ICFP '15 Paper #117

cyology+icfp15@gmail.com Profile · Help · Sign out

Your submissions (All) Search

Main Edit

#117 Modularly Metaprogrammable Type Structure

Email notification
Select to receive email on updates to reviews and comments.

PC conflicts
Neelakantan Krishnaswami

Submitted 372kB 28 Feb 2015 5:55:43am EST
dab7885d3397c68ddb01e6ce78c4148290053900
Supplementary material/atlam-icfp15-supplement.pdf (497kB)

You are an **author** of this paper.

Abstract
We introduce Verse, a programming language specified, like many modern languages are, by a typed translation semantics targeting a common intermediate language (e.g. CIL [more]

Authors
C. Omar, J. Aldrich [details]

Supplementary material
atlam-icfp15-supplement.pdf

Topics

	OveMer	RevExp
Review #117A	1	3
Review #117B	4	2
Review #117C	2	3

Edit paper · Add response

Reviews in plain text

Review #117A

Plain text

Overall merit
1. Reject

Reviewer expertise
3. Knowledgeable

Paper summary
This paper addresses the problem of definable type structure: instead of building rich type structure as primitive constructs, the Verse allows various forms of type to be defined. The semantics of those types is expressed via translation to a simple core language (System F plus products, sums and recursive types). The source language includes concepts for type abstraction
and metaprogramming to enable type definition.

Evaluation
The introduction makes bold claims about the sorts of type structure that can
be expressed in Verse, but I'm not convinced that the language lives up to those promises.

In particular, the paper does not compare itself to other languages that support programmable type structure, including Haskell, Agda and other dependently-typed languages. The primary functionality seems to be type level
computation, and there are many examples of encoded type structure in these
languages.

For example, both Haskell and Agda can define record types using the encoding
presented in the paper. As the paper points out, there are many different ways
to encode records, with enough trade offs such that no one encoding dominates.
How is the Verse encoding better? Is there something special about Verse that
will enable it to avoid this tar pit?

In particular, it doesn't appear that Verse is expressive enough to reach all of the encodings that are available in Haskell or Agda. The problem is that source Verse does not include any form of polymorphism! This lack of polymorphism is a significant limitation for Verse given its primary role in typed functional languages. It also obscures limitations of the type encodings
themselves.

This paper does not address the real problem with the interaction between definable type structure and polymorphism. In that situation, one often wants
stronger type equalities than are derived from type-level computation. For example, when extensible records are based on finite maps, it is useful to have a theory of such maps available during type checking. For example, for
any two concrete maps m1 and m2, we can use type-level computation to show
that m1 `union` m2 = m2 `union` m1. However, if either (or both) of m1 and m2
are type variables, then we lose this equivalence.

That said, I don't see how one could add polymorphism to source Verse and
still support elaboration to the core language. Although the core language has
polymorphism, I doubt that it is expressive enough. For comparison, the GHC
core language FC is similar to Verse core, but also includes equality coercions so that it can support type-level computation (among other things).

Comments for author(s)

This metaprogramming part of this design seems to be motivated by a goal of
making sure that all trace of source-language type definitions are gone by the

time programs are compiled to core. Why is this important? The core
language
cannot express source language abstractions, is that useful in practice?
I'm
not sure that I follow the FFI argument: certainly two different programs
using different source-level type structure cannot communicate directly if
their abstractions are preserved to core. But an expressive core language
should be able to describe how to convert the abstractions from one
language
into the abstractions of another. That is essentially what an FFI is, and
being able to type this FFI more strongly seems useful.

Why introduce the new oxymoronic term "static dynamics" to refer to
compile-time reduction? What is the motivation for "equality kinds"? (Many
languages, like F-omega or Coq/Agda allow the type checker to compare
type-level lambda-abstractions.)

Why not use similar syntax for quote and unquote in the concrete syntax
(Figure 1) and the abstract syntax? (And the syntax in Figure 1 needs to be
explained. I'm only assuming that % is unquote.)

The kinding rules in section 3.4 (k-ity-unquote) and (k-ity-trans) involve
several forms in the conclusion. This seems strange to me and usually
means
that the system is not very compositional.

Review #117B

Plain text

Overall merit

4. Accept

Reviewer expertise

2. Some familiarity

Paper summary

The paper presents a programming language ("Verse") with only function
types built in as primitive: other type constructors are defined by
the programmer using "tycon structures" (record types and regular
expression types are given as example). Translation into a core
language is given, along with bidirectional type inference and
metatheory.

Evaluation

This is a nice idea, motivated with well-chosen examples, and with
solid technical development to back it up.

The idea could be promoted even better perhaps by considering the
analogous situation for term-level constructs, where the trend has
been for more advanced languages to push as much as possible into
libraries. Compare C built-in operators, for example, with the more
extensible approach of Haskell (put them in a Prelude, supported by
type classes). Or Haskell's do notation, for user-defined monads; also
F#'s computation expressions, that serve a similar role.

A significant omission is any discussion of quantification over types in the external language, either universal (for polymorphic definitions) or existential. The very last paragraph of Section 6 conjectures that it is simply complementary. I suspect that there is more to it than that. If the type translation specified by a tycon is not stable under substitution of types for type variables then the translation may not work.

A couple of other pieces of related work might be worth mentioning. In addition to proof assistants, there are "logical frameworks" such as Twelf. These are a bit different, perhaps closer to the aims of Verse, though typically used to build metatheory rather than programming language implementations.

Also F# type providers, which also in a sense extend the core type system of a language, with the focus being on interpreting external data through static types and operators.

Review #117C

Plain text

Overall merit

2. Weak reject

Reviewer expertise

3. Knowledgeable

Paper summary

The paper introduces "Verse", a programming language that consists of a minimal core with basically only function types built in.
To make up for this, Verse provides a meta-programming mechanism that gives library writers the ability to modularly define new type structures. In particular, one can express types like records this way.

Evaluation

This paper tackles an exciting subject: programmable compilers. By building
on Wyvern, the Verse language gets library extensible syntax, and this paper
describes the next step where a library can provide new type structures and associated operations.

Unfortunately, the paper does not explain Verse very well. Even after reading
it multiple times and trying to understand it all I still have no idea how much expressive power there is in the type system and what its limitations are. This is not helped by having just a single concrete example giving at the
beginning. Moreover, the conference example is fairly simple: this can be done
in a straightforward way with template Haskell too (and that is also fully typed and safe). Now, since Verse integrates deeply into the type system, there
must be more expressiveness but from the text in the paper I am unable to figure out what that exactly is -- and where its boundaries are. Some hints about the boundaries are given in the conclusion (Section 7): variable

binding cannot be changed, and "opcons" cannot directly invoke each other.

The typing rules are very complex; with the bidirectional type inference it is very hard to figure out what programs would be accepted and which ones
would need type annotations. What makes this harder is that the usage of names seems inconsistent:
- Figure 2, internal types, uses just *\tau for types* (include polymorphic ones)
- But Figure 3, the external language, uses *\sigma for types* annotations
- And then Figure 4, the static language, uses *\sigma for static terms*!
After reading the text carefully, it becomes clear that "\sigma" indeed stands
for the static terms but of kind Ty and that there are only three possible forms. It would help so much to put that in a separate figure and name it differently,
ie. \sigma is usually used for polymorphic types and this is very confusing.
Given the above, and looking at the type rules, it seems there is no polymorphism.
Is that the case?

Further confusion: In the example given, "year = paper#conf#2", the "conf#2"
projects the second capture as the year. This means that "#" is also defined for "rstr" (how does Verse deal with that kind of static overloading?). However,
in Figure 7 I cannot find the "#" operation for the rstr type, just for the record
type? And what is "conc"?

Overall, even after multiple reads this reviewer could not understand all the technicalities and does not feel confident accepting the paper in the current form.
The presentation should be improved to be more accessible with clear explanations,
examples, and discussion about the limitations.

Response

Cyrus Omar <cyology+icfp15@gmail.com> 24 Apr 2015 5:10:25am EDT

Response submitted.

The authors' response should address reviewer concerns and correct misunderstandings. Make it short and to the point; the conference deadline has passed. Try to stay within 500 words.

Thank you for your careful readings!

All three reviewers asked about parameteric polymorphism in the EL. Our intent is to give a _minimal_ account of a new approach, so it is reasonable to base it on the STLC rather than System F. That said, the only question would be regarding how tycase interacts with universally quantified type variables (which would be static values of kind Ty, not static variables). The simplest approach is to raise an error when an opcon attempts to case analyze it. R2's concern is addressed: an opcon's semantics would then be stable under substitution of types for type variables in well-typed terms (which is all that is necessary). Again, a full technical exposition is beyond the scope of the paper but we can more explicitly emphasize that the EL is based on the STLC early in the final version.

R1

R1 states that 1) Verse's "primary functionality seems to be type-level computation" (TLC) and 2) that you can express the same semantics using TLC in Agda and Haskell. Neither are accurate:

1) Verse's primary functionality is a form of metaprogramming where terms and types are both statically manipulated.

2) Extensible records must be built primitively into Agda and are not available in Haskell. At best, one can try to define a sort of "modular" encoding using closed type families in Haskell [1] but the static semantics are not preserved, e.g. projecting out a row that doesn't exist does not give you a static type error. The underlying implementation is also asymptotically slower -- this is why we are concerned with explicitly controlling translation and maintaining a phase separation.

We discussed type equality in Secs 3.3 and 4.1. We acknowledge that one could take a more sophisticated approach to type equality, perhaps using equality coercions, but this is complexity that would distract from the basic mechanism the paper introduces.

[1] https://wiki.haskell.org/CTRex

R3

Template Haskell cannot be used to define new type structure (only term rewriting at an existing type).

Submit Cancel Delete response

0 words left

HotCRP