$iCoLa^+$: An Extensible Meta-Language with Support for Exploratory Language Development

Damian Frölich[a], L. Thomas van Binsbergen[a]

[a] *Informatics Institute, University of Amsterdam, Postbus 94216, 1090 GE, Amsterdam, The Netherlands*

Abstract

Programming languages providing high-level abstractions can increase a programmers' productivity and the safety of a program. Language-oriented programming is a paradigm in which domain-specific languages are developed to solve problems within specific domains with (high-level) abstractions relevant to those domains. However, language development involves complex design and engineering processes. These processes can be simplified by reusing (parts of) existing languages and by offering language-parametric tooling.

In this paper we present $iCoLa^+$, an extensible meta-language implemented in Haskell supporting incremental (meta-)programming based on reusable components. We demonstrate $iCoLa^+$ through the construction of the *Imp*, *SIMPLE*, and *MiniJava* languages via the composition and restriction of language fragments, demonstrate the variability of our approach through the construction of several languages using a fixed-set of operators, and demonstrate the different forms of extensions possible in $iCoLa^+$.

Keywords: language composition, domain-specific languages, meta-languages, exploratory language development, syntax and semantics

1. Introduction

This paper is an extended version of the paper introducing *iCoLa* as presented at SLE2022 [1].

High-level programming languages increase programmer productivity, program safety, program correctness, and maintainability, among other qualities. Language-Oriented Programming (LOP) [2] is a programming paradigm utilizing the advantages of higher-level programming through the development of new languages specialized to the problem domain at hand via domain-specific abstractions. However, the development of a programming language requires significant engineering efforts, for example to build an interpreter or compiler, to build tooling for the language users, to guarantee performance, etc.

To reduce the engineering effort, a variety of approaches and tools can be utilized. Some examples of such methods and tools are language workbenches and meta-languages [3], techniques for modular and reusable specification of syntax [4] and semantics [5, 6], and component-based approaches to semantics [7].

Besides being a huge engineering effort, creating a programming language is also a *design* process, and navigating design choices is not straightforward. This is evident in the frequent revisions seen in the historical development of general-purpose programming languages as well as in the context of Domain-Specific Languages (DSLs). In the context of DSLs, the design of a language must reflect the concepts known to the domain experts using the language and the design of a language is often updated based on user experience. The design process is thus an iterative process in which language developers and domain experts continue to reflect on the existing design.

Exploration of a programmable design space can be aided by incremental programming. Incremental programming is a style of software development in which a user repeatedly submits small snippets of code on which they receive immediate feedback, constructing a larger system via this feedback-loop. As such, incremental programming delivers early feedback on design decisions in the software development process, enabling rapid prototyping and experimentation. This programming style is supported by Read-Eval-Print Loop (REPL) environments and systems like Jupyter notebooks [8].

In this paper we introduce $iCoLa^+$, an extensible meta-language with a focus on the language design process via exploratory language development and rapid prototyping, achieved by utilizing reusable language components and incremental programming. The work presented in this paper builds upon our earlier work on *iCoLa* [1] by providing an extensible implementation that supports user-defined environment and DSL based domain definitions, as well as extending the approach with concrete syntax, and also supporting arbitrary amount of semantic domains. Specifically, we make the following contributions:

*Corresponding author
Email address: `dfrolich@acm.org` (Damian Frölich)

- We extend the approach of *iCoLa* [1] with concrete syntax and support for an arbitrary number of semantic domains (Section 4).
- We provide an extensible implementation of the extended model in the form of a DSL. The implementation also functions as an alternative to the implementation presented in [1] (Section 5).
- We evaluate the extended approach and compare the new implementation with the *iCoLa* implementation (Section 6).

The remainder of this paper is outlined as follows: in Section 2 we give the required background. In Section 3 we detail the formal model of our approach as presented in [1] and give an overview of the implementation. The formal model is extended with concrete syntax definitions and support for arbitrary semantic domains in Section 4. In Section 5, a DSL for the extended model and its implementation in Haskell is presented. The extended approach is evaluated via an extended exact replication in Section 6. A discussion on the two implementations and the extended model is held in Section 7. Finally, related work is discussed in Section 8 and we end with a conclusion in Section 9.

2. Background

The approach presented in this paper combines insights from earlier works to achieve language composition. The implementation of the approach is based on certain advanced functional programming techniques described in this section.

2.1. Language design, implementation, and evaluation

Language-oriented programming [2] is a paradigm that puts the construction of a programming language at the center of the development process by developing domain-specific languages for the problem at hand. Defining the solution in a domain-specific language, improves productivity and simplifies maintenance. In addition, languages often share similar constructs, promoting reuse with language-oriented programming.

Domain-specific languages can be implemented in a variety of ways [9]. One way is to embed the domain-specific language within an already existing general-purpose language [10], known as an embedded domain-specific language (eDSL). The benefit of such an approach is that no new parser is needed and the tooling of the general-purpose language can be used. An embedding is often either shallow or deep. With a shallow embedding, the operations of the DSL are directly mapped to operations in the general-purpose language. Thus, no abstract-syntax tree (AST) is built. This makes the implementation simpler but also more difficult to add different interpretations. With a deep embedding, the DSL operations build an AST which can then be interpreted in different ways.

Erdweg et al. provide a framework for discussing and comparing meta-languages, tools and formalisms that support various forms of incremental language development [11]. In particular, the authors define the concepts of (modular) language extension, restriction, and unification, which they apply to both the syntax (concrete & abstract), static semantics, operational semantics and IDE services of languages. Extension occurs when a base language is extended by another language that has a dependency on the base language. Restriction is a special form of extension, where a language is restricted, making the new language a subset of the original language. Unification is the process of combining two independent languages with the help of glue code to unify the two languages. The paper also distinguishes between different forms of extension: no extension composition, incremental extension, and extension unification. In case a method does not support extension composition, it is impossible to combine multiple extensions. For incremental extension, extension can be performed in layers where one extension extends the base and another extension extends the extensions, etc. With extension unification, two extensions are unified and the unification is used as the extension on a base language. In this paper we adopt their terminology and use their framework as part of our evaluation.

2.2. Programming language semantics

The initial algebra semantics of Goguen et al. [12], concisely described by Mosses in [13], provides the formal foundation and terminology to our work. Initial algebra semantics captures the essential elements of many existing semantic specification formalisms, such as denotational semantics and attribute grammars. A multi-sorted signature (Σ) lays out the operators of a language in terms of a set of sorts — a set of symbols functioning as an index set, such as $\{\textbf{int}, \textbf{bool}\}$. A Σ-algebra assigns carrier sets to these sorts. When taking term-constructors as the carriers, we obtain the abstract syntax of the language. The algebra formed this way is initial in the class of Σ-algebras. Due to its initiality, there is a unique homomorphism from the initial algebra to any algebra in the class of Σ-algebras — also known as a catamorphism [14]. Algebras give meaning to the operators of a signature by assigning a semantic function to each. Following initiality, any abstract syntax can be mapped to the semantics of an algebra.

The component-based approach to operational semantics presented in [15] is centered around reusable definitions of the *fun*damental *cons*tructs of (general-purpose) programming languages – referred to as *funcons* for short. An example funcon term is **print**(**integer-add**$(1, 2)$), which outputs the result of $1 + 2$ and is constructed using the **print** and **integer-add** funcons. Throughout the text, funcons are indicated with a maroon color, except when used within code snippets. As explained in [7], 'micro-interpreters' can be generated from funcon definitions. The micro-interpreters are compositional evaluation functions expressing the behavior of an individual funcon that can be generated and compiled separately. In this paper, we leverage the generality of the Funcons-Beta library [16] to be able to express the semantics of language constructs in a shared base language. Effectively, the generated micro-interpreters for funcons are applied as the constructs of an embedded DSL.

2.3. Incremental and Exploratory Programming

Exploratory programming [17, 18] is a style of programming in which the goal worked towards is open and by experimenting with code this goal is advanced. This style of development constitutes the creation of different variants for experimentation and also entails discarding written code. Exploratory programming is in a limited form supported by incremental programming environments such as read-eval-print loops (REPLs) and notebooks. Incremental programming supports the submission of small snippets of code to obtain immediate feedback, resulting in a tight feedback loop which is useful during prototyping. However, these environments generally do not have first-class support for the exploration of multiple variants simultaneously nor managing explorations that can be discarded. Previous work [19] provides a principled approach to (defining and developing) REPL interpreters. The approach involves adapting an existing language to a 'sequential' variant that naturally supports incremental programming. Sequential languages are defined as languages in which any two programs can be sequenced together to form a new program, and the interpretation of the sequence is identical to the composition of the interpretation of the two programs in isolation. In this definition there is an assumption on the interpretation function (I), namely it assigns semantics to a program (p) as a function over configurations, i.e. $I(p) : \Gamma \to \Gamma$, for some set of configurations Γ. Configurations represent the context in which a particular program is evaluated. Visually, a language is sequential when the diagram in Figure 1 commutes, where $p_1; p_2$ is the sequence containing the programs p_1 and p_2.

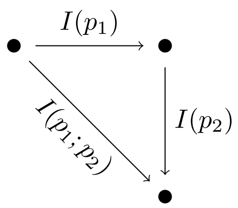

Figure 1: A visual view of the concept of sequential languages. I is the interpretation function over a set of configurations. $p_1; p_2$ is the sequencing of the programs p_1 and p_2.

For sequential languages, tooling for incremental programming such as REPLs, Jupyter Notebooks [8], and even exploratory programming environments [20, 21], can be obtained for free. In this paper, we apply the idea of sequential languages to support incremental programming in our *meta-language iCoLa*$^+$ (i.e. incremental language development) and to obtain REPL interpreters for the *object languages* defined with *iCoLa*$^+$.

3. Summary of the original work

In this section we summarize our earlier work on which this paper is an extension [1]. The insight of incremental language development via composition and the separation between operator (or language construct) definitions on the one hand and language definitions on the other hand, is essential to our approach. A language definition can freely choose from the available operators and constrains the flexibility with which the chosen operators can be used. The definition of an operator consists of an abstract syntax definition and a denotational semantics, choosing funcon terms as a semantic domain. The separation between operator and language definitions is enabled by an alternative take on abstract syntax definitions.

3.1. Abstract syntax

A common approach to define the abstract syntax of a language is to use algebraic datatypes (ADTs), of which the operator[1] signatures determine, in a mutually recursive fashion, the set of terms that forms the abstract syntax of the language. For example, the abstract syntax of a lambda calculus can be represented as follows, where Var_O, Abs_O, and App_O are operators (as indiciated by the subscript) and *String* and *Expr* are sorts.

$$\begin{aligned} Var_{\mathcal{O}} &: String \to Expr \\ Abs_{\mathcal{O}} &: String \times Expr \to Expr \\ App_{\mathcal{O}} &: Expr \times Expr \to Expr \end{aligned}$$

In this style, the signature of an operator simultaneously identifies the sort of terms constructed by applications of the operator, the arity of the operator, and the sort of terms required at each operand position in valid applications of the operator.

A key insight of our approach is to delay the decisions related to sorts (but not the arity) until the definition of a language, rather than making these part of operator definitions. This is achieved by (1) using a unique sort at every position in the signature and by (2) introducing separate *sort constraints* to establish the relations between the sorts. Following (1), the sorts are effectively naming operand positions. The right-hand side of a signature is made redundant and can be removed as every operator already has a unique name. With these changes, the operators are defined as follows:

$$\begin{aligned} Var_{\mathcal{O}} &: VarVar \\ Abs_{\mathcal{O}} &: AbsVar \times AbsBody \\ App_{\mathcal{O}} &: AppAbs \times AppArg \end{aligned}$$

[1] Such as *constructors* in Haskell and *variants* in the ML family of languages.

In contrast to the conventional approach, the signatures do not share any sorts, and the three operators are (as of yet) completely unrelated. To re-establish the relationships, we introduce sort constraints. Sort constraints are based on the interpretation of sorts as sets of operators. For example, the following sort constraints indicates that strings serve as identifiers in both variable references and abstractions:

$$\begin{aligned} \mathit{String} &\subseteq \mathit{VarVar} \\ \mathit{String} &\subseteq \mathit{AbsVar} \end{aligned}$$

This kind of sort constraint is referred to as a *sub-sort declaration*.

The other kind of sort constraint, referred to as an *operator assignment*, indicates that terms constructed by the $\mathit{Var}_{\mathcal{O}}$ operator can be used as the body of an abstraction:

$$\mathit{Var}_{\mathcal{O}} \in \mathit{AbsBody}$$

To express the same relations between the operators as in the initial example, operator assignments can be written for every pair of an operator and sort taken from the sets $\{\mathit{Var}_{\mathcal{O}}, \mathit{Abs}_{\mathcal{O}}, \mathit{App}_{\mathcal{O}}\}$ and $\{\mathit{AbsBody}, \mathit{AppAbs}, \mathit{AppArg}\}$. Writing down these operator assignments grows increasingly tedious (and error-prone) as more and more operators are added to a language. Therefore, as a convenience, sort constraints can also be used to introduce auxiliary sorts that serve as a level of indirection and enable reuse. The following sort constraints utilize the auxiliary sort *Expr*, stating that all operators assigned to *Expr* are also assigned to *AbsBody*, *AppAbs* and *AppArg*:

$$\begin{aligned} \mathit{Expr} &\subseteq \mathit{AbsBody} \\ \mathit{Expr} &\subseteq \mathit{AppAbs} \\ \mathit{Expr} &\subseteq \mathit{AppArg} \end{aligned}$$

The relations of the original example are then expressed by assigning the operators to *Expr*.

$$\begin{aligned} \mathit{Var}_{\mathcal{O}} &\in \mathit{Expr} \\ \mathit{App}_{\mathcal{O}} &\in \mathit{Expr} \\ \mathit{Abs}_{\mathcal{O}} &\in \mathit{Expr} \end{aligned}$$

A language designer can introduce new operators with full flexibility and without modifying existing operator definitions because our approach separates operators from constraints detailing where operators can be used. For example, extending the lambda calculus with integer addition can be achieved by defining an *Add* operator and assigning this operator to the sorts where we want to use the *Add* operator.

$$\begin{aligned} \mathit{Add}_{\mathcal{O}} &: \mathit{AddLeft} \times \mathit{AddRight} \\ \mathit{Add}_{\mathcal{O}} &\in \mathit{Expr} \end{aligned}$$

This definition adds $\mathit{Add}_{\mathcal{O}}$ to *Expr*, such that the *Add* operator can be used at the operand positions over which we distributed *Expr* earlier. Interestingly, no operators have been assigned to the operands of the *Add* operator yet. Consider the following sort constraints:

$$\begin{aligned} \mathit{Integer} &\subseteq \mathit{AddLeft} \\ \mathit{Integer} &\subseteq \mathit{AddRight} \\ \mathit{Integer} &\subseteq \mathit{Expr} \\ \mathit{Add}_{\mathcal{O}} &\in \mathit{AddRight} \end{aligned}$$

, these constraints express that integer literals can appear as operands of $\mathit{Add}_{\mathcal{O}}$ in both positions. However, since the *Add* operator is only added to the *AddRight* location, the constraints allow only nested occurrences of $\mathit{Add}_{\mathcal{O}}$ on the right side, encoding right-associativity. This example demonstrates the flexibility of sort constraints: integer expressions can be used in lambda-expressions — owing to the constraints $\mathit{Add}_{\mathcal{O}} \in \mathit{Expr}$ and $\mathit{Integer} \subseteq \mathit{Expr}$ — whereas lambda-expessions cannot be used in integer expressions. Such rules of composition can be changed simply by selecting a different set of sort constraints without affecting the definitions of the operators themselves. As discussed in §3.4, selecting sort constraints is done as part of a language definition.

3.2. Compositional semantics

To retain the disjoint property of the operators, their semantics must be defined independently as well. This is achieved by defining semantic functions that together form an algebra. Semantic functions translate an operator into a specific semantic domain. For example, our previous operators defining the lambda calculus can have the following semantic functions, with funcons being our semantic domain[2].

$$\begin{aligned} \mathit{Var}_{\mathcal{F}}(\mathit{lit}) &= \textbf{bound string}\ \mathit{lit} \\ \mathit{Abs}_{\mathcal{F}}(x, b) &= \textbf{function closure scope}(\\ &\qquad \textbf{bind}(\textbf{string}\ x, \textbf{given}), b) \\ \mathit{App}_{\mathcal{F}}(\mathit{abs}, \mathit{arg}) &= \textbf{apply}(\mathit{abs}, \mathit{arg}) \end{aligned}$$

Through the catamorphism, the operands of an operator are already translated by their respective translation function when an operator is translated. Hence, an operator only needs to translate itself into the semantic domain while having access to the already translated operands.

3.3. Operator specialization

In certain circumstances, it may be necessary to adapt the semantics of language constructs in order to make them suitable for the language in mind. The so-called 'glue code', which adapts an existing semantic definition, is often used in these circumstances. This glue code is to be

[2] In the right-hand side, juxtaposition is the right-associative application of a funcon to a (single) funcon term, i.e. **bound string** lit == **bound**(**string**(lit)).

written modularly and in isolation, without anticipating, or constraining, future interactions. These observations can be exemplified by the following example: Consider an *if* operator encoding if-expressions or if-statements.

$$If_{\mathcal{O}} : IfCond \times IfTrue \times IfFalse$$
$$If_{\mathcal{F}}(c,t,f) = \textbf{if-true-else}(c,t,f)$$

The **if-then-else** funcon expects that the conditional evaluates to a boolean. However, in C-like languages, if-statements are defined in terms of integers. Therefore, to utilize $If_{\mathcal{O}}$, we need a mechanism to convert the integer-expression into a boolean-expression. The example below shows that glue code is added to a sub-sort declaration to achieve the necessary conversion within the semantic domain of funcons.[3]

$$CExpr \subseteq IfCond \qquad \text{(Sort constraint)}$$
$$\hookrightarrow \textbf{not is-equal}(0, CExpr_{\mathcal{F}}) \qquad \text{(with glue code)}$$

The sub-sort declaration determines that C-expressions can be used as conditionals. The added glue code defines a function that determines how the funcon terms produced for C-expressions are modified/extended when C-expressions occur as conditionals, i.e. occur at the *IfCond* operand position. The placeholder $CExpr_{\mathcal{F}}$ refers to the result of the translation of the C-expression before the glue-code, which is implicitly defined in terms of the translation functions given for the operators contained in the sort *CExpr*.

We thus have specialized the *If* operator to the semantics of our specific language without modifying the existing definition of the *If* operator nor do we need to define a different operator for all possible variations. In addition, by applying glue-code conditionally, it does not affect other operands assigned to the *IfCond* location and removing C-like expressions from the language does not leave any stale glue code.

3.4. Language definition

Languages can freely choose from the available operators and use sort-constraints to constrain the operator usage. We define a language as follows.

Definition 3.1. *Given a set O of operators, with every operator having an arity, denoted with $|o|$, a set of operand positions, denoted with $\overrightarrow{o} = \{1, \cdots, |o|\}$, and a semantic function $F(o) : \mathcal{F}^{|o|} \rightarrow \mathcal{F}$ where $\mathcal{F}$ is the set of funcon terms, we define a language as a structure $\langle T, S, G, I\rangle_O$ in terms of O, with $T \subseteq O$ being the set of top-level operators; S is a family $\langle S_{o\in O, w\in\overrightarrow{o}}\rangle$ of sets indexed by $O \times \mathbb{N}$. $S_{o,w}$ is the set of operators assigned to the operand position w of operand o. G is a family $\langle G_{o\in O, w\in\overrightarrow{o}}\rangle$ of functions indexed by $O \times \mathbb{N}$. $G_{o,w}$ is the glue function $O \times \mathcal{F} \rightarrow \mathcal{F}$ for operators assigned to the operand position w of operand o. I is a family $\langle I_{t\in T}\rangle$ of functions indexed by the top-level operators. $I_t : \mathcal{F} \rightarrow \mathcal{F}$ denotes the top-level initialization function for the specific top-level operator.*

We do not distinguish between sub-sort declarations and operator assignments, since all sub-sort declarations can be described in terms of operator assignments. Furthermore, the definition does not use names to refer to operand positions. Instead, integers are used. Nevertheless, we do use names in our examples as a notation convenience, ensuring that there is a one-to-one mapping between operand names and operand positions.

The top-level operators are present in a language to determine the entry points of the language. This can be used in generation of grammar/parsers for languages, generation of tooling, generation of language structure diagrams, etc. Initialization functions can be used to modify the top-level behavior of the language, affecting the behavior of code fragments when executed, for example, in a REPL interpreter. In this case, the effects of a code fragment are to be summarized by printing certain information to the screen such as the value computed for an expression or any new bindings that have been introduced by a declaration. This behavior is specified by the top-level specialization code. Doing so as an individual language (extension) makes it possible to define alternatives and to effortlessly switch between them. Another common use of top-level behavior is the handling of uncaught exceptions, e.g., to determine what run-time error message is printed to report uncaught exceptions.

3.5. Language composition

New languages can be defined by composing existing languages together.

Definition 3.2. *Language composition of two languages, L_1 and L_2, specified in terms of the same operator set, is defined as follows: $L_1 \diamond_O L_2 = \langle T_1 \cup T_2, S, G, I\rangle_O$, where*

$$S = \{S_{1\langle o,w\rangle} \cup S_{2\langle o,w\rangle} | o \in O, w \in \overrightarrow{o}\}$$
$$G = \{G_{2\langle o,w\rangle} \circ G_{1\langle o,w\rangle} | o \in O, w \in \overrightarrow{o}\}$$
$$I = \{I_{2\langle t\rangle} \circ I_{1\langle t\rangle} \mid t \in T_1 \cup T_2\}$$

From the associativity of the operations used on the elements of the languages, it follows that language composition is associative. Language composition, however, is not commutative due to the usage of function composition with G and I. With language composition, languages form a monoid. The neutral language can be defined by taking the empty set for T, letting the family S assign the empty set to every index, and letting the families G and I assign the identity function over funcon terms to every index.

Although languages are defined in terms of some operator set, this does not restrict language composition or

[3]The example is simplified to save space. When performing such glue on C, the checks need to be extended to supports floats, doubles, etc. This is easily achieved by dispatching on the type of the current value.

the incrementality we provide. A language does not need to utilize all operators of the operator set. As a consequence, we can simply add a new operator and grow the operator set. Existing composition can then be lifted to the new operator set for free. *iCoLa* thus exists out of two languages: one for defining operators and their semantics, and one for defining languages, such that the interpretation of the operator language results in a set of operators that is used in the interpretation of the language-definition language. Addition of a new operator is thus evaluated before the language definitions are evaluated, hence the free lifting of language composition to a larger operator set. Furthermore, because both operators and languages are compositional, from a users perspective there is no difference, and language and operator definitions can be freely mixed.

Being able to freely grow the operator set by introducing new operators while automatically lifting existing composition is based on achieving sequentiality via composition. This insight is best explained according to Figure 1. In essence, when composition is supported, incrementality is almost obtained for free, because we can always reformulate an incremental step as a composition from our starting point. This approach of creating a sequential language does require that the evaluation of a composition scales and is not much slower than simply evaluating the incremental step.

3.6. Implementation

The implementation supporting the conceptual model is a domain-specific language embedded in Haskell with the Template Haskell (meta-programming) extension. An example definition, in this case of the lambda calculus, in the eDSL is given in Listing 1. The Template Haskell function `genLanguage` takes an operator set and a language definition and returns an implementation for the defined language.

An operator is defined using a GADT and is identified by the constructor of the GADT and a so-called 'meta-type' – a reification of the conceptual sort containing only the defined operator. An abstraction operator for the lambda calculus might be defined as follows in the eDSL.

```
data Abs u t where
    Abs :: IsTrue (AbsBody t) =>
        String -> u t -> Abs u AbsType
type family AbsBody t
```

The constructor signature determines the arity and operand locations. The return type of the constructor indicates the data type of the operator and the meta-type. In this example, the meta-type is `AbsType`. Meta-types are used in operator assignments and are implemented as data types with no constructors. The operator definition in this example has an arity of two and the second parameter is the operand location identified by the `AbsBody` name. The linkage of the `AbsBody` name to the second parameter happens in the constraint-head of the constructor, where `t` refers to the meta-type of the second parameter. The constraint is met when an operator is assigned to the operand location identified by `AbsBody`. When the constraint is not met, a Haskell type error is given. Therefore, it is impossible to construct terms that violate the constraints.

To assign semantics to an operator, an instance for the type-class representing the semantic domain is defined. For example, the translation to the funcons semantic domain for the `Abs` operator is defined as follows.

```
instance ToFuncons Abs where
    toFuncons (Abs s (K body)) = K $ function_ [closure_
        [scope_ [bind_ [T.string_ s, given_], body]]]
```

The usage of the K constructor is (necessary) boilerplate to ensure kind-correctness. Furthermore, funcon constructors take a variable number of arguments, hence the usage of lists. The operands of an operator are already translated to the semantic domain because an algebra is being defined.

A language definition is a data type in terms of Template Haskell, containing operator assignments, sub-sort declarations, operator specialization code, and initialization code. Listing 1 gives an example language definition of the lambda calculus in the *iCoLa* eDSL. In this example, `(op, ''Expr)` demonstrates an operator assignment in the language, assigning the operator bound to the `op` variable, which is bound in the list-comprehension, to the auxiliary sort `Expr` — defined as a type family. Since language definitions are at the level of Template Haskell, Haskell constructs need to be quoted, hence the `''` in front of the `Expr` auxiliary sort. Sub-sort declarations follow a similar pattern as operator assignments, except the first element of the tuple must be an auxiliary sort or an operand location instead of an operator meta-type.

Because language definitions are Template Haskell data types, languages form first-class citizens [22]. As a result, languages can be manipulated via Haskell functions, which is how the extension, unification, and restriction operators defined by Erdweg et al [11] are implemented in *iCoLa*.

4. Extended compositional definitions

In this section we extend the original model with concrete syntax and by generalizing to an arbitrary amount of semantic domains.

To generalize our approach to an arbitrary amount of semantic domains, we update the notion of an operator set to a family as follows:

Definition 4.1. *Let D be a set of domains. Every domain gives rise to a set of valid terms in that domain, denoted with A_d for a given domain $d \in D$. An operator set O_D is a family $\langle O_d \rangle$ indexed by D. O_d is the set of operator symbols with a translation to the semantic domain d, i.e. there exists a function $A_d^{|o|} \to A_d$ for every operator symbol $o \in O_d$, where $|o|$ is the arity of the operator symbol o.*

Listing 1 Definition of the lambda calculus in the eDSL implementation of *iCoLa*.

```
$(genLanguage [(''Var, ''VarType), (''Abs, ''AbsType),
               (''App, ''AppType)] lambdaLanguage
  where
     lambdaLanguage = Language
        { op_assign = [(op, ''Expr) | op <- [''VarType, ''AbsType, ''AppType]]
        , sub_sorts = [(''Expr, t)  | t <- [''AbsBody, ''AppLeft, ''AppRight, ''TopLevel]],
        , glue_code = []
        , init_code = []
        })
type family Expr t
```

As a notation convenience, we use O to refer to all operator symbols in an operator set, i.e. $O = \bigcup O_D$.

When all operator symbols in an operator set have a translation to a specific domain, we call the set complete with respect to the domain.

Definition 4.2. *An operator set, O_D, is complete with respect to a domain $d \in D$ iff all operator symbols have a translation to the semantic domain d.*

We also update our language definition to handle the arbitrary domains and extend the language definition with a (concrete) syntax component. Syntax is defined at the language level instead of the operator level, because operator definitions are immutable. Syntax, however, can vary widely for the same operator between languages.

Definition 4.3. *A language is a structure $\langle T, S, G, I, P\rangle_{O_D}$ in terms of O_D, with*

- *$T \subseteq O$ being the set of top-level operator symbols;*
- *S is a family $\langle S_{o \in O, w \in \overrightarrow{\sigma}}\rangle$ of sets indexed by $O \times \mathbb{N}$. $S_{o,w}$ is the set of operator symbols assigned to the operand position w of operand o;*
- *G is a family $\langle G_{o \in O, w \in \overrightarrow{\sigma}, d \in D}\rangle$ of functions indexed by $O \times \mathbb{N} \times D$. $G_{o,w,d}$ is the specialization function $O \times A_d \rightarrow A_d$ in the domain d for operators assigned to the operand position w of operand o;*
- *I is a family $\langle I_{t \in T, d \in D}\rangle$ of functions indexed by $T \times D$. $I_{t,d} : A_d \rightarrow A_d$ denotes the top-level initialization function for the specific top-level operator in the given domain;*
- *P is a family $\langle P_{o \in O}\rangle$ of sets indexed by O. $P_o \subset (\Sigma \cup \overrightarrow{\sigma})^*$ is the singleton set representing the syntax rule that produces the operator identified by the operator symbol o, and Σ is a set of terminal symbols.*

The update to language composition (Definition 3.2) is trivial with a right-biased operator for syntax rules and the empty rule (ϵ) as the neutral element.

In contrast to productions as used in the definition of a context-free grammar, our definition does not include the notion of a non-terminal. Instead, in syntax rules, placeholders are available to refer to the non-terminals generated for the sorts of the operand positions.

With the used syntax definition, a context-free grammar $G = \langle V, A, P, S\rangle$ can be generated using the algorithm specified in Algorithm 1. The process is as follows: We take the alphabet to be all operator symbols in the operator set, all operand locations, all terminal symbols, and add a symbol to represent the top-level. The set of terminals in the CFG correspond to the set of terminals used in the syntax rules. Productions for operand location non-terminals are obtained by creating an alternative for every non-terminal corresponding to the operators assigned to the location (in the algorithmic description, we use `choice` to combine alternatives into one production). Productions for operator symbols are obtained by taking the syntax rules and turning them into productions by replacing the location placeholders with their corresponding generated non-terminal. Finally, productions for the top-level non-terminal are obtained by creating an alternative for every non-terminal corresponding to the operators assigned to the top-level, and we denote the top-level as the distinguished element.

Algorithm 1 Algorithm to turn meta-productions into a context-free grammar

function LANGTOCFG($\langle T, S, G, I, P_m\rangle_{O_d}$)
    $A \leftarrow$ *terminals* P_m
    $L \leftarrow \{o_w \mid o \in O, w \in \overrightarrow{\sigma}\}$ ▷ Create non-terminals for all operand locations
    $V \leftarrow O \cup L \cup \{TopLevel\} \cup A$
    $P \leftarrow \{\}$
    $S \leftarrow \{TopLevel\}$
    for $o_w \leftarrow L$ **do** ▷ Create productions for the operand location non-terminals
        $P \leftarrow P \cup \{(o_w, reduce\ G_{o,w}\ using\ choice)\}$
    end for
    for $p \leftarrow P_m$ **do**
        $P \leftarrow P \cup concrete\ p$ ▷ Replace location placeholders with corresponding operand location non-terminal
    end for
    $P \leftarrow P \cup \{(TopLevel, reduce\ T\ using\ choice)\}$
    return $\langle V, A, P, S\rangle$
end function

To illustrate this process we take our example lambda calculus and extend it with the following syntax rules.

$$\begin{aligned}\langle P_{var}\rangle &::= \langle 0\rangle \\ \langle P_{abs}\rangle &::= \langle 0\rangle\texttt{->}\langle 1\rangle \\ \langle P_{app}\rangle &::= \langle 0\rangle\langle 1\rangle\end{aligned}$$

With these rules we obtain the following values for the V and A sets: $V = \{abs, var, app, TopLevel, \texttt{-}, \texttt{>}\}$, $A = \{\texttt{-}, \texttt{>}\}$. We then create the productions from the syntax rules, with operand location productions first.

$$\begin{aligned}\langle var_0\rangle &::= \texttt{String} \\ \langle abs_0\rangle &::= \texttt{String} \\ \langle abs_1\rangle &::= \langle\text{Var}\rangle \mid \langle\text{Abs}\rangle \mid \langle\text{App}\rangle \\ \langle app_0\rangle &::= \langle\text{Var}\rangle \mid \langle\text{Abs}\rangle \mid \langle\text{App}\rangle \\ \langle app_1\rangle &::= \langle\text{Var}\rangle \mid \langle\text{Abs}\rangle \mid \langle\text{App}\rangle\end{aligned}$$

We continue by filling the locations in the syntax rules turning the rules into productions for the CFG, and define the top-level production. The result of this process is a CFG for our definition of the lambda calculus.

$$\begin{aligned}\langle Var\rangle &::= \langle var_0\rangle \\ \langle Abs\rangle &::= \langle abs_0\rangle\texttt{->}\langle abs_1\rangle \\ \langle App\rangle &::= \langle app_0\rangle\langle app_1\rangle \\ \langle TopLevel\rangle &::= \langle App\rangle \mid \langle Abs\rangle \mid \langle Var\rangle\end{aligned}$$

5. Implementation

In this section we discuss our DSL implementation of *iCoLa*$^+$. The DSL is tool-oriented in the sense that *iCoLa*$^+$ is by itself not executable, it needs to be embedded within a larger tool that orchestrates different components, one of which is *iCoLa*$^+$. The tool-oriented design enables the construction of a pyramid abstraction, displayed in Figure 5, in which the different layers are operated by different kind of users. Users who operate in the lower parts of the pyramid require more expertise and provide abstractions as foundations for the users at higher-levels.

We start this section by explaining the design of the meta-language, then we detail the internal implementation, and we finalize by demonstrating the embedding of *iCoLa*$^+$ inside tooling.

5.1. The iCoLa$^+$ Language

5.1.1. Operators

Operator are defined using the `operator` keyword and require a unique name and a variable amount of operand locations. Operand locations are identified by names and these names must be unique among the locations for an operator, but not among locations across operators.[4] The following example demonstrates the definition of the three operators present in the lambda calculus.

```
operator Var : Var
operator Abs : Var Body
operator App : Fun Arg
```

The name before the `:` symbol indicates the name of the operator. The names after the `:` symbol are the names for operand locations. In case of our definition of the `Abs` operator, there are two locations: `Var` and `Body`.

Certain operators have operand locations that can take a variable number of values when instantiating the operator. An example of such an operator is a list. To support this, operand locations can have a modifier indicating the degree of values an operand location accepts. There are three modifiers currently available: the `+` modifier, indicating one or more values; the `*` modifier, indicating zero or more values; and the `?` modifier, indicating zero values or one value. For example, a list operator can be defined as follows in *iCoLa*$^+$.

```
operator List : Item*
```

Indicating that the `Item` location can have zero or more values.

5.1.2. Operator semantics

Operator semantics is given by translating an operator to a chosen semantic domain. Every semantic domain is uniquely identified by its name. The name of a semantic domain identifies a translation function in the DSL. To illustrate, we demonstrate the translation for the three operators of the lambda calculus to the semantic domain of funcons.

```
funcons Var = bound(@Var)
funcons Abs = function closure scope(bind(@Var, given), @Body)
funcons App = apply(@Fun, @Arg)
```

The example starts with the name of the semantic domain — `funcons` — to indicate that the translation function is into the semantic domain of funcons. After the name of the semantic domain, the operator being translated is identified by its name, and is followed by the `=` operator to indicate the start of the body of the translation function. The syntax available in the body of a translation depends on the domain in which the translation function is being defined. A domain definition is free in its choice of syntax as long as it supports the `@` operator for naming holes. The names for holes correspond to operand locations for the operator being translated. Essentially, a domain definition can define its own DSL inside the *iCoLa*$^+$ DSL. In our example, the body of the `funcon` translation function uses funcon terms as the concrete syntax, corresponding to the syntax used in the formal model of Section 3 and the syntax used by the PLanComps projects.

[4] Essentially, operators introduce a namespace for the locations. Since operator names are unique, we retain the uniqueness of operand locations as required by the conceptual approach.

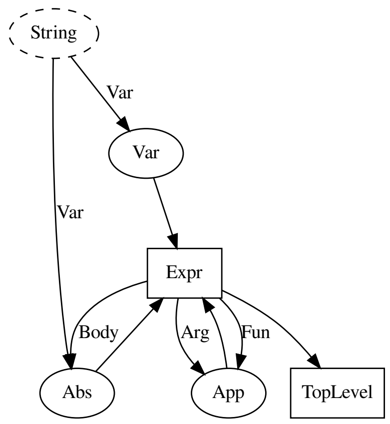

Figure 2: Constraint graph for the lambda calculus example. Ellipses denote operators, dashed ellipses denote built-in operators, and rectangles denote auxiliary sorts. Edges denote assignment of operator to the operand location determined by the edge label.

5.1.3. Language definitions

Languages are defined via the `language` keyword and can contain sort constraints, parser definitions, and operator specialization code. To illustrate, we construct a variant of the lambda calculus in steps, starting with the constraints defining the language.

```
sort Expr
language Lambda
{ {Var, Abs, App} < Expr
, { String } < Var[Var]
, { String } < Abs[Var]
, Expr < Abs[Body]
, Expr < App[Fun, Arg]
, Expr < TopLevel
}
```

We first define the `Expr` auxiliary sort. Then a new language, named `Lambda`, is introduced. Inside the language definition, we start with the `{Var, Abs, App} < Expr` constraint. This states that the `Expr` sort must contain those three operators to be a correct instantiation of the `Lambda` definition. After, we constraint the `String` operator over the operand location `Var` for the `Var` and `Abs` operators. The syntax `operator[name]` is used to reference the operand location identified by `name` for the specified operator. This is followed by distributing the `Expr` sort over the operand locations where expression can occur, and assigning the `Expr` sort to the top-level. With this definition, we get the constraint graph displayed in Figure 2 for our `Lambda` definition.

In language definitions there is no separate syntax between operator assignments and sub-sort constraints. Instead, operator assignments are defined as sub-sort constraints using an anonymous sort — identified by the inline set notation.

Our current definition has several repetitions. To reduce repetition, we can utilize several forms of syntactic sugar supported by the *iCoLa*$^+$ DSL. First, usage of a specific sort in multiple constraints can be merged via sequencing. In our example, this occurs with the `String` operator and the `Expr` sort. We can update our definition as follows.

```
sort Expr
language Lambda
{ {Var, Abs, App} < Expr
, { String } < Var[Var] ; Abs[Var]
, Expr < Abs[Body] ; App[Fun, Arg] ; TopLevel
}
```

The sequence operator (`;`) groups sorts together and applies the constraints as if it were individual constraints. In the updated definition, sequencing occurs on the right-hand side of the `<` operator. Nevertheless, sequencing can also occur on the left-hand side. Besides sequencing to reduce duplication, some constraints use all operand locations of an operator. In our example this occurs within the constraints referencing `Var[Var]` and `App[Fun, Arg]`. Instead of naming all locations, an empty indexing expression `[]` can be used. With this form of syntactic sugar, our example definition can be updated as follows.

```
sort Expr
language Lambda
{ {Var, Abs, App} < Expr
, { String } < Var[] ; Abs[Var]
, Expr < Abs[Body] ; App[] ; TopLevel
}
```

Our updated definition has no need for the `Expr` sort for the definition of our `Lambda` language.

```
language Lambda
{ {Var, Abs, App} < Abs[Body] ; App[] ; TopLevel
, { String } < Var[] ; Abs[Var]
}
```

However, removing such an auxiliary sort makes it more work to add a new operator that is usable at the same operand locations as the three existing operators. Whether to use an auxiliary sort depends on the intended usage of the language and the used operators. There is no correct way and one choice does not restrict future usage of a defined language.

Having finalized our constraints for the lambda calculus, we move on to the concrete syntax. Concrete syntax is added to a language by defining syntax rules using the `::=` symbol. Our lambda calculus language can thus be extended with concrete syntax as follows.

```
Lambda
{ Var ::= @Var
, Abs ::= '\\' @Var "->" @Body
, App ::= @Fun @Arg
}
```

Instead of using the `language` keyword to introduce a new language, we extend the existing definition by referring to the name identifying the language. In the extension, we define syntax rules for the three operators in the lambda calculus. The left-hand side of the `::=` symbol identifies the operator for which the syntax rule is being defined, and the right-hand side contains the actual body of the syntax rule. The right-hand side has access to the locations of the current operator via the @ symbol. Besides locations, syntax rules can contain literal strings — enclosed between “ ” — and literal characters — enclosed between ‘ ’. In addition, when an operand location can take a variable number of values, parsing modifications can be used to control how the values are parsed. For example, a parser for our previous list operator can be defined as follows: `list ::= @Item{,}`, denoting that it will parse items separated by the separation character, which is a comma in this example. In case no parser modification is given, it will parse multiple items without any separation character separating them.

Our current definition of the lambda calculus can be slightly extended by showing evaluation results. To show evaluation results, we add initialization semantics to the language. In the DSL, initialization code is achieved by specializing the `TopLevel` sort.

```
Lambda
{ funcons TopLevel when Expr => print @this
}
```

This example specializes the `TopLevel` sort for the `funcons` domain when the operator currently bound to the top-level is part of the `Expr` sort, by printing the evaluation result. In specialization, the current term being specialized in the semantic domain is referenced using the `@this` construct. In contrast to the formal model, in the DSL we make no distinction between initialization and specialization code. Instead, initialization is achieved by specializing the `TopLevel` sort, which is built-in. Without a `when` expression, specialization is always applied to all operators assigned to the specialized location.

5.2. Internal representation

iCoLa$^+$ is implemented in Haskell and in this section we detail the internal representation and the evaluation pipeline that turns an *iCoLa*$^+$ specification into an executable language given a language definition and a semantic domain. Figure 3 gives an overview of this process.

5.2.1. Operators

Operators are implemented as functors — type transformations that come with a function (`fmap` in Haskell) to lift functions from the original type to functions on the transformed type. Using functors, we can use the same data type for the different representations of an operator. When an operator is defined in the DSL, it has a string representation. When a language is instantiated, operators have a fix-point representation, allowing us to represent terms of the defined language.

```
data Operator a = Operator ID [a]
                | OBuiltIn (BuiltInOp a)

data BuiltInOp a = OTuple [a]
                 | OInt Int
                 | OOString String
                 | OBool Bool
                 {- ... -}
```

The `Operator` constructor is used for operator definitions. The remaining constructor is used for the built-in operator data type, of which a selection of the constructors is shown.

5.2.2. Semantic domains

A semantic domain is represented by a name, a parse function, an algebra definition, and an instance of the substitution type-class. The parse function takes the body of a semantic function and translates it into an internal representation with holes. Then the algebra is applied on this internal representation, which translates built-in operators to the semantic domain. The substitution type-class implements the replacement of named holes with the terms to which the names are mapped, and is used to automatically translate the `Operator` constructor to the semantic domain. To illustrate, let us look at the definition for the funcons domain.

```
funconsDomain :: Domain Funcons
funconsDomain = Domain "funcons" Funcons.parse funconsAlg

funconsAlg :: BuiltInOp FT.Funcons -> FT.Funcons
funconsAlg (OInt i) = FT.int_ i
funconsAlg (OOString s) = FT.string_ s
funconsAlg (OBool b) = FT.bool_ b
funconsAlg (OTuple t) = FT.tuple_ t

instance Substitution Funcons where
    subst = applyFuncon

data Domain a =
    Domain String (String -> a) (BuiltInOp a -> a)

class Substitution a where
    subst :: M.Map ID a -> a -> a
```

The `Funcons.parse` function parses the concrete syntax of funcons into a funcons term with holes, and the `applyFuncon` function substitutes named holes with the term to which the name is mapped.

5.2.3. Parser concretization

To handle ambiguity in syntax rules we utilize Generalized LL (GLL) parsing [23]. Syntax rules are implemented by translating them to combinators defined by the GLL combinators library [24], and follows the process as outlined in Algorithm 1.

For example, parsers for operand locations are achieved by mapping to the `<|>` combinator, denoting choice. operand locations that can parse multiple values are mapped to the `multiple` or `multiple1` combinators, depending on the presence of an operand location modifier. In case a parser modifier is present, one of the `sepBy`, `sepBy1`, or `optional` combinators is used. Essentially, there is a mapping for every construct present in parser definitions to a combinator in the GLL library.

To handle the parse results of operand location modifiers, the `OTuple` constructor is used. When the `?` modifier

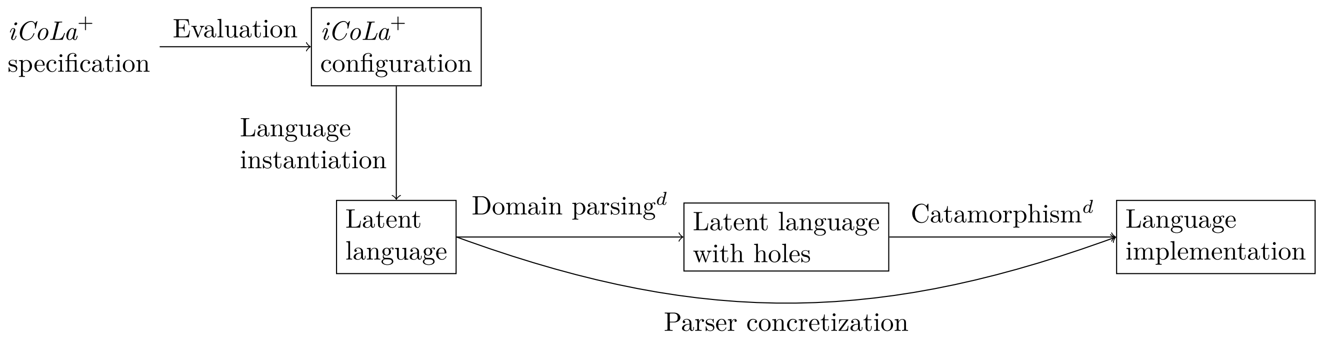

Figure 3: Overview of the *iCoLa*+ evaluation pipeline. Domain parsing and the catamorphism are parameterized by a domain definition.

is used, the result is either a tuple with one element containing the result or a tuple with zero elements. In case of the + and * modifiers, the `OTuple` constructor containing all the parsed values is used. Semantic domains thus must support operations on tuples. For example, the translation to the funcon domain for the `List` operator is as follows.

```
funcons List = list tuple-elements @Item
```

The `tuple-elements` extracts a tuple into a sequence of values, which is accepted by the funcons `list` constructor. The method in which a semantic domain supports operation on tuples can be done differently. In case of the funcons domain, this needs to be explicitly encoded in the translation functions. Alternatively, a semantic domain can encode this internally without requiring the unpacking of tuples to happen inside translation functions. Then, for example, the `list` constructor would accept a tuple as an argument.

5.3. Environment definitions

So far, we have looked at the *iCoLa*+ implementation using the semantic domain of funcons and have not detailed how users interact with *iCoLa*+. User interaction and the full capabilities of *iCoLa*+ are determined by environment definitions. Figure 4 gives an abstract view of environment definitions, which consists out of two environments: the meta-environment and the language environment. A language designer interacts with a meta-environment and a user of the defined language interacts with the language environment. Of course, these two might be the same. The method of interaction is determined by the environment definition, which orchestras the interaction between the different users with their respective environments, and in case of the meta-environment determines the capabilities via the selection of domain definitions.

Inside an environment definition, there is a dependency of the language environment on the meta-environment in the form of generated language implementations. Internally, the `instantiateLanguage` function is used to obtain a language implementation for a given language and a given domain.

```
instantiateLanguage :: String -> Domain a
    -> ICoLa (String -> FOperator, FOperator -> a)
```

The first parameter to the function is the name of the language being instantiated, and the second parameter is the domain for which the language is being instantiated. The result of this function is a parser and an evaluator, which can be used by the environment definition to connect to user interaction in whatever way fit. The parser and evaluator are separate instead of composed to give flexibility to the environment definition.

When instantiating a language, we require that the language is complete with respect to the chosen domain (Definition 4.2) If not, an error is thrown.

iCoLa+ specifications can contain domain definitions which are not used by a specific environment. *iCoLa*+ only checks the correctness of the domain when a language is instantiated. As a result, a specification does not need to change when used in an alternative environment.

5.4. The interaction layers of iCoLa+

The *iCoLa*+ implementation can be divided in three layers: the DSL for language and operator definitions, the environment definitions, and the domain definitions. The aim of this separation is to allow different kind of users with different expertise and knowledge to interact with *iCoLa*+. The layers form the earlier mentioned pyramid abstraction as displayed in Figure 5.

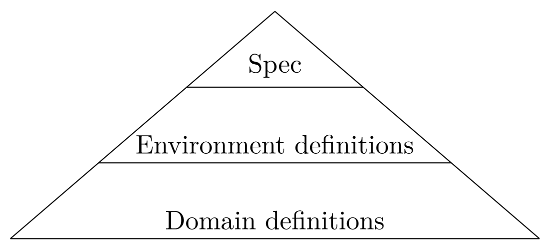

Figure 5: Abstraction pyramid showing the different interaction layers of *iCoLa*+.

Most users will be interacting with the top layer of the pyramid, which is the layer where languages and operators are being defined. Users that want to integrate language environments into tooling or want to work on user interfaces for language development will be interacting with the middle layer. Users that are language engineering experts

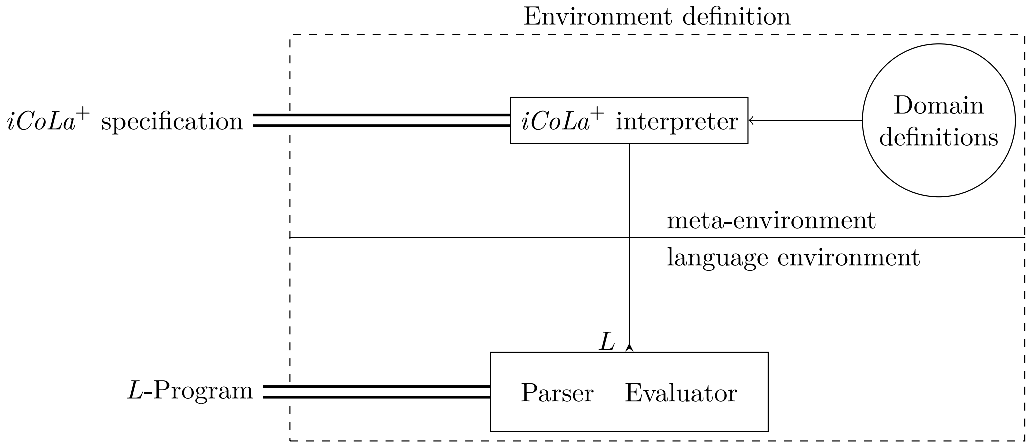

Figure 4: Visual view showing the embedding of *iCoLa*$^+$ inside an environment definition. Normal arrows denote a dependency. Stealthed arrows denote generation. Pipes denote abstract interaction between two components. The concrete interactions patters are determined by the environment definitions and therefore unknown to *iCoLa*$^+$. Inspired by a diagram found in previous work [25].

and want to experiment with alternative approaches to semantic specifications will be interacting with the last layer. Of course, a user can interact with multiple layers as well, but the intention behind the abstraction pyramid is that it is, in the general case, not needed. The different layers thus also require different levels of understanding of the system. Interaction with the *Spec* layer requires no Haskell knowledge and no knowledge of the implementation; it does require understanding of the semantic domains being used by the user, and the conceptual idea of *iCoLa*$^+$ on the usage of sort-constraints. Users that interact with *iCoLa*$^+$ via the *Environment definitions* layer need to understand the basics of Haskell. They do not need to understand the actual details of the evaluation pipeline. Users interacting with the *Domain definitions* layer need to have an intermediate understanding of Haskell, and need to understand the basics of the *iCoLa*$^+$ evaluation pipeline as shown in Figure 3, especially the usage of catamorphisms since that guides the domain definitions.

6. A Demonstration of *iCoLa*$^+$

In this section we evaluate *iCoLa*$^+$ by conducting an extended exact replication of the evaluation performed on *iCoLa*. We opted for this kind of evaluation to enable us to focus on the extensions proposed in this paper, and to be able to compare the implementation introduced in this paper to the already existing implementation, and by extending *iCoLa*$^+$ via the provided extension points.

As part of the evaluation, we construct the following three languages via the composition, extension, and refinement of existing languages and language fragments — language definitions that are not executable by themselves. *Imp* [26], a simple imperative language; *SIMPLE* [27], a more complex procedural language; and *MiniJava* [28], a strict subset of the Java language. These languages are chosen because they have their semantics described in terms of funcons as part of the case studies for the PlanCompS project.[5] We have various reasons for picking languages that already have their structure and semantics expressed. With this choice, we are able to demonstrate that our approach is applicable to already existing language definitions, and is effective as shown by taking existing language definitions and turning them into the compositions of smaller languages. In addition, we are able to show that our approach promotes reusability by reusing language definitions within new definitions. It is incremental because the chosen languages are defined in an incremental and step-wise manner, and is flexible by showing that language design choices do not restrict future usage of language definitions.

The structure of this section is based on the abstraction pyramid introduced in the previous section. Sections 6.1-6.5 correspond to the *Spec* layer; Section 6.6 corresponds to *Environment definitions* layer; and Section 6.7 corresponds to the *Domain definitions* layer.

6.1. Specification of Imp

We define *Imp* as the composition of the following four languages:

```
language Imp = ImpArith <> ImpBExpr
    <> ImpStmt <> ImpProgram
```

The composition is comprised of a simple arithmetic language with support for integer addition and division; a boolean expression language with support for less-than-equal comparison and (binary) conjunction; a statement-language containing if-statements, while-statements, and sequencing of statements; and a program language that unifies these languages together by defining the top-level in accordance to the top-level of *Imp* (in that order). The structural definition of these languages are displayed in

[5] https://plancomps.github.io/CBS-beta/docs/Languages-beta/index.html

Listing 2, and the concrete syntax definitions for operators used in the fragments are displayed in Listing 3.

Except for the definition of `ImpProgram`, the top-level is not constrained. As a result, these language definitions are not executable on themselves due to there being no entry points. Such languages will be referred to as language fragments. In addition, `ImpArith` defines specialization code over the *Div* operator, wrapping the divide in a check. When division by zero occurs, the program is terminated following the application of the funcon **checked**. This behavior is not directly encoded in the semantics of the *Div* operator, because other languages handle this differently, for example by throwing an exception. Finally, `ImpBExpr` uses the *Aexpr* auxiliary sort in its definition (`Aexpr < Leq[]`) but does not assign any operators to this sort. Hence, `ImpBExpr` is an extension on `ImpArith`.

Instead of extending `ImpArith`, we can define `ImpBExpr` independently and use language unification to connect the two languages. Therefore, we define a weaker variant of the `ImpBExpr` language as follows.

```
language ImpBExpr⁻
{ {Bool, Leq, Not, And } < Bexpr
, Bexpr < Not[] ; And[]
}
```

And we unify this with the `impArith` language via a glue language to come back to our original definition.

```
language ImpBGlue = { Aexpr < Leq[] }
language ImpBExpr = impArith <> impBExpr⁻ <> ImpBGlue
```

6.2. Specification of SIMPLE

Since language composition is a main concept in our implementation, we can utilize *Imp* and the language fragments used to define it in the definition of other languages. Even when the definition of *Imp* does not directly correspond to the definition of the to be defined language. To illustrate this, we will define *SIMPLE* using the *Imp* language or the language fragments making up *Imp*.

There are two adaptations required to *Imp* to define *SIMPLE* as an extension of *Imp*: removing the top-level definition of *Imp* and removing the distinction *Imp* makes between arithmetic and boolean expressions. Note that *Imp* variables can only occur inside arithmetic expressions. To alleviate the problem of the top-level, we opt to define the base using the language fragments of *Imp* without the `ImpProgram` fragment. To alleviate the second problem we define a new language that glues the two different expressions into a new sort:

```
sort Expr
language UnifiedExpr
{ Aexpr ; Bexpr < Expr
, Expr < Leq[] ; Add[] ; Div[] ; While[Cond] ; If[Cond]
}
```

We first constrain the new `Expr` sort with both the `Aexpr` and `Bexpr` sort. Then we constraint the operand locations of *Imp* operators which also occur in *SIMPLE* and use expressions with the new `Expr` sort. Using this language, we removed structural choices of *Imp* to align with *SIMPLE*. Now, we can reuse the fragments to define a base for *SIMPLE*:

```
language SimpleBase = UnifiedExpr <> ImpArith
   <> ImpBExpr <> ImpStmt
```

Having defined a suitable base, we can extend our base with the constructs that are present in *SIMPLE* but not in *Imp*. A subset of these constructs is displayed in Table 1. The table is not exhaustive. Most operators that only occur within *SIMPLE* have been omitted for brevity. Operators are grouped to indicate their relation within a possible language fragment.

	Imp	*MiniJava*	*SIMPLE*
Arith			
Addition	●	●	●
Division	◓	◓	◓
Substraction	○	●	●
Multiplication	○	●	●
Bool			
Negation	●	●	●
≤	●	○	●
And	●	●	●
Or	○	○	●
<	○	●	●
Statements			
If*	●	●	●
While	●	●	●
Assignments*	●	●	●
Input/Output			
Ouput*	○	◓	●
Input	○	○	●
Classes	○	●	○
Arrays			
Length*	○	●	●
Indexing	○	◓	◓
Exceptions			
Throw	○	○	●
TryCatch	○	○	●

Table 1: The rows indicate operators used during the evaluation and the columns the constructed languages from the collection. The ● indicates that the operator is used as is; ◓ indicates that an operator is used with glue code; and ○ indicates that an operator is not used. A * next to an operator indicates that the concrete syntax for the operator is not identical between the languages.

6.3. Specification of MiniJava

To define *MiniJava*, we can reuse the definition of the base for *SIMPLE*. However, *MiniJava* requires one more step, because the less-than-equal operator does not occur in *MiniJava*. Therefore, we define a refinement which removes the less-than-equal operator from the *SIMPLE* base.

```
language MiniJavaBase = refine SimpleBase $ { Leq }
```

Building on our *MiniJava* base, we can add operators not present in *Imp*, such as classes and arrays.

Listing 2 Structure definitions for the language fragments used in the definition of the *Imp* language.

```
language ImpArith
{ {Int, Id, Div, Add, AParen} < Aexpr
, {String} < Id[]
, Aexpr < Add[] ; Div[] ; AParen[]
, funcons Div ⇒ checked @this
}

language ImpStmt
{ {Assign, If, While, Block} < Stmt
, { String } < Assign[Id]
, Aexpr < Assign[Expr]
, Bexpr < While[Cond] ; If[Cond]
, Stmt < Block[]
, { Block } < If[True, False] ; While[Body]
}
```

```
language ImpBExpr
{ { Bool, Leq, Not, And, BParen } < Bexpr
, Bexpr < Not[] ; And[] ; BParen[]
, Aexpr < Leq[]
}

language ImpProgram
{ { String } < SIdList[Id]
, { SIdList } < IdList[] ; Program[Ids]
, { IdList } < IdList[Rem] ; Program[Ids]
, Stmt < Program[Program]
, { Program } < TopLevel
}
```

Listing 3 Concrete syntax extensions to the language fragments used in the definition of the *Imp* language.

```
ImpArith
{ Add ::= @Left '+' @Right
, Div ::= @Left '/' @Right
, AParen ::= '(' @Expr ')'
, Id ::= @Var
}

ImpStmt
{ While ::= "while" '(' @Cond ')' @Body
, If ::= "if" '(' @Cond ')' @True
   "else" @False
, Block ::= '{' @Stmt @Rem '}'
, Assign ::= @Id '=' @Expr ';'
}
```

```
ImpBExpr
{ Not ::= '!' @Expr
, Leq ::= @Left "<=" @Right
, And ::= @Left "&&" @Right
, BParen ::= '(' @Expr ')'
}

ImpProgram
{ Program ::= "int" @Ids ';' @Program
, IdList ::= @Id ',' @Rem
, SIdList ::= @Id
}
```

Many of the operators used within *MiniJava* are also present within *SIMPLE*. However, the usage of these operators is not always identical. For example, in *MiniJava* output is always followed by a newline, which is not the case in *SIMPLE*. Such differences are alleviated using glue code, which enables us to still share operators and languages even when the usage of operators does not fully align.

MiniJava is interesting because variations of *MiniJava* exist that have been introduced for teaching purposes. Flexibility regarding the constructs included in the language enable a teacher to adapt to student expertise. This flexibility is naturally supported by our system since the (full) *MiniJava* language can be given as the composition of multiple smaller language variants. For instance, our version of *MiniJava* actually deviates slightly from the original definition. The slight deviation is the presence of the division operator. This is not present in the original definition due to the requirement of exceptions to support this. In our version, we simply inherit this from the *Imp* language and obtain division for free. To obtain the original definition, we can simply refine our version of *MiniJava* and remove the division operator. Utilizing existing language definition gives a teacher the means to simply create new variants with more or less language features based on the needs of the assignment and teaching objectives. Another possible extension we could have included is the inclusion of exceptions as present in the *SIMPLE* language definition. Again, extending *MiniJava* without much effort.

6.4. Object language variability

Language variability is not only useful for teaching purposes; it is also useful when designing a programming language. With *iCoLa*$^+$, different variants of a language can be defined and tested with relative ease. Multiple variants can exist side-by-side, making it easy to compare and contrast variations and gather feedback early, on both the concrete syntax and semantics, to include in the design process. Table 2 demonstrates some of the variability one can obtain with a relative small set of operators. The language definitions were defined in isolation or via composition. For instance, $lambda_{cbn}$ is defined by composing the *lambda* language with a language consisting (only) of glue-code that inserts the semantics of call-by-name using thunks [29].

```
language cbnGlue
{ funcons App[Arg] ⇒ thunk @this
, funcons Var ⇒ force @this
}
lambdaCBN = lambda <> cbnGlue
```

In this definition, we assume that all variables are assigned to thunked values. This is not always the case, e.g. in a procedural language with global variables. Type information can be used to distinguish variables based on whether their values are thunked. This, however, is not possible in our glue code definitions because glue code is context-free. Nonetheless, it can be realized within the semantic domain of funcons, as funcon terms are dynamically typed. The table shows an overlap between different languages and the two forms of variability in our approach: we can add new operators to existing languages and add new languages using existing operators, without modification of existing code.

6.5. Exploratory language development

The variability obtained in the previous section was achieved purely via incremental programming by introducing unique names for variants, as illustrated with the different lambda variants. In an exploratory setting, such variants can instead be explicitly defined as variants with the same name, supporting the utilization of both variants throughout an exploration. For instance, we could have defined our two lambda versions as two explicit variants by introducing a branch for both variants. An advantage of this approach is that we can experiment on both branches without having to duplicate our steps by replicating the actions executed on one branch automatically on the other branch, a concept we call mirroring. With mirroring, we can explore multiple branches simultaneously while operating on a single branch. A visual presentation of the idea of mirroring is displayed in Figure 6.

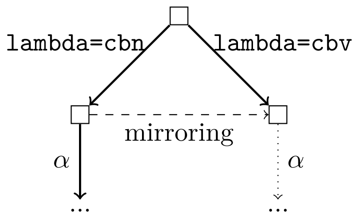

Figure 6: Visual idea of mirroring during exploratory programming, where a branch mirrors another branch explore different paths without duplication. α represents the execution of an arbitrary *iCoLa*$^+$ program, boxes denote configurations, solid lines denote actions taken by the user, dashed lines denote meta-actions, and dotted lines denote actions automatically done by system.

Mirroring is not the only advantage of first-class support for exploratory programming. With first-class support it is also trivial to experiment with different combinations of languages within different context, for example functional vs object-oriented, and switch between the contexts easily while also being able to compare the explorations. Furthermore, handling dead-ends during exploration is also supported by being able to go back to earlier points of the exploration. Since exploratory programming is an open ended task, such dead-ends are not unusual and having to restart a session, thus losing all context, hampers the experimentation. The support for exploratory programming within *iCoLa*$^+$ is essentially achieved for free via the generic back-end of [30] and the design of our DSL as a sequential language.

6.6. Environment definitions

The session as described by Table 2 was performed in the *iCoLa*$^+$-shell. The *iCoLa*$^+$-shell is a construction of the original *iCoLa*-shell as an environment definition with support for the aforementioned exploratory programming. In the *iCoLa*$^+$-shell, users can define operators and languages, and commit a language which results in a REPL for the defined language. After experimenting with this language, they can return back to their session in the *iCoLa*$^+$-shell, adapt the language definition and then commit the newly defined language. Furthermore, within the *iCoLa*$^+$-shell users can manage the exploration state via two meta-commands: `jump` and `revert`. With `jump`, users can jump to arbitrary states already seen during the exploration, which can be used to introduce branching. With `revert`, users can prune the exploration tree to throw away futile paths. In addition, users have access to the `mirror` meta-command, which results in mirroring of program execution across multiple branches. The `mirror` meta-command is fully defined in terms of `jump` and the execution of programs.

6.7. Domain definitions

The main requirement for a domain definition is that its evaluation model must abide by the initial algebra semantics approach. To illustrate the extensibility this provides, we give an alternative semantic domain. The example semantic domain we use to demonstrate is rendering operators as strings.

```
renderDomain :: Domain String
renderDomain = Domain "render" Render.parse renderAlg

renderAlg :: BuiltInOp String -> String
renderAlg (OInt i) = show i
renderAlg (OOString s) = s
renderAlg (OBool b) = show b
renderAlg (OTuple t) = show t
{- ... -}
```

With the render domain definition, we can give pretty printing semantics to operators.

```
pretty Var = @Var
pretty Abs = lambda @Var : @Body
pretty App = @Fun(@Arg)
```

7. Discussion

In this section we discuss the results of our extended exact replication. We start by comparing *iCoLa* and *iCoLa*$^+$ in Section 7.1. The remainder of the discussion focuses on the *iCoLa*$^+$ specific parts.

Table 2: Table demonstrating a view from a session in the *iCoLa*$^+$-shell, constructing several languages with a fixed-set of operators. Columns indicate the operators used during the evaluation and the rows are the languages constructed with (some) operators from the collection. The ● indicates that the operator is used as is; ◓ indicates that an operator is used with glue code; and ○ indicates that an operator is not used.

	Var	Abs	App$_{cbv}$	Addition	Int	Return	Call/cc	If	Throw	Catch
lambda	●	●	●	○	○	○	○	○	○	○
arithmetic	○	○	○	●	●	○	○	○	○	○
exceptions	○	○	○	○	○	○	○	○	●	●
proc	●	◓	●	○	○	●	○	○	○	○
lambda$_{cbn}$	◓	●	◓	○	○	○	○	○	○	○
functional	●	●	●	●	●	○	●	●	●	○
procedural	●	◓	●	●	●	●	○	●	●	●
procedural + functional	●	◓	●	●	●	●	●	●	●	●

7.1. iCoLa vs iCoLa$^+$

Without looking at concrete syntax, we fully replicate the results obtained with *iCoLa*. This is explainable because the formal model implemented by *iCoLa*$^+$ is an extension that fully encompasses the formal model implemented by *iCoLa*. However, when including concrete syntax in the comparison, we do see some decline in operator reuse as indicated in Table 1. Nevertheless, as Table 1 highlights, not all operators required a different concrete syntax definition. This indicates that the introduction of concrete syntax does not nullify the reusability obtained with our approach. Furthermore, during exploration one might opt to not modify the concrete syntax definition of some operators but only do this after the choice of operators is finalized. Nevertheless, concrete syntax can have an impact on language ergonomics. In *iCoLa*, concrete syntax had to be defined separately from *iCoLa*, and is therefore not incremental, and was not integrated within the *iCoLa*-shell. With *iCoLa*$^+$, concrete syntax is defined within *iCoLa*$^+$, and integrated within environment definitions.

7.1.1. eDSL vs DSL

Since both *iCoLa*$^+$ and *iCoLa* are built upon the same foundation, some of their main differences arise in the way users interact with the implementations. *iCoLa* provides an eDSL for interaction, while *iCoLa*$^+$ provides a DSL. The benefits and limitations of this design choice are discussed in this section.

The DSL implementation provides concise definitions of languages and operators by removing boilerplate code. For example in operator definitions, the eDSL requires that the operand locations have type families and are matched in the constructor for the operator. In the DSL this is done automatically, as illustrated in Listing 4 by comparison of the Abs operator.

Definition of semantic functions is rather similar between the implementations, with the main difference that the DSL allows domains to define their own concrete syntax, resulting in more concise definitions. In our comparison this is mostly noted by the fact that the eDSL requires wrapping arguments in lists due to the variable number of arguments funcon constructors can take. In the DSL implementation, this is not needed.

Language definitions in the eDSL and DSL are also rather similar, with some differences in available syntax. For example in the eDSL, list comprehension can be used. Similar expressiveness is obtained with constraint sequencing in the DSL. In addition, several forms of syntactic sugar are available in the DSL that make language definitions slightly more concise.

Another benefit of the DSL is error reporting to the user. The eDSL was based on Template Haskell, and errors resulted in Haskell type errors. With the DSL implementation, we can give domain-specific errors instead and use the terms available in our language, such as `operator`, inside error message.

The DSL does not provide the full power of Haskell, making it more difficult for users to create abstractions, such as function composition or using `where` clauses to improve readability, and languages are not first-class citizens anymore. Difficulty in the creation of abstractions is clearly visible with refinements. In the eDSL, refinement functions can be composed, which is not possible in the DSL implementation because refinements are directly applied on languages instead of taking languages as parameters, resulting in some duplication. Not having languages be first-class citizens makes abstractions like parameterized languages very convoluted. Some of the abstraction capabilities can still be achieved via the extension points provided by *iCoLa*$^+$. Nevertheless, in future work we would like to explore how to enable users to introduce abstractions natively in the DSL. One possible direction to achieve this is by enabling operator inheritance and parameterized refinements. With operator inheritance, operators can be built by composing other operators together and thus inheriting their semantic descriptions. With parameterized refinements, refinements can take a language as a parameter, making composition of refinements possible.

Implementation wise, the DSL implementation required around two to three times as much code as the eDSL, with the remark that the tooling for the external DSL is almost non-existing. So far, we have implemented the *iCoLa*$^+$-shell, similarly to the *iCoLa*-shell. However, no support for debugging, profiling, and editor services are available with the DSL, while the eDSL inherits this from the embedding in Haskell.

To summarize, *iCoLa*$^+$ required a more substantial development effort compared to *iCoLa*. With the choice of a DSL, *iCoLa*$^+$ sacrifices some expressiveness for more concise definitions, better error reporting and more flexibility. In addition, users can define operators and languages without any Haskell knowledge with *iCoLa*$^+$, in contrast to *iCoLa* where Haskell knowledge is required.

7.2. Restrictions and Scalability

With the flexibility our approach provides, language definitions can become unwieldy where it is unclear where operators are exactly assigned to, which operators are part of the language, and how they are affected by specialization code. In our experience, the development is often done in layers, where prototyping is done at the current layer and when done, the layer is fixed. This keeps modifications local and prototyping focused on specific areas. It is important, however, that the first layer is well understood before such a development process can be applied.

In future work, we want to explore tooling that can help in quickly understanding the effects of new operator assignments and language composition. For example, by utilizing visual views of the defined languages, as we have shown in Figure 2. One possible direction is to move away from text-based language engineering towards a visual style, by connecting operators with operand locations by drawing an edge between them. This allows a developer to see the structure of a language easier compared to a text based approach. How this scales to larger languages and how it affects the ergonomics of a developer is something that needs to be investigated. These extensions can be achieved in future work via environment definitions that manage the interaction. Consequently, these extensions require no modifications to the existing *iCoLa*$^+$ implementation.

7.3. Domain definitions and domain fusion

Through a standard example of a pretty printing extensions, we have demonstrated that it is possible to add new domain definitions and modify the *iCoLa*$^+$ DSL this way. With the possibility to add new domains, *iCoLa*$^+$ can also be used as a vessel for the evaluation of new semantic specification languages. Currently, we are in the process of defining domains for static semantics and scope graphs [31]. In future work, we will report on these efforts and investigate how to fuse domain definitions together. With domain fusion, translation functions can utilize other domains. Using information from another domain can result in more efficient specifications, for example, by utilizing typing information.

7.4. Language composition and syntax ambiguity

Through the construction of several languages via composition, we have shown that our approach supports the extension, refinement and unification operators for semantic and syntax, as introduced by Erdweg [11]. However, currently disambiguation can only be achieved by encoding the disambiguation rules inside the structure of the language, which is difficult to manage when composing languages and not always sufficient. As a result, our definitions of the *Imp*, *MiniJava*, and *SIMPLE* languages slightly deviate from the definitions present in the PLanCompS case-study.

We are still exploring the best way to introduce disambiguation inside *iCoLa*$^+$. The easiest approach is to extend the concrete syntax definition with many of the combinators available in the used GLL library. Alternatively, we can take a similar approach as used in the SDF formalism, but then adapted to GLL where applicable. Another possible alternative is to use pattern matching to do disambiguation [32]. In addition, we aim to investigate in future work whether the debug friendliness of GLL parsing can aid disambiguation in an interactive style, such that it can be integrated into the design process.

7.5. Extensible built-in operators

In the current implementation we have decided upon a selection of built-in operators. This selection is based on the requirement we found during the construction of the several languages and fragments demonstrated in this paper. The current selection will not be enough for all possible language definitions. Currently, addition of new built-in operators requires extension to the core of *iCoLa*$^+$. Since built-in operators correspond to lexemes of a language, we expect that a better system for the addition of built-in operators is required. For now, we see two ways to achieve this. The first option is to add a catch-all built-in operator that bypasses the Haskell type system and then require domain definitions to handle these built-in operators. This approach is easy to implement in the current system and easy to understand from a users perspective. Since this approach bypasses the Haskell type system, it can result in run-time errors when a domain definition needs to handle an operator for which it has no handler. The second option is to use data-types à la carte [4]. With data types à la carte, built-in operators can be defined independently and composed together. The composition is then given to *iCoLa*$^+$ together with a lexing definition for the operator. Algebra definitions then become type-class instances which need to be implemented for all the built-in operators supported by the domain, which is checked by the Haskell compiler. This approach requires a bigger engineering effort to achieve in the current implementation, makes built-in operators mutable, and results in more complex operator definitions.

Listing 4 Comparison of the definition for the Abs operator in the eDSL (left) and the DSL (right).

```
data Abs u t where
    Abs :: IsTrue (AbsBody t) => String -> u t -> Abs u AbsType
type family AbsBody t
```

```
operator Abs : Var Body
```

7.6. Exploration within iCoLa$^+$

iCoLa$^+$ support exploratory programming as a first-class citizen via its design as a sequential language and embedding within existing tooling. Exploring multiple ideas via the creation of many variants and discarding futile paths is fully supported via the `jump` and `revert` meta-commands present in *iCoLa*$^+$. Although we currently only provide an exploratory REPL, in future work we want to investigate alternative interfaces with an explicit focus on exploratory language development, which can be defined as environment definitions. Towards this idea, we also aim to utilize the presented implementation to investigate exploratory patterns within language development to further guide interface design. In addition, within the *iCoLa*$^+$-shell a user can commit to a language and experiment with the object language within an object REPL. However, after exiting the object REPL, the object REPL session is lost. But, within an exploration session, a user might want to compare two languages by their usage within the object REPL. Currently, that is possible by scrolling back in the REPL, but there is no real support for it, yet. In future work, we aim to investigate how to support retention of the object REPL sessions within the exploration session in an ergonomic and efficient manner.

7.7. Limitations and Threats to validity

The primary evaluation of our approach is based on the semantic domain which uses funcons. This limits the scope to the class of languages which can have their semantics expressed in funcons. Since the funcon library is open-ended [7], this class is mostly characterized by the fixed set of semantic entities. Nevertheless, a variety of languages already have their semantics expressed in funcon terms [33, 19]. Furthermore, our approach is extensible through new semantic domains. An interesting foundation for an alternative semantic domain is algebraic effects and handlers [34, 35], which provide a mathematical approach for reasoning about effects in programming languages and support composition [36, 37].

The usage of catamorphisms to guide the translation from initial algebra to semantic algebra constitutes a fundamental functionality to our approach. Nevertheless, the initial algebra semantics presents a unified approach to formal semantics of programming languages [12], and therefore supports different approaches. However, this choice of abstraction puts certain restrictions on the translation functions used in our approach, which can affect the manner in which semantic domains are defined.

8. Related work

Developing languages via some form of composition is supported by a wide variety of language-development environments [38, 39, 40, 41, 42]. Erdweg et al. [11], performed a systematic evaluation of existing environments and their support for the different forms composition (extension and unification). Out of the considered environments, only JastAdd [43], which is an environment for the construction of Java like languages, supported unification at the semantic level. For syntax, both Spoofax [38] and SugarJ [40] support unification. To handle ambiguous grammars, Spoofax and SugarJ use the SDF formalism [44]. Our approach supports unification at both the syntax and semantic level, with the remark that disambiguation at the syntax level is minimal. Much of the syntax available in concrete syntax definitions of *iCoLa*$^+$ are influenced by SDF. In contrast to SDF, we use GLL parsing instead of scannerless Generalized LR parsing.

Lisa [45] is a full-fledged interactive environment for programming language development based on attribute grammars with support for incremental language development [46] via multiple attribute grammar inheritance [47]. Lisa also supports visual based development of programming languages. Compared to our approach, no distinction between operators and where operators are used is made.

Melange [48] is a meta-language involving meta-models and aspect oriented programming. It uses aspects to implement the semantics of languages, and supports both extension and unification. Our operator specialization closely resembles the idea of aspects as seen in Melange. Compared to our approach, Melange makes no distinction between operator semantics and operator specialization; does not make a distinction between operator definitions and language definitions; and operators are not immutable, instead a renaming mechanism is provided to solve conflicting abstract syntax. Multi-level modeling [49] supports more than two meta-modeling levels and has been used in language development to achieve extensible meta-models via linguistic extensions [50] and by specializing meta-models to specific domains via instantiation [51]. However, to support optionality of language primitives (closed variability), multi-level modeling needs to be combined with the product lines approach [52]. Within our approach, such optionality is achieved via refinements on language definitions. Perspectives [53] are a layer above the model layer and are used to describe the relations between multiple languages and consistency requirements among them, or to exclude language concepts from the model layer. The approach has similar characteristics as Melange, with the addition that it enforces consistency requirements among languages. Our approach explicitly has few restrictions to promote the exploration phase. However, outside the exploration phase, more refined restrictions might be beneficial.

In Feature-oriented programming [54], a system is decomposed in the features it provides. This style of programming aims to increase structure, reuse and variation by making features user configurable such that a system can be developed by picking and configuring the correct features. Neverlang [55] is a Java-based development environment with support for language unification, modeled around the idea of feature-oriented programming. Neverlang uses evaluation phases for semantic specifications and supports evaluation phases depending on other phases. Compared to our approach, syntax definitions are not restricted, resulting in dependencies among syntactic definitions, and no distinction between operator definitions and operator usage is made.

Software product lines [56] is a development paradigm that models the software development process as a product line, where a system is constructed by selecting components from a repository, adapting the components to the use case, and integrating the components together. Compared to feature-oriented programming, software product lines focus on similarities between systems, also known as families. This gives a high variability where variants of systems can be quickly created. Feature-oriented programming can be used to implement software product lines, which is done by AiDE [57]. AiDE provides an environment for language-development based on software product lines by building an environment on top of Neverlang [55]. Besides AiDE, there are several other environments integrating software product lines in the context of language development — also known as language product lines [58].

A focus on language families [59], a set of related languages, is inherent in the language product lines style of development. As a result, the variability of these systems is high, enabling the construction of a wide variety of languages in an incremental manner. However, because the focus is on language families, there is a restriction on the structure of the different variations. Nevertheless, correctness of model properties can be efficiently checked, which opens the door to promote the variability offered by product lines to more areas such as model editors and code generators [60]. Language product lines have been combined with multi-level modeling at the (abstract) syntax level [61] to enable both extension and selection based on a feature model. The approach supports bottom-up extensions where a meta-model is extended from below, which can be useful during the exploration process. To also enable modularity at the semantic level, graph transformation have been used [62]. With graph transformations, consistency of semantic constraints can be enforced among the languages within a language family. Our approach achieves modularity at the semantic level by supporting the introduction of new semantic domains and the introduction of new operators accompanied by semantic translation functions. Although our approach essentially describes a constraint graph, as indicated by Figure 2, we have not yet explored whether this can be utilized to enforce constraints without affecting the exploration capabilities, or aid the exploration process.

Concern-oriented language development [63] moves away from the family constraint by combining different modularity approaches at the language development level via so called concerns: reusable piece of language artifacts. Concerns have a variation interface, a customization interface, and a usage interface. The variation interface represents configurable components and the customization interface describes how a concern can be integrated into a different context. The exploratory capabilities of a concern are thus determined by the flexibility of these interfaces and the inherit restrictions present in concern definitions. The ideas of concern-oriented language development are used to reuse language components that are textual, external, and translational [64]. The approach uses specific composition operator to ensure compatibility within the used technologies, which makes it impossible to remove parts of a language.

9. Conclusion

In this paper we introduced *iCoLa*$^+$, an extensible meta-language aimed at improving the language design process via rapid prototyping with reusable components and incremental programming. *iCoLa*$^+$ extends the *iCoLa* meta-language by adding support for concrete syntax, by providing a DSL for language definitions, and by supporting an arbitrary amount of semantic domains. The *iCoLa*$^+$ implementation is extensible via Haskell defined environment definitions and domain definitions. Environment definitions determine how users interact with *iCoLa*$^+$ and the defined languages. Domain definitions determine the capabilities of semantic translation functions.

By constructing several languages with our approach, we have demonstrated to which extent our approach simplifies the construction of new languages as well as variants of existing languages. Through the construction of the *iCoLa*$^+$-shell and by adding a new domain definition, we have shown the possibilities of extending *iCoLa*$^+$. The flexibility provided by *iCoLa*$^+$ makes it easy to modify existing language design choices, but also increases the difficulty of tracking the precise composition of languages when applied at (large) scale. In addition, disambiguation of concrete syntax is only supported in a limited form. Methods to improve *iCoLa*$^+$ in these regards are to be explored in future work.

References

[1] D. Frölich, L. T. van Binsbergen, iCoLa: A Compositional Meta-language with Support for Incremental Language Development, in: B. Fischer, L. Burgueño, W. Cazzola (Eds.), Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Auckland,

New Zealand, December 6-7, 2022, ACM, 2022, pp. 202–215. doi:10.1145/3567512.3567529.

[2] M. P. Ward, Language-oriented programming, Softw. Concepts Tools 15 (4) (1994) 147–161.

[3] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, J. van der Woning, The state of the art in language workbenches - conclusions from the language workbench challenge, in: M. Erwig, R. F. Paige, E. V. Wyk (Eds.), Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings, Vol. 8225 of Lecture Notes in Computer Science, Springer, 2013, pp. 197–217. doi:10.1007/978-3-319-02654-1_11.

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

[5] B. C. d. S. Oliveira, W. R. Cook, Extensibility for the masses - practical extensibility with object algebras, in: J. Noble (Ed.), ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings, Vol. 7313 of Lecture Notes in Computer Science, Springer, 2012, pp. 2–27. doi:10.1007/978-3-642-31057-7_2.

[6] B. van den Berg, T. Schrijvers, C. B. Poulsen, N. Wu, Latent effects for reusable language components, in: H. Oh (Ed.), Programming Languages and Systems, Springer International Publishing, Cham, 2021, pp. 182–201. doi:10.1007/978-3-030-89051-3_11.

[7] L. T. van Binsbergen, P. D. Mosses, N. Sculthorpe, Executable component-based semantics, J. Log. Algebraic Methods Program. 103 (2019) 184–212. doi:10.1016/j.jlamp.2018.12.004.

[8] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, J. D. Team, Jupyter notebooks - a publishing format for reproducible computational workflows, in: F. Loizides, B. Schmidt (Eds.), Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, June 7-9, 2016, IOS Press, 2016, pp. 87–90. doi:10.3233/978-1-61499-649-1-87.

[9] M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM Comput. Surv. 37 (4) (2005) 316–344. doi:10.1145/1118890.1118892.

[10] P. Hudak, Building domain-specific embedded languages, ACM Comput. Surv. 28 (4es) (1996) 196. doi:10.1145/242224.242477.

[11] S. Erdweg, P. G. Giarrusso, T. Rendel, Language composition untangled, in: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, LDTA '12, ACM, 2012. doi:10.1145/2427048.2427055.

[12] J. A. Goguen, J. W. Thatcher, E. G. Wagner, J. B. Wright, Initial algebra semantics and continuous algebras, Journal of the ACM 24 (1) (1977) 68–95. doi:10.1145/321992.321997.

[13] P. D. Mosses, Denotational semantics, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, Elsevier and MIT Press, 1990, pp. 575–631. doi:10.1016/b978-0-444-88074-1.50016-0.

[14] E. Meijer, M. M. Fokkinga, R. Paterson, Functional programming with bananas, lenses, envelopes and barbed wire, in: J. Hughes (Ed.), Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings, Vol. 523 of Lecture Notes in Computer Science, Springer, 1991, pp. 124–144. doi:10.1007/3540543961_7.

[15] P. D. Mosses, Software meta-language engineering and CBS, Journal of Computer Languages 50 (2019) 39–48. doi:10.1016/j.jvlc.2018.11.003.

[16] P. D. Mosses, N. Sculthorpe, L. T. Van Binsbergen, Funcons-Beta, online GitHub repository (2021).
URL https://plancomps.github.io/CBS-beta/Funcons-beta/

[17] M. B. Kery, B. A. Myers, Exploring exploratory programming, in: A. Z. Henley, P. Rogers, A. Sarma (Eds.), 2017 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2017, Raleigh, NC, USA, October 11-14, 2017, IEEE Computer Society, 2017, pp. 25–29. doi:10.1109/VLHCC.2017.8103446.

[18] P. Rein, S. Ramson, J. Lincke, R. Hirschfeld, T. Pape, Exploratory and live, programming and coding - A literature study comparing perspectives on liveness, Art Sci. Eng. Program. 3 (1) (2019) 1. doi:10.22152/programming-journal.org/2019/3/1.

[19] L. T. van Binsbergen, M. Verano Merino, P. Jeanjean, T. van der Storm, B. Combemale, O. Barais, A Principled Approach to REPL Interpreters, ACM, 2020, pp. 84–100. doi:10.1145/3426428.3426917.

[20] D. Frolich, L. T. van Binsbergen, A generic back-end for exploratory programming, in: V. Zsók, J. Hughes (Eds.), Trends in Functional Programming - 22nd International Symposium, TFP 2021, Virtual Event, February 17-19, 2021, Revised Selected Papers, Vol. 12834 of Lecture Notes in Computer Science, Springer, 2021, pp. 24–43. doi:10.1007/978-3-030-83978-9_2.

[21] L. T. van Binsbergen, D. Frölich, M. V. Merino, J. Lai, P. Jeanjean, T. van der Storm, B. Combemale, O. Barais, A language-parametric approach to exploratory programming environments, in: B. Fischer, L. Burgueño, W. Cazzola (Eds.), Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Auckland, New Zealand, December 6-7, 2022, ACM, 2022, pp. 175–188. doi:10.1145/3567512.3567527.

[22] M. Cimini, Languages as first-class citizens (vision paper), in: D. J. Pearce, T. Mayerhofer, F. Steimann (Eds.), Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018, ACM, 2018, pp. 65–69. doi:10.1145/3276604.3276983.

[23] E. Scott, A. Johnstone, GLL parsing, Electronic Notes in Theoretical Computer Science 253 (7) (2010) 177 – 189, proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009). doi:10.1016/j.entcs.2010.08.041.

[24] L. T. van Binsbergen, E. Scott, A. Johnstone, Purely functional GLL parsing, J. Comput. Lang. 58 (2020) 100945. doi:10.1016/j.cola.2020.100945.

[25] P. Klint, A meta-environment for generating programming environments, ACM Trans. Softw. Eng. Methodol. 2 (2) (1993) 176–201. doi:10.1145/151257.151260.

[26] G. Rosu, T. Serbanuta, An overview of the K semantic framework, J. Log. Algebraic Methods Program. 79 (6) (2010) 397–434. doi:10.1016/j.jlap.2010.03.012.

[27] G. Rosu, T. Serbanuta, K overview and SIMPLE case study, Electron. Notes Theor. Comput. Sci. 304 (2014) 3–56. doi:10.1016/j.entcs.2014.05.002.

[28] A. W. Appel, J. Palsberg, Modern Compiler Implementation in Java, 2nd edition, Cambridge University Press, 2002.

[29] L. T. van Binsbergen, Funcons for HGMP: the fundamental constructs of homogeneous generative meta-programming (short paper), in: E. V. Wyk, T. Rompf (Eds.), Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2018, Boston, MA, USA, November 5-6, 2018, ACM, 2018, pp. 168–174. doi:10.1145/3278122.3278132.

[30] D. Frölich, L. T. van Binsbergen, A generic back-end for exploratory programming, in: The 22nd International Symposium on Trends in Functional Programming (TFP 2021), Vol. 12834 of LNCS, Springer, 2021. doi:10.1007/978-3-030-83978-9_2.

[31] P. Neron, A. P. Tolmach, E. Visser, G. Wachsmuth, A theory of name resolution, in: J. Vitek (Ed.), Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, Vol. 9032 of Lecture Notes in Computer Science, Springer, 2015, pp. 205–231.

doi:10.1007/978-3-662-46669-8_9.

[32] A. Afroozeh, J. Bach, M. van den Brand, A. Johnstone, M. Manders, P. Moreau, E. Scott, Island grammar-based parsing using GLL and tom, in: K. Czarnecki, G. Hedin (Eds.), Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers, Vol. 7745 of Lecture Notes in Computer Science, Springer, 2012, pp. 224–243. doi:10.1007/978-3-642-36089-3_13.

[33] P. D. Mosses, Fundamental constructs in programming languages, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation - 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2021, Rhodes, Greece, October 17-29, 2021, Proceedings, Vol. 13036 of Lecture Notes in Computer Science, Springer, 2021, pp. 296–321. doi:10.1007/978-3-030-89159-6_19.

[34] G. D. Plotkin, J. Power, Adequacy for algebraic effects, in: F. Honsell, M. Miculan (Eds.), Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings, Vol. 2030 of Lecture Notes in Computer Science, Springer, 2001, pp. 1–24. doi:10.1007/3-540-45315-6_1.

[35] G. D. Plotkin, M. Pretnar, Handlers of algebraic effects, in: G. Castagna (Ed.), 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, Vol. 5502 of Lecture Notes in Computer Science, Springer, 2009, pp. 80–94. doi:10.1007/978-3-642-00590-9_7.

[36] C. van der Rest, C. B. Poulsen, Towards a language for defining reusable programming language components - (project paper), in: W. Swierstra, N. Wu (Eds.), Trends in Functional Programming - 23rd International Symposium, TFP 2022, Virtual Event, March 17-18, 2022, Revised Selected Papers, Vol. 13401 of Lecture Notes in Computer Science, Springer, 2022, pp. 18–38. doi:10.1007/978-3-031-21314-4_2.

[37] E. C. Brady, Programming and reasoning with algebraic effects and dependent types, in: G. Morrisett, T. Uustalu (Eds.), ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013, ACM, 2013, pp. 133–144. doi:10.1145/2500365.2500581.

[38] L. C. L. Kats, E. Visser, The spoofax language workbench: rules for declarative specification of languages and ides, in: W. R. Cook, S. Clarke, M. C. Rinard (Eds.), Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA, ACM, 2010, pp. 444–463. doi:10.1145/1869459.1869497.

[39] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, M. Felleisen, Languages as libraries, in: M. W. Hall, D. A. Padua (Eds.), Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011, ACM, 2011, pp. 132–141. doi:10.1145/1993498.1993514.

[40] S. Erdweg, T. Rendel, C. Kästner, K. Ostermann, Sugarj: library-based syntactic language extensibility, in: C. V. Lopes, K. Fisher (Eds.), Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011, ACM, 2011, pp. 391–406. doi:10.1145/2048066.2048099.

[41] M. Voelter, K. Solomatov, Language modularization and composition with projectional language workbenches illustrated with mps. software language engineering (2010).

[42] L. Tratt, Domain specific language implementation via compile-time meta-programming, ACM Trans. Program. Lang. Syst. 30 (6) (2008) 31:1–31:40. doi:10.1145/1391956.1391958.

[43] T. Ekman, G. Hedin, The jastadd extensible java compiler, in: R. P. Gabriel, D. F. Bacon, C. V. Lopes, G. L. S. Jr. (Eds.), Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada, ACM, 2007, pp. 1–18. doi:10.1145/1297027.1297029.

[44] J. Heering, P. R. H. Hendriks, P. Klint, J. Rekers, The syntax definition formalism SDF - reference manual, ACM SIGPLAN Notices 24 (11) (1989) 43–75. doi:10.1145/71605.71607.

[45] M. Mernik, M. Lenic, E. Avdicausevic, V. Zumer, LISA: an interactive environment for programming language development, in: R. N. Horspool (Ed.), Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings, Vol. 2304 of Lecture Notes in Computer Science, Springer, 2002, pp. 1–4. doi:10.1007/3-540-45937-5_1.

[46] M. Mernik, V. Zumer, Incremental programming language development, Comput. Lang. Syst. Struct. 31 (1) (2005) 1–16. doi:10.1016/j.cl.2004.02.001.

[47] M. Mernik, M. Lenic, E. Avdicausevic, V. Zumer, Multiple attribute grammar inheritance, Informatica (Slovenia) 24 (3) (2000).

[48] T. Degueule, B. Combemale, A. Blouin, O. Barais, J. Jézéquel, Melange: a meta-language for modular and reusable development of dsls, in: R. F. Paige, D. D. Ruscio, M. Völter (Eds.), Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015, ACM, 2015, pp. 25–36. doi:10.1145/2814251.2814252.

[49] C. Atkinson, T. Kühne, The essence of multilevel metamodeling, in: M. Gogolla, C. Kobryn (Eds.), UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings, Vol. 2185 of Lecture Notes in Computer Science, Springer, 2001, pp. 19–33. doi:10.1007/3-540-45441-1_3.

[50] J. de Lara, E. Guerra, Deep meta-modelling with metadepth, in: J. Vitek (Ed.), Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings, Vol. 6141 of Lecture Notes in Computer Science, Springer, 2010, pp. 1–20. doi:10.1007/978-3-642-13953-6_1.

[51] J. de Lara, E. Guerra, J. S. Cuadrado, Model-driven engineering with domain-specific meta-modelling languages, Softw. Syst. Model. 14 (1) (2015) 429–459. doi:10.1007/s10270-013-0367-z.

[52] J. de Lara, E. Guerra, Multi-level model product lines - open and closed variability for modelling language families, in: H. Wehrheim, J. Cabot (Eds.), Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Vol. 12076 of Lecture Notes in Computer Science, Springer, 2020, pp. 161–181. doi:10.1007/978-3-030-45234-6_8.

[53] H. Ali, G. Mussbacher, J. Kienzle, Perspectives to promote modularity, reusability, and consistency in multi-language systems, Innov. Syst. Softw. Eng. 18 (1) (2022) 5–37. doi:10.1007/s11334-021-00425-3.

[54] S. Apel, C. Kästner, An overview of feature-oriented software development, J. Object Technol. 8 (5) (2009) 49–84. doi:10.5381/jot.2009.8.5.c5.

[55] W. Cazzola, Domain-specific languages in few steps - the neverlang approach, in: T. Gschwind, F. D. Paoli, V. Gruhn, M. Book (Eds.), Software Composition - 11th International Conference, SC@TOOLS 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings, Vol. 7306 of Lecture Notes in Computer Science, Springer, 2012, pp. 162–177. doi:10.1007/978-3-642-30564-1_11.

[56] P. Clements, L. Northrop, Software product lines, Addison-Wesley Boston, 2002.

[57] T. Kühn, W. Cazzola, D. M. Olivares, Choosy and picky: configuration of language product lines, in: D. C. Schmidt (Ed.), Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015, ACM, 2015, pp. 71–80. doi:10.1145/2791060.2791092.

[58] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, B. Baudry, Leveraging software product lines engineering in the development of external dsls: A systematic literature review, Comput. Lang. Syst. Struct. 46 (2016) 206–235. doi:10.1016/j.cl.2016.09.004.

[59] J. Liebig, R. Daniel, S. Apel, Feature-oriented language families: A case study, in: Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, 2013, pp. 1–8.

[60] E. Guerra, J. de Lara, M. Chechik, R. Salay, Property satisfiability analysis for product lines of modelling languages, IEEE Trans. Software Eng. 48 (2) (2022) 397–416. doi:10.1109/TSE.2020.2989506.

[61] J. de Lara, E. Guerra, Language family engineering with product lines of multi-level models, Formal Aspects Comput. 33 (6) (2021) 1173–1208. doi:10.1007/s00165-021-00554-3.

[62] J. de Lara, E. Guerra, P. Bottoni, Modular language product lines: a graph transformation approach, in: E. Syriani, H. A. Sahraoui, N. Bencomo, M. Wimmer (Eds.), Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, Montreal, Quebec, Canada, October 23-28, 2022, ACM, 2022, pp. 334–344. doi:10.1145/3550355.3552444.

[63] B. Combemale, J. Kienzle, et al., Concern-oriented language development (COLD): fostering reuse in language engineering, Comput. Lang. Syst. Struct. 54 (2018) 139–155. doi:10.1016/j.cl.2018.05.004.

[64] A. Butting, J. Pfeiffer, B. Rumpe, A. Wortmann, A compositional framework for systematic modeling language reuse, in: E. Syriani, H. A. Sahraoui, J. de Lara, S. Abrahão (Eds.), MoDELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, ACM, 2020, pp. 35–46. doi:10.1145/3365438.3410934.