Modular Type Constructors

Conservatively Composing Typed Language Fragments

Cyrus Omar and Jonathan Aldrich

Carnegie Mellon University, Pittsburgh, PA 15213, USA
`{comar,aldrich}@cs.cmu.edu`

Abstract. Abstraction providers sometimes need to extend a programming language with new base types and their corresponding operators. Ideally, such language fragments could be defined and reasoned about separately. Unfortunately, existing tools and techniques do not come equipped with strong modular reasoning principle – fundamentally, each combination of fragments becomes its own dialect of the language for which metatheoretic and compiler correctness results must be established monolithically. Recent work has started to address this problem, e.g. by showing how to modularly reason about desugarings atop a fixed type system. Our focus here is on safely composing type system fragments. We organize fragments around type constructors, as is usual practice when describing type systems, and sidestep the difficulties of composing abstract syntax (i.e. the expression problem) by delegating control over a small, uniform abstract syntax in a type-directed manner to logic associated with a type constructor. The paper is organized around a minimal calculus, @λ, that permits surprisingly practical examples. We establish several strong semantic guarantees, notably *type safety*, *stability of typing* under extension and *conservativity*: that the *type invariants* that a finite set of fragments maintain are conserved under extension. This involves lifting typed compilation techniques into the semantics and enforcing an abstraction barrier around each fragment using a form of "internal" type abstraction. Conservativity then follows from classical parametricity results, so these *modular type constructors* can be reasoned about like modules in an ML-like language.

1 Introduction

Typed programming languages are often described in *fragments*, each defining contributions to a language's concrete syntax, abstract syntax, static semantics and dynamic semantics. In his textbook, Harper organizes fragments around type constructors, each introduced in a different chapter [6]. A language is then identified by a set of type constructors, e.g. $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1 \times +\}$ is the language that builds in partial function types, polymorphic types, recursive types, nullary and binary product types and binary sum types (its syntax is shown in Figure **??**, discussed further below). It is left implied that the metatheory developed separately is conserved when fragments like these are composed to form a language.

Luckily, fragment composition is not an everyday programming task, because fragments like these are "general purpose" in that they make it possible to construct *isomorphic embeddings* of many other fragments as "libraries". For example, lists can be placed in isomorphism with polymorphic recursive sum of products, $\forall(\alpha.\mu(t.1 + (\alpha \times t))))$. Languages providing datatypes in the style of ML are perhaps most directly oriented around embeddings like this (preserving type disequality requires adding some form of generativity, as ML datatypes also expose).

Unfortunately, situations do sometimes arise where using these fragments to establish an isomorphic embedding that preserves a desirable fragment's static and dynamic semantics (including bounds specified by its cost semantics) is not possible. Embeddings can also sometimes be unsatisfyingly *complex*, as measured by the cost of the extralinguistic computations that are needed to map in and out of the embedding and, if these must be performed mentally, considering cognitive metrics like error message comprehensibility.

When an embedding is too complex, abstraction providers have a few options, discussed in Sec. **??**. When an embedding is not possible, however, or when these options are insufficient, providers are often compelled to introduce these fragments by extending an existing language, thereby forming a new *dialect*. To save effort, they may do so by forking existing artifacts or leverage tools like compiler generators or language frameworks, also reviewed in Sec. **??**. Within the ML lineage, dialects that go beyond $\mathcal{L}\{\rightharpoonup \forall\, \mu\, 1 \times +\}$ abound:

1. **General Purpose Fragments:** A number of variations on product types, for example, have been introduced in dialects: n-ary tuples, labeled tuples, records (identified up to reordering), structurally typed or row polymorphic records [4], records with update and extension operators [9], mutable fields [9], and "methods" (i.e. pure objects [2, **?**]).[1] Sum types are also exposed in various ways: finite datatypes, open datatypes [10], hierarchically open datatypes [13], polymorphic variants [9] and ML-style exception types. Other generally useful fragments are also built in, e.g. `sprintf` in the OCaml dialect statically distinguishes format strings from strings.
2. **Specialized Fragments:** Fragments that track specialized static invariants to provide stronger correctness guarantees, manage unwieldy lower-level abstractions or control cost are also often introduced in dialects, e.g. for data parallelism [5], distributed programming [14], reactive programming [11], authenticated data structures [12], databases [15] and units of measure [8].
3. **Foreign Fragments:** A safe and natural foreign function interface (FFI) can be a valuable feature (particularly given this proliferation of dialects). However, this requires enforcing the type system of the foreign language in the calling language. For example, MLj builds in a safe FFI to Java [3].

[1] The Haskell wiki notes that "No, extensible records are not implemented in GHC. The problem is that the record design space is large, and seems to lack local optima. And all reasonable variants break backward compatibility. As a result, nothing much happens." [1]

This dialect-oriented state of affairs is, we argue, unsatisfying: a programmer can choose either a dialect supporting a principled approach to distributed programming, or one that builds in support for statically reasoning about units of measure, but there may not be an available dialect supporting both. Combining dialects is non-trivial in general, as we will discuss. Using different dialects separately for different components of a program is also untenable: components written in different dialects cannot always interface safely with one another (i.e. an FFI, item 3 above, is needed).

These problems do not arise when a fragment can be exposed as an isomorphic embedding because modern *module systems* can enforce abstraction barriers that ensure that the isomorphism need only be established in the "closed world" of the module. This is useful because it does not impose proof obligations on clients in the "open world". For situations where an embedding is not evident, however, mechanisms are needed that make specifying, implementing and reasoning about direct fragment composition similarly modular. Such mechanisms could ultimately be integrated directly into a language, blurring the distinction between fragments and libraries and decreasing the need for new dialects. Importing fragments that introduce new syntax and semantics would be as easy as importing a new module is today. In the limit, the community could rely on modularly mechanized metatheory and compiler correctness results, rather than requiring heroic efforts on the part of individual research groups as the situation exists today.

Contributions In this paper, we take substantial steps towards this goal by constructing a minimal but powerful core calculus, @λ (the "actively typed" lambda calculus). This calculus is structured in a manner similar to the Harper-Stone semantics for Standard ML [7], consisting of an *external language* (EL) governed by a typed translation semantics targeting a fixed *internal language* (IL). Rather than building in a monolithic set of type constructors, however, the typechecking judgement is indexed by a *tycon context*. Each tycon defines the semantics of its associated operators via functions written in a *static language* (SL). Types are values in the SL.

We introduce @λ by example in Sec. **??**, demonstrating its surprising expressive power and detailing its semantics, then examine its key metatheoretic properties in Sec. **??**, beginning with *type safety* and *unicity of typing* and touching on *decidability of typechecking*. We then consider properties that relate to composition of tycon definitions: *hygiene*, *stability of typing* and a key modularity result, which we refer to as *conservativity*: any invariants that can be established about all values of a type under *some* tycon context (i.e. in some "closed world") are conserved in any further extended tycon context (i.e. in the "open world").

We combine several interesting type theoretic techniques, applying them to novel ends: 1) a bidirectional type system permits flexible reuse of a fixed syntax; 2) the SL serves both as an extension language and as the type-level language; we give it its own statics (i.e. a *kind system*); 3) we use a typed intermediate language and leverage corresponding *typed compilation* techniques, here lifted into the semantics of the EL; 4) we leverage internal type abstraction implicitly as an effect during normalization of the SL to enforce abstraction barriers between type constructors. As a result, conservativity follows from the same elegant parametricity results that underly abstraction theorems for module systems. Like modules, reasoning about these *modular type constructors* does not require mechanized specifications or proofs: correctness issues in the type constructor logic necessarily causes typechecking to fail, so even extensions that are not proven correct can be distributed and "tested" in the wild without compromising the integrity of an entire program (at worst, only values of types constructed by the tycon being tested may exhibit undesirable properties). We conclude by discussing some limitations, proposing some richer variants of the calculus and discussing related work (Sec. **??**).

References

1. GHC/FAQ. `http://www.haskell.org/haskellwiki/GHC:FAQ#Extensible_Records`.
2. J. Aldrich. The power of interoperability: why objects are inevitable. In A. L. Hosking, P. T. Eugster, and R. Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 101–116. ACM, 2013.
3. N. Benton and A. Kennedy. Interlanguage Working Without Tears: Blending SML with Java. In *ICFP '99*, pages 126–137, New York, NY, USA, 1999. ACM.
4. L. Cardelli. A semantics of multiple inheritance. In *Proc. Of the International Symposium on Semantics of Data Types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.
5. M. Chakravarty, R. Leshchinskiy, S. Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, pages 10–18. ACM, 2007.
6. R. Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, 2012.
7. R. Harper and C. Stone. A Type-Theoretic Interpretation of Standard ML. In *IN Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
8. A. Kennedy. Types for units-of-measure: Theory and practice. In Z. Horváth, R. Plasmeijer, and V. Zsók, editors, *CEFP*, volume 6299 of *Lecture Notes in Computer Science*, pages 268–305. Springer, 2009.
9. X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system release 4.01 Documentation and user's manual.* Institut National de Recherche en Informatique et en Automatique, September 2013.
10. A. Löh and R. Hinze. Open data types and open functions. In *PPDP*, pages 133–144. ACM, 2006.
11. L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *PPDP*, pages 82–93. ACM, 2005.
12. A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated Data Structures, Generically. In *POPL '14*, pages 411–423, New York, NY, USA, 2014. ACM.
13. T. D. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Trans. Program. Lang. Syst*, 26(5):836–889, 2004.

14. T. Murphy, VII., K. Crary, and R. 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.
15. A. Ohori and K. Ueno. Making Standard ML a Practical Database Programming Language. In *ICFP*, ICFP '11, pages 307–319, New York, NY, USA, 2011. ACM.

A Appendix