ESOP 2019 (author)

Docs / Log out

Submission 65 | Help | Conference | News | EasyChair

Rebuttal

ESOP 2019 Submission 65

Submission information updates are disabled.

Click "Rebuttal" to **see the reviews sent to you for the rebuttal period and your response**.

For all questions related to processing your submission you should contact the conference organizers. Click here to see information about this conference.

All **reviews sent to you** can be found at the bottom of this page.

Submission 65	
Title	Handling Recursion in Generic Programming Using Closed Type Families
Paper:	(Nov 17, 09:43 GMT)
Author keywords	Datatype-generic programming Sums of products Recursion Overlapping instances Closed type families Zipper Mutually recursive datatypes Haskell
Abstract	Many of the extensively used libraries for datatype-generic programming offer a fixed-point view on datatypes to express their recursive structure. However, some other approaches, especially the ones based on sums of products, avoid the fixed point encoding. They facilitate implementation of generic functions that do not require to look at the recursive knots in a datatype representation, but raise issues otherwise. A widely used and unwelcome solution to the problem resorts to overlapping instances. In this paper, we present an alternative approach that uses closed type families to eliminate the need of overlap for handling recursion in datatypes. Moreover, we show that this idiom allows for generic programming with families of mutually recursive datatypes.
Submitted	Nov 09, 16:21 GMT
Last update	Nov 09, 16:21 GMT

Authors

first name	last name	email	country	affiliation	Web page	corresponding?
Anna	Bolotina	ann-bolotina@yandex.ru	Russia	Southern Federal University		✔

Artem	Pelenitsyn	a.pelenitsyn@gmail.com	United States	Northeastern University		✔

Reviews

Review 1	
Overall evaluation	**-1**: (weak reject) *Summary* This paper develops a technique for generic programming. The technique involves encoding *Comments* I am a regular functional programmer and have some experience with basic type theory (dependent types, singletons, higher kinds), but I do not normally use Haskell. I do see Haskell in conference papers from time to time, but I'm not up on the latest syntax for type families, higher kinds and the like. As a result, I found this paper very difficult to understand. From the beginning, there is a fair amount of terminology that I had a difficult time getting a grip on. The basic idea that one might prefer to encode datatypes as n-ary sums of products rather than binary sums of products, I could understand. Though, if doing so adds significant complexity, it is not obvious to me that it is worth doing. Is a cascade of pairs really all that problematic? Why? The authors merely write "They form an expressive instrument for defining generic functions in a more succinct and high-level style as compared to the classical binary sum-of-products views." --- that is not all that motivation for this paper (and the others the authors cite solving the same problem). When it comes to the issue of recursion, the authors write: "However, the SOP view does not reflect recursion points in generic representation types. So it naturally supports definitions of functions that do not require a knowledge about recursive occurrences, but otherwise becomes unhandy." I suppose a "recursion point" is some kind of indicator that a recursive type equation should be unrolled or something. But what is the real problem here? "unhandy" is somewhat vague and does not convince me there is a real problem to be solved. (There may be, but it doesn't convince someone unfamiliar with the area.) A solution to the problem is encode recursive positions "using the fixed-point approach" but unfortunately, I do not know what that is. My point, which I hope can help the authors in future drafts of their work, is that the outside reader has a great deal of difficulty following the introduction of the paper. Exactly what

the problem is and what barriers to writing real programs exist is not so well explained. Hence, the overall importance of the problem is somewhat unclear. It does seem to be a rather narrow problem and a problem that has been tackled before, but perhaps in ways that the authors can incrementally improve. Consequently, my hunch is that this work is likely better suited for a more specialized workshop than a broad conference like ESOP.

My difficulties with the paper did not end in the introduction as I was also unfamiliar with the Haskell type definition technology at work. For instance, I was uncertain even about the very first technical definition:

type family Code (a :: *) :: [[*]]

I wasn't sure what sort of thing that defined. As a result, I simply could not get very far through this paper. Some explaination of the syntax would have been helpful.

I asked a colleague about related work. My colleague suggested that there were a number of papers that may be related. If the authors could explain the relation to the following work, that would be helpful:

* Bahr's "Composing and decomposing data types: a closed type families implementation of data types à la carte" (WGP'14) seems relevant.

* "Sums of products for mutually recursive datatypes: the appropriationist's view on generic programming" (TyDe'18) by Victor Cacciari Miraldo and Alejandro Serrano seems very relevant -- perhaps to the point of subsuming this work.

* "Generic programming of all kinds" (Haskell'18), by Serrano and Miraldo, covers relevant territory, as well.

Reviewer's confidence **2**: (low)

Review 2

Overall evaluation **-3**: (strong reject)

The paper describes a library providing generic programming techniques for recursive datatypes. Unfortunately, there is very little new to be found here. The translations between datatypes and codes is (as the authors acknowledge) existing generic programming technology, while the representation of types using type-level lists appears first in Oliveira et al. [1] (uncited in the current paper). The type families the authors introduce to avoid overlap in defining instances over these code are also standard, and appear in (at least) Bahr [2] and Morris [3] (also both uncited). The only remaining difference arises from the treatment of recursion. However, this difference is entirely superficials. The previous work separates recursion not because of any fundamental difficulty, but to provide extensibility. The current paper uses GHC generics for the same purpose. The irrelevance is made clear by observing that the actual type-level machinery (type families and type classes) are identical in the two cases.

[1] Bruno C. d. S. Oliveira, Shin-Cheng Mu, and Shu-Hung You. 2015. Modular reifiable matching: a list-of-functors approach to two-level types. In Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15). ACM, New York, NY, USA, 82-93. DOI=http://dx.doi.org/10.1145/2804302.2804315

[2] Patrick Bahr. 2014. Composing and decomposing data types: a closed type families implementation of data types à la carte. In Proceedings of the 10th ACM SIGPLAN workshop on Generic programming (WGP '14). ACM, New York, NY, USA, 71-82. DOI=http://dx.doi.org/10.1145/2633628.2633635

[3] J. Garrett Morris. 2015. Variations on variants. In Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15). ACM, New York, NY, USA, 71-81. DOI=http://dx.doi.org/10.1145/2804302.2804320

Review 2

Overall evaluation	**-3**: (strong reject) PC discussion: the PC encourages the authors to take the related work seriously, and look forward to seeing this work in the future. The paper describes a library providing generic programming techniques for recursive datatypes. Unfortunately, there is very little new to be found here. The translations between datatypes and codes is (as the authors acknowledge) existing generic programming technology, while the representation of types using type-level lists appears first in Oliveira et al. [1] (uncited in the current paper). The type families the authors introduce to avoid overlap in defining instances over these code are also standard, and appear in (at least) Bahr [2] and Morris [3] (also both uncited). The only remaining difference arises from the treatment of recursion. However, this difference is entirely superficials. The previous work separates recursion not because of any fundamental difficulty, but to provide extensibility. The current paper uses GHC generics for the same purpose. The irrelevance is made clear by observing that the actual type-level machinery (type families and type classes) are identical in the two cases. [1] Bruno C. d. S. Oliveira, Shin-Cheng Mu, and Shu-Hung You. 2015. Modular reifiable matching: a list-of-functors approach to two-level types. In Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15). ACM, New York, NY, USA, 82-93. DOI=http://dx.doi.org/10.1145/2804302.2804315 [2] Patrick Bahr. 2014. Composing and decomposing data types: a closed type families implementation of data types à la carte. In Proceedings of the 10th ACM SIGPLAN workshop on Generic programming (WGP '14). ACM, New York, NY, USA, 71-82. DOI=http://dx.doi.org/10.1145/2633628.2633635 [3] J. Garrett Morris. 2015. Variations on variants. In Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15). ACM, New York, NY, USA, 71-81. DOI=http://dx.doi.org/10.1145/2804302.2804320
Reviewer's confidence	**5**: (expert)

Review 3

Overall evaluation	**1**: (weak accept) SUMMARY The paper presents a new approach for the sum-of-product style generic programming that improves on the current state of the art by avoiding the use of overlapping instances. In particular, it exploits the feature of closed type families provided by Haskell's extension. The new approach is described and then applied to the generic zipper recursive data structure to demonstrate adequacy. EVALUATION Overall, I enjoyed reading the paper. Its presentation is very clear and it is well-written. The paper also does a nice job at presenting the context and the motivating problem, which I appreciate. Personally, I am less certain about the advantages of avoiding fixed-points. In practice recursion seems to always be more conveniently handled through them, and there is quite enough infrastructure, both theoretical and practical, to support them. Thus, claims advocating for other approaches stating that "They facilitate implementation of generic functions that do not require to look at the recursive knots in a datatype representation…" are rather unsupported, in my view. Nevertheless, I appreciate that there are works in this line of work, and that the current paper contributes to the field. The general approach seems interesting and applicable, however, I have two main concerns about the actual contribution of the paper which prevent me from giving it a higher score. First, the approach is described via examples, namely: subterms, show, and fold and compos. However, not formal general theorem is given. On the other hand, there are several strong claims such as "…we claim that any generic function accessing recursive knots in the underlying datatype structure can be defined in the way described above for the task of subterms ." on page 8, that are left unsupported. Section 3.4 aims to provide the abstract description of the design pattern, but it lacked a formal statement and a soundness claim. Section 4 than takes the generic zipper as a test case for establishing the adequacy of the approach. However, it is unclear why this should constitute as an illuminating example. The fact that it is complex is not enough, and there is no discussion on its particular features that may result in it being a particularly important example. My second concern is that the entire work presented is done with respect to a particular implementation of a specific SOP approach – generics-sop. Thus, while you claim in the introduction that "We believe that the idea presented is suitable for any sum-of-products approach that does not employ the fixed point view…" no evidence that supports this claim is provided. Moreover, it is unclear how tailored the "solutions" to the examples presented are tailored to this specific implementation and approach, and will require a different solution in other approaches, if at all possible. As an example, in the solution to the subterms example, you rely on a specific machinery provided by generics-sop. The paper would highly benefit, in my view, from a more general view, or at least a discussion of some of the other approaches to SOP and how does the approach presented fits into them. OTHER COMMENTS TO THE AUTHORS 1. The paper could be more approachable to non-experts by providing more explanations/citations to some concepts, even if considered well-known. For example, GHC, XSafe, GADT, rose tree, etc. 2. Section 5 defines a translation between the two approaches, but it falls short at providing a correctness proof. 3. Page 23.:"The approach, described in this paper, is applicable to a range of datatypes that are monomorphically recursive". Is this meant to be a consequence of the general theorem that was never formally stated? In general, the discussion section is rather vague and incomplete, and in my opinion, does not add much to the paper.

	4. The related works section lack comparison to the current work. For instance, you state that there is another approach for restricting overlap using functional dependencies that resembles yours – in what way? How does your approach relates to [1]?
Reviewer's confidence	**3**: (medium)