Towards Incremental Language Definition with Reusable Components

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

This paper introduces a novel method for defining software languages incrementally as the composition of smaller languages, starting from reusable components for the specification of syntax and semantics. The method is enabled by the combined application of several advanced techniques implemented in functional languages: datatypes à la carte for the fine-grained composition of (abstract) syntactic categories, generalised top-down parsing enabling the composition of executable (concrete) syntax specifications and composable micro-interpreters that implement the operational semantics of certain reusable components known as 'funcons'. We demonstrate the method makes it possible to perform incremental language development with prototyping. The generality of the method is demonstrated through a variety of case studies.

KEYWORDS

software language engineering, language composition, syntax, semantics, interpretation

1 INTRODUCTION

Incremental programming is a style of programming in which software is built in a step-by-step fashion by submitting code fragments that, for example, declare a single type or execute a single statement with immediate feedback on the validity and effect of the code fragment. This style of programming is naturally supported by read-eval-print-loop (REPL) interpreters (also referred to as interactive shells) such as JShell and IPython and computational notebooks such as Jupyter [5] and Mathematica [4]. In the context of software engineering, a common usage of a REPL is to test a library under development by loading its latest version and interacting with the functions it defines, perhaps in combination with functions from other libraries (under development). In the context of data science, a common usage of a computational notebook is the simultaneous development and testing of a scientific workflow. In the context of language-oriented programming [18], an incremental programming environment should support the definition and testing of language constructs (akin to library functions), also in combination with the constructs of other languages, by extending language definitions and by running test programs written in the language(s) currently under construction. However, there are various challenges to realising such a system. For example, simultaneously extending the syntax and semantics of a language definition without modifying or recompiling existing parts are requirements of solutions to the well-documented expression problem coined by Wadler [17]. As another example, the composition of two deterministic context-free grammars may produce a non-deterministic context-free grammar to which the parsing technology of choice is not applicable or which results in ambiguities that need to be resolved.

In this paper we experiment with an approach to incremental language development enabled by certain functional techniques for modular language specification: data types á la carte [11], combinators for concrete syntax specification and generalised LL (GLL) parsing [14, 15], and micro-interpreters implementing the operational semantics of a reusable library of fundamental programming languages constructs known as 'funcons' [13]. The funcons of the Funcons-beta library [8] are used as a common base language on top of which object languages are defined. The practicality of this approach is to be evaluated in this paper.

The full version of this paper is to contribute by:

- Presenting a novel approach for incremental language definition with reusable components
- Presenting an implementation of the approach as a framework consisting of existing Haskell EDSLs resulting from recent advances in modular language specification techniques
- Demonstrating the applicability and generality of the approach through various case studies and in comparison with existing approaches to language extension and unification

2 BACKGROUND

The initial algebra semantics of Goguen et al. [3], concisely described by Mosses in [6], provides important formal foundations and terminology to our work, as it can be seen to capture the essential elements of many existing semantic specification formalism such as denotational semantics and attribute grammar semantics. In initial algebra semantics, a multi-sorted signature lays out the operations of a language. Abstract syntax is defined by assigning value constructors to the operations (as an initial algebra) according to their structure, as determined by the signature. Semantics are defined by assigning 'semantic domains' to sorts and functions to operations (within an evaluation algebra) that map the values (within their respective semantic domains) computed for operands to the value describing the result of applying the operation to these operands.

As a solution to the expression problem [17], *data types á la carte* [11] provides a method for assembling data types and functions from individual components to form signatures and initial

IFL '21, September 01–03, 2021, Online

algebras, and evaluation algebras respectively. With this technique, signatures of independent languages can be freely composed, enabling mixing of syntactic constructs of different languages.

The GLL combinator library [14] forms an EDSL for the specification of (context-free) grammars using BNF-like operators. The combinator expressions formed by applying the core combinators of this library can themselves be parameterised; achieving 'reuse through abstraction'. This way, reusable components for (concrete) syntax specification can be defined [12]. The components are executable since combinator expressions can be evaluated according to the GLL parsing algorithm [9, 10, 15]. By applying generalised parsing, there is no need to restructure the grammar to suit the parsing technology as the composition of two or more context-free grammars is a context-free grammars (and generalised parsing algorithms work on all context free grammars). However, the result of a composition may be an ambiguous grammar, even when the input grammars are unambiguous, demanding the integration of ambiguity reduction strategies.

The component-based approach to operational semantics presented in [7] is centred around reusable definitions of the fundamental constructs of programming – *funcons* for shorts. As explained in [13], 'micro-intepreters' can be generated from funcon definitions. The micro-interpreters are compositional evaluation functions expressing the behaviour of an individual funcon that can be generated and compiled separately. In this paper we leverage the generality of the Funcons-beta [8] library to be able to express the semantics of various languages in a shared based language, applying the micro-interpreters generated for funcons as the constructs of an EDSL.

By combining the three described techniques, language extensions and compositions can be written in a highly modular fashion, permitting rapid prototyping. Via the method proposed in [16] and implemented in [2], REPLs for the resulting languages can be obtained with minimal effort.

Erdweg et al. provide a framework for discussing and comparing meta-languages, tools and formalisms that support various form of incremental language development [1]. In particular, the authors define the concepts of (modular) language extension, restriction, and unification which they apply to both the syntax, static semantics, operational semantics and IDE services of languages. In this paper we adopt their terminology and use their framework as the basis for our evaluation.

3 MOTIVATING EXAMPLE

To illustrate the approach, we start with a simple *While* language supporting while-loops, assignments, integer expressions, and less-than-equal comparison expressions. The language is implemented with several building blocks, which can be combined to construct the concrete *While* language. The lines beginning with *ghci>* denote operations in the Haskell REPL and lines beginning with *repl>* denote operations in the REPL for the constructed language.

```
ghci> import Whilelang
ghci> repl pWhile
repl> x = 1
repl> while (x =< 5) x = x + 1 done
```

In the example, the *Whilelang* module is imported which exports the definition of *While* syntax and funcon translation. The `repl` function starts a read-eval-print-loop session for the language definition given as an argument.

As is, *While* does not support for printing. To overcome this, we define the printing construct as a language extension.

```
ghci> data Print a = Print a deriving Functor
ghci> instance ToFuncons Print where
|toFuncons (Print a) = print_ [a, string_ "\n"]
ghci> pPrint pArg = "print" <::=>
   iPrint <$$> keyword "print" **> pArg
```

Viewed as a language, the printing language is abstract in the sense that it requires an argument to determine for what kind of expressions it is capable of printing the value. For example, support for printing in *While* can be added by choosing *While* expressions as the argument to the printing language.

```
...
ghci> let pWhilePrint :: Parser (Whilelang.Sig :+: Print)
          = pWhile <||> pPrint pExpr
ghci> repl pWhilePrint
repl> x = 1
repl> while (x =< 5) x = x + 1 done
repl> print(x)
6
```

The example uses the $\langle||\rangle$ operator to compose two languages, which allows syntax of both languages to occur at the top-level. Such a composition is a coarse-grained composition, because language constructs of both languages can be used but not freely mixed. Furthermore, the final parser of the composed language has type *Parser (Whilelang.Sig :+: Print)*[1], which composes the signature of *While* with the signature of the print language, allowing language constructs from both languages to occur in the parsing result.

Coarse-grained composition is not the only style, it is also possible to perform fine-grained composition such that language elements can be freely mixed. Our previous example can be defined as a fine-grained composition as follows.

```
...
ghci> repl $ pWhileFine (pPrint pExpr)
repl> x = 1
repl> while (x =< 5) print(x); x = x + 1 done
1
...
5
```

Instead of using the choice operator, the concrete print language is passed as an argument to a finer definition of *While*. Languages are free to define different forms of fineness via their parsers. This enables different styles of compositions with other languages. In the example, the *Whilelang* module exports a finer definition that takes one argument which is used to extend the language constructs possible inside while loops and at the top-level. In the example, this style of composition makes it possible to use print statements inside while loops, which was not possible with the coarse-grained composition.

With our approach it is also possible to re-use existing language components to perform extension, for example by extending a language with syntactic sugar. In case of *While*, for-loops can be

[1] In further examples, parser types are omitted for brevity.

added with only a parser definition by re-using the signature and semantics of *While*.

```
ghci> import Whilelang
...
ghci> pFor = "for" <::=> buildFor <$$>
    pHeader <**> pWhileFine (pPrint pExpr)
  where
    buildFor = \var -> \start -> \end -> \body ->
      iSeq (iAssign var start) iWhile (iLeq var end)
           (iSeq body (iAssign var (iAdd var iLitInt 1)))
ghci> repl $ pWhileFine $ (pPrint pExpr) <||> pFor
repl> for(x = 1 to 5) print(x) done
1
...
5
```

The for extension uses smart constructors from the *Whilelang* module to construct the actual for components, and thus no new instance definition or signature components are required.

Currently, *While* is extended by introducing new languages or introducing new language components. Nonetheless, *While* can also be extended by using existing languages.

```
ghci> import Whilelang as W
ghci> import Labmbdalang as L
...
ghci> repl $ W.pWhileExprFine
    ((pPrint pExpr) pExpr) pExpr
    where
        pExpr = W.pExprFine pExpr
            <||> L.pLambdaFine pExpr
repl> add5 = lambda x x + 5 done
repl> x = 0
repl> while (x =< 10) print(x); x = add5(x) done
0
5
10
```

In this example, we import the *Whilelang* module and the *Lambdalang* module. The last module exports an implementation of call-by-value lambda calculus. In this composition, we utilise the *pWhileExprFine* parser exported by the *Whilelang* module. This parser, takes two arguments, one to extend the statements of *While* and one to define the expressions of *While*. To define the expressions of this language, a coarse-grained composition, composing the *While* expressions and lambda expressions together. However, both parsers used in this composition are fine-grained and are made concrete by self-referencing the coarse-grained composition. As a result of self-referencing, *While* expressions and lambda expressions can be used interchangeably. In the example, this occurs in the definition of the *add5* function. The definition of the *add5* function uses assignments from *While*, a lambda expression from lambda, and inside the lambda expression integer addition from *While*. Finally, this self-referencing composition is passed as the argument for the *While* expressions, allowing usage of both lambda and *While* expressions inside while loops.

4 CONCLUSIONS AND FUTURE WORK

In this paper we have shown an approach to incremental language development enabled by functional techniques for modular language specification. With the approach, a language can be incrementally defined by writing new languages, using existing language components, or using existing languages. Furthermore, by utilising higher-order parsers, higher-order languages are possible, enabling both coarse- and fine-grained compositions. With fine-grained compositions, a level of flexibility is achieved that enables the extension of languages with existing languages and mix language construct from both languages arbitrarily. In addition, languages are also executable, enabling prototyping in a REPL. Via these interactions, a language can be quickly tested and adapted where required.

Currently, we have given an overview of our approach using an example that extends *While* with support for printing, for-loops and functions. In the final paper, we go in-depth on the technical details of the approach and we perform case-studies to demonstrate the genericity and applicability of the approach. Via these case-studies, we identify the drawbacks and applicability of the approach, and compare our approach to existing approaches for incremental language development.

REFERENCES

[1] 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

[2] Damian Frölich and L. Thomas van Binsbergen. 2021. A Generic Back-End for Exploratory Programming. In *The 22nd International Symposium on Trends in Functional Programming (TFP 2021) (LNCS, Vol. 12834)*. Springer. https://doi.org/10.1007/978-3-030-83978-9_2

[3] 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

[4] Brian Hayes. 1990. Thoughts on Mathematica. *Pixel* 1, January/February (1990), 28–34.

[5] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica 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*, Fernando Loizides and Birgit Scmidt (Eds.). IOS Press, Netherlands, 87–90. https://doi.org/10.3233/978-1-61499-649-1-87

[6] Peter Mosses. 1990. Denotational Semantics. In *Handbook of Theoretical Computer Science (Vol. B): Formal Models and Semantics*, Jan van Leeuwen (Ed.). MIT Press, 577–631.

[7] 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

[8] Peter D. Mosses, Neil Sculthorpe, and L. Thomas Van Binsbergen. [n.d.]. Funcons-Beta. Online GitHub repository, https://plancomps.github.io/CBS-beta/Funcons-beta/.

[9] Elizabeth Scott and Adrian Johnstone. 2010. GLL Parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (2010), 177 – 189. https://doi.org/10.1016/j.entcs.2010.08.041 Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[10] Elizabeth Scott and Adrian Johnstone. 2013. GLL parse-tree generation. *Science of Computer Programming* 78, 10 (2013), 1828 – 1844. https://doi.org/10.1016/j.scico.2012.03.005

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

[12] L. Thomas van Binsbergen. 2019. *Executable Formal Specification of Programming Languages with Reusable Components.* Ph.D. Dissertation. Royal Holloway, University of London.

[13] 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

[14] L. Thomas van Binsbergen, Elizabeth Scott, and Adrian Johnstone. 2018. GLL Parsing with Flexible Combinators. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2018)*.

[15] L. Thomas van Binsbergen, Elizabeth Scott, and Adrian Johnstone. 2020. Purely functional GLL parsing. *Journal of Computer Languages* 58 (2020), 100945. https://doi.org/10.1016/j.cola.2020.100945

[16] 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

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

[18] Martin P. Ward. 1994. Language-Oriented Programming. *Softw. Concepts Tools* 15, 4 (1994), 147–161. https://doi.org/10.1007/978-1-4302-2390-0_12