iCoLa: A Compositional Meta-language with Support for Incremental Language Development

Damian Frölich
dfrolich@acm.org
Informatics Institute, University of Amsterdam
Amsterdam, the Netherlands

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org
Informatics Institute, University of Amsterdam
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*, a meta-language supporting incremental (meta-)programming based on reusable components. In our implementation of *iCoLa*, languages are first-class citizens, providing the full power of the host-language (Haskell) to compose and manipulate languages. We demonstrate *iCoLa* through the construction of the *Imp*, *SIMPLE*, and *MiniJava* languages via the composition and restriction of language fragments and demonstrate the variability of our approach through the construction of several languages using a fixed-set of operators.

CCS Concepts: • **Software and its engineering → Domain specific languages**.

Keywords: language composition, prototyping, domain-specific languages, meta-language, funcons

ACM Reference Format:
Damian Frölich and L. Thomas van Binsbergen. 2022. *iCoLa*: A Compositional Meta-language with Support for Incremental Language Development. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), December 06–07, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3567512.3567529

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SLE '22, December 06–07, 2022, Auckland, New Zealand
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9919-7/22/12…$15.00
https://doi.org/10.1145/3567512.3567529

1 Introduction

High-level programming languages increase programmer productivity, program safety, program correctness, and maintainability, among other qualities. Language-Oriented Programming (LOP) [48] is a programming paradigm utilizing the advantages of higher-level programming by developing new languages specialized to the problem domain at hand by offering 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, and to guarantee performance.

Language workbenches and meta-languages are created to support language *engineering* efforts [14], often delivering tooling such as IDEs for object languages for free. Techniques for the modular and reusable specification of syntax [39] and semantics [8, 23, 45] have been developed. Reusable fundamental constructs (funcons) have been identified for general-purpose languages [43] to describe the semantics of (new) languages in terms of existing components.

Language *design* is also a time-consuming effort as 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). For example, 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 an iterative process in which language developers and domain experts continue to reflect on the existing design.

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. In this work we apply incremental programming (also) in the context of language development to support rapid prototyping of languages.

In this paper we introduce *iCoLa*, a meta-language focused on the language design process by supporting reusable components and incremental language development to enable rapid prototyping of languages. Concretely, we make the following contributions:

- We present a new approach to defining languages (section 3) which supports unconstrained composition of languages through incremental programming.
- The approach is implemented as an Embedded Domain-Specific Language (EDSL) with (Template) Haskell as the host language (section 4). The full power of Haskell is available to define (operations over) languages, as languages are first-class citizens [6].
- We demonstrate reusability and incrementality through case-studies (section 5) and relate our approach to existing meta-languages by applying Erdweg's evaluation framework [12] (section 6).

2 Background

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

Initial algebra semantics. The initial algebra semantics of Goguen et al. [16], concisely described by Mosses in [31], 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 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 [26]. 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.

data types à la carte. As a solution to the expression problem [47], *data types à la carte* [39] provides a method for assembling data types and functions from individual components to form signatures, an initial algebra for every signature, and evaluation algebras, respectively.

With the approach, data types are defined as functors and combined by taking the functor co-product, which is, again, a functor. Using this technique, a simple integer addition language can be defined as follows.

```
data Val a = Val Int
data Add a = Add a a

data (f :+: g) e = Inl (f e) | Inr (g e)
```

The `:+:` operator implements the functor co-product.

To allow nested expressions and cross-usage of constructs, the recursive knot must be tied. This is achieved by taking the fix-point of the co-product functor [26]: `data Term f = In (f (Term f))`.

Although nested expressions are now supported, we still need to place our expressions on the correct side of our co-product. To automate this, automatic injections into the co-product type via type classes are used.

```
class (Functor sub, Functor sup) => sub :<: sup where
    inj :: sub a -> sup a

instance f :<: f where
    inj = id

instance f :<: g where
    inj = Inl

instance (f :<: g) => f :<: (g :+: h) where
    inj = Inr . inj
```

The `:<:` operator defines a typing relation such that if `f :<: g`, it means that f is subsumed by g, i.e. values of type f can be constructed as part of values of type g.

Since both the data types and the functor co-product are functors, catamorphisms can be used to operate on the composition of data types.

```
foldTerm :: Functor f => (f a -> a) -> Expr f -> a
foldTerm f (In t) = f (fmap (foldTerm f) t)
```

The first argument to the `foldTerm` function is called an algebra and defines how the resulting value is constructed. To enable the extension of new cases at the function level, the approach implements algebras via type classes. Type classes are used because they provide ad-hoc polymorphism and are open for extension via instance definitions.

The comp-data library [3] is a comprehensive Haskell library implementing the data types à la carte approach with some extensions, including support for Generalized Algebraic Data Types (GADTs) [19], contexts, and automatically deriving several type class instances using Template Haskell. The library also supports higher-order functors [19] to implement signatures. A consequence of using higher-order functors is that algebras become natural transformations instead of functions, affecting the kind of the algebra.

Funcons. The component-based approach to operational semantics presented in [32] is centered around reusable definitions of the *fun*damental *con*structs of (general-purpose) programming languages – referred to as *funcons* for short. As explained in [43], '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 [33] 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 EDSL.

Language composition evaluation framework. Erdweg et al. provide a framework for discussing and comparing meta-languages, tools and formalisms that support various forms of incremental language development [12]. 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 the basis for our evaluation.

Template Haskell. Template Haskell is a Haskell extension permitting compile-time meta-programming [38]. With Template Haskell, users can write programs that transform programs. For instance, it is useful when generating boilerplate code or to perform calculations at compile-time to improve run-time performance. The extension provides several facilities to inspect and operate on Haskell programs, including a quotation monad that enables reification of Haskell names, giving the programmer access to the internal representation of the compiler. Names are obtained by using prefix quotes, with one quote operating in the expression context and two quotes operate in the type context. The `$` construct is used to evaluate, or splice, a Template Haskell expression. For example, the following splice derives something based on a name obtained from the expression level and a name obtained from the type level: `$(derive 'f ''Bool)`. In this example, `f` is a function and its name is obtained by prefixing it with single quote, and `Bool` is a type and its name is obtained by prefixing it with double quotes. These obtained names can now be reified to get the internal Haskell representation of the underlying constructs.

A Principled Approach to REPL Interpreters. Previous work [44] 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. The paper defines sequential languages as languages in which we can take any two programs and sequence these programs 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. There is the assumption that an interpreter I assigns semantics to a program as a function over configurations representing execution context, i.e. $I(p) : \Gamma \rightarrow \Gamma$ for some set of configurations Γ. Visually, sequentiality means that the diagram in Figure 1 commutes, where $p_1; p_2$ is the sequence containing the programs p_1 and p_2.

$$\begin{tikzcd}
\bullet \arrow[r, "I(p_1)"] \arrow[dr, "I(p_1;p_2)"'] & \bullet \arrow[d, "I(p_2)"] \\
 & \bullet
\end{tikzcd}$$

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 [21], and even exploratory programming environments [15], 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 Compositional definitions

In this section we describe our approach to language development conceptually. 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 defining the abstract syntax of a language is to give 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,

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

and App_O are operators (as indiciated by the subscript) and *String* and *Expr* are sorts.

$$\begin{aligned} Var_O &: String \longrightarrow Expr \\ Abs_O &: String \times Expr \longrightarrow Expr \\ App_O &: Expr \times Expr \longrightarrow Expr \end{aligned}$$

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:

$$\begin{aligned} Var_O &: VarVar \\ Abs_O &: AbsVar \times AbsBody \\ App_O &: AppAbs \times AppArg \end{aligned}$$

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 constraint indicates that strings serve as identifiers in both variable references and abstractions:

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

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 Var_O operator can be used as the body of an abstraction:

$$Var_O \in AbsBody$$

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 $\{Var_O, Abs_O, App_O\}$ and $\{AbsBody, AppAbs, AppArg\}$. 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*:

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

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

$$\begin{aligned} Var_O &\in Expr \\ App_O &\in Expr \\ Abs_O &\in Expr \end{aligned}$$

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.

$$\begin{aligned} Add_O &: AddLeft \times AddRight \\ Add_O &\in Expr \end{aligned}$$

This definition adds Add_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:

$$\begin{aligned} Integer &\subseteq AddLeft \\ Integer &\subseteq AddRight \\ Integer &\subseteq Expr \\ Add_O &\in AddRight \end{aligned}$$

These constraints express that integer literals can appear as operands of *Add* in both positions. However, since the *Add* operator is only added to *AddRight*, the constraints allow only nested occurrences of *Add* 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 $Add_O \in Expr$ and $Integer \subseteq Expr$ — 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].

$$\begin{aligned} Var_{\mathcal{F}}(lit) &= \textbf{bound string } lit \\ Abs_{\mathcal{F}}(x, b) &= \textbf{function closure scope}(\\ &\qquad \textbf{bind}(\textbf{string } x, \textbf{given}), b) \\ App_{\mathcal{F}}(abs, arg) &= \textbf{apply}(abs, arg) \end{aligned}$$

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 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.

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

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_O we need to glue it into our C-like language, as below.[3]

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

In the example, we perform a sub-sort declaration, linking the *CExpr* sort — containing the C expression operators — to the *IfCond* operand. As part of that declaration, we define glue code which is only applied when the translated operator is part of the *CExpr* group, since glue code is conditional. Furthermore, the glue code glues the result of the translation function of the operator on which glue code is defined. Thus in our example, $CExpr_{\mathcal{F}}$ is the result of the translation function associated with the sort *CExpr*, which is implicitly defined in terms of the translation functions given for the operators contained in the sort *CExpr*, i.e. the catamorphism.

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 *IfCond* and removing C-like expressions from the language does not leave any stale glue code around.

[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*)).

[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 can be easily done by dispatching on the type of the current value.

3.4 Language definition

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 $F(o) : \mathcal{F}^{|o|} \to \mathcal{F}$ 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 $O \times \mathcal{F} \to \mathcal{F}$ 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. $I_t : \mathcal{F} \to \mathcal{F}$ 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 introduce operand names. Instead, operand positions 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 unambiguous 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. For example, in a REPL, returned values might be printed and unhandled exceptions are caught and displayed while not resulting in termination of the REPL. This behavior is however not preferred when not running in a REPL-like environment. With the initialization functions, this behavior can be encoded as an extension on a language, creating a clear distinction between the different languages.

3.5 Language composition

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

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

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.

While in our definitions, languages are defined in terms of some set of operators, this does not restrict the incrementality we provide nor does it prevent new operators from being introduced. New operators can be added at any time, because operators and their semantics are compositional as well, as inherited from the conceptual model of data types à la carte and our usage of initial algebra semantics. The compositional nature provides enough to support the incremental aspect. 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. Since every incremental step is then evaluated as a composition from the starting point, we do assume that the interpretation of this composition scales and is not much slower than just doing the incremental step.

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. Because both operators and languages are compositional, from a users perspective there is no difference, and language and operator definitions can be freely mixed. This, again, is a result of our approach of achieving incrementality via composition.

4 Implementation

In this section we demonstrate an EDSL in Haskell of the approach introduced in the previous section. The EDSL is partly embedded in Haskell and partly embedded in Template Haskell. In case definitions in the EDSL must be given in terms of Template Haskell, we first give the Haskell definition and after the corresponding encoding in terms of Template Haskell, which demonstrates to which Haskell expression the Template Haskell encoding is evaluated by our approach.

4.1 Operators

Operators are implemented as GADTs with two type parameters, `o` and `t`, corresponding to the set of operators, which is needed for injection into the co-product type, and a so-called meta-type (explained below) of the operator, respectively. GADTs are needed to support the delayed decision regarding sort-constraints via class instances in the constructor definition of the operator. To illustrate, we take the definition for $Abs_{\mathcal{O}}$ as an example.

```
data Abs u t where
    Abs :: IsTrue (AbsBody t)1a =>
        String -> u t1b -> Abs u AbsType2a
```

```
type family AbsBody1c t
data AbsType2b
```

The sort-constraint (1a) enforces that the second parameter (1b) is assigned to the `AbsBody` sort, and sorts are implemented as type-families (1c). Because we carry around the meta-type in arguments (1b) to enforce sort-constraints, type `u` is a type taking a type as a parameter, i.e., it is a functor. Consequently, our operators are higher-order functors. In addition, every operand is assigned a meta-type (2a), which are implemented as empty data types (2b) — data types without constructors. Furthermore, `IsTrue` is a type class for which only one instance is defined — the instance for the type-level boolean `True`.

```
class IsTrue bool
instance IsTrue True
data True
```

To encode $Abs_{\mathcal{O}} \in AbsBody$ — the assignment of the abstraction operator to the body of abstractions — we define the meta-type of the operator as an instance evaluating to `True` of the *AbsBody* type family.

```
type instance AbsBody AbsType = True
```

To retain adaptability of operator assignments, we delay the instantiation of such instances by defining them in terms of Template Haskell.

```
(''AbsType, ''AbsBody) :: OperatorAssignment
type OperatorAssignment = (MetaType, Sort)
type MetaType - Name
type Sort = Name
```

Auxiliary sorts are also implemented using type families. For example, our earlier convenience sort, `Expr`, is defined as follows.

```
type family Expr t
```

As such, instances can be added to an auxiliary sort with an operator assignment.

To perform sub-sort assignments, a sort is linked to another sort, again in terms of template Haskell. Thus, the encoding for $Expr \in AbsBody$ is as follows.

```
(''Expr, ''AbsBody) :: SubSort
type SubSort = (Sort, Sort)
```

4.2 Semantic functions

As is the case in data types à la carte, semantic functions are defined modularly as type class instances and are applied by the fold of an algebra. For example, the following instance encodes the definition of $Abs_{\mathcal{F}}(x, b)$ given in subsection 3.2.

```
instance ToFuncons Abs where[4]
    toFuncons (Abs s (K body)) = K $ function_ [closure_
        [scope_ [bind_ [T.string_ s, given_], body]]]
```

The `ToFuncons` type class captures those types for which a translation to 'Funcons' is available and is defined as follows.

[4]The `T` module provides helper functions to transform Haskell values into funcon values. Funcon smart constructors — identified by the trailing underscore — take a variable number of arguments, hence the usage of lists.

```
class HFunctor f => ToFuncons f where⁵
    toFuncons :: Alg f (K Funcons)
newtype K a i = K {unK :: a} deriving (Functor)
type Alg f e = f e :-> e
type (:->) f g = forall i. f i -> g i
```

Since operators are higher-order functors and we carry around the meta-type, the carrier of our algebra must be a parameterized functor, which funcons are not. Therefore, we wrap funcons with the `K` constructor.

4.3 Glue code

Glue code is implemented as functions of type `Funcons -> Funcons` and to link glue code to an operand of a specific operator, we use multi-parameter type classes and an adaptation on the catamorphism that applies glue code before application of the semantic function on the operator being evaluated. For example, the glue definition from section 3 is achieved with the following instance definition.

```
instance GetGlue IfGlue CExpr Funcons where
    getGlue IfCondGlue _ = \l -> not_ [is_equal_ [0, l]]
    getGlue _ _ = id
class GetGlue operand (f :: (* -> *) -> * -> *) target where
    getGlue :: operand -> f (Term a) b -> target -> target
    getGlue _ _ = id
```

In this example, `CExpr` refers to all operators assigned to the `CExpr` sort, and should be read as the generation of this instance for all operators assigned to the `CExpr` sort.

To identify the operands, we generate a data type for every operator, `IfGlue` in the example, where the cases identify the operand positions for the application of glue code, which in the example are identified by the constructors suffixed with `Glue`.

Again, to be able to modify glue code instances, we delay such instances by defining them in terms of Template Haskell constructs instead.

```
(''CExpr, ''IfCond, 'cExprGlue) :: GlueDefinition
type GlueDefinition = (MetaType, Sort, GlueFunction)
type GlueFunction = Name
cExprGlue l = not_ [is_equal_ [0, l]]
```

The Template Haskell definition does not refer to the glue data type. Instead, the right glue data-type is automatically determined based on the sort in the glue code definition. The glue data-type is thus fully abstracted away from language designers.

4.4 Language definition

Because we defined the components of a language in terms of template Haskell constructs, we can define a language as data type in terms of those template Haskell constructs.

```
data Language = Language
    { op_assign :: [OperatorAssignment]
    , sub_sorts :: [SubSort]
    , glue_code :: [GlueDefinition]
    , init_code :: [(MetaType, GlueFunction)]
    } deriving (Show)
instance Semigroup Language
instance Monoid Language
```

The language definition does not contain a special entry for top-level operators. Instead, we utilize a special sort — `TopLevel` — that can be used inside the operator assignment and sub-sort definitions. As a result, initialization code is defined as a tuple linking operator — via their meta-type — to glue-code functions. In addition, we make languages an instance of the `Monoid` type class, allowing usage of the `(<>)` operator to compose languages.

To instantiate a language, the `genLanguage` function is spliced in.

```
genLanguage :: [Operator] -> Language -> Q [Dec]
type Operator = (Constructor, MetaType)
```

The `genLanguage` function distributes the sub-sort declarations over the operands, generates type family instances for the operator assignments, generates smart constructors that automatically inject the operator into the co-product type (representing the set of operators), generates glue code data types for the operators, and the glue code definitions are transformed into `GetGlue` type class instances. Furthermore, the function performs several checks. First, it checks if all operators mentioned in the language are present in the set of operators. Secondly, it requires that the sub-sort declarations form a directed acyclic graph. If any of these conditions are not met, the compilation is stopped with an error indicating the unsatisfied condition.

Our original definition of the lambda calculus is thus obtained by the definition in Listing 1.

4.5 Parser generation and the *iCoLa*-shell

The structure present in language definitions is enough to use in the generation of parsers for the defined languages. Currently, we generate a parser that parsers a language in which operator application is written in a style similar to LISP [25]. We use such syntax to ensure that the generated grammar is not ambiguous. For example, `(add (intv 1) (add (intv 2) (intv 3)))` demonstrates an expression using the generated syntax for a simple integer addition language. In this example, `add` and `intv` are operators and applications of operators are always surrounded by parentheses. The operands are separated by spaces. So this example simply encodes the arithmetic expression $1 + (2 + 3)$.

Using these generated parsers, we provide the *iCoLa*-shell, which is a 'meta-REPL' that accepts any Haskell declaration. This enables users to define languages and operations over languages inside the meta-REPL. Furthermore, a meta-command `commit` is provided that can be used to commit to a specific language and start an 'object-REPL' for the chosen object language. In the object-REPL, the user can experiment with the defined language using the LISP-style generated concrete syntax. When stopping the object-REPL, the user returns to the meta-REPL, continuing the same session where

[5]The `K`, `Alg` and `(:->)` types are defined by the comp-data library [3]

```
$(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 = []
        })
```

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

```
meta> intAdd = intLanguage <> addLanguage
meta> :commit intAdd
intAdd> (add (intv 1) (intv 10))
11
intAdd> :exit
meta>
```

Listing 2. Example session inside the *iCoLa*-shell.

they left off. Listing 2 illustrates an example session in the meta-REPL and an object-REPL for a simple integer addition language, where meta> denotes execution inside the meta-REPL, otherwise execution is happening inside the object-REPL.

In the same *iCoLa*-shell session, the user can extend the language and start a new object-REPL, or focus on a subset of the language by removing part of the composition or focusing on a subset of languages used in the composition.

5 Examples of incremental compositional language definitions

In this section we demonstrate *iCoLa* by defining several languages in terms of other languages. The used languages are: *Imp* [34], a simple imperative language; *SIMPLE* [35], a more complex procedural language; and *MiniJava* [2], a strict subset of the Java language. These languages have their semantics described in terms of funcons as part of the case studies for the PlanCompS project[6]. This enables us to focus on the incremental and flexibility aspects of our approach and to demonstrate the reuse achieved via operator definitions and glue code.

5.1 The construction of *Imp*

We define *Imp* as the composition of the following four languages: impArith <> impBExpr <> impStmts <> impPrograms, 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 statements 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*, respectively. The definition of impArith and impBExpr are as follows.

```
impArith = Language
    { op_assign = [(e, ''ArithExpr) |
        e <- [''IntType, ''AddType, ''DivType, ''IdType]]
    , sub_sorts = [(''ArithExpr, s) |
        s <- [''AddLeft, ''AddRight, ''DivLeft, ''DivRight]]
    , glue_code = [(''DivType, ''Always, 'check_divide)]
    ...} where
        check_divide f = checked_ [f]

impBExpr = Language
    { op_assign = [(e, ''BExpr) |
        e <- [''BoolType, ''LeqType, ''NotType] ]
    , sub_sorts = [(''BExpr, s) |
        s <- [''NotExpr, ''AndLeft, ''AndRight]]
        ++ [(''ArithExpr, s) | s <- [''LeqLeft, ''LeqRight]]
    ...}
```

In these definitions, there are three things that need to be highlighted. Firstly, both languages do not define a top-level, which means that these languages on themselves are not executable since there are no entry points. Secondly, impArith defines glue code over the *Div* operator that wraps the divide in a check. When division by zero occurs, the program is terminated due to the check. 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. In the description for the glue code, we see usage of the shorthand ''Always sort. This sort is a convenience and denotes that the glue code needs to always be applied on the *Div* operator, it thus encodes the assignment of this glue code to all operands to which *Div* is assigned. Finally, impBExpr uses the *ArithExpr* auxiliary sort in its definitions but does not assign any operators to this sort. Thus, impBExpr is an extension on the arithmetic language.

Alternatively, we can define a refined version of impBExpr that is independent from impArith as follows.

```
impBExpr⁻ = Language
    { op_assign = [(e, ''BExpr) |
        e <- [''BoolType, ''LeqType, ''NotType]]
    , sub_sorts = [(''BExpr, s) |
        s <- [''NotExpr, ''AndLeft, ''AndRight]]
    ...}
```

In this definition, we removed the mentioning of the auxiliary sort. As a result, the refinement of impBExpr is not an extension. To get back to our original definition of impBExpr,

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

we can define a new language that glues `impArith` and the refined version of `impBExpr` together. This glue language only contains the sub-sort declaration that we removed, modeling language unification.

```
impArith <> impBExpr^- <> glueLanguage
    where
    glueLanguage = Language
        { op_assign = []
        , sub_sorts = [(''ArithExpr, s)
            | s <- [''LeqLeft, ''LeqRight]]
        ...}
```

Alternatively, owing to our languages being first-class citizens in Haskell, we can define `impBExpr` as a function with one parameter denoting the sort that can occur in less-then-equal expressions.

```
impBExpr^+ leqSort = Language
    { op_assign = [(e, ''BExpr) |
        e <- [''BoolType, ''LeqType, ''NotType] ]
    , sub_sorts = [(''BExpr, s) |
        s <- [''NotExpr, ''AndLeft, ''AndRight]]
        ++ [(leqSort, s) | s <- [''LeqLeft, ''LeqRight]]
    ...}
```

This makes the parameterized version of `impBExpr` configurable and removes the hard dependency on the auxiliary sort while also removing the requirement of a glue language. Instead, we can decide the correct auxiliary sort when composition occurs.

5.2 Reusing *Imp* to define *SIMPLE* and *MiniJava*

We can reuse the definition of *Imp* to define *SIMPLE* and *MiniJava*, utilizing *Imp* in different ways. *SIMPLE* is a much more elaborate language that almost fully subsumes *Imp*. *MiniJava* is less elaborate since it does not contain all constructs present in *Imp*, but still shares a significant part. However, the way *Imp* is defined does not fully correspond with both *SIMPLE* and *MiniJava*, since the top-level of *Imp* is different and *Imp* contains certain constructs not present in *MiniJava*. To align *Imp* with these requirement we refine the language to a language that aligns with both *SIMPLE* and *MiniJava*. A refinement is an endofunction over languages implemented directly as a Haskell function. This way, all components of a language can be refined, closely resembling the idea of restriction as presented by Erdweg.

To align *Imp*, we define two refinement functions, one, denoted with ψ, that removes the operators from the top-level as defined by *Imp* and one, denoted with ϕ, that removes the operators from the less-than-equal operands, removing the less-than-equal operator from the language.

```
ψ lang = lang { sub_sorts = removeTopLevels . sub_sorts $ lang}
    where
        removeTopLevels = filter $ not . (==''TopLevel) . snd

φ lang = lang { op_assign = removeLeq . op_assign $ lang
              , sub_sorts = removeLeqOps . sub_sorts $ lang }
    where
        removeLeq = filter $ not . (==''LeqType) . fst
```

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.

	Imp	*MiniJava*	*SIMPLE*
Int + Addition	●	●	●
Substraction + Multiplication	○	●	●
Division	◓	◓	◓
If + While	●	●	●
Variables	●	●	●
Ouput	○	◓	●
Input	○	○	●
Classes	○	●	○
Arrays	○	◓	◓
Throw + Catch	○	○	●

```
        removeLeqOps = filter $ not .
            (flip elem [''LeqLeft, ''LeqRight]) . snd
```

For *SIMPLE*, ψ is enough to make *Imp* suitable to use in the definition. In case of *MiniJava*, both ψ and ϕ are needed, thus the composition of these refinement functions is the required refinement function for *MiniJava*.

Besides the required refinement, *Imp* makes a distinction between two types of expressions: arithmetic expressions and boolean expressions. Variables can only occur inside arithmetic expressions and not in boolean expression. In the definitions of *SIMPLE* and *MiniJava* this distinction is not made. Nevertheless, because we make a distinction between operators and sorts in our approach, these structure choices do not prevent the usage of *Imp* when defining both *SIMPLE* and *MiniJava*, because we can define a new auxiliary sort and link both *Imp* sorts to this new sort and then distribute the new sort over the required operands. This demonstrates the flexibility of our approach and that existing language structure choices do not restrict in which compositions a language can be used.

Both *SIMPLE* and *MiniJava* also add new constructs that are not present in *Imp*. Some of the constructs occur in both *SIMPLE* and *MiniJava*. Table 1 highlights some of the language constructs used and their presence in the languages. This table is not extensive and we sometimes group operators together due to space limitations, since *SIMPLE* alone already contains 40 language constructs. Nevertheless, it demonstrates a selection of constructs that are often present in multiple of the defined languages. This highlights the reusability obtained via our approach.

While operators might occur in multiple languages, their usage is not always identical. For example, in *MiniJava*, output is always followed by a newline, which is not the case in *SIMPLE*. Also, both languages check out-of-bounds array access, hence the required glue code.

5.3 Object language variability

MiniJava is interesting because variations of *MiniJava* exist that have been introduced for teaching purposes.[7] 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. This enables a teacher to exclude or include languages to create new variants. Furthermore, a teacher is not restricted to the existing core of *MiniJava*, because with sort-constraints, a teacher can freely alter the language to their needs. For instance, a teacher can remove the object-oriented aspect of *MiniJava* and start with procedural programming before introducing objects and classes. Alternatively, a teacher can include the `Exceptions` from *SIMPLE* to add exceptions to *MiniJava*.

Language variability is also useful when designing a programming language. In the *iCoLa*-shell, 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 early feedback to include in the design process. In Table 2, the outcome of such a session is listed as a table. In this session, a fixed set of operators is used to define a variety of languages. 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.

```
lambdaCBN = lambda <> Language
{glue_code = [(''Always, ''AppArg, 'thunk),
              (''VarType, ''Always, 'force)]
...} where
    thunk f = thunk_ [f]
    force f = force_ [f]
```

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. However, 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 Discussion

In *iCoLa*, some of the techniques discussed in section 2 are combined as follows. The syntax and semantic functions of operators are defined modularly using data types à la carte. The funcons of Funcons-beta are used to express the semantics of all operators in the same semantic domain. This makes it possible to define languages by selecting (top-level) operators from the set of all available operators. This is consistent with the methodology of [44] and ensures object languages in *iCoLa* are 'sequential languages' by definition. As such, REPLs for the object languages are obtained for free. The meta-language is sequential in itself, thereby supporting incremental meta-programming in the *iCoLa*-shell. This is achieved by defining operators in isolation using 'sort constraints' as explained in Section 3. The sort contraints are enforced statically by applying (Template) Haskell in the implementation of *iCoLa* as an EDSL. In this section we reflect on further details of our approach.

6.1 *iCoLa* as an EDSL

The presented implementation is in the form of a Haskell EDSL. The EDSL offers static guarantees such that every operator in a language has a semantic function and sort constraints are respected. In addition, language designers have the full power of Haskell available to them when defining and manipulating languages. However, we are also restricted by our choice of implementation. Because we utilize Template Haskell, a compilation step is needed before a defined language can be used, only one language can be generated per module, and there is a stage restriction enforced by Template Haskell. This requires us to implement the *iCoLa*-shell separately instead of reusing the REPL provided by the Haskell compiler (e.g. GHCi). Furthermore, we assume Haskell familiarity from language designers, for example to understand Haskell type errors when operators are incorrectly used. In addition, operators, semantic functions, and language definitions involve some boilerplate code. For example, the introduction of type families for operators, the need for constraints on operator definitions, and occurrences of the `K` constructor.

Some boilerplate code can be removed by using quasi-quotation. With quasi-quotation, we could provide a small layer of syntactic sugar that removes most of the boilerplate code currently present in our approach. The concern regarding Haskell type-errors can be mitigated by providing custom type-errors or other strategies for type-error customization in EDSLs [37].

Another alternative is an implementation with an external DSL. This gives the possibility to provide syntax that is much closer to the mathematical approach of section 3 and also allows us to give more domain-specific error messages. However, such an implementation is more complex and puts a restriction on the semantic functions. Currently, our implementation is easily extended with new semantic functions; and because semantic functions are implemented in Haskell, the possibilities are endless. When providing an external DSL, this flexibility is lost, requiring either a constraint on the semantic functions or complicating the implementation

[7] https://courses.cs.washington.edu/courses/cse401/13wi/project/MiniJava.html; http://teaching.up.edu/cs358/miniJava.pdf

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	●	◓	●	●	●	●	●	●	●	●

to support more complex semantic functions. Using an external DSL also removes many of the benefits we currently have by providing languages as first-class citizens inside Haskell. Alternatively, a hybrid approach can be implemented that provides a DSL layer on top that compiles down to Haskell such that semantic functions and language refinement can still be defined as Haskell functions. How these different implementation techniques affect usability and expressiveness is to be explored in future work.

6.2 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 glue code. In our experience from our case study, 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, via the structure of the language, a BNF-like grammar can be generated that shows the structure in an uncluttered fashion. In addition, we envision tooling that highlights the effects of language composition and allows one to zoom in on specific operators and see how they are affected by glue code. Furthermore, the algebra used in our approach has no context which requires that the translation must be done in a context independent manner, deferring a lot of the work to the semantic domain, but keeping operator implementations simple. In future work, we would like to explore having additional algebras defined as semantic functions, especially to express static semantics, without losing the flexible compositional capabilities of our approach.

6.3 Concrete syntax

In the presented framework, a LISP-style concrete syntax definition is available for object languages automatically. However, a language designer might want to define their own concrete syntax. To support user-defined syntax, we can extend operator definitions with a notion of concrete syntax or extend language definitions to attach concrete syntax to the chosen operators of the language. In both cases, problems with ambiguity can arise when combining concrete syntax definitions. Generalized parsing techniques, such as Early parsing [10], GLR [41], and GLL [36], can be used to accept ambiguous grammars. Handling the ambiguities in a low-effort way is important as even the introduction of a single operator into a language can result in ambiguity, hampering rapid prototyping. When a language is finalized, the grammar can be inspected and adapted to possibly remove ambiguities.

An alternative approach is to mix user-defined parsers and the generated parsers. This way, operators are not extended with a notion of concrete syntax. Instead, a language designer develops a parser alongside the language definition and when prototyping with new operators uses the generated parser for those operators. This way, the parser is incrementally defined just like the language itself. However, this approach requires an efficient way to connect user-defined parser and generated parser and further integration to ensure the language development process to occur fully inside the *iCoLa*-shell. Nevertheless, in both cases the existing implementation of parser generation can be mostly reused, since the generation of operand parsers is generic.

In future work, we will investigate approaches to incorporate user-defined concrete syntax into our current model, by experimenting with the different options mentioned and evaluating how these affect the interactive, reusable, and compositional aspects of our approach.

7 Related work

Developing languages via some form of composition is supported by a wide variety of language-development environments [13, 20, 40, 42, 46]. Erdweg et al. [12], 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 [11], which is an environment for the construction of Java like languages, supported unification. Nevertheless, extension-unification is supported by most environments.

Lisa [29] is a full-fledged interactive environment for programming language development based on attribute grammars with support for incremental language development [30] via multiple attribute grammar inheritance [28]. Compared to our approach, the incremental focus is more linear and distinction between operators and where operators are used is not made.

Melange [9] 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. In contrast to our approach, Melange makes no distinction between operator semantics and operator specialization, and does not make a distinction between operator definitions and language definitions.

In Feature-oriented programming [1], 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 [5] is a development environment modeled around the idea of feature-oriented programming, where features are implemented using an object-oriented approach.

Software product lines [7] 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, somewhat resembling our idea of an operator universe from the language point of view, 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 family systems. 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 [22]. AiDE provides an environment for language-development based on software product lines by building an environment on top of Neverlang [5]. Besides AiDE, there are several other environments integrating software product lines in the context of language development — also known as language product lines [27].

Focus on language families [24], 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.

Solutions to the expression problem, such as finally tagless [4], object algebras [8], and, data types à la carte [39], naturally lead to an extensible approach for operators and can be used to implement languages in a modular fashion, as demonstrated by several systems based on the solutions to the expression problem [17, 18]. However, composition and variability are not necessarily obtained. Nevertheless, it would be interesting to see if these solutions to the expression problem can function as a compilation target, for which we use data types à la carte now, in our implementation.

LANG-N-PLAY [6] is a proof-of-concept language introducing the idea of *languages as first-class citizens*. *LANG-N-PLAY* is a statically typed functional language in which languages are just expressions. Consequently, operators over languages are defined as functions. In contrast to our approach, *LANG-N-PLAY* is a newly developed language and not achieved via template programming. In addition, *LANG-N-PLAY* is statically typed. Our approach is statically typed as well, but via Template Haskell, which performs type-checking in stages: the Template Haskell expression is not type checked, but the resulting program is [38]. This requires a translation between the two layers when interpreting type errors.

8 Conclusion

This paper introduced *iCoLa*, a meta-language aimed at improving the language design process through rapid prototyping with reusable components and incremental programming. The *iCoLa*-shell enables fast prototyping by supporting the simultaneous definition of multiple languages that can be composed, unified, extended, restricted and tested within a shared REPL session. In *iCoLa*, languages are first-class citizens such that the users are given the full power of the host language (Haskell) to define languages. Operators over languages can be defined (e.g. composition) and languages can be parameterized, including by other languages (e.g. refinement). 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.

The flexibility provided by *iCoLa* makes it difficult to track the precise composition of a language when applied at (large) scale and user-defined concrete syntax is currently not supported. Methods to improve *iCoLa* in these regards are to be explored in future work.

References

[1] Sven Apel and Christian Kästner. 2009. An Overview of Feature-Oriented Software Development. *J. Object Technol.* 8, 5 (2009), 49–84. https://doi.org/10.5381/jot.2009.8.5.c5

[2] Andrew W. Appel and Jens Palsberg. 2002. *Modern Compiler Implementation in Java, 2nd edition.* Cambridge University Press.

[3] Patrick Bahr and Tom Hvitved. 2011. Compositional data types. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Jaakko Järvi and Shin-Cheng Mu (Eds.). ACM, 83–94. https://doi.org/10.1145/2036918.2036930

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

[5] Walter Cazzola. 2012. Domain-Specific Languages in Few Steps - The Neverlang Approach. In *Software Composition - 11th International Conference, SC@TOOLS 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7306)*, Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book (Eds.). Springer, 162–177. https://doi.org/10.1007/978-3-642-30564-1_11

[6] Matteo Cimini. 2018. Languages as first-class citizens (vision paper). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*, David J. Pearce, Tanja Mayerhofer, and Friedrich Steimann (Eds.). ACM, 65–69. https://doi.org/10.1145/3276604.3276983

[7] Paul Clements and Linda Northrop. 2002. *Software product lines.* Addison-Wesley Boston.

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

[9] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: a meta-language for modular and reusable development of DSLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, Richard F. Paige, Davide Di Ruscio, and Markus Völter (Eds.). ACM, 25–36. https://doi.org/10.1145/2814251.2814252

[10] Jay Earley. 1970. An Efficient Context-Free Parsing Algorithm. *Commun. ACM* 13, 2 (1970), 94–102. https://doi.org/10.1145/362007.362035

[11] Torbjörn Ekman and Görel Hedin. 2007. The jastadd extensible java compiler. In *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*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 1–18. https://doi.org/10.1145/1297027.1297029

[12] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. 2012. Language Composition Untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications* (Tallinn, Estonia) *(LDTA '12)*. ACM, Article 7, 8 pages. https://doi.org/10.1145/2427048.2427055

[13] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: library-based syntactic language extensibility. In *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*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 391–406. https://doi.org/10.1145/2048066.2048099

[14] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge. In *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8225)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer, 197–217. https://doi.org/10.1007/978-3-319-02654-1_11

[15] Damian Frolich and L. Thomas van Binsbergen. 2021. A Generic Back-End for Exploratory Programming. In *Trends in Functional Programming - 22nd International Symposium, TFP 2021, Virtual Event, February 17-19, 2021, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12834)*, Viktória Zsók and John Hughes (Eds.). Springer, 24–43. https://doi.org/10.1007/978-3-030-83978-9_2

[16] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. 1977. Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM* 24, 1 (1977), 68–95. https://doi.org/10.1145/321992.321997

[17] Maria Gouseti, Chiel Peters, and Tijs van der Storm. 2014. Extensible language implementation with object algebras (short paper). In *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014*, Ulrik Pagh Schultz and Matthew Flatt (Eds.). ACM, 25–28. https://doi.org/10.1145/2658761.2658765

[18] Pablo Inostroza and Tijs van der Storm. 2017. Modular interpreters with implicit context propagation. *Comput. Lang. Syst. Struct.* 48 (2017), 39–67. https://doi.org/10.1016/j.cl.2016.08.001

[19] Patricia Johann and Neil Ghani. 2008. Foundations for structured programming with GADTs. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 297–308. https://doi.org/10.1145/1328438.1328475

[20] Lennart C. L. Kats and Eelco Visser. 2010. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *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*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 444–463. https://doi.org/10.1145/1869459.1869497

[21] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. 2016. Jupyter Notebooks - a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, June 7-9, 2016*, Fernando Loizides and Birgit Schmidt (Eds.). IOS Press, 87–90. https://doi.org/10.3233/978-1-61499-649-1-87

[22] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. 2015. Choosy and picky: configuration of language product lines. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, Douglas C. Schmidt (Ed.). ACM, 71–80. https://doi.org/10.1145/2791060.2791092

[23] Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *22nd Symposium on Principles of Programming Languages*. ACM, 333–343. https://doi.org/10.1145/199448.199528

[24] Jörg Liebig, Rolf Daniel, and Sven Apel. 2013. Feature-oriented language families: A case study. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. 1–8.

[25] John McCarthy. 1978. History of LISP. In *History of Programming Languages, from the ACM SIGPLAN History of Programming Languages Conference, June 1-3, 1978, Los Angeles, California, USA*, Richard L.

Wexelblat (Ed.). Academic Press / ACM, 173–185. https://doi.org/10.1145/800025.1198360

[26] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 523)*, John Hughes (Ed.). Springer, 124–144. https://doi.org/10.1007/3540543961_7

[27] David Méndez-Acuña, José Angel Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. 2016. Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review. *Comput. Lang. Syst. Struct.* 46 (2016), 206–235. https://doi.org/10.1016/j.cl.2016.09.004

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

[29] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. 2002. LISA: An Interactive Environment for Programming Language Development. In *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 (Lecture Notes in Computer Science, Vol. 2304)*, R. Nigel Horspool (Ed.). Springer, 1–4. https://doi.org/10.1007/3-540-45937-5_1

[30] Marjan Mernik and Viljem Zumer. 2005. Incremental programming language development. *Comput. Lang. Syst. Struct.* 31, 1 (2005), 1–16. https://doi.org/10.1016/j.cl.2004.02.001

[31] Peter D. Mosses. 1990. Denotational Semantics. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Jan van Leeuwen (Ed.). Elsevier and MIT Press, 575–631. https://doi.org/10.1016/b978-0-444-88074-1.50016-0

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

[33] Peter D. Mosses, Neil Sculthorpe, and L. Thomas Van Binsbergen. 2021. *Funcons-Beta*. Retrieved August 17, 2022 from https://plancomps.github.io/CBS-beta/Funcons-beta/ Online GitHub repository.

[34] Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

[35] Grigore Rosu and Traian-Florin Serbanuta. 2014. K Overview and SIMPLE Case Study. *Electron. Notes Theor. Comput. Sci.* 304 (2014), 3–56. https://doi.org/10.1016/j.entcs.2014.05.002

[36] Elizabeth Scott and Adrian Johnstone. 2010. GLL Parsing. *Electron. Notes Theor. Comput. Sci.* 253, 7 (2010), 177–189. https://doi.org/10.1016/j.entcs.2010.08.041

[37] Alejandro Serrano. 2018. *Type Error Customization for Embedded Domain-Specific Languages*. Ph. D. Dissertation. Utrecht University, Netherlands. http://dspace.library.uu.nl/handle/1874/363523

[38] Tim Sheard and Simon L. Peyton Jones. 2002. Template meta-programming for Haskell. *ACM SIGPLAN Notices* 37, 12 (2002), 60–75. https://doi.org/10.1145/636517.636528

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

[40] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 132–141. https://doi.org/10.1145/1993498.1993514

[41] Masaru Tomita. 1985. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers.

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

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

[44] L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. 2020. *A Principled Approach to REPL Interpreters*. ACM, 84–100. https://doi.org/10.1145/3426428.3426917

[45] Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. 2021. Latent Effects for Reusable Language Components. In *Programming Languages and Systems*, Hakjoo Oh (Ed.). Springer International Publishing, Cham, 182–201. https://doi.org/10.1007/978-3-030-89051-3_11

[46] M Voelter and K Solomatov. 2010. Language modularization and composition with projectional language workbenches illustrated with MPS. Software Language Engineering.

[47] Philip Wadler et al. 1998. The expression problem. *Posted on the Java Genericity mailing list* (1998).

[48] Martin P. Ward. 1994. Language-Oriented Programming. *Softw. Concepts Tools* 15, 4 (1994), 147–161.