An Elementary Semantics for PackageFormer with Applications to Universal Algebra (Short Paper)

—Draft—

Musa Al-hassy, Jacques Carette, Wolfram Kahl

Abstract

Folklore has held that any 'semantic unit' is essentially a type-theoretic context —this includes, for example, records and algebraic datatypes. Recently a flexible implementation of general contexts has risen in the setting of Martin-Lof Type Theory as so-called PackageFormer. These contexts come equipped with a number of so-called variationals that allow them to be viewed as concrete Agda packaging constructs —such as records, algebraic datatypes, and modules.

PackageFormers are implemented as an editor extension for Agda, but their theoretical boundaries are unclear. In this paper, we provide a simple semantics to the useful editor extension. Moreover, to demonstrate that the semantics is sufficient to capture a large number of use cases, we show how homomorphism constructions can be *mechanically derived* using the PackageFormer mechanism in a correct-by-construction fashion for over 300 equational theories —we are serving more than just a classical mathematical audience by considering tiny theories near the theory of Groups. This is the second contribution of this paper: Ensuring that a common pattern can be mechanically derived for a large number of use cases that people generally have written by hand.

MA:

- Group = Carrier × Identity × Operation × UnitLaws × AssocitivityLaw × InvOp × InvLaws
- 2 ⇐ There are two choices to whether we want a carrier or the empty theory.
- 2 ⇐ There are two choices to whether we want an elected element or not.
  - 2^2 ⇐ If we have the element, there are 4 choices whether we want left/right unit laws.
- 2 ⇐ There are two choices to whether we want a binary operation or not.
  - - 2 ⇐ If we have an bop, there are two choices to whether we want the AssocitivityLaw.
  - 2 ⇐ Two choices whether we have a unary operator or not.
    - 2^2 ⇐ If we have an InvOp, there are 4 choices whether we want left/right inverse laws.

  Total: $2 \times 2 \times (1 + 1 \times 2^2) \times (1 + 1 \times 2) \times (1 + 1 \times 2^2) = 300$
- Maybe we can jump to categories instead and obtain functors!
- Right now, I've tried M-sets; but simply have not tried if the existing setups works for cats —something to do.
  - If it doesn't work, discuss why not.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00

https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:

Musa Al-hassy, Jacques Carette, Wolfram Kahl. 2019. An Elementary Semantics for PackageFormer with Applications to Universal Algebra (Short Paper) —Draft—. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction [0/4] **BORING:UNCLEAR**

- ☐ Show example of a PackageFormer.
  - Demonstrate how: PackageFormer ≈ named context + header.
- ☐ Show example of how it can be used to give a record.
- ☐ Show how it can be used to give us a homomorphism definition.
- ☐ What are the pre- and post-conditions of the homomorphism construction?
  - This is what we are trying to solve.

2 A Grammar for PackageFormer [0/5] **RATHER:PROMISING**

- ☐ Grammar for PackageFormer heading.
- ☐ Grammar for element datatype.
- ☐ Grammar for "types".
  - We clearly cannot use any Agda/MLTT types.
- ☐ Define a fold for PackageFormer —the homepage currently calls this `graph-map` due to the graph theoretic nature of element dependencies.
- ☐ Prove that this fold preserves well-formedness & well-typedness of PackageFormers.
  - This is the semantics function!

- **PackageFormers are an M-Set and fold is an M-Set homomorphism!**
  Call this M-Set "**PF**".
  1. Two sorts: `PackageFormer` and `Element`.
  2. Action: `_◁_ : PackageFormer → Element → PackageFormer`
  3. Monoid on `PackageFormer`
     - Unit: The empty PackageFormer
     - Bop: Union of contexts
       - If they agree on their intersection, then union of element lists; otherwise 'crash' by yielding ANN.
       - ANN is the annihilating PackageFormer: It is a postulated value that acts as the zero of union.
       - This ensures that a crash propagates and so a union of PF's is ANN if any two items conflict.
       - E.g., "crash : PackageFormer⊥ → PackageFormer⊥ → Boolean" is defined with "crash ⊥ x ≈ true" and symmetrically so.
       - Taking ANN = ⊥, as a bottom element; e.g., `nothing`.
       - Proof outline of associativity:
       - Case 1: No crashes, then ordinary list catenation, which is associative.
       - Case 2: Some two items conflict, then ANN is propagated and both sides equal ANN.

2.1 Deriving Fold

1. Define a "Right M-Set" (close, but not really):

```
PackageFormer M-Set : Set_1 where
   Carrier_1     : Set
   Carrier_2     : Set
   _◁_           : Carrier_1 → Carrier_2 → Carrier_1
   ∅             : Carrier_1
   _∪_           : Carrier_1 → Carrier_1 → Carrier_1
   leftId        : {v : Carrier_2}  →  ∅ ◁v  ≡ v
   assoc         : {a b : Carrier_1} {v : Carrier_2} → (a ∪ b) ◁v  ≡  a ∪ (b ◁v)
```

2. Let $\mathcal{M}$ denote an M-Set.
3. For `fold : PF` ⟶ $\mathcal{M}$ to be an M-Set homomorphism, we are **forced** to have …
4. Two maps, `fold`$_i$ `: PF.Carrier`$_i$ → $\mathcal{M}$`.Carrier`$_i$
5. `fold`$_1$ is a monoid homomorphism
   a. Unit$_1$: `fold`$_1$ `∅ ≈ ∅`
   b. Assoc$_1$: `fold`$_1$ `(p ∪ q) ≈ fold`$_1$ `p ∪ fold`$_1$ `q`
6. Equivariance: `fold`$_1$ `(p ◁ e) ≈ fold`$_1$ `p ◁ fold`$_2$ `e`
7. Since a PackageFormer, by extensionality, can always be expressed as a finite sequence of extensions we find:

```
  fold_1 p
= {- Extensionality, with e_i elements of p -}
  fold_1 (∅ ◁ e_1 ◁ e_2 ◁ ⋯ ◁ e_n)
= {- Equivariance (6) -}
  fold_1 ∅ ◁ fold_2 e_1 ◁ ⋯ ◁ fold_2 e_n
= {- Unit (5.1) -}
  ∅ ◁ fold_2 e_1 ◁ ⋯ ◁ fold_2 e_n
= {- M-Set.leftId -}
  fold_2 e_1 ◁ ⋯ ◁ fold_2 e_n
```

8. Whence it seems `fold`$_1$ is defined uniquely in terms of `fold`$_2$ —which is unsurprising: **PackageFormers are an inductive type!**
9. TODO: Realise this argument within Agda!

3 An Application to Universal Algebra

SUPER_SKETCHY

- ☐ Grammar for the minimal language necessary to form homomorphism contexts.
  - How? What? Huh!?
  - I'm not sure I know what I'm thinking here.
  - I'm trying to "know" what the hom variational, from the webpage does!
- ☐ Define a function: `H : PFSyntax → List HomoSyntax`.
- ☐ Show a coherence such as `H(T ◁ e) = H T ◁ H e` where ◁ denotes context extension; i.e., append.
  - This would ensure that we have a 'modular' way to define homomorphisms.

Applications to structures that CS people are interested in?

- Monoids ⇐ for-loops
- Graphs ⇐ databases
- Lattices ⇐ optimisation

4 Conclusion & Next Steps **SKETCHY**

- Initial semantics is enough?
- Limitations?
- Dependent-type?
- A counterexample not covered by the semantics?
- Soundness?