What is a package?

Musa Al-hassy
Department of Computing And Software, McMaster University
Supervisors: Jacques Carette & Wolfram Kahl

Why care about 'packaging'?

How do you write an algorithm for the dot product?
This way? —With every piece mentioned in detail.

$$dot_{six} \ : \ \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$
$$dot_{six} \ x_1 \ x_2 \ x_3 \ y_1 \ y_2 \ y_3 \ = \ x_1 * y_1 + x_2 * y_2 + x_3 * y_3$$

Or with the pieces bundled up into "logically related units"?

$$dot_{vec} \ : \ \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$$
$$dot_{vec} \ \vec{x} \ \vec{y} \ = \ \vec{x}_1 * \vec{y}_1 + \vec{x}_2 * \vec{y}_2 + \vec{x}_3 * \vec{y}_3$$

What if we want to ignore the second dimension?

$$dot_{noTwo} \ : \ \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$$
$$dot_{noTwo} \ \vec{x} \ \vec{y} \ = \ \vec{x}_1 * \vec{y}_1 + \vec{x}_3 * \vec{y}_3$$

What if we only want to product along a given vector $\vec{y}$?

$$dot_{fixed} \ : \ \mathbb{R}^3 \to \mathbb{R}$$
$$dot_{fixed} \ \vec{x} \ = \ \vec{x}_1 * \vec{y}_1 + \vec{x}_2 * \vec{y}_2 + \vec{x}_3 * \vec{y}_3$$

All the 'forms' are useful, slightly distinct, but **behave essentially the same!**

Let's write one form, and generate the rest!

Write once, Generate many

```
// Write once,
Vector = Int[3]
Int dot_vec (Vector x, Vector y)
  =   x[0] * y[0]  +  x[1] * y[1]  +  x[2] * y[2]

// Generate many,
dot_six    =  dot_vec #flattened#
dot_noTwo  =  dot_vec #with# x[1] := 0, y[1] := 0
dot_fixed  =  dot_vec #with# y := [2, 3, 5]
```

Problem

There are many ways to bundle up data; rather than write each form by hand, we should "write one, generate many"!

Why the need?

The dot product example is small and could have been done *manually, by hand*; so why not?

- Tedious & error-prone,
- Obscures the corresponding general principles underlying them,
- No machine support to ensure implementation is correct,
- Does not scale!

Objective

Without jeopardising runtime performance [TB15], we want

Machine support deriving mechanisms for different 'views' of packages.

Which should be well-defined, with coherent semantic underpinning —such as an associated Cartesian Closed Category [Mog91].

Spot the Subtlety

Packaging forms are essentially problems of variable introduction,

$$\begin{array}{lll} & \forall x_1, x_2, x_3, y_1, y_2, y_3 & \bullet \ \cdots \\ \cong & \forall \vec{x}, \vec{y} & \bullet \ \cdots \\ \cong & \forall \vec{y} \ \bullet \ \forall \vec{x} & \bullet \ \cdots \end{array}$$

Possibly with constraints,

$$\forall \vec{x}, \vec{y} \quad | \quad \vec{x}_2 = 0 \ \wedge \ \vec{y}_2 = 0 \quad \bullet \ \cdots$$

The Next 700 Module Systems

We begin by manually writing a "package former" –a package whose concrete form is unspecified, akin to type constructors. –Look at the demo.

Novel Idea: *Higher-order packages*

With such a generic mechanism, we could derive nearly all other forms of packaging!

✓ Classes ✓ Records ✓ Interfaces ✓ Typeclasses ✓ Traits ✓ Dependent Products ✓ Dependent Sums

Write a method on the package former *once* then have it apply to all deviations!
Moreover, it supports the expected features of modularisation.

✓ Namespacing	✓ Information Hiding	✓ Polymorphism	✓ First- and Second-class Citizenship
✓ Applicative modules	✓ Generative modules	✓ Subtyping	✓ Computation Sharing

Modifying Containers

When we have a package, we can do a number of things with it [CO12]:

Flatten	'unpack' some of its nested packages
Instantiate	'pick values' for some of its contents
Parameterise	'fix' some of its contents
Hide	'ignore' some of its contents
Rename	'relabel' some of its contents
Mixin	'combine' packages over a common interface
Package-up	'bundle' some of its contents

Preliminary Research

- Emacs prototype actually producing derivatives from "package formers" —see the demo.
- Packages, top level modules, can be treated like any other *value*.
  - Uniform syntax for variations on packages and uniform computational definitions.
- Prototype interface is practical and accessible.
- Prototype appears promising to eliminate massive renaming boilerplate in [Kah18; Tea19] developments.

Next Steps

- Realise proposal in an existing industrial-strength compiler.
  - Hence, implementations are more than just 'research quality' [MRK18] but actually ready for a broad audience.
  - Users should be able to extend it within the core language –without resorting to inspecting the compiler.
- Utilise the opportunities provided by dependent-types [BD08; McK06] to provide a module system suitable for dependently-typed languages whose constructs are orthogonal. —*Dependent-types blur many lines!*
- Provide denotational semantics [Bui06] for the resulting module system.
  - Ensure all proofs are machine-checked, using Agda [Mar85].

References

[BD08] Ana Bove and Peter Dybjer. "Dependent Types at Work". 2008.
[Bui06] Alexandre Buisse. *Categorical Models of Dependent Type Theory*. 2006.
[CO12] Jacques Carette and Russell O'Connor. "Theory Presentation Combinators" (2012).
[Kah18] Wolfram Kahl. *Relation-Algebraic Theories in Agda*. 2018.
[Mar85] P. Martin-Löf. "Constructive Mathematics and Computer Programming". 1985.
[McK06] James McKinna. "Why dependent types matter". 2006.
[Mog91] Eugenio Moggi. "A Cateogry-Theoretic Account of Program Modules" (1991).
[MRK18] Dennis Müller, Florian Rabe, and Michael Kohlhase. "Theories as Types". 2018.
[TB15] Matus Tejiscak and Edwin Brady. "Practical Erasure in Dependently Typed Languages" (2015).
[Tea19] The Agda Team. *Agda Standard Library*. 2019.