A Survey on Modular Interpreters and Language Composition

Damian Frölich
University of Amsterdam, The Netherlands
d.frolich@uva.nl

ABSTRACT

Programming languages increase programmers' productivity significantly. Language oriented programming (LOP) is a programming paradigm that makes languages a first class citizen in problem solving. The paradigm puts the language at the centre of problem solving by developing a language specifically for the problem, abstracting away complexities of the domain into the language implementation. The created language allows a more concise solution to the problem and can support static checking of domain violations and perform domain specific performance optimisations. However, constructing interpreters for languages is complex and takes a significant amount of time, making a LOP approach less effective. Nonetheless, by applying modular techniques to define programming languages, the engineering effort required for a language can be reduced.

In this survey I give an overview of functional, object oriented, and attribute grammar approaches facilitating the modular construction of interpreters, and evaluate the approaches according to the Language Extension Problem (LEP). Based on this evaluation, the strength and weaknesses of the different approaches are identified and an overview of the most suitable approach for specific LEP criteria is given.

1 INTRODUCTION

Programming languages provide crucial abstractions for programmers to increase productivity, maintainability and safety. In addition, a programming language can be defined that encodes domain knowledge, increasing efficiency, and enabling end-user programming. Language oriented programming (LOP) [25] is a programming paradigm that puts the language at the centre for problem solving by developing a language that captures the requirements, enabling a concise solution to the problem at hand, allowing invariant checking, and enabling domain specific performance optimisations.

However, development of a programming language is not trivial. A language often requires multiple components to be useful, like an interpreter, pretty printer, debugger, and so on. Furthermore, language design is complex and implementing a useful performant language from scratch is hard. The complexity of language development makes it difficult to achieve benefits with a LOP approach. Nonetheless, by applying modular language development techniques, using pre-defined components, or extending a base language, the complexity of language construction can be reduced, making a LOP approach more feasible.

In this work, I perform a systematic search of the current literature to identify approaches suitable for the construction of modular interpreters. In addition, the approaches are evaluated according to a modified version of the Language Extension Problem (LEP). This results in the following contributions.

- An overview of the current literature on approaches for constructing modular interpreters, classified on the paradigm provided by the approach.
- A modified version of the language extension problem that defines several extension on the original problem and redefines some existing criteria, making it suitable as a filter for future literature reviews and usable as an evaluation framework for evaluating approaches focusing on the construction of modular interpreters.
- Application of the modified LEP to evaluate several approaches and gain insight in how approaches compare on different criteria, which can be used to decide the best approach for specific requirements.
- Identify a main trade-off when choosing an approach for the construction of modular interpreters.

2 BACKGROUND

In this section I give an informal introduction to the theory of initial algebra semantics, which is used as an abstraction to discuss the different approaches. Furthermore, I introduce the expression problem which formulates the difficulty of modularity, and the Language Extension Problem (LEP) which focuses on the difficulty of modularity in language engineering.

2.1 Initial algebra semantics

Initial algebra semantics [15] takes a categorical approach to formal semantics of programming languages and provides an abstraction to discuss the different approaches. In initial algebra semantics, a signature Σ is family of disjoint sets indexed by $S^* \times S$ where S is the set of sorts. A Σ-algebra A assigns to each sort $s \in S$ a carrier A_s and to each operation $\sigma \in \Sigma$ an operation σ_A. An algebra S is initial in a class C iff there exists a unique homomorphism $h_A : S \longrightarrow A$ for every $A \in C$. In the context of programming languages, abstract syntax is an initial algebra in the class of signature algebras and a homomorphism — also referred to as a semantic function — takes the abstract syntax to a semantic algebra, assigning semantics to the abstract syntax.

A simple integer expression language can be described with the theory of initial algebra semantics by defining the signature as containing the sorts $\{Int, Expr\}$ with the indexed family of sets: $\Sigma_{Num,Expr} = \{Num\}$ and $\Sigma_{(Expr,Expr),Expr} = \{Add\}$. The resulting initial algebra contains terms Num 5, Add(Num 1, Num 2), Add(Add(Num 1, Num 2), Num 3), etc. An example semantic algebra is the algebra which takes the integers as the carrier for both sorts and assigns integer addition semantics to the signature.

This simple expression language can be defined in a functional language, Haskell in this case, as follows:

```
data Expr = Num Int
          | Add Expr Expr

eval :: Expr -> Int
eval (Num x) = x
eval (Add x y) = eval x + eval y
```

The data type definition corresponds to the initial algebra and the eval function is a semantic function for the language.

The language can also be described using an object oriented approach:

```
trait Expr {
  def eval(): Int
}

class IInt(v: Int) extends Expr {
  def eval() = this.v
}

class Add(l: Expr, r: Expr) extends Expr {
  def eval() =
    this.l.eval() + this.r.eval()
}
```

In this case, the initial algebra is given by the base class and the classes extending the base class. The semantic function is given by the *eval* method defined in the base class.

2.2 Language extension

A language can be extended on two parts: the signature can be extended which requires an update to all existing algebras or a semantic function can be added which requires an implementation for all operations in the signature.

In the functional example, the first extension corresponds to extending the data type with a new case and the second extension corresponds to defining a new function over the data type. The first extension is difficult and requires modification, because the data type must be modified and all functions operating over the data type must modified to handle the new case. The other extension is trivial: just define a new function over the data type.

In the object oriented example, the first extension corresponds to adding a new class that extends the base class and the second extensions corresponds to adding a new method to the base class and implementing it for all classes extending the base class. The first extension is trivial but the second requires modification of the base class and all classes extending the base class.

The problem showcased, of extending data types with new cases and defining new operations over existing data types, was named the expression problem [37] by Philip Wadler. Wadler formulated the problem with four requirements which a solution must fulfil: it must be possible to add new cases to existing data types; it must be possible to add new operations over existing data types; extending existing data types or adding new operations must not require recompilation of existing code; and type safety must be statically guaranteed.

The Language Extension Problem (LEP) [20] takes the expression problem and specialises it for language engineering. Since this is a specialisation, the authors formulate the problem in a similar style as Wadler did:

> The Language Extension Problem (LEP) is a new name for an old problem. The goal is to define a family of languages, where one can add a new language to the family by adding new syntax (i.e., new constructors for existing syntactic categories as well as new categories) and also new semantics over existing and new syntax, while conforming to constraints similar to those in the Expression Problem but specialized to language extension.

The authors formulate the criteria of the language extension problem as follows:

(1) **Extensibility in both dimensions:** It should be possible to extend the syntax and adapt existing semantics accordingly. Furthermore, it should be possible to introduce new semantics on top of existing syntax.
(2) **Strong static type safety:** All semantics should be defined for all syntax.
(3) **No modification or duplication:** Existing language specifications and implementations should neither be modified nor duplicated.
(4) **Separate compilation:** Compiling a new language (e.g., syntactic extension or new semantics) should not encompass re-compiling the original syntax or semantics.
(5) **Independent extensibility:** It should be possible to combine and used jointly language extensions (syntax or semantics) independently developed.

Additionally, the paper introduces the concept of 'glue code', and Open and Closed systems. Glue code is described as "code dedicated solely to the interconnection of extensions", and should be kept to a minimum. In a Closed system, all possible language are known. In an Open system that is not the case. This makes an Open system more flexible but also more complex.

3 STUDY

In this section I detail the design and execution of the study, and introduce a modified version of the LEP, which is used to evaluate the approaches.

3.1 Research Questions

To have a concrete evaluation for the study, I define the following two research questions:

(1) What approaches for constructing modular interpreters are currently available in the literature?
(2) How do these approaches compare to each other?

To answer the second question, an adaptation of the LEP is used as an evaluation framework.

3.2 Modified LEP

Overall, the LEP defines good criteria for the evaluation of approaches in language engineering. However, some changes to the criteria are required for our evaluation because they are too strict or not concrete enough for a clear evaluation. For example, the LEP defines modification and duplication as separate criteria, but

the first criteria is used as an inclusion requirement in our search. Without modification and duplication in the **Extensibility in both dimensions** criteria, most approach simply fulfil the criteria by modifying or duplicating the original code. Furthermore, I redefine the idea of adaptable semantics and make it a separate criteria, and extend the type safety criteria to be more focused on semantics.

Therefore, I reformulate the LEP criteria as follows and use the reformulated criteria as our evaluation criteria:

(1) **Extensibility in both dimensions:** It should be possible to extend the syntax and adapt existing semantics accordingly, and extend existing syntax with new semantics. Extension should require no modification or duplication of existing language components.
(2) **Adaptable semantics:** It should be possible to adapt semantics of existing syntax.
(3) **Static semantic guarantees:** All semantics should be defined for all syntax and semantics should not be dependent on order of extension.
(4) **Separate compilation:** Compiling a new language (e.g., syntactic extension or new semantics) should not encompass recompiling the original syntax or semantics.
(5) **Independent extensibility:** It should be possible to combine language extensions independently developed, ideally without any glue code requirements.

Compared to the original formulation, I extracted the **adaptable semantics** in a separate criteria and made it more concrete; changed the **Extensibility in both dimensions** criteria to require that extension is not achieved via modification or duplication, and adapted the **No modification or duplication** criteria accordingly; extended the **Independent extensibility** criteria with the glue code requirement; changed the **Strong static type safety** criteria to the **Static semantic guarantees** criteria by requiring that order of extension does not change the semantics. The last addition is to evaluate if performing an extension is is commutative when combining multiple components. If this is not commutative, changing the order of extension can have a significant impact on the semantics of the language, which requires awareness from the programmer when using the approach.

3.3 Search

A systematic search is used to answer the first question. The systematic search used a snowballing approach. The snowballing approach started with searching the 'modular interpreters' query in the DBLP search engine[1]. DBLP was chosen because it received a good evaluation on uniqueness and completeness in a study on search engines for computer science research [3]. The DBLP search resulted in 11 results, for which three results fulfilled our evaluation criteria. These three results formed the start of the snowballing approach.

With the snowballing approach, I looked at papers citing one of the three selected paper (Citations) and at papers cited by one of the three selected papers (References). To find Citations and References both Google Scholar[2] and Semantic Scholar[3] were used.

[1]https://dblp.org
[2]https://scholar.google.com/
[3]https://www.semanticscholar.org/

The Citations and References were evaluated according to the evaluation criteria. First, papers were evaluated on their abstract, introduction, and conclusion using inclusion criteria 1, 4, 5, 6, and 7. If a paper fulfilled those criteria, the full paper was used to evaluate inclusion criteria 2 and 3. Papers fulfilling all criteria were selected for the next round of snowballing. In total, four rounds were performed resulting in over 1000 Citations and over 200 References (results include duplicates). From those results, 16 papers fulfilled all the evaluation criteria.

The used evaluation criteria are detailed as follows.

(1) The study must introduce an approach that can be used in the construction of modular interpreters.
(2) The study must focus on the abstract syntax and the semantics of a programming language.
(3) The study must give an example of constructing an interpreter with the approach or be accompanied by another paper constructing an interpreter using the approach. This ensures that the approaches can be evaluated according to actual implementations.
(4) The study must be a primary source. With a primary source, I mean a source that introduces the approach. For example, a study using the approach without modification as part of a bigger system is not considered a 'primary' source in our evaluation.
(5) The study must be available in the English language.
(6) The study must be peer reviewed, ensuring the quality of the study.
(7) The study must be available and not a duplicate of an existing study.

4 RESULTS

The results of the survey and the evaluations of the selected approaches are displayed in Table 1. For clarity, the approaches are divided into several categories: functional, object oriented, and attribute grammars. In this section, the evaluation for every approach is explained and is structured according to the earlier mentioned categories.

4.1 Functional

4.1.1 Monads. The first two approaches are centred around the usage of monads to handle computations in an interpreter. Monads provide a framework to describe the common properties of computations in a categorical semantics [24]; and are a concept from category theory that is applied in functional programming — especially pure functional programming [35]. However, monad composition is not trivial [17], which makes extending a monadic interpreter — an interpreter expressed as a monad — difficult.

To overcome this difficulty, [21] introduces monad transformers for the monadic interpreter. With monad transformers, monads can be stacked on top of each other making it possible to combine monads in a modular way. Stacking monads influences **Static semantics guarantees** since a different stacking order can result in different semantics for the whole stack. In addition, monad transformers alone are not enough to achieve **Extensibility in both dimensions**, because monad transformers make the computations

Table 1: Evaluation of the different approaches according to our evaluation criteria. The Extensibility in both dimensions criteria is omitted from the table, since all approaches fulfil that criteria as part of our search evaluation. The ● indicates that a criteria is fully met by the approach; ◓ indicates that a criteria is partially met; and ○ indicates that a criteria is not met by an approach.

	Adaptable semantics	Strong static type safety	Separate compilation	Independent extensibility	Open/Closed
4.1 **Functional**					
Monad transformers [21]	●	◓	○	●	Open
Monad transformers [8]	●	◓	●	●	Open
Data types á la carte [30]	●	●	●	●	Closed
Polymorphic variants [12]	●	◓	●	◓	Closed
Open data types and Open functions [22]	○	●	○	●	Closed
Tagless [2]	●	●	●	●	Closed
4.2 **Object oriented**					
Object algebras [4]	●	●	●	○	Open
Object algebras with implicit context propagation [16]	●	●	●	◓	Open
Javascript modules [5]	●	○	N/A	●	Open
Mixin modules [7]	●	◓	○	●	Closed
Extensible algebraic data types with defaults [39]	●	◓	○	○	Closed
4.3 **Attribute grammars**					
UUAG (AspectAG) [33]	●	●	○	○	Open
Composable attribute grammars [10]	●	●	○	◓	Open
Composable attribute grammars (separate evaluation) [10]	●	●	●	◓	Closed
Modular attribute grammars [6]	●	○	○	◓	Closed
Multiple attribute inheritance [23]	●	◓	○	●	Closed

of an interpreter extendable, but not the signature or the semantics. Therefore, [21] combines monad transformers with extensible union types and type classes. Extensible union types can be used to combine types together into one, providing modular construction of signatures. Type classes [36] provide ad hoc polymorphism, making semantics extendable and provides a common denominator for the evaluation of other language components.

By using type classes, the approach decouples semantics from the signature and the various components from each other, providing **Independent extensibility**. Since a type class provides ad hoc polymorphism, component evaluations can write their evaluation using the type class methods, which makes the component open to be combined with any other component of the type class. This combination of components requires no shared knowledge between the components, because there is a common denominator for the evaluation: being a member of the type class. In addition, type classes also promote **Adaptable semantics**, because the instance definition of a type class is not required to be in the same module as the type for which the instance is defined. This can be used to define multiple instances for the same data type in different modules, such that the desirable semantics can be included by including the corresponding module.

To allow recursive signatures, the monadic interpreter is defined as a function mapping terms of type *Term* to monadic computations. A Haskell type signature of such an interpreter is displayed in Listing 1. The concrete *Term* type can be used by signatures to tie the recursive knot. This is illustrated in Listing 2, which defines an addition term block that uses the *Term* type for its add expressions. As a result, the add expressions can contain any type part of the *Term* type, including itself. However, when the *Term* type is extended, the components defined using the concrete type require recompilation, preventing **Separate compilation**.

Listing 1: Haskell type signature of a monadic interpreter.

```
interpreter :: Term -> InterpM Value
```

Listing 2: Modular building block defining the signature of an integer arithmetic language [21]

```
data TermAdd = Num Int
             | IntAdd Term Term
```

The constraint of the concrete *Term* type was identified by [8], which proposes an alternative approach for extensible union types by modelling these to allow processing by catamorphisms, achieving **Separate compilation**. Processing by catamorphisms can be done by defining the signatures as functors, as displayed in Listing 3. With the approach, the signature is defined in terms of a

type parameter, preventing the dependency on the concrete *Term* type, which can now be passed as the parameter. The approach retains the other techniques described by [21], and therefore the evaluation of the other criteria remain unchanged.

Listing 3: Example of a signature defined as a functor [8]

```
data Arith x
    = Num Int
    | Add x x
```

4.1.2 Algebraic effects. Algebraic effects [26, 27] in combination with effect handlers [28, 29] offer an alternative to monad transformers for modular computations. With algebraic effect handlers, there is a clear distinction between the signature and the semantics — the effects make up the signature and the handlers give the semantics. Because of this decoupling, different semantics can be given to the same effect by writing multiple handlers. Furthermore, effects are easier to compose than monads, preventing the need for something similar as monad transformers in the context of effects.

Algebraic effects with handlers is used by [30] in the construction of modular interpreters. Swierstra uses a similar technique as [8] to handle extension of signatures and semantics, but uses effects and handlers to support effectful computations instead of monad transformers. To achieve this, the approach defines effects as free monads, which keep their underlying functor co-product, making it possible to combine effects via their co-product. This gives a finer granularity for specifying which effects occur in an interpreter compared to monad transformers. Listing 4 displays the usage of the *Teletype* and *FileSystem* effect to specify in more detail what kind of IO occurs in the computation.

Listing 4: Example of extracting the IO monad into separate effects. The example shows a computation using the teletype and filesystem effect [30].

```
cat :: FilePath -> Term (Teletype :+: FileSystem) ()
cat fp = do
    contents <- readFile fp
    mapM putChar contents
    return ()
```

To handle effects, the approach uses type classes to implement handlers, giving every effect unique operations. Furthermore, combining effects is commutative with the approach in contrast to monad stacking with monad transformers, making **Static semantics guarantees** fully possible. Listing 4.1.2 shows how the *Teletype* effect is defined and how the handler for the effect is implemented.

```
class Functor f => Exec f where
    execAlgebra :: f (IO a) -> IO a
instance Exec Teletype where
    execAlgebra (GetChar f ) = getChar >>= f
    execAlgebra (PutChar c io) = putChar c >> io
data Teletype a =
    GetChar (Char -> a)
    | PutChar Char a
```

However, the type class operations to run effects are explicit in the effect dependencies. In the *execAlgebra* case, this is the IO component. With an interpreter this might be semantic entities as displayed in Listing 5, which shows the usage of a store semantic entity operated via effects and handlers. Because the run function is explicit, adding a new semantic entity requires modification of this run function and therefore modification of all the type class instances, making the approach only applicable as a Closed system.

Listing 5: Example of an algebra being concrete with respect to semantic entities. Based on an example from [30].

```
newtype Mem = Mem (Map String Expr)
class Functor f => Run f where
    runAlgebra :: f (Mem -> (a,Mem))
        -> (Mem -> (a,Mem))
```

4.1.3 Polymorphic variants. Polymorphic variants [11] offer a similar type of functionality as extensible unions. They decouple the type from the constructor, making data type sharing, constructors overloading, and implicit typing possible. Listing 6 gives an example of using polymorphic variants — prefixed by the "' symbol — to define a family of *expr* types consisting of various variants independently defined. In the example, the *'Add* variant takes two arguments of type *'expr*. Because both the *'Add* and *'Num* variant are defined as type *'expr*, they can be used as arguments: *'Add('Num 2, 'Num 3)*. Extending the syntax is thus achieved by defining a new polymorphic variant of type *'expr*.

Listing 6: Example of defining a type family *'expr* where types consist out of polymorphic variants, like *'Num*. Based on an example by [12].

```
type 'expr num = [ `Num of int ]
type 'expr add = [ `Add of 'expr * 'expr ]
```

Usage of polymorphic variants in constructing modular interpreters is made concrete by [12]. The approach combines polymorphic variants types with an open recursive style to achieve **Extensibility in both dimensions**. An open recursive style defines evaluation functions with an extra parameter: the evaluation function to use for recursive evaluations. Listing 7 shows the usage of an open recursive style to define isolated evaluation functions for the *num* and *add* expression types.

Listing 7: Example of defining evaluation functions for *'expr* type family using an open recursive style.

```
let eval_num = function `Num (n : int) -> `Num n
let map_add eval =
    function `Add(e1, e2) -> `Add(eval e1, eval e2)
let eval_add eval_rec expr =
    match map_add eval_rec expr with
        | `Add(`Num n1, `Num n2) -> `Num (n1 + n2)
let rec eval_add_num = function
  | #num as n -> eval_num n
  | #add as e -> eval_add eval_add_num e
```

The evaluation function for add takes as its first argument the recursive evaluation function to use for the further evaluation of add expressions. Via this approach, earlier defined evaluation functions can be re-used with newly defined signatures without any duplication or modification of earlier implementations. The example shows this by defining an evaluation function that combines the evaluation of the *num* and *add* expression.

An open recursive style gives flexibility in the evaluation function used, allowing **Adaptable semantics**. However, the open recursive style requires a new evaluation function when combining components, resulting in boilerplate which affects the **Independent extensibility** criteria. For example, if the earlier example language is extended with a new component, a new evaluation function must be written in a similar style as the *eval_add_num* function that incorporates the new component. Nonetheless, with an open recursive style, the effect of an extension on the semantics is under full control of the developer, achieving that part of **Static semantics guarantees**.

The approach does not guarantee that a function is defined for every variant, preventing the defined semantics requirement of the **Static semantics guarantees** criteria. Because this guarantee is not made, it is possible to define new variants anywhere without affecting existing functions, achieving **Separate compilation**.

To handle effectful computations, the approach performs explicit context propagation, i.e. the context is an explicit parameter in the evaluation function. Hence, the full effects of a system must be known when constructing an evaluation function. Otherwise, evaluation functions of extensions are incompatible, preventing propagation of effect because of the open recursive style. Due to these requirements, the approach is only applicable in a Closed system.

4.1.4 Open data types and open functions. With polymorphic variants, only the data types were open for extensions. Open data types and open functions [22] make both the data types and function extensible, achieving **Extensibility in both dimensions** without an open recursive style. Data types can be defined as open by prefixing with the *open* keyword. Constructors can then be defined for this open datatype separate from the datatype definition, as illustrated in Listing 8.

Listing 8: Example of an open data type definition and a constructor for the open data type [22].

```
open data Expr :: *
Num :: Int -> Expr
```

Since open data types can be extended by defining a new constructor, functions over these data types must handle the extension. To achieve this, the paper introduces open functions. Open functions are extended by defining a new case for the function, as illustrated in Listing 9. Because new variants and cases can be added from anywhere by only introducing a constructor or function case, **Independent extensibility** is achieved.

Listing 9: Example of an open function definition and a case defining the implementation for the *Num* open data type [22].

```
open eval :: Expr -> Int
eval (Num n) = n
```

To prevent duplicate definitions of variants in a data type or cases in a function, the approach requires that the open definitions can be transformed into closed definitions. This transformation to a closed form is also how the approach implements the open definitions. Effectively, this requires a full recompilation when an open definition is extended, prohibiting **Separate compilation**. Nonetheless, the compilation approach allows the comparison of definitions, making **Static semantics guarantees** possible by applying a different matching strategy than first match. With first match, a catch-all clause closes the function for new cases. Instead, the paper introduces best-fit left-to-right matching which takes the best fit and ensures that the extension order does not influence semantics.

However, **Adaptable semantics** is not possible because the approach provides no way of overwriting an existing function definition, and defining multiple definitions results in a compile error. In addition, the paper does not detail the usage of the semantic entities. Therefore, the approach is only suitable in a Closed system, because an open function can always be defined with pre-known semantic entities.

4.1.5 Tagless. Instead of using data types to encode the initial algebra, finally tagless [2] uses type classes to encode signatures which allows an algebra to be defined via an instance definition. The type class consists out of ordinary function which allows function composition to compose across languages. An encoding of a simple integer expression signature is given in Listing 10.

Listing 10: Example of a signature description using the tagless approach. Based on the example given in [2].

```
class ExpSYM repr where
    lit :: Int -> repr
    add :: repr -> repr -> repr
```

The *repr* argument for the type class allows an implementation to specify the carrier of the algebra. Listing 11 gives an example of specifying the *repr* as type *Int*, describing an algebra with the carrier being the *Int* type, assigning integer addition semantics to the language.

Listing 11: Example of an implementation of the earlier given signature description. Based on the example given in [2].

```
instance ExpSYM Int where
    lit n = n
    add = (+)
```

To extend the signature, a new class can be written that defines a new signature. Because a signature class contains simple functions, as long as the used carrier is identical, the functions from different classes can be composed. To illustrate, Listing 12 displays an extension on the expression language by adding negation.

Listing 12: Example of a signature description using the tagless approach. Based on the example given in [2].

```
class NegSYM repr where
    neg :: repr -> repr
```

When the repr is equal, the *neg* function can be used on the expressions constructed by the *lit* and *add* functions, achieving **Independent extensibility**. Furthermore, extension of semantics is achieved by implementing a new instance with a new carrier, achieving **Extensibility in both dimensions** and **Separate compilation**. By using type classes, as seen before, **Adaptable semantics** is possible because the instance implementations can be defined in different modules.

A language in the tagless approach consists out of a class and its instance implementations. Loading multiple languages, which by function composition allows language composition, does not influence the semantics of languages since only one instance specification can exist in a module at a time. In addition, the Haskell type system ensures the semantic guarantees of the language because it is not possible to create an expression that has no semantics, achieving **Static semantics guarantees**. However, the approach is explicit in the context required for evaluation, making it only applicable in a Closed system.

4.2 Object Oriented

4.2.1 Object algebras. Object algebras [4] provide a separation of algebraic signatures and algebras over those signatures in object oriented languages. A signature is defined via an object algebra interface, implemented via an object oriented interface; and an object algebra is a class that implements the signature interface making the interface concrete. An example of such a signature interface is given in Listing 13, and an object algebra implementing the signature is given in Listing 14.

Listing 13: An example of an object algebra interface defining a signature for a simple arithmetic language.

```
interface IntAlg<A> {
    A lit(int x);
    A add(A e1, A e2);
}
```

The separation of signatures and algebras in object algebras provide a natural way to construct modular interpreters [4], because **Extensibility in both dimensions** can be achieved via object inheritance. A signature can be extended by extending the object algebra interface with a new interface containing the new syntactic constructs. An algebra for the new signature can be created by extending the algebra for the original signature with evaluation methods for the new syntactic constructs. Since both methods use object inheritance, the approach achieves **Adaptable semantics**, but also makes **Independent extensibility** impossible because extension is linear, meaning that extensions always form a stack and when combining two components, one of the two must extend the other. When these components are independently developed, modification of one of the two components is required, because one component must perform the extension.

Nonetheless, because the approach relies on simple inheritance, both **Separate compilation** and **Static semantics guarantees** are achieved.

Listing 14: An example of an object algebra implementing the interface. *Exp* is a class that the *Lit* and the *Add* class both inherit.

```
class IntFactory implements IntAlg<Exp> {
    public Exp lit(int x) {
        return new Lit(x);
    }
    public Exp add(Exp e1, Exp e2) {
        return new Add(e1, e2);}
    }
}
```

Instead of object inheritance, [16] uses trait inheritance to construct modular interpreters with object algebras. With traits, multiple inheritance is possible, opening up an implementation that supports **Independent extensibility**.

However, the combination of two independent components might not be immediately possible because of required context. For example, one component requires state and the other does not. To overcome this, the paper introduces implicit context propagation for object algebras. Implicit context propagation is achieved by lifting the interpreter into the context of another interpreter. An example of such a lifting process is displayed in Listing 15, which shows how *EvArith* algebra can be lifted into the *EvEArith* algebra. As a result of this lifting process, the *EvArith* algebra can be combined with algebras requiring environments.

Listing 15: Lifting of an algebra not supporting environments to an algebra supporting environments.

```
trait EvEArith extends Arith[EvE] {
    private val base = new EvArith {}
    def add(l: EvE, r: EvE): EvE = env =>
     base.add(() => l(env), () => r(env))()
    def lit(n: Int): EvE = env =>
     base.lit(n)()
}
```

A lifting operation requires some boilerplate, but the lifted interpreter is defined in terms of the interpreter being lifted, and therefore, there is a minimum amount of duplication. To reduce the amount of boilerplate and prevent duplication, the paper uses code annotations to automate the lifting process. Listing 16 shows the same lifting process but this time performed using code annotations. Thus, independent components can be combined, but a lifting annotation might be required to make the context parameters identical, affecting **Independent extensibility**.

Listing 16: An example of using annotations to automate the lifting process.

```
@lift[Arith[_], ()=> Val, EvArith, (Env) => Val]
trait EvEArith
```

4.2.2 Modules. Modules provide an interface for importing and exporting functionality, creating a modular system of re-usability and control, because a module can control what is used by others and can include functionality exported by others.

The concept of modules is used by [5] to construct modular language components in Javascript. Javascript is used because objects allow assignments of new functions and overriding existing functions, in contrast to many statically typed object oriented languages. This gives Javascript objects **Extensibility in both dimensions** by default, of course, this is heavily reliant on Javascript being a dynamically typed language with support for duck-typing[4]. This is illustrated in Listing 17, which defines two components both with an *eval* function.

[4]A style of run-time polymorphism where functions look at methods of an object instead of its type to determine if an operation is valid.

Listing 17: Example of defining two components of a simple expression language in Javascript. Based on the examples given by [5]

```
var num = {
    new(n) { return { __proto__: this, n}},
    eval() { return this.n }
}

var plus = {
    new(l, r) { return {__proto__:this, l, r} },
    eval() { return this.l.eval() + this.r.eval() }}
}
```

By using duck-typing, these two components can be combined freely, achieving **Independent extensibility**. Of course, this works because of duck-typing and no compilation step. Since there is no compilation, I opted to leave out the evaluation of the **Separate compilation** for this approach, such that there is a distinction between approaches providing compilation but not achieving separate compilation and approaches not providing a compilation step.

The dynamic nature has another consequence: **Static semantics guarantees** can not be made, because there is no compilation step. As a result, components can be freely combined that might not have the same evaluation functions defined, resulting in a run-time type error form the Javascript engine.

Since Javascript object functions can be freely overridden, **Adaptable semantics** is naturally possible, however, overriding a function does modify the actual object which might introduce semantic conflicts. To overcome this, the paper models extensions as modules — a function that accepts imports and returns exports. The import is the type to be extended, preventing changes to the original object. This is illustrated in Listing 18, which defines a 'module' changing the semantics of the *num* node. The *base* argument describes the import of the modules and requires the num node to adapt. The adapted num node is returned by the function, modelling an export. Via this technique, **Adaptable semantics** is achieved without modification of the original object.

Listing 18: Example of a 'module', changing the semantics of the *num* node. Based on an example from [5]

```
var newEval = function(base) {
    var num = {__proto__: base.num,
        eval() { return this.n + 1 }
    }

    return {num}
}
```

4.2.3 Mixins. A mixin [1] is an abstract subclass and can be used to specify the behaviour of several parent classes. Mixins are often 'included' and not inherited, and a prime characteristics of mixins is that they are defined in terms of parent methods while not actually having a parent. These parent methods become concrete when a mixin is included by another class. Overall, mixins provide functionality to abstract away boilerplate shared by many classes.

The concept of mixins is used by [7] to implement mixin modules in the standard ML language. A mixin module contains an extensible *body* section. A body sections consists out of data types and functions. All data types and functions in a body section are open for extension, which is achieved by merging mixins, making **Extensibility in both dimensions** possible. On such a merge, definitions with the same name are merged into one. An example of two mixin modules is given in Listing 19.

Listing 19: Example of two mixin modules defining part of a language with integers and lambda calculus [7]. For brevity, the implementation of some evaluation functions are omitted, which is denoted by ellipsis.

```
structure Num =
mixin body
    datatype term = CONST of int
    datatype value = NUM of int
    type env = string -> value
    fun eval (CONST i) _ = NUM i
      | eval tm (e:env) = inner tm e
end

structure Func =
mixin body
    datatype term = VAR of string
        | ABS of string * term
        | APP of term * term
    datatype value = CLOS of term * env
    withtype env = string -> value
    fun eval (VAR x) (env:env) = env x
      | eval (f as ABS _) e = CLOS (f, e)
      | eval (APP (rator, rand)) e = ...
      | eval tm e = inner tm e
end
```

The example defines a *Num* mixin and a *Func* mixin. Both mixins define a *term* and *value* data type, and an *eval* function. When these two mixins are merged, the *term*, *value*, and *eval* definitions are merged as well, which results in a language with both numbers and functions. However, the *eval* function defines context parameters explicitly, in this case an environment, making the approach only applicable in a Closed system.

One crucial thing to note in the example is the usage of the *inner* function in the *eval* function of both mixins. This *inner* function is the earlier mentioned prime characteristic of a mixin, and determines when control is passed to the next mixin. Via the *inner* function, mixins can interoperate with other mixins without predefined knowledge or duplicate specification, achieving **Independent extensibility**. However, because of the *inner* function, combining mixins is not commutative, preventing that part of **Static semantics guarantees**.

The approach does not allow mixins to share constructors. Therefore, on merge, the merged mixins must be checked to not violate this requirement, preventing **Separate compilation** but achieving the semantic requirement of **Static semantics guarantees**. Nonetheless, the approach allows shadowing of functions as long as the types do not change, achieving **Adaptable semantics**.

4.2.4 Algebraic data types with defaults. Algebraic data types with defaults [39] models data types using classes which allows using inheritance to extend data types. To make the extensions usable,

the approach defines a default rule for data types. When such a data type is extended, the variants of the extending object are subsumed by the default rule. As a result, the variants also become part of the extended object. Listing 20 illustrates extending using a default rule. In the example, the *ExtTerm* variants also become part of the *Term* variants, which makes usage of *ExtTerm* variants as arguments where *Term* variants are expected possible. For example, by giving a variable as an argument to the plus constructor.

Listing 20: Example of definition of two data types using data types with defaults, and an evaluator and extended evaluator for evaluating the terms. Implementations are left out for brevity. Based on the example given by [39].

```
class Term {
    case Number(int v);
    case Plus(Term l, Term r);
}
class ExtTerm extends Term {
    case Variable(String name);
    case Apply(ExtTerm fn, ExtTerm arg);
    case Lambda(Stringname, ExtTerm body);
}
class Evaluator {
    Term eval(Term term, Env env) {
        switch(term) {
            case Number(int v): ...
            case Plus(Term l, Term r): ...
            default: return term;
        }
    }
}
class ExtendedEvaluator extends Evaluator {
    Term eval(Term term, Env env) {
        switch(term) {
            case Variable(String name): ...
            case Apply(ExtTerm fn, ExtTerm arg): ...
            ...
            default: return super.eval(term, env);
        }
    }
}
```

Evaluation happens by writing a class with functions operating on the data types, as illustrated in Listing 20 with the *Evaluator* classes. Because of the new typing relation, an evaluator can be extended to add the semantic rules for newly added variants in terms of the extended object, achieving **Extensibility in both dimensions**. Defining in terms of the extended object is clearly visible in Listing 20 by looking at the eval method of the extended evaluator. The eval method takes as argument a term of type *Term*. Because of the default rule, the evaluator can pattern match on the cases introduced in the *ExtTerm* class and forward to the base evaluator when there is no match on any of the extended cases.

However, the approach is explicit in the usage of semantic entities, making the approach only applicable in a Closed system.

Since the approach uses inheritance and does not support multiple inheritance, **Adaptable semantics** is possible but **Independent extensibility** is not because a component must be defined as an extension. In addition, extending an existing component requires recompilation because the sub-types of the extended type changes, preventing **Separate compilation**. Furthermore, **Static semantics guarantees** can not be fully guaranteed, because it is possible that an operation is performed on a variant of an extended type for which the operation is not defined.

4.3 Attribute grammars

Attribute grammar [18] formalisms are declarative formalisms for describing the semantics of a programming language. Attribute grammars assign attributes to the symbols in a context free grammar (CFG). A CFG consists out of grammar symbols — the terminals and non-terminals of the grammar — productions, which construct a non-terminal out of other non-terminal and terminal symbols, and a start state.

With an attribute grammar, productions and grammar symbols can be associated with attributes. There are synthesized and inherited attributes. Synthesized attributes denote values from children and inherited attributes denote data from the parent and siblings. For a production, the output occurrences are the synthesized attributes of the left hand side, and the inherited attributes of the right hand side of the production. Every production in the attribute grammar is associated with semantic equations describing how the values of output attribute occurrences are computed in terms of the other attributes in the production.

4.3.1 UUAG with AspectAG. UUAG [33] is a domain specific language for attribute grammar specifications and compiles down to AspectAG [34]. AspectAG provides a combinator library in Haskell to construct attribute grammars. The approach uses heterogeneous lists to store attributes and values, making it possible to compose attribute definitions to create more complex definitions and combine independently developed semantic definitions.

UUAG achieves **Extensibility in both dimensions** by combining semantic functions via the attribute composition provided by AspectAG, and extension of signatures is achieved by extending the defined data type. Using AspectAG also makes **Adaptable semantics** possible because of the type-level heterogeneous lists.

Listing 21 shows a base definition for a simple expression language using the UUAG system. This base definition is extended in Listing 22. The extension extends the expression data type with support for negation, and adds error handling to the root and expression type.

Listing 21: An example expression language with integer addition semantics using UUAG. Based on an example given by [33].

```
DATA Root | Root main:Expr
DATA Expr | Add l:Expr r:Expr
          | Val value:Int
ATTR Root Expr SYN sval:Int
SEM Root | Root lhs.sval = main.sval
SEM Expr | Add lhs.sval = l.sval + r.sval
         | Val lhs.sval = value
```

Listing 22: extension of the above base language by adding support for negating statements and error handling. Semantic definitions for negation and errors for expr have been omitted for brevity.

```
Extends "Base"
DATA Expr | Neg expr:Expr
ATTR Root Decls Expr SYN serr: [String]
SEM Root | Root lhs.serr = main.serr
```

However, an extension requires to be explicitly denoted in a module and a module can only extend one base definition. Therefore, to combine independently developed components, a stack of extensions must be created. However, this requires modification of the components, preventing **Independent extensibility**. Furthermore, to extend an existing data type, UUAG regenerates the corresponding data type definition in the Haskell code, preventing **Separate compilation**.

Nonetheless, by compiling down to AspectAG, UUAG uses the Haskell type system to achieve **Static semantics guarantees**, ensuring that attribute definitions are type safe, adapting existing attribute definitions retains type safety, and ensuring that attribute composition is correct. But, the type system does not check circularity in attribute grammars and UUAG does not perform any checking itself, requiring the user to understand the compilation process to AspectAG to understand possible errors produced by Haskell on the generated code.

4.3.2 Composable attribute grammars. Introduced by [10], composable attribute grammars achieve **Extensibility in both dimensions** by extracting components into composable attribute grammars (CAG). A composable attribute grammar is a specification of a signature and the corresponding semantics, with optional parameters and results. Parameters are parameterised inherited attributes, and the results are parameterised synthesized attributes, making the CAG reusable for different types and use-cases. An example of a CAG implementing a very simple environment lookup component is given in Listing 23. The example defines an environment that maps names to objects. Names can be used to retrieve an object from the environment. The CAG defines the parameter *tp_NTX* which represents the name and is made concrete when using the CAG. Furthermore, it defines the results *OBJ* and *MSGS*, denoting the result of a lookup and possible error messages, respectively.

To use a component, a glue grammar must be written. A glue grammar threads input and output through different components. For example, the environment CAG can be used to perform a simple environment lookup to get the object belonging to a variable for further processing — for example type checking. Type checking might be a different CAG that is completely unrelated to the environment CAG, but can be used together via the glue grammar. So, **Independent extensibility** is possible, however, there is some glue code required to join the separate components together. In addition, **Adaptable semantics** is possible by defining a new component that imitates the original component but with semantics modified.

To handle compilation, the approach provides two methods. One method combines all the CAGs and glue grammar into one monolithic attribute grammar, which prevents **Separate compilation** because the monolithic glue grammar must be fully rebuilt on an addition. The other method is based on separable CAGs — CAGs are separable iff they either provide parameters to other CAGs or receive values from others CAGs, but not both. With separable CAGs the approach achieves **Separate compilation**, but is only applicable in a Closed system because of the separable restriction. Nonetheless, both compilation methods provide **Static semantics guarantees**.

Listing 23: An example of a CAG describing the signature and semantics of a simple variable lookup component. This is a shortened version of the example given by [10]

```
envREF : t NAME:in:tp_NTX, OBJ:out:entry,
    MSGS:out:tp:msgs;
envitems: n ENVin:h:symbol_table,
    ENVout:s:symbol_table;

env_ref : envitems ::= envREF.
    envREF.OBJ =
        envLookup(envitems.ENVin, envREF.name),
    envREF.MSGS = if envREF.obj == nullObj
    then "Invalid reference"
    else "" fi
envitems.ENVout = envitems.ENVin;
```

4.3.3 Modular attribute grammars. A different approach to modularity is taken by [6], who constructs modular components around patterns. These patterns can then be matched on a context free grammar (CFG) to specify a full attribute grammar.

Listing 24: Example of a modular attribute grammar defining the patterns and semantics for a simple arithmetic language. Based on an example given by [6]

```
module val
'digit' -> '0
    'digit.val = 0;
'digit' -> '1
    'digit.val = 1;
binexpr -> Lop op Rop
    binexpr.val = callop(op.operator, Lop.val,
                         Rop.val)
```

Listing 24 gives an example of a modular component in this approach. The example defines the signature and semantics of a value component with support for binary expressions. The patterns described in the module are compared to the patterns in the CFG. When patterns match, the semantics of the patterns in the module are assigned to the CFG. An example of a CFG is given in Listing 25, which defines the CFG on which the value module can match.

Listing 25: Example of a CFG describing the patterns to match on for the modular attribute grammars [6].

```
goal -> expr
expr -> term
expr -> expr addop term
term -> digit
term -> term digit
digit -> 0
digit -> 1
```

```
addop -> '+'
```

The usage of the CFG to match, influences **Independent extensibility** because a module can only be used when a match is present in the CFG. Thus, to use a module, a matching pattern must first be introduced in the CFG, resulting in a boilerplate requirement.

In addition, **Separate compilation** is not possible, because the approach performs a compilation step that transforms all MAGs and the CFG into one attribute grammar. The compilation step cannot, however, guarantee that the resulting attribute grammar is consistent, hence, the approach does not achieve the guarantees criteria of **Static semantics guarantees**.

To handle duplicate patterns in modules, the system chooses the first matching pattern it processed, preventing **Static semantics guarantees** fully. Nonetheless, this makes **Adaptable semantics** possible with a very fine granularity, because a component can be written that only changes the semantics of a very specific pattern, and by including it before the other modules the semantics can be adapted.

4.3.4 Multiple attribute inheritance. Multiple attribute inheritance is a technique to inherit multiple attributes in a new attribute grammar [23], allowing the new grammar to extend, specify or override the attributes it inherits, naturally achieving **Extensibility in both dimensions**. This is illustrated in Listing 26, which shows how an *Expr* language is defined and extended by a new language, the *ExprEnv* language. The *ExprEnv* language extends certain rules — indicated by the 'extends' keyword — overwrites others — indicated by the 'overwrites' keyword — and adds new rules to the *Expr* language. Overwriting fully overwrites existing rules for a production and makes **Adaptable semantics** possible.

The example only shows extending one language, but the approach allows multiple inheritance, thus it is possible to extend multiple languages at once, achieving **Independent extensibility**. However, the order of languages being extended determines how the rules are merged, an earlier extension takes precedence over later extensions, preventing extension order of **Static semantics guarantees**. In addition, the attribute grammars are combined into one, which prevents **Separate compilation**, because a new component might overwrite existing components, thus requiring a full recompilation when a new module is added. Nonetheless, by using a monolithic approach, the semantic guarantees for **Separate compilation** are achieved.

Listing 26: Example of an attribute grammar for a simple expression language that is extended via inheritance to support environments. The 'overwrite' keyword overwrites existing equations with equation defined. The 'extends' keyword extends existing equations. Based on examples given by [23]. For brevity, implementations have been omitted, which is indicated with ellipsis.

```
language Expr {
    ...
    attributes int *.val;
    rule Expression1 {
        EXPR ::= EXPR + TERM compute {
            EXPR[0].val = EXPR[1].val + TERM.val;
        };
    }
    rule Expression2 {
        EXPR ::= EXPR + TERM compute {
            ...
        };
    }
    rule Term1 {
        TERM ::= #Number compute {
            ...
        };
    }
}

language ExprEnv extends Expr {
    attributes Hashtable *.inEnv, *.outEnv;
    ...
    rule extends Expression1 {
        EXPR ::= EXPR + TERM compute {
            valueDistribution<EXPR[0].inEnv,
                [TERM.inEnv, EXPR[1].inEnv]>
        };
    }

    rule overwrites Expression2 {
        EXPR ::= EXPR + TERM compute {
            ...
        }
    }

    rule Term2 {
        TERM ::= #Identifier compute {
            TERM.val = TERM.inEnv.get(
                #Identifier.value());
        };
    }
}
```

5 DISCUSSION

In this section I discuss the evaluation presented in section 4. This discussion starts by discussing the approaches within their respective category, and is followed up by a discussion on the approaches outside their category. Furthermore, I discuss a common trade-off present in the approaches, the possible threats to the study, and some approaches not selected for the evaluation.

5.1 Categorised evaluation

5.1.1 Functional. In the functional category, the monad approach with catamorphism [8] is the most flexible since it fulfils most of the evaluation criteria fully, and only the **Static semantics guarantees** partially. There is no functional approach that ensures **Static semantics guarantees** in an Open system. For Closed systems, the data types á la carte [30] approach and the tagless [2] approach are most flexible, since they achieve all criteria.

5.1.2 Object oriented. Among approaches using an object oriented style or language, object algebras with implicit context propagation [16] is the most flexible. The approach fulfils all criteria fully, except the **Independent extensibility** criteria which it fulfils only partially. If that criteria is a requirement, the Javascript module [5] approach is the best fit, since it is the only Open object oriented approach fully achieving **Independent extensibility**. For object oriented approaches, there is no advantage to using a Closed approach because both approaches only applicable in a Closed system evaluate worse than the object algebra approach. As such, the object algebra with implicit object propagation can be chosen, which, in a Closed system, does not require the implicit propagation, positively affecting the **Independent extensibility** of the approach.

5.1.3 Attribute grammars. For attribute grammar approaches, composable attribute grammars [10] are overall the best fit for both Open and Closed systems. As a Closed system, the approach meets all criteria except the **Independent extensibility** which it partially meets. If that criteria is a requirement, the multiple attribute inheritance [23] approach is the only option. Furthermore, composable attribute grammars in the Closed approach is the only approach achieving **Separate compilation**.

5.2 Full evaluation

For approaches supporting an Open system, both the monad transformers with catamorphisms and object algebras with implicit context propagation evaluate well. Both fulfil all criteria except one, which they partially fulfil. For the monad transformers approach, this is the **Static semantics guarantees** criteria, and for the object algebra approach it is the **Independent extensibility** criteria. As such, the best approach from our evaluation depends on the importance of the **Independent extensibility** and **Static semantics guarantees** criteria.

If the semantic entities are known a priori, a Closed system with the data types á la carte approach or the tagless approach is a good choice, since they fulfil all the criteria. In addition, the object algebras with implicit context propagation can be adapted for a Closed system, removing the need for the boilerplate when combining independent elements, achieving the same evaluation as the data types á la carte and tagless approaches.

5.3 Features and complexity

Among the approaches there is a constant choice to be made for features. Most approaches achieving many of the criteria also use a combination of language features from the implementation language, complicating the approach.

This choice is seen in the case of separate compilation. Some approaches not achieving separate compilation do so because they apply a monolithic compilation process, where all code is taken together to generate a solution. Other approaches that do have separate compilation need to implement some kind of interface system since certain information is required to allow separate compilation, for example, typing information, already defined data types, and so on. By choosing separate compilation, the language the approach is written in needs to have more features. Furthermore, these interfaces often contain limited information, making performance optimisations harder and sometimes impossible compared to a monolithic approach.

The choice regarding features of the underlying language and criteria met by an approach is also visible when looking at adaptable semantics in combination with independent extensibility. With independent extensibility, adaptable semantics comes almost for free by making the extensibility not commutative, which prevents static semantic guarantees. If an approach also wants static semantic guarantees, it must make composition commutative and achieve adaptable semantics in another way. This might be via inheritance as seen with the object algebras approach. Nonetheless, it requires another feature of the language in which the approach is implemented, increasing the complexity of the approach.

Thus, there is a trade-off to be made between choosing the number of criteria an approach meets and the overall complexity of the approach.

5.4 Threats to validity

There are three threats to validity that need to be discussed for this study: effectiveness of the search process (internal), the selection and classification process (internal), and the correctness and completeness of the considered studies (external).

To ensure the search process is effective, I used multiple search engines with the snowballing approach, since not all sources are available in one search engine. Furthermore, I applied techniques presented in other systematic search studies [31, 40]. In addition, by using a snowballing approach I was able to cast a wide net of papers while starting with rudimentary knowledge of the field being studied, minimising ineffectiveness of the search compared to an approach using specific search terms which require deep knowledge of the field being studied.

The inclusion criteria were used in two stages. The first criteria and the last four were used to determine if a paper focused on the construction of modular interpreters while having a certain level of quality and available information for a fair evaluation. With these criteria, the abstract, introduction and conclusion were evaluated. The other criteria were used to select papers that provide enough information for an equal validation by using interpreters constructed by domain experts due to time constraints. These criteria were used when evaluating the full paper.

The external threat occurs because a paper might provide incorrect information about a system. By requiring a built interpreter with an approach and requiring a paper to be published, this threat is minimised by utilising the peer review system and validation possibilities of the interpreter. However, it must be noted that due to time constraints I was unable to use the approaches to construct an interpreter.

5.5 Excluded papers

Not every paper focused on (modular) construction of interpreters was selected for evaluation. This section contains some of those papers with argumentation why they were excluded from the evaluation. It is not possible to include every paper not selected. Nonetheless, this section gives insight in the exclusion of papers and the

argumentation given for these papers can be extrapolated to the other papers not selected.

Staging interpreters in a host languages provides performance benefits to the interpreter by specialising the interpreter to a specific program, eliminating unnecessary evaluations. [38] combines staging interpreters with an abstract interpreter which provides static analysis, allowing both static checking and increase in performance for language implementations. However, the approach is not focused on the modular construction of interpreters and has hard coded case patterns in functions, preventing extensibility as described in the LEP.

Using catamorphisms in combination with monad transformers to construct modular interpreters is also described by [13]. In the abstract, the papers outlines an approach rather similar to the approach presented by [8]. However, during the time of the search, I was only able to obtain the extended abstract[5] from the paper, making it unfit for evaluation.

Another approach worth mentioning is component based semantics (CBS) [32]. CBS defines language semantics in terms of funcons. A funcon describes the signature and semantics of a fundamental programming language construct, for example, an if-else construct. A funcon definition is fixed, but new funcons can be freely added. Translating a language to funcons corresponds to a semantic function and funcon definitions are solely related to the semantic domain. The paper also provides a system for writing translation rules that translate abstract syntax into funcons. However, the paper only shows translating to funcons. Furthermore, it is unclear if the system allows extension of the language syntax, constituting the exclusion from the evaluation.

Finally, Silver is a full attribute grammar system with a focus on the extension of a base language. However, Silver only allows extension by defining extension in terms of components of the base language, which is a really restrictive form of extension. Furthermore, from the current literature it is unclear how Silver performs the compilation process. Due to these two reasons, I decided to omit Silver from the evaluation.

6 RELATED WORK

Modularity in interpreters and language design is a broad topic that has different perspectives in what is of critical importance. Also, there are different levels of abstraction and different requirements. For example, in our survey we do not focus on modular construction of concrete syntax, but purely on abstract syntax and its semantics. In this section I discuss surveys that are also focused on modularity in language construction but evaluate different criteria.

Performance of an approach and tool support for interpreter construction are examples of alternative criteria, which are present in the survey by [19] on modularity in language engineering. The survey is more broad than ours since it also evaluates approaches that combine several techniques into one, like language workbenches, but also evaluate approaches that only showcase one technique, like object algebras. In addition, the survey looks at tool support around an approach, like integration with an integrated development environment (IDE). However, the survey does not evaluate **Independent extensibility** nor does it make a distinction between Open and Closed systems.

Language workbenches are also evaluated by [9]. Compared to [19], this survey only focuses on language workbenches and performs the evaluation by implementing the same language in every workbench. Both surveys evaluate tool support and performance. Compared to our survey, this survey does not evaluate approaches in isolation, but evaluates a whole system with multiple components. As such, the survey includes evaluation of testing, validation, and editing support.

Several of the approaches in our survey are described using a specific semantic specification technique. For example, object algebras with implicit context propagation is specified using denotational semantics. Modularity of these specification techniques is not evaluated in our survey, but is evaluated by [14]. In the survey, the authors evaluate different semantic specification techniques on several criteria, including readability, flexibility, and modularity. Compared to our survey, this survey is more focused on specification techniques instead of implementation techniques.

7 CONCLUSION

This survey was structured around the evaluation of approaches for constructing modular interpreters. To perform this evaluation, a modified version of the Language Extension Problem (LEP), a framework for evaluating approaches for modular language construction, was used. The survey was performed via a systematic search using a snowballing approach up to four levels, which resulted in 16 papers fulfilling our evaluation criteria, divided into functional approaches, object oriented approaches and attribute grammar approaches.

Out of those papers, object algebras with implicit context propagation (Inostroza and van der Storm) and monad transformers combined with extensible unions based on catamorphisms (Duponcheel), are most suitable when the scope of the language being constructed is unknown, because they fulfil almost all our evaluation criteria. If the scope is known, the data types à la carte approach (Swierstra), tagless (Carette et al.), or an adapted version of object algebras with implicit context propagation (Inostroza and van der Storm), come out at top, fulfilling all evaluation criteria.

7.1 Future work

As discussed in the discussion (section 5), many approaches fulfilling many criteria are also rather complex, because many language features are included in the approach. This survey does not look further at the complexity of approaches. Therefore, for future work it is interesting to evaluate the approaches on the more practical criteria, like performance, complexity, and tooling. This evaluation can make the trade-offs an approach makes concrete and look at multiple criteria were trade-offs are made.

REFERENCES

[1] Gilad Bracha and William R. Cook. 1990. Mixin-based Inheritance. In *Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming (OOPSLA/ECOOP), Ottawa, Canada,*

[5] http://di002.edv.uniovi.es/ labra/FTP/Papers/nwpt.pdf

October 21-25, 1990, Proceedings, Akinori Yonezawa (Ed.). ACM, 303–311. https://doi.org/10.1145/97945.97982

[2] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

[3] Antonio Cavacini. 2015. What is the best database for computer science journal articles? *Scientometrics* 102, 3 (2015), 2059–2071. https://doi.org/10.1007/s11192-014-1506-1

[4] Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses - Practical Extensibility with Object Algebras. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7313)*, James Noble (Ed.). Springer, 2–27. https://doi.org/10.1007/978-3-642-31057-7_2

[5] Florent Marchand de Kerchove, Jacques Noyé, and Mario Südholt. 2015. Towards modular instrumentation of interpreters in JavaScript. In *Companion Proceedings of the 14th International Conference on Modularity, MODULARITY 2015, Fort Collins, CO, USA, March 16 - 19, 2015*, Robert B. France, Sudipto Ghosh, and Gary T. Leavens (Eds.). ACM, 64–69. https://doi.org/10.1145/2735386.2736753

[6] G. D. P. Dueck and Gordon V. Cormack. 1990. Modular Attribute Grammars. *Comput. J.* 33, 2 (1990), 164–172. https://doi.org/10.1093/comjnl/33.2.164

[7] Dominic Duggan and Constantinos Sourelis. 1996. Mixin Modules. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, Robert Harper and Richard L. Wexelblat (Eds.). ACM, 262–273. https://doi.org/10.1145/232627.232654

[8] Luc Duponcheel. 1995. Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters. *Utrecht University* (1995).

[9] Martin Erwig, Richard F Paige, and Eric Van Wyk. 2013. The state of the art in language workbenches-conclusions from the language workbench challenge. In *Software Language Engineering-6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, Vol. 8225. Springer, 197–217.

[10] R. Farrow, T. Marlowe, and D. Yellin. 1992. Composable attribute grammars: support for modularity in translator design and implementation. In *POPL '92*.

[11] Jacques Garrigue. 1998. Programming with polymorphic variants. In *ML Workshop*, Vol. 13. Baltimore, 7.

[12] Jacques Garrigue. 2000. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Vol. 13. Citeseer.

[13] José Emilio Labra Gayo, Juan Manuel Cueva Lovelle, María Cándida Luengo Díez, and Agustín Cernuda del Río. 2001. Modular Development of Interpreters from Semantic Building Blocks. *Nord. J. Comput.* 8, 3 (2001), 391–407. http://www.cs.helsinki.fi/njc/References/labragayoclc2001:391.html

[14] JE Labra Gayo. 2002. Reusable semantic specifications of programming languages. In *6th Brazilian Symposium on Programming Languages*.

[15] Joseph A. Goguen and James W. Thatcher. 1974. Initial Algebra Semantics. In *15th Annual Symposium on Switching and Automata Theory, New Orleans, Louisiana, USA, October 14-16, 1974*. IEEE Computer Society, 63–77. https://doi.org/10.1109/SWAT.1974.13

[16] Pablo Inostroza and Tijs van der Storm. 2015. Modular interpreters for the masses: implicit context propagation using object algebras. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2015, Pittsburgh, PA, USA, October 26-27, 2015*, Christian Kästner and Aniruddha S. Gokhale (Eds.). ACM, 171–180. https://doi.org/10.1145/2814204.2814209

[17] Mark P Jones and Luc Duponcheel. 1993. *Composing monads*. Technical Report. Technical Report YALEU/DCS/RR-1004, Department of Computer Science. Yale ....

[18] Donald E. Knuth. 1990. The Genesis of Attribute Grammars. In *Attribute Grammars and their Applications, International Conference WAGA, Paris, France, September 19-21, 1990, Proceedings (Lecture Notes in Computer Science, Vol. 461)*, Pierre Deransart and Martin Jourdan (Eds.). Springer, 1–12. https://doi.org/10.1007/3-540-53101-7_1

[19] Manuel Leduc. 2019. *On modularity and performance of External Domain-Specific Language implementations. (Modularité et performance des implémentations de langages dédiés externes).* Ph.D. Dissertation. University of Rennes 1, France. https://tel.archives-ouvertes.fr/tel-02972666

[20] Manuel Leduc, Thomas Degueule, Eric Van Wyk, and Benoît Combemale. 2020. The Software Language Extension Problem. *Softw. Syst. Model.* 19, 2 (2020), 263–267. https://doi.org/10.1007/s10270-019-00772-7

[21] Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 333–343.

[22] Andres Löh and Ralf Hinze. 2006. Open data types and open functions. In *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, Annalisa Bossi and Michael J. Maher (Eds.). ACM, 133–144. https://doi.org/10.1145/1140335.1140352

[23] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. 2000. Multiple Attribute Grammar Inheritance. *Informatica (Slovenia)* 24, 3 (2000).

[24] Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

[25] Robert Pickering. 2009. Language-oriented programming. In *Beginning F#*. Springer, 327–349.

[26] Gordon D. Plotkin and A. John Power. 2004. Computational Effects and Operations: An Overview. *Electron. Notes Theor. Comput. Sci.* 73 (2004), 149–163. https://doi.org/10.1016/j.entcs.2004.08.008

[27] Gordon D. Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2303)*, Mogens Nielsen and Uffe Engberg (Eds.). Springer, 342–356. https://doi.org/10.1007/3-540-45931-6_24

[28] Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

[29] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Log. Methods Comput. Sci.* 9, 4 (2013). https://doi.org/10.2168/LMCS-9(4:23)2013

[30] Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

[31] Damian A Tamburri, Patricia Lago, and Hans van Vliet. 2013. Organizational social structures for software engineering. *ACM Computing Surveys (CSUR)* 46, 1 (2013), 1–35.

[32] L. Thomas van Binsbergen, Peter D. Mosses, and Neil Sculthorpe. 2019. Executable component-based semantics. *J. Log. Algebraic Methods Program.* 103 (2019), 184–212. https://doi.org/10.1016/j.jlamp.2018.12.004

[33] Marcos Viera, S. Doaitse Swierstra, and Arie Middelkoop. 2012. UUAG meets AspectAG: how to make attribute grammars first-class. In *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*, Anthony Sloane and Suzana Andova (Eds.). ACM, 6. https://doi.org/10.1145/2427048.2427054

[34] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. 2009. Attribute grammars fly first-class: how to do aspect oriented programming in Haskell. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, Graham Hutton and Andrew P. Tolmach (Eds.). ACM, 245–256. https://doi.org/10.1145/1596550.1596586

[35] Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*. ACM, 61–78. https://doi.org/10.1145/91556.91592

[36] Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 60–76. https://doi.org/10.1145/75277.75283

[37] Philip Walder. [n.d.]. *The expression problem.* Retrieved 18 January, 2021 from http://www.ctan.org/pkg/acmart

[38] Guannan Wei, Yuxuan Chen, and Tiark Rompf. 2019. Staged abstract interpreters: fast and modular whole-program analysis via meta-programming. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 126:1–126:32. https://doi.org/10.1145/3360552

[39] Matthias Zenger and Martin Odersky. 2001. Extensible Algebraic Datatypes with Defaults. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, Benjamin C. Pierce (Ed.). ACM, 241–252. https://doi.org/10.1145/507635.507665

[40] Cheng Zhang and David Budgen. 2011. What do we know about the effectiveness of software design patterns? *IEEE Transactions on Software Engineering* 38, 5 (2011), 1213–1231.