do not need pattern synonyms. One might, however, always want to use pattern synonyms for consistency and to make it clearer which instantiation of the AST is being used. The latter becomes more

[9]In reality, we use a macro to generate all type family instances at once. We discuss this, and give the actual code for the parser, in section 3.4.1.

[10]Again, we actually use a macro to generate type family instances.

important for code that interacts with multiple instantiations because the constructor names would indicate whether the data has been processed by the phase or not.

Previously, the AST decorated using parametric polymorphism had an associated type class, `Annotated` (we use the terms *decoration* and *annotation* synonymously), that defined two methods for common operations.

```
class Functor ast => Annotated ast where
  amap :: (a -> a) -> ast a -> ast a
  ann  :: ast a -> a
```

The first method, `amap`, is a specialization of `fmap`. Since the class requires all instances to also be `Functor`s, `amap` can be implemented in terms of `fmap`. The parser utilizes neither `amap` nor `fmap`,[11] so we leave the discussion of these methods to section 3.4.

The second method, `ann`, was used to extract the decoration from AST nodes. The implementations of the method were generated automatically. As with `fmap`, we cannot implement this method for our new AST. Therefore, we do not go into detail about how the instance generation worked. An example of where the `ann` function was used is in a function for constructing names. Both `Name` and `Ident` are AST types.

```
mkUniformName :: (a -> a -> a) -> NameType -> [Ident a] -> Name a
mkUniformName f nt ids = mkName' (reverse ids)
  where mkName' [] = panic (syntaxModule ++ ".mkUniformName")
                     $ "Empty list of idents"
        mkName' [i] = Name (ann i) nt Nothing i
        mkName' (i:is) =
          let pre = mkName' is
              a   = f (ann pre) (ann i)
          in Name a nt (Just pre) i
```

The method `ann` is applied to different types at different locations. Specifically, it is applied to both `Ident`s (the variable `i`) and `Name`s (the variable `pre`). The old signature for `ann` is no longer valid. Since we are dealing with only two types, we get two different signatures, and we can quickly implement one function for each.

[11] More accurately, the parser does not use `fmap` over any AST types, but over other types.

```
annId :: Ident ξ -> XIdent ξ
annId (Ident d _) = d

annName :: Name ξ -> XName ξ
annName (Name d _ _ _) = d
```

If we were to simply substitute the occurrences of `ann` with our new functions, we would run into an issue. For our use case in the parser, we know that both `Ident` and `Name` are decorated with `SourcePos`. That is:

```
type instance XIdent PA = SourcePos
type instance XName  PA = SourcePos
```

In general, however, `XIdent` and `XName` are not necessarily instantiated to the same type. To get around this issue, we can constrain the function only to be applicable when the type family instances are the same. The tilde syntax (from the type family language extension) expresses type equality.

```
mkUniformName :: XName ξ ~ XIdent ξ
              => (XName ξ -> XName ξ -> XName ξ)
              -> NameType -> [Ident ξ] -> Name ξ
mkUniformName f nt ids = mkName' (reverse ids)
    where mkName' [] = panic (syntaxModule ++ ".mkUniformName")
                             "Empty list of idents"
          mkName' [i] = Name ( annId i ) nt Nothing i
          mkName' (i:is) =
              let pre = mkName' is
                  a = f (annName pre) (annId i)
              in Name a nt (Just pre) i
```

This solution is sufficient for the parser phase since only two types are involved. However, the method scales poorly, both because one function has to be declared for each type and because constraints can become large and cumbersome. We have not looked into other possible solutions, but suggest that something similar to the solution we present for replacing `fmap`, in section 3.4, may be applicable.

3.4 Reducing Boilerplate

The adoption of TTG resulted in repetitive code — boilerplate. This section presents how, using libraries and language extensions for generic programming and template programming, we were able to reduce this boilerplate.

3.4.1 Type Family Instances

For most extension descriptors (compiler phases), the majority of all of the AST's type families are instantiated to the same type. For instance, after the parsing phase, every constructor but two are extended with a source position field, the naive implementation of which would consist of hundreds of repetitive lines on the form:

```
type instance SomeExtensionFamily PA = SourcePos
```

Instead of writing the instances manually, we generate them using a *Template Haskell* macro [15]. Template Haskell is a GHC language extension and library for metaprogramming. It allows us to inspect and manipulate Haskell's abstract syntax and, thus, generate code. We demonstrate how our macro is used for the parsing phase in the code snippet below, where:

- `makeTypeInsts` is the Template Haskell macro.
- `PA` is the extension descriptor for the parsing phase.
- `SourcePos` is the type of source positions.
- The last argument is a list of type families, in this example specified as all families except `XTypeArgumentExp` and `XRefTypeArrayType`,[12] which are removed by using the list-difference operator, \\.

```
$(makeTypeInsts ''PA ''SourcePos
  (allFamilies \\ [''XTypeArgumentExp, ''XRefTypeArrayType]))
```

The macro implementation is straightforward. An auxiliary function generates a single instance:

```
makeTypeInst :: Name -> Name -> Name -> Q [Dec]
makeTypeInst ind typ fam = return [ TySynInstD fam $
                                    TySynEqn
                                    [ConT ind]
                                    (ConT typ)
                                  ]
```

All the types occurring in this definition are part of Template Haskell's represen-

[12]We exclude `XTypeArgumentExp` and `XRefTypeArrayType` because they are instantiated to other types.

tation of Haskell AST. We do not take advantage of Template Haskell's syntax for *declaration quotations*, although we would like to since it would be more readable than our current implementation. The reason is that we have been unable to handle the type family variable correctly, which appears to be due to a limitation of Template Haskell related to splicing names into declarations [16]. We give an implementation that almost works:

```
makeTypeInst' ind typ fam = [d| type instance $fam $i = $t |]
 where
   i = conT ind
   t = conT typ
```

We then use our auxiliary macro in the definition of the macro that works with a list of type families. We apply `makeTypeInst` to each family in the list and join the resulting quotation monads (`Q`) and lists of declarations to `Q [Dec]`.

```
makeTypeInsts :: Name -> Name -> [Name] -> Q [Dec]
makeTypeInsts ind typ fams =
  join <$> mapM (makeTypeInst ind typ) fams
```

3.4.2 Type Class Instances

Type families complicate the derivation of type class instances. We use open type families for the extensions. *Open*, as opposed by *closed*, type families, allow instances to be declared separately from the family itself, which is suitable for extensible data types. Unfortunately, the open type families prevent us from automatically deriving type instances by appending, e.g., `deriving (Eq, Show)` to data type definitions. The reason is that GHC cannot deduce that all family instances are instances of the respective classes. However, using standalone deriving, we can add the constraint that all the types contained in the data types have to be members of the type classes (requires the language extension `StandaloneDeriving`), which enables the compiler to generate the instances.

The code for the deriving is very repetitive in two ways. Firstly, every `deriving instance` declaration needs a long list of constraints containing every type family that occurs somewhere (recursively) in the type for which to derive the instances. Secondly, every pair of type class and data type needs a `deriving instance` declaration.

The solution to the first problem is described by Najd and Jones [9] and uses the `ConstraintKinds` language extension. We define a *constraint synonym* that, given a type class and an extension descriptor, gives a constraint requiring all the

AST types (instantiated with the given extension descriptor) to be instances of the given class.[13,14]

```
type ForallXFamilies (f :: * -> Constraint) ξ =
  ( f (XCompilationUnit ξ), f (XPackageDecl ξ), ...)
```

We solve the second problem using Template Haskell. A macro produces the desired `deriving instance`-declarations for each combination of type class and data types, both given as lists to the macro. Like with the type family instances, we use a separate macro to generate each instance. However, we are now able to use quotation syntax.

```
deriveInstance :: Name -> Name -> Name -> Q [Dec]
deriveInstance constraint clazz typ =
  [d| deriving instance $con $c x => $c ($t x) |]
  where con = conT constraint
        c   = conT clazz
        t   = conT typ
```

We generate an instance for each pair of AST type and type class by using Haskell's list comprehensions. Like with the type family instance macro, the last operation is to join all the generated declarations together into a single list contained in the quotation monad.

```
deriveInstances :: Name -> [Name] -> [Name] -> Q [Dec]
deriveInstances constraint clazzes types
  = fmap join $ sequence $
    [deriveInstance constraint clazz typ | clazz <- clazzes
                                         , typ   <- types]
```

3.4.3 Pattern Synonyms

Like with the type family instances, it turns out that many pattern synonyms are similar, and writing them out for each data constructor, for each phase, is tiresome. Again, we employ Template Haskell to generate them en masse. In contrast to class and type family instances that are highly uniform and thus simple to generate, pattern synonyms are a bit more involved. The arities of the data

[13]GHC limits the size of constraint tuples to 62 elements [17], which is not enough to cover all the type families in the AST. To get around the problem, the 62nd element of the tuple can itself be a tuple of size 62, et cetera.

[14]The language extension `UndecidableInstances` is required for the compiler to accept this, since we would otherwise have the constraint contain the very thing that should be derived (something akin to `deriving instance f a => f a`).

constructors differ, and the macro must thus inspect their structure. Figure 3.4 gives a (made up) example of how to use the macro, complete with the declaration of the extension descriptor, an AST type (`Example`), and a type instance.

```
--| Extension descriptor
data De
--| Example type
data Example ξ = Exam1 (XExam ξ)
               | Exam2 (XExam ξ) Int
type family XExam ξ
--| Extension field type instance
type instance XExam De = (String, (Bool, Maybe Char))

--| Template Haskell macro usage
$(makePatternSyns "De" ['Exam1, 'Exam2] [p| (s, (b, Just c)) |])

-- At compilation, the above line generates:
-- pattern DeExam1 s b c   = Exam1 (s, (b, Just c))
-- pattern DeExam2 s b c i = Exam2 (s, (b, Just c)) i
```

Figure 3.4: An example of generating pattern synonyms using Template Haskell. For simplicity, we use the same type family for both constructors in this example.

The arguments to the Template Haskell macro, `makePatternSyns`, are:

1. A (`String`) prefix for the pattern synonym, we use the name of the extension descriptor for clarity, but it is arbitrary.

2. A list of constructors for which to generate the synonym.

3. A *pattern* used on the right-hand side of the pattern synonym for the first field — the extension field.[15]

The macro extracts all variables from the given pattern and prepends them to the argument list on the left-hand side of the pattern, in a flattened structure. Note that the pattern synonym also captures all other fields (the `Int` in `Exam2`). The macro is a bit involved; therefore, we refrain from presenting it here and refer the interested reader to appendix C.

[15]Note that the extension field has to be the first field in each constructor.

3.4.4 Preservation and Simple Transformations

In most compiler phases, subtrees of the AST are either preserved or changed only by a simple transformation on the extension fields. These cases are very easily handled when using parametric polymorphism for the AST decorations, but they become considerably more involved with the adoption of TTG.

An example of where we desire to preserve a subtree of the AST is with Paragon's import declarations (they correspond to Java's import declarations), which are unaffected by several of the compiler phases. When using parametric polymorphism for the AST decorations, this problem is trivial — the entire subtree can be kept as is. After adopting the TTG idiom, however, the subtrees before and after each phase are only conceptually equivalent but differ nominally in their extension descriptors, as demonstrated in figure 3.5.

```
-- Extension descriptors
data A
data B

type family Fam ξ where
  Fam A = Int
  Fam B = Int

newtype W ξ = W (Fam ξ)

-- Type error: types A and B are not equal.
wAtoB :: W A -> W B
wAtoB x = x
```

Figure 3.5: Example of how types which are isomorphic, but not equal, result in a type error.

The problem in figure 3.5 is solvable using *coercions*, shown below [18]. Coercing one type to another relies on the types having the same runtime representation, which is the case when using `newtype`. However, when using `data` declarations instead of `newtype`, the types are merely *isomorphic*, and the solution will, therefore, not work.

```
wAtoB' :: W A -> W B
wAtoB' = coerce
```

Instead, we can define a type class using the `MultiParamTypeClasses` language extension with a single method to describe the preservation function. All types of the subtree of the AST we would like to preserve, need to implement this class.

```
class Preserve ast ξ ζ where
  preserve :: ast ξ -> ast ζ
```

Using GHC Generics [19], we can generate these instances automatically in three steps. First, we convert the data type to a generic representation. Second, a function, analogous to the preservation function (called `gPreserve`) replaces the extension descriptor. Third, we convert the generic representation back into the AST type. The code below shows how the default implementation of the `Preserve` class is defined in terms of `gPreserve` (`Rep` and `Generic` are classes from `GHC.Generics`).[16]

```
class Preserve ast ξ ζ where
  preserve :: ast ξ -> ast ζ
  default preserve :: ( GPreserve (Rep (ast ξ)) (Rep (ast ζ))
                      , Generic (ast ξ), Generic (ast ζ) )
                   => ast ξ -> ast ζ
  preserve = to . gPreserve . from
```

To be able to use the `preserve` function, the following three requirements must be met:

1. All involved data types must be members of the `Generic` class, accomplished by adding `deriving Generic` to their declarations.

2. Instance declarations for the relevant AST types must be declared:

   ```
   instance Preserve ast ξ ζ
   ```

   where ξ and ζ are concrete extension descriptors, and `ast` is a type constructor for an AST type with kind[17] `* -> *`.

3. For every data constructor `C` with type family extension field `XC` of `ast`, it must hold that `XC` $\xi \sim$ `XC` ζ, where $\sim$ is type equality [21].

The preservation problem is a special case of the more general problem of applying transformations to the extension fields (preservation is something akin to applying an identity transformation). A particularly simple example of where we use such a

[16]The `default` syntax allows default definitions of class methods with restricted signatures. It requires the GHC language extension `DefaultSignatures`.

[17]Kinds are the types of type-level entities [20].

transformation in the compiler is in the code generation phase, which removes all the extension fields (sets them to the unit value, `()`). The original compiler leveraged the fact that the AST types were `Functor`s to perform such transformations, thus offloading most of the work on the automatically derived `Functor` instances for the AST types.[18]

With the adoption of TTG, the AST is no longer a `Functor`. Indeed, a primary reason for adopting TTG is that we are not limited to having the same type of decoration on every node of the AST. It should be noted, though, that each AST type must be handled explicitly, turning what previously was simple `fmap` statements into hundreds of lines of code. In practice, however, most of the AST types have the same extension fields in most phases. Thus, it would be desirable to be able to use a blanket implementation for most AST nodes and only implement the divergent cases manually.

Our solution to the preservation problem can be extended to support transformations. We add a type class with two parameters, one for the source type and one for the target type.

```
class Conv a b where
    conv :: a -> b
```

Most of the implementation closely follows the one for preservation. The primary difference is in one of the instances for transforming the generic representation of the data type, which allows us always to convert fields of the AST types as long as they are members of the `Conv` class.

```
instance Conv a b => GTransform (K1 i a) (K1 j b) where
    gTransform (K1 x) = K1 (conv x)
```

It is important to note that the `conv` instance is dispatched (selected) depending on both the source *and* target types, meaning we can specify multiple conversions from the same source type as long as they have different target types. Frequently, conversions with the same source and target type are just identity functions. We can readily implement all these cases at once with:

```
instance Conv a a where conv = id
```

We give a complete implementation of the transformation code in appendix B.

[18] Automatically deriving `Functor` instances requires a GHC language extension [22].

As a final note, a `Conv` must be declared for each phase pair that one wants to convert between, to allow different transformations in different phases. It might be possible to extend the `Conv` class with additional type parameters for the phase descriptors, reducing all conversion classes to a single class with instances for different phases.

3.4.5 Alternative Methods for Generic Programming

The methods and libraries for generic programming that we have presented are not the only ones available in Haskell. For general surveys of generic programming in Haskell, see [23] and [24].

We find the library Uniplate [25] and its extension Multiplate [26] to be particularly interesting. These libraries provide mechanisms for generic traversals, primarily by handling the recursive calls, allowing the user to focus on the transformations themselves. Multiplate extends Uniplate to support mutually recursive data types (which Paragon's AST contains). Unfortunately, we have not been successful in solving our boilerplate problems using these libraries due to the lack of uniformity of our AST types.

It should be possible to replace GHC Generics with Template Haskell in the cases where we have used it. However, we believe that the implementation would be considerably more complex since we would need to traverse a large part of Haskell's AST. Nonetheless, Template Haskell is more flexible, and in combination with Uniplate or Multiplate, it might be a better fit.

3.5 Readability Issues

We have found that the adoption of the TTG idiom comes with unwanted consequences for readability. Partially, it is intrinsic to the idiom; partially, it is a consequence of our methods for decreasing boilerplate.

Parametric polymorphism offers clarity. The types of decorations are readily available in functions' type signatures, and the types of data constructors and their fields are immediately visible in the data type's definition. In contrast, TTG hides the type of decorations in the type family instantiations for the relevant extension descriptor, which is often located in separate files from the AST types themselves. The same applies to the types of data constructors and their fields, including any extension constructors. Also, pattern synonyms add another step when examining the definitions.

Moreover, our Template Haskell macros are significant culprits in obfuscating code, primarily when generating pattern synonyms. Because we generate names (pattern synonyms) at compile-time, it can be hard to find the source of pattern synonyms by inspecting source code. Likewise, but to a lesser extent, generating type class instances can make their implementations hard to trace.

Of course, GHC is always able to deduce relevant information, so loading the entire project into an interactive shell (GHCI) allows us to query for types and other information via the `:type` and `:info` commands. Similar functionality is available through many text editor integrations. However, it is our experience that such integrations work poorly with the Paragon compiler codebase, at least under recent versions of GHC (8.6.5). We speculate that the cause is one or multiple of the many language extensions used in the project.

Finally, our solution to the decoration-transformation problem is somewhat opaque. Although it is akin to other automatically derived type classes, it is somewhat more complex since three different classes are involved. The relation between these may not be obvious. In particular, it can be hard to identify when the `transform` method calls the `conv` method (from the `Conv` class), and which instance it dispatches. We do believe that proper organization of the code and clear documentation mostly relieves this last issue.

To make the readability issues more concrete for the reader, we dedicate the remainder of this section to a more extensive example that partially builds on code from previous sections, shown in figure 3.6. The provided code is incomplete and only showcases some of the previously described readability issues.

Imagine that we want to find the type of `vs` in the following line of code:

```
myFun :: Expr EV -> ...
myFun (EvAnd sp vs e1 e2) = ...
```

We need to find what `EvAnd` is. However, a simple text search fails to find its definition because it is generated at compile time from the Template Haskell macro on line 20 in figure 3.6. Now, we do not have access to the pattern synonym's type signature, but we can see that it corresponds to the second element in the pair used for the decoration. The decoration type is specified on line 19, once again by a Template Haskell macro. Finally, line 17 informs us that the sought type is `[Variable]`. Consider that we also want to know the type of `e1`. As before, we have to go through the pattern synonym to find that it is an alias for the `And` constructor (in reality, we are probably able to guess this immediately).

```
-- File containing declaration of the extensible data type

type Variable = Char
data Expr ξ = Var      (XVar ξ)    Variable
            | Neg      (XNeg ξ)    Expr
            | And      (XAnd ξ)    Expr     Expr
            | ExprExt  (XExprExt ξ)

type family XVar ξ
type family XNeg ξ
type family XAnd ξ

-- File containing instantiation for a specific phase

data EV -- Extension descriptor
type SourcePos = Int
type Dec = (SourcePos, [Variable]) -- The decoration type

$(makeTypeInsts ''EV ''Dec [''XLit, ''XNeg, ''XAnd])
$(makePatternSyns  "Ev" [p | (sp, vs) ] [''Lit, ''Neg, ''And])

-- Extension constructor
type instance XExprExt X = Bool
pattern EvVal :: Bool -> Expr X
pattern EvVal b = ExprExt b
```

Figure 3.6: Example of a declaration of a data type, instances and pattern synonyms.

Another line of the declaration of `myFun` may be:

```
myFun (EvVal b) = ...
```

From the function signature, we can deduce that `EvVal` is a data constructor for `Expr`, and, following the previous pattern, we incorrectly guess that it is an alias for a constructor `Val`. In reality, it is an alias for the extension constructor `ExprExt`, but the mistake is easy to make.

4

Final Thoughts

A significant amount of work remains to produce a modular Paragon compiler. We have primarily focused on AST representation but also conducted some work on other aspects of modularization (our code is available on GitHub [27]). We end this report with a discussion of our results.

4.1 Is TTG Suitable for the Paragon Compiler?

We have converted the AST in the Paragon compiler to use the TTG idiom. We found that the AST became more extensible than before and that the precision in its types increased. However, the switch entailed problems with incompatibility with existing code (see 3.3), large amounts of boilerplate code (see 3.4), and a decline in readability. Our solutions to the first and second problems increased the severity of the third problem.[19] Moreover, we use a multitude of GHC extensions, making the project less accessible and possibly less likely to be compatible with future versions of Haskell and GHC. Also, the project's compilation time has increased substantially.

We cannot say for certain whether TTG, or our boilerplate reduction methods, are suitable for the Paragon compiler, or not. However, we believe that the TTG implementation is sufficiently extensible and makes many illegal states unrepresentable but that it is substantially more complex than decoration by parametric polymorphism.

It is also worth reiterating that there are other alternatives to AST representation than those explored in this report, and a better solution may be attainable. The same is true for our boilerplate reduction methods. Although we were unsuccessful

[19]It is, however, easy to replace our Template Haskell macros with the code they generate, which can be obtained by passing the `-ddump-splices` flag to GHC.

in leveraging Uniplate or Multiplate for the specific cases we discussed, it is possible they, or other libraries, can solve the problems in a superior way. Hopefully, the description of our methods can be of utility for others who intend to apply the TTG idiom.

4.2 Modularity

Finding a representation for the AST is an important step towards a modular compiler for Paragon. In addition to the AST data type itself, we have also worked on other aspects of the compiler. Specifically, we have:

- updated the parser, name resolver, and code generator (and parts of the monolithic type checker) to work with the new AST,
- updated various functions and data types that were incompatible with the new AST,
- constructed a new test bench driver that uses Cabal,
- implemented the first of the five phases (the type checker) for field declarations,
- begun implementation of the second phase, policy type evaluation.

We hope that our work, including this report, will aid future efforts to modularize the compiler. This includes both our discussions regarding the AST, but also compiler phase descriptions and other discussed issues.

Bibliography

[1] N. Broberg, B. van Delft, and D. Sands, “Paragon for Practical Programming with Information-Flow Control,” *Asian Symposium on Programming Languages and Systems (APLAS)*, 2013.

[2] V. Simonet, *Flowcaml.* [Online]. Available: `https://www.normalesup.org/~simonet/soft/flowcaml/`.

[3] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, *Jif 3.0: Java information flow*, Jul. 2006. [Online]. Available: `http://www.cs.cornell.edu/jif`.

[4] N. Broberg and D. Sands, “Paralocks — Role-Based Information Flow Control and Beyond,” *POPL’10, Proceedings of the 37th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2010.

[5] ——, “Flow locks: Towards a core calculus for dynamic flow policies,” in *Programming Languages and Systems*, P. Sestoft, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 180–196, ISBN: 978-3-540-33096-7.

[6] N. Broberg, *Flowlocks-framework: Generalized flow locks framework*, Apr. 2019. [Online]. Available: `https://hackage.haskell.org/package/flowlocks-framework`.

[7] ——, *Paragon’s github repository.* [Online]. Available: `https://github.com/niklasbroberg/paragon2`, Accessed February 14, 2020.

[8] Cabal Development Team, *The haskell cabal.* [Online]. Available: `https://www.haskell.org/cabal/`, Accessed April 29, 2020.

[9] S. Najd and S. P. Jones, “Trees that grow,” *Journal of Universal Computer Science*, vol. 23, pp. 42–62, Jan. 2017. [Online]. Available: `http://www.jucs.org/jucs_23_1/trees_that_grow/jucs_23_01_0042_0062_najd.pdf`.

[10] N. Broberg, B. van Delft, and D. Sands, "Paragon for Practical Programming with Information-Flow Control – Technical Report," 2013. [Online]. Available: `http://www.cse.chalmers.se/research/group/paragon/publications/BDS13-TR.pdf`.

[11] GHC Team, *How to compile mutually recursive modules*, Apr. 2020. [Online]. Available: `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/separate_compilation.html#mutual-recursion`.

[12] M. Torgersen, "The expression problem revisited," vol. 3086, Jun. 2004, pp. 123–143. DOI: `10.1007/978-3-540-24851-4_6`.

[13] GHC Team, *Ghc typefamilies extension*, Apr. 2020. [Online]. Available: `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#type-families`.

[14] haskell-language-server developers, *Haskell-language-server*. [Online]. Available: `https://github.com/haskell/haskell-language-server`.

[15] T. Sheard and S. Peyton Jones, "Template meta-programming for haskell," in *Proceedings of the 2002 Haskell Workshop, Pittsburgh*, Oct. 2002, pp. 1–16. [Online]. Available: `https://www.microsoft.com/en-us/research/publication/template-meta-programming-for-haskell/`.

[16] N. Collins, *Support spliced function names in type signatures in th declaration quotes*. [Online]. Available: `https://gitlab.haskell.org/ghc/ghc/issues/15298`, Accessed June 3, 2020.

[17] GHC Team, *Github: Ghc tuples*, Apr. 2020. [Online]. Available: `https://github.com/ghc/ghc/blob/master/libraries/ghc-prim/GHC/Tuple.hs`.

[18] J. Breitner, R. A. Eisenberg, S. Peyton Jones, and S. Weirich, "Safe zero-cost coercions for haskell," *SIGPLAN Not.*, vol. 49, no. 9, pp. 189–202, Aug. 2014, ISSN: 0362-1340. DOI: `10.1145/2692915.2628141`. [Online]. Available: `https://dl.acm.org/doi/10.1145/2692915.2628141`.

[19] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh, "A generic deriving mechanism for haskell," *SIGPLAN Not.*, vol. 45, no. 11, pp. 37–48, Sep. 2010, ISSN: 0362-1340. DOI: `10.1145/2088456.1863529`. [Online]. Available: `https://dl.acm.org/doi/10.1145/2088456.1863529`.

[20] S. P. J. et al., *The haskell 98 report*, Dec. 2002. [Online]. Available: `https://www.haskell.org/onlinereport/decls.html#sect4.1.1`.

[21] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly, "System f with type equality coercions," in *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, ser. TLDI '07, Nice, Nice, France: Association for Computing Machinery, 2007, pp. 53–66, ISBN: 159593393X. DOI: `10.1145/1190315.1190324`. [Online]. Available: `https://dl.acm.org/doi/10.1145/1190315.1190324`.

[22] GHC, *Derive functor instances.* [Online]. Available: `https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/derive-functor`.

[23] R. Hinze and J. Jeuring, "Generic haskell: Practice and theory.," Jan. 2003, pp. 1–56. DOI: `10.1007/b12027`.

[24] J. Jeuring, S. Leather, J. P. Magalhães, and A. Yakushev, "Libraries for generic programming in haskell," May 2008, pp. 165–229. DOI: `10.1007/978-3-642-04652-0_4`.

[25] N. Mitchell and C. Runciman, "Uniform boilerplate and list processing or: Scrap your scary types," Jan. 2007, pp. 49–60. DOI: `10.1145/1291201.1291208`. [Online]. Available: `https://dl.acm.org/doi/10.1145/2692915.2628141`.

[26] R. O'Connor, *Multiplate: Lightweight generic library for mutually recursive data types.* [Online]. Available: `https://hackage.haskell.org/package/multiplate`, Accessed May 13, 2020.

[27] J. A. et al, *Our fork of paragon on github.* [Online]. Available: `https://github.com/Lukas-Skystedt/paragon2`.

[28] *Ghc generics.* [Online]. Available: `https://wiki.haskell.org/GHC.Generics`.

Appendix A

Dependencies

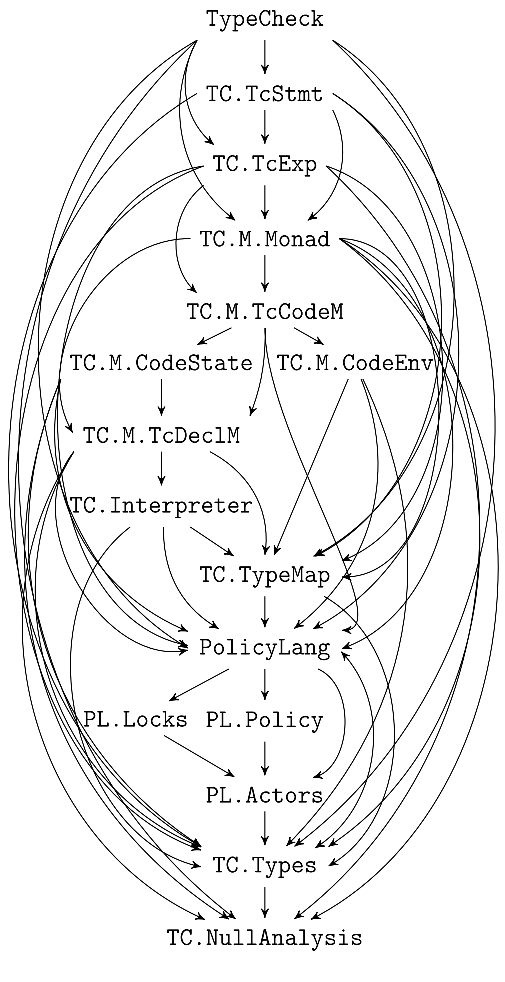

Figure A.1: Imports between different modules involved in type checking.

Appendix B

Transform Code

The part of the below code that uses `GTransform` is loosely based on an example from the Haskell Wiki on GHC Generics [28].

```
-- | Captures conversions of decoration types
class Conv a b where
  conv :: a -> b

-- Every type can be converted to itself
instance Conv a a where
  conv = id


class Transform ast x y where
  -- | The actual method that converts an AST type from one
  -- extension descriptor to another.
  transform :: ast x -> ast y
  -- In most cases, we can create the instance automatically
  -- using GHC generics. (However, it is still possible to
  -- define custom instances.)
  default transform :: ( GTransform (Rep (ast x)) (Rep (ast y))
                       , Generic (ast x)
                       , Generic (ast y) )
                    => ast x -> ast y
  transform = to . gTransform . from

-- | A class for the conversion of the generic representation of
-- the AST type(s) (for use with GHC Generics).
class GTransform f g where
  gTransform :: f a -> g a

--
instance GTransform V1 V1 where
  gTransform = id
```

```
-- Constructors without arguments are unchanged
instance GTransform U1 U1 where
  gTransform = id

-- This is the interesting case. We can convert types when a
-- Conv instance exists.
instance Conv a b => GTransform (K1 i a) (K1 j b) where
  gTransform (K1 x) = K1 (conv x)

-- We can transform an AST type with one extension descriptor
-- to the same type with another extension descriptor when the
-- conversion is specified by a Transform instance.
instance Transform ast x y
    => GTransform (K1 i (ast x)) (K1 j (ast y)) where
  gTransform (K1 x) = K1 (transform x)

-- Meta-information. Simply transform the contained value.
instance GTransform f g
    => GTransform (M1 i c f) (M1 j d g) where
  gTransform (M1 x) = M1 (gTransform x)

-- Sum types (constructor alternatives) are converted by
-- converting the contained value.
instance (GTransform f1 f2, GTransform g1 g2)
    => GTransform (f1 :+: g1) (f2 :+: g2) where
  gTransform (L1 l) = L1 (gTransform l)
  gTransform (R1 r) = R1 (gTransform r)

-- Product types (multiple fields in a constructor) are
-- converted by converting all contained values.
instance (GTransform f1 f2, GTransform g1 g2)
    => GTransform (f1 :*: g1) (f2 :*: g2) where
  gTransform (l :*: r) = gTransform l :*: gTransform r
```

Appendix C

Pattern Synonym Macro

```
makePatternSyn :: String -> Name -> Q Pat -> DecsQ
makePatternSyn prefix conName rhPatQ = do
  rhPat <- rhPatQ

  let (_pre, suff) = takeUnqualified $ show conName
  let newConName = mkName $ prefix ++ suff

  -- The type given here is annoying to work with since it consists of
  -- applications..
  (DataConI _nam _typ parNam) <- reify conName
  let (VarP temp) = rhPat
  -- ..Instead we extract its parent (the type it constructs),..
  (DataConI name typ par) <- reify conName
  -- ..get its declaration..
  (TyConI dec) <- reify par
  -- ..and find the constructor again,..
  let [NormalC _ bangTypes] = case dec of
    (DataD    _ctx _name _binds _kind cons _deriv) ->
        filter (\(NormalC n _) -> n == conName) cons
    (NewtypeD _ctx _name _binds _kind con  _deriv) -> [con]
  -- now with the type given as a list. We throw away the first field,
  -- which should be the TTG extension field.
  let (_extfield:conArgs) = map snd bangTypes :: [Type]

  -- Extract all the names that are used in a pattern in the right hand
  -- side.
  let patNames = patternNames rhPat
  -- Generate names for the remaining constructor fields.
  bindingNames <- mapM (const (newName "a")) conArgs

  let lhsPattern = PrefixPatSyn $ patNames ++ bindingNames
  let rhsPattern = ConP conName $ rhPat : map VarP bindingNames

  return [ PatSynD newConName lhsPattern ImplBidir rhsPattern ]


-- | Given a pattern, find all 'VarP' recursively and extract their names.
-- That is, find all variable bindings in a pattern.
patternNames :: Pat -> [Name]
patternNames (LitP _)              = []
patternNames (VarP name)           = [name]
patternNames (TupP pats)           = concatMap patternNames pats
patternNames (UnboxedTupP pats)    = concatMap patternNames pats
```

```haskell
patternNames (UnboxedSumP pat _ _) = patternNames pat
patternNames (ConP _ pats)         = concatMap patternNames pats
patternNames (InfixP pat1 _ pat2)  = patternNames pat1 ++ patternNames pat2
patternNames (UInfixP pat1 _ pat2) = patternNames pat1 ++ patternNames pat2
patternNames (ParensP pat)         = patternNames pat
patternNames (TildeP pat)          = patternNames pat
patternNames (BangP pat)           = patternNames pat
patternNames (AsP _ pat)           = patternNames pat
patternNames WildP                 = []
patternNames (RecP _ fpats)        = concatMap (patternNames . snd) fpats
patternNames (ListP pats)          = concatMap patternNames pats
patternNames (SigP pat _)          = patternNames pat
patternNames (ViewP _ pat)         = patternNames pat

-- | Split a qualified name to an unqualified name and the prefix. Eg.
-- "GHC.Maybe.Just" should return ("GHC.Maybe.", "Just").
takeUnqualified :: String -> (String, String)
takeUnqualified name = let (suffR, preR) = break (=='.') $ reverse name
                       in (reverse preR, reverse suffR)
```