Modularly Programmable Syntax and Type Structure

Cyrus Omar

August 5, 2015

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich, Chair
TODO: confirm rest of committee

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: TODO: keywords

TODO: dedication

Abstract

Full-scale functional programming languages often make *ad hoc* choices in the design of their concrete syntax and type structure. For example, nearly all major functional languages include derived forms for lists, but introducing derived forms for other library constructs requires forming a new dialect of the language. Unfortunately, there is generally no way to modularly combine such dialects, limiting the choices available to client programmers. We describe and formally specify new primitives that mitigate the need for dialects by giving library providers the ability to programmatically control syntactic expansion, typechecking and translation (to a small typed internal language) in a safe and modular manner.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Contributions 3
1.3 Disclaimers 5

2 Language Overview **7**
2.1 External Language 7
2.2 Internal Language 7
2.3 Static Language 7
2.4 Module Language 7

I Modularly Programmable Syntax **9**

3 Motivation **11**
3.1 Motivating Examples 11
3.1.1 Lists 11
3.1.2 HTML 11
3.1.3 Regular Expressions 11
3.1.4 Monadic Commands 11
3.1.5 Quasiquotation 11
3.2 Existing Approaches 11
3.2.1 Dynamic String Parsing 11
3.2.2 Direct Syntax Extension 11
3.2.3 Term Rewriting 11

4 Typed Syntax Macros **13**
4.1 Examples 13
4.2 Minimal Formalization 13
4.3 Parameterized TSMs 13

5 Type-Specific Languages **15**
5.1 Examples 15
5.2 Minimal Formalization 15

5.3 Parameterized TSLs 15
5.4 Conclusion & Future Work 15

II Modularly Programmable Type Structure **17**

6 Motivation **19**
6.1 Motivating Examples 19
6.2 Existing Approaches 19

7 Metamodules **21**

8 Conclusion & Future Work **23**

Bibliography **25**

List of Figures

Chapter 1

Introduction

1.1 Motivation

Functional programming language designers often study small typed lambda calculi to develop a principled understanding of fundamental metatheoretic issues, like type safety, and to examine the mathematical character of language primitives of interest in isolation. These studies inform the design of "full-scale"[1] functional languages, which combine several such primitives and also typically feature various generalizations and primitive "embellishments" motivated by a consideration of human factors. For example, major languages like Standard ML (SML) [12, 17], OCaml [14] and Haskell [13] all primitively build in record types, generalizing the nullary and binary product types that suffice in simpler calculi, because explicitly labeled components are cognitively useful to human programmers. Similarly, these languages build in derived syntactic forms (colloquially, "syntactic sugar") that decrease the syntactic cost of working with common library constructs, like lists.

The hope amongst many language designers is that a limited number of primitives like these will suffice to produce a "general-purpose" programming language, i.e. one where programmers can direct their efforts almost exclusively toward the development of useful libraries. However, a stable language design that fully achieves this ideal has yet to emerge, as evidenced by the diverse array of "dialects" that continue to proliferate around all major contemporary languages. In fact, tools that aid in the construction of so-called "domain-specific" language dialects (DSLs)[2] seem only to be becoming increasingly prominent.

Why are there so many dialects?

This calls for an investigation: why is it that programmers and researchers are still so often unable to satisfyingly express the constructs that they need in libraries, as modes of use of the "general-purpose" primitives already available in major languages today, and instead see a need for new language dialects?

[1]Throughout this work, phrases that should be read as having an intuitive meaning, rather than a strict mathematical meaning, will be first introduced with quotation marks.

[2]In some parts of the literature, such dialects are called "external DSLs", to distinguish them from "internal" or "embedded DSLs", which are actually library interfaces that only "resemble" distinct dialects [9].

Perhaps the most common motivation may be that the *syntactic cost* of expressing a construct of interest using contemporary general-purpose primitives is not always ideal. In response, library providers construct *syntactic dialects* – dialects that introduce only new derived syntactic forms. For example, Ur/Web is a syntactic dialect of Ur (a language that itself descends from ML [2]) that builds in derived forms for SQL queries, HTML elements and other datatypes used in the domain of web programming [3]. These are certainly not the only types of data that could similarly benefit from the availability of specialized derived forms – we will consider a number of other examples in Sec. 3.1. Tools like Camlp4 [14], Sugar* [4, 5] and Racket [7], which we will discuss in Sec. 3.2, have lowered the engineering costs of constructing syntactic dialects in such situations, contributing to their proliferation.

More advanced dialects introduce new type structure, going beyond what is possible with only new derived forms. As a simple example, the static and dynamic semantics of records cannot be expressed by context-independent expansion to a language with only nullary and binary products. Various languages have explored "record-like" primitives that go further, supporting functional update operators, width and depth coercions (sometimes implicit), methods, prototypic dispatch and other such "semantic embellishments" that in turn cannot be expressed by context-independent expansion to a language with only standard record types (we will detail an example in Sec. 6.1). OCaml primitively builds in the type structure of polymorphic variants, open datatypes and operations that use format strings like `sprintf` [14]. ReactiveML builds in primitives for functional reactive programming [15]. ML5 builds in high-level primitives for distributed programming based on a modal lambda calculus [18]. Manticore [8] and AliceML [20] build in parallel programming primitives with a more elaborate type structure than is found in simpler accounts of parallelism. MLj builds in the type structure of the Java object system (motivated by a desire to interface safely and naturally with Java libraries) [1]. Other dialects do the same for other foreign languages, e.g. Furr and Foster describe a dialect of OCaml that builds in the type structure of C [10]. Tools like proof assistants and logical frameworks are used to specify and reason metatheoretically about dialects like these, and tools like compiler generators and language frameworks [6] lower their implementation cost, again contributing to their proliferation.

Dialects Considered Harmful

One might view the ongoing proliferation of dialects described above as harmless or even positive, because programmers can simply choose the right language for the job at hand [22]. However, this "dialect-oriented" approach is, in an important sense, anti-modular: a library written in one dialect cannot, in general, safely and idiomatically interface with a library written in another dialect. As the study of MLj demonstrated, addressing this interoperability problem requires somehow "combining" the dialects into a single language. However, in the most general setting where the dialects in question might be specified by judgements of arbitrary form, this is not a well-defined notion. Even if we restrict our interest to dialects specified using formalisms that do operationalize some notion of dialect combination, there is generally no guarantee that the combined dialect will conserve important syntactic and semantic properties that can be established about the dialects in isolation. For example, consider two syntactic dialects, one specifying derived syntax for finite mappings, the other specifying a similar syntax for *ordered* finite mappings. Though each dialect can be specified by an unambiguous grammar in isolation, when these grammars are naïvely combined by, for example, Camlp4, ambiguities arise. Due to this paucity of modular reasoning principles, the "language-oriented" approach is problematic for software development "in the large".

Dialect designers have instead had to take a less direct approach to have an impact on large-scale software development: they have had to convince the designers in control of comparatively popular languages, like OCaml and Scala, to include some suitable variant of the primitives they've espoused into backwards compatible language revisions. This *ad hoc* approach is not sustainable, for three main reasons. First, as suggested by the diversity of examples given above, there are simply too many potentially useful such primitives, and many of these are only relevant in relatively narrow application domains (for derived syntax, our group has gathered initial data speaking to this [19]). Second, primitives introduced earlier in a language's lifespan can end up monopolizing finite "syntactic resources", forcing subsequent primitives to use ever more esoteric forms. And third, primitives that prove to be flawed in some way cannot be removed or changed without breaking backwards compatibility.

This suggests that language designers should be quite careful and strive to keep general-purpose languages small and free of *ad hoc* primitives. Given that we also want to avoid dialect formation, for the reasons described above, this leaves two possible paths forward. One, exemplified (arguably) by SML, is to simply eschew further syntactic conveniences and innovative type structure and settle on the existing primitives, which might be considered to sit at a "sweet spot" in the overall language design space. The other path forward is to search for a small number of highly general primitives that allow us degrade many of the constructs that are built primitively into languages today insead to modularly composable library constructs. Encouragingly, primitives of this sort do occasionally arise. For example, a recent revision of OCaml added support for "generalized algebraic data types" (GADTs), based on research on guarded recursive datatype constructors [23]. Using GADTs, OCaml was able to move some of the *ad hoc* machinery for typechecking operations that use format strings, like `sprintf`, out of the language and into a library (note, however, that syntactic machinery remains built in).

1.2 Contributions

In this work, we introduce primitive language constructs that take further steps down the second path just described. In particular, we introduce the following primitives:

1. **Typed syntax macros** (TSMs), introduced in Part I, reduce the need to primitively build in derived concrete syntactic forms specific to library constructs (e.g. list syntax as in SML or XML syntax as in Scala and Ur/Web), by giving library providers static control over the parsing and expansion of delimited segments of textual syntax (at a specified type or parameterized family of types). We begin by showing explicitly-invoked TSMs, then show how to associate a privileged TSM with a type declaration and then rely on a local type inference scheme to invoke these implicitly.
2. **Metamodules**, introduced in Part II, reduce the need to primitively build in the type structure of constructs like records (and variants thereof), labeled sums and other interesting constructs that we will introduce later by giving library providers programmatic "hooks"

directly into the semantics, which are specified as a *type-directed translation semantics* (introduced in Chap. 2).

Both TSMs and metamodules make extensive use of *static code generation* (also called *static* or *compile-time metaprogramming*), meaning that the relevant rules in the static semantics of the language call for the evaluation of *static functions* that generate static representations of expressions and types. This design also has conceptual roots in earlier work on *active libraries*, which similarly envisioned using compile-time computation to give library providers more control over aspects of the language and compilation process [21].

As vehicles for this work, we plan to formally specify small typed lambda calculi that capture each of the novel primitives that we introduce "minimally". We will also describe (but not formally specify) a new "full-scale" functional language called Verse.[3] The reason we will not follow Standard ML [17] in giving a complete formal specification of Verse is both to emphasize that the primitives we introduce can be considered for inclusion in a variety of language designs, and to avoid distracting the reader with specifications for "orthogonal" primitives that are already well-understood in the literature. We will give a brief overview covering how these languages are organized in Chap. 2.

The main challenge in the design of these primitives will come in ensuring that they are metatheoretically well-behaved. If we are not careful, many of the problems that arise when combining language dialects, discussed earlier, could simply shift into the semantics of these primitives.[4] Our main technical contributions will be in rigorously showing how to address these problems in a principled manner. In particular, syntactic conflicts will be impossible by construction and the semantics will validate code statically generated by TSMs and metamodules to maintain a strong *hygienic type discipline* and, most uniquely, powerful *modular reasoning principles*. In other words, library providers will have the ability to reason about the constructs that they have defined in isolation, and clients will be able to use them safely in any program context and in any combination, without the possibility of conflict.[5] We will make these notions completely precise as we continue.

Thesis Statement

In summary, we propose a thesis defending the following statement:

> A functional programming language can give library providers the ability to express new syntactic expansions and new types and operators atop a small type-theoretic internal language while maintaining a hygienic type discipline and modular reasoning principles.

[3]We distinguish Verse from Wyvern, which is the language referred to in prior publications about some of the work that we will describe, because Wyvern is a group effort evolving independently in some important ways.

[4]This is why languages like Verse are often called "extensible languages", though this is somewhat of a misnomer. The defining characteristic of an extensible language is that it *doesn't* need to be extended in situations where other languages would need to be extended. We will avoid this somewhat confusing terminology.

[5]This is not quite true – simple naming conflicts can arise. We will tacitly assume that they are being avoided extrinsically, e.g. by using a URI-based naming scheme as in the Java ecosystem.

1.3 Disclaimers

Before we continue, it may be useful to explicitly acknowledge that completely eliminating the need for dialects would indeed be asking for too much: certain design decisions are fundamentally incompatible with others or require coordination across a language design. We aim only to decrease the need for dialects.

It may also be useful to explicitly acknowledge that library providers could leverage the primitives we introduce to define constructs that are in "poor taste". We expect that in practice, languages like Verse will come with a standard library defining a carefully curated collection of standard constructs, as well as guidelines for advanced users regarding when it would be sensible to use the mechanisms we introduce (following the example of languages that support features like operator overloading or type classes [11], which also have the potential for such "abuse"). For most programmers, using Verse should not be substantially different from using a language like ML or one of its dialects.

Finally, Verse is by design not a dependently-typed language like Coq, Agda or Idris, because these languages do not maintain a phase separation between "compile-time" and "run-time". This phase separation is useful for programming tasks (where one would like to be able to discover errors before running a program, particularly programs that may have an effect) but less so for theorem proving tasks (where it is primarily the fact that a pure expression is well-typed that is of interest, by the propositions-as-types principle). Verse is designed to be used for programming tasks where SML, OCaml, Haskell or Scala would be used today, not for advanced theorem proving tasks. That said, we conjecture that the primitives we describe could be added to languages like Gallina (the "external language" of the Coq proof assistant [16]) or to the program extraction mechanisms of proof assistants like Coq with modifications, but do not plan to pursue this line of research in this dissertation.

Chapter 2

Language Overview

TODO: integrate this point An important observation is that dialects that have substantially different type structure are still often implemented by translation to a simple and stable intermediate language. For example, the Glasgow Haskell Compiler's intermediate language has remained stable for many years, even as the number of Haskell dialects that the compiler supports has grown. Similarly, the primary Scala compiler uses the Java virtual machine (JVM) bytecode language as an intermediate language, which has also evolved relatively slowly.

2.1 External Language

2.2 Internal Language

2.3 Static Language

2.4 Module Language

Part I

Modularly Programmable Syntax

Chapter 3

Motivation

3.1 Motivating Examples

3.1.1 Lists

3.1.2 HTML

3.1.3 Regular Expressions

3.1.4 Monadic Commands

3.1.5 Quasiquotation

3.2 Existing Approaches

3.2.1 Dynamic String Parsing

3.2.2 Direct Syntax Extension

Related work I haven't mentioned yet:

- Fan: http://zhanghongbo.me/fan/start.html
- Well-Typed Islands Parse Faster: `http://www.ccs.neu.edu/home/ejs/papers/tfp12-island.pdf`

3.2.3 Term Rewriting

Chapter 4

Typed Syntax Macros

4.1 Examples

4.2 Minimal Formalization

4.3 Parameterized TSMs

Chapter 5

Type-Specific Languages

5.1 Examples

5.2 Minimal Formalization

5.3 Parameterized TSLs

5.4 Conclusion & Future Work

Part II

Modularly Programmable Type Structure

Chapter 6

Motivation

6.1 Motivating Examples

6.2 Existing Approaches

Chapter 7

Metamodules

Chapter 8

Conclusion & Future Work

Bibliography

TODO (Later): List conference abbreviations.
TODO (Later): Remove extraneous nonsense from entries.

[1] Nick Benton and Andrew Kennedy. Interlanguage Working Without Tears: Blending SML with Java. In *ICFP '99*, pages 126–137, 1999. ISBN 1-58113-111-9. doi: 10.1145/317636.317791. URL `http://doi.acm.org/10.1145/317636.317791`. 1.1

[2] Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 122–133. ACM, 2010. ISBN 978-1-4503-0019-3. URL `http://doi.acm.org/10.1145/1806596.1806612`. 1.1

[3] Adam Chlipala. Ur/Web: A simple model for programming the web. In *POPL '15*, pages 153–165, 2015. ISBN 978-1-4503-3300-9. URL `http://dl.acm.org/citation.cfm?id=2676726`. 1.1

[4] Sebastian Erdweg and Felix Rieger. A framework for extensible languages. In *GPCE '13*, pages 3–12, 2013. 1.1

[5] Sebastian Erdweg, Tillmann Rendel, Christian Kastner, and Klaus Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOPSLA '11*, pages 187–188, 2011. 1.1

[6] 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 H. Wachsmuth, and Jimi van der Woning. The state of the art in language workbenches. In *Software Language Engineering (SLE '13)*. 2013. 1.1

[7] Matthew Flatt. Creating languages in Racket. *Commun. ACM*, 55(1):48–56, January 2012. ISSN 0001-0782. doi: 10.1145/2063176.2063195. URL `http://doi.acm.org/10.1145/2063176.2063195`. 1.1

[8] Matthew Fluet, Mike Rainey, John H. Reppy, Adam Shaw, and Yingqi Xiao. Manticore: a heterogeneous parallel language. In *Workshop on Declarative Aspects of Multicore Programming (DAMP '07)*, pages 37–44. ACM, 2007. ISBN 978-1-59593-690-5. URL `http://doi.acm.org/10.1145/1248648.1248656`. 1.1

[9] M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley Professional, 2010. 2

[10] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. In *PLDI '05*, pages 62–72, 2005. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065019. URL `http://doi.acm.org/10.1145/1065010.1065019`. 1.1

[11] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, March 1996. ISSN 0164-0925. doi: 10.1145/227699.227700. URL `http://doi.acm.org/10.1145/227699.227700`. 1.3

[12] Robert Harper. Programming in Standard ML. `http://www.cs.cmu.edu/~rwh/smlbook/book.pdf`. Retrieved June 21, 2015., 1997. 1.1

[13] Simon L Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003. 1.1

[14] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.01 Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, September 2013. 1.1, 1.1

[15] Louis Mandel and Marc Pouzet. ReactiveML: a reactive extension to ML. In *PPDP '05*, pages 82–93. ACM, 2005. 1.1

[16] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0. 1.3

[17] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. 1.1, 1.2

[18] Tom Murphy, VII., Karl Crary, and Robert Harper. Type-safe Distributed Programming with ML5. In *Proceedings of the 3rd Conference on Trustworthy Global Computing*, TGC'07, pages 108–123, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78662-7, 978-3-540-78662-7. URL `http://dl.acm.org/citation.cfm?id=1793574.1793585`. 1.1

[19] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In *ECOOP '14*, 2014. 1.1

[20] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*, pages 79–96. Intellect Books, Bristol, UK, ISBN 1-84150144-1, Munich, Germany, February 2006. 1.1

[21] Todd L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University, 2004. 1.2

[22] Martin P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4): 147–161, 1994. 1.1

[23] Xi, Chen, and Chen. Guarded recursive datatype constructors. In *POPL '03*, 2003. 1.1