Generic Proofs and Constructions in Agda

Samuel Klumpers
6057314

April 14, 2023

Outline

In this document I propose a master thesis project, in which I will investigate and attempt to counter the obstacles one can encounter when replacing one datastructure with a more complicated one, while keeping the end-result verifiable.

Section 1 describes some challenges that can arise when replacing structures, along with known ways to tackle them.

Section 2 lists literature and frameworks relevant to these questions.

Section 3, Section 4, and Section 5 summarize and contain the preliminary work done for this project.

Section 6 lists some remaining and new questions regarding the replacement of datastructures and proof transport, and proposes methods and ideas to tackle and answer the open challenges and questions.

Section 7 proposes a planning for the described project.

1 Program Vivisection

Agda [Tea23] is a functional programming language and a proof assistant, taking inspiration from languages like Haskell and other proof assistants such as Coq. We can write programs as we would in Haskell, and then express and prove their properties all inside Agda. This allows us to demonstrate the correctness of programs by formal proof rather than by testing. However, this level of formality also trades-off the uncertainty of testing for a time-investment to produce these proofs. In this thesis, we will explore a variety of methods of proving properties of our programs, focusing on the problems that one may encounter, presenting solutions as they arise. Let us sketch some of these problems.

First, merely adapting a program to Agda may already require changes to the datatypes used in it; for example, if a program manipulating a List uses the unsafe head function, then one is forced to replace the List by a datatype that ensures non-emptiness, such as a NonEmpty list or a length-aware vector Vec. On the other hand, there might be sections of a program where the concrete length is not relevant for correctness and only gets in the way. As a result, one

might find themselves duplicating common functions like concatenation _++_ to only alter their signatures.

However, the "new" datatype (Vec) is typically a simple variation on the old datatype (List) making small adjustments to the existing constructors; in this case, we decorate the nil and cons constructors with natural numbers representing the length. This kind of modification of types falls in the framework of *ornamentation* [KG16]; if two types are reified to their *descriptions*, then *ornaments* express whether the types are "similar" by acting as a recipe to produce one type from the other. By restricting the operations to the copying of corresponding parts, and the introduction of fields or dropping of indices, the existence of such an ornament ensures that the types have the same recursive structure. In general, ornaments allow us to introduce invariants into existing types, so that, as an example, one can derive ordered versions of lists or trees from their ordinary variants. Furthermore, using *patches* [DM14], we can in one direction ensure that _++_ on Vec agrees with its version for List under the ornament; in the other direction, a patch can also help us while defining this lifted variant.

Using ornaments, we can organize similar datatypes using ornaments; but we will also make use of relations between dissimilar datatypes. It is conventional to prototype a program using simpler types or implementations, and only replace these with more performant alternatives in critical places. While this may quickly turn into a refactoring nightmare in the general case, we can hope for a more satisfying transition if we restrict our attention to a narrower scope. As an example, we might start programming using Lists, but replace this with a Tree if we notice that the program spends most of its time in lookup operations. To gain a speedup, we will have to reimplement the operations on Tree. This would also double the number of necessary proofs; however, we have two ways to avoid this problem.

We will look at the more specific solution first. This solution is guided by the realization that even though List and Tree have different recursive structures, they have one commonality; namely, both resemble a number system. Lists and Braun trees[1] can both be presented by deriving them from unary and binary numbers respectively, as is made formal by Hinze and Swierstra [HS22]. One can then apply this *numerical representation* [Oka98] to simplify or trivialize properties of these datastructures. We will also see that we can interpret numerical interpretations more literally, and construct the representation directly as an ornament.

In the general case, we can apply representation independence. Equality of indiscernables ensures that substituting terms for equal terms cannot change the behaviour of a program, and, as types are terms, the same should hold for types. If we consider two types implementing a given interface, with an operation-preserving isomorphism, then representation independence tells us that the implementations must be functionally equivalent. In the case of trees and lists, this states that since converting a list to a tree preserves lookup, the

[1] Braun trees are a kind of binary tree, of which the shape is determined by its size.

outcome of a program that only uses lookup cannot change when substituting trees for lists. While a proof of this statement usually either exists in the metatheory, or is produced by manually weaving the conversions through our proofs, Cubical Agda allows us to internalize this independence [Ang+20].

We will first take a closer look at SIP [Ang+20] and give concrete examples of proof transport, which we can use to characterize equivalences of flexible two-sided arrays. Then we recall the constructions of numerical representations [HS22] and ornamentation [KG16], illustrating how we can define arrays from simpler types by providing interpretations into naturals. We will test these methods by using them to simplify the presentation of finger trees[2] [HP06]. After that, we will investigate other generic operations, such as the presentation of certain type transformations as ornaments, and the fair enumeration of recursive datatypes.

2 Related work

2.1 The Structure Identity Principle

If we write a program, and replace an expression by an equal one, then we can prove that the behaviour of the program can not change. Likewise, if we replace one implementation of an interface with another, in such a way that the correspondence respects all operations in the interface, then the implementations should be equal when viewed through the interface. Observations like these are instances of "representation indepencence", but even in languages with an internal notation of type equality, the applicability is usually exclusive to the metatheory.

In our case, moving from Agda's "usual type theory" to Cubical Agda, a cubical homotopy type theory, *univalence* [VMA19] lets us internalize a kind of representation independence known as the Structure Identity Principle [Ang+20], and even generalize it from equivalences to quasi-equivalence relations. We will also be able to apply univalence to get a true "equational reasoning" for types when we are looking at numerical representations.

Still, representation independence in non-homotopical settings may be internalized in some cases [Kap23], and remains of interest in the context of generic constructions that conflict with cubical.

2.2 Numerical Representations

Rather than equating implementations after the fact, we can also "compute" datastructures by imposing equations. In the case of container types, one may observe similarities to number systems [Oka98] and call such containers numerical representations. One can then use these representations to prototype new

[2]A finger tree is a nested type representing a sequence, designed to support amortized constant time en-/dequeueing at both ends, and logarithmic time concatenation and lookup.

datastructures that automatically inherit properties and equalities from their underlying number systems [HS22].

From another perspective, numerical representations run by using representability as a kind of "strictification" of types, suggesting that we may be able to generalize the approach of numerical representations, using that any (non-indexed) infinitary inductive-recursive type supports a lookup operation [DS16].

2.3 Ornamentation

While we can derive datastructures from number systems by going through their index types [HS22], we may also interpret numerical representations more literally as intstructions to rewrite a number system to a container type. We can record this transformation internally using ornaments, which can then be used to derive an indexed version of the container [McB14], or can be modified further to naturally integrate other constraints, e.g., ordering, into the resulting structure [KG16]. Furthermore, we can also use the forgetful functions induced by ornaments to generate specifications for functions defined on the ornamented types [DM14].

2.4 Generic constructions

Being able to define a datatype and reflect its structure in the same language opens doors to many more interesting constructions [EC22]; a lot of "recipes" we recognize, such as defining the eliminators for a given datatype, can be formalized and automated using reflection and macros. We expect that other type transformations can also be interpreted as ornaments, like the extraction of heterogeneous binary trees from level-polymorphic binary trees [SWI20].

3 Proof Transport via the Structure Identity Principle

Write here about: ""Summarize" the hidden sections here"

4 Types from Specifications: Ornamentation and Calculation

Write here about: ""Summarize" the hidden sections here"

5 More equivalences for less effort

Write here about: ""Summarize" the hidden sections here"

6 Research Question and Contributions

The research question of this project will be: *can we describe finger trees [HP06] in the frameworks of numerical representations and ornamentation [KG16], simplifying the verification of their properties as flexible two-sided arrays?* This question generates a number of interesting subproblems, such as that the number system corresponding to finger trees has many representations for the same number, which we expect to describe using quotients [VMA19] and reason about using representation independence [Ang+20]. If this is accomplished or deemed infeasible at an early stage, we can generalize the results we have to other related problems; for example, we may view the problem of generating arbitrary values for testing as an instance of an enumeration problem, through the lens of ornaments.

7 Planning

Date	Target
2023-04-24	Define and work out (better) simplified finger trees
2023-05-01	Force representability onto conventional finger trees
2023-05-08	Is there an ethical numrep with the bounds of finger trees?
2023-05-15	Experiment with enumeration
2023-05-22	How fair is enumeration/can we make better use of sharing?
2023-05-29	”
2023-06-05	Vectors are indexed, finger trees are not, SIP for indexed types?
2023-06-12	”
2023-06-19	Small universe ornaments
2023-06-26	Find out what HSIP means for us
2023-07-03	Holiday
2023-07-10	?
2023-07-17	”
2023-07-24	”
2023-07-31	”
2023-08-07	”
2023-08-14	”
2023-08-21	”
2023-08-28	?
2023-09-04	?
2023-09-11	Find more generic constructions
2023-09-18	”
2023-09-25	Can patches work better in C-c C-,
2023-10-02	Write
2023-10-09	”
2023-10-16	”
2023-10-23	”

2023-10-30	”
2023-11-06	”
2023-11-13	”
2023-11-20	Prepare presentation
2023-11-27	”
2023-12-04	”
2023-12-11	Present thesis
2023-12-18	-
2023-12-22	End date of research project

Table 1: The proposed planning for the research project.

References

[Ang+20] Carlo Angiuli et al. *Internalizing Representation Independence with Univalence*. 2020. DOI: 10.48550/ARXIV.2009.05547. URL: https://arxiv.org/abs/2009.05547.

[DM14] PIERRE-ÉVARISTE DAGAND and CONOR McBRIDE. “Transporting functions across ornaments”. In: *Journal of Functional Programming* 24.2-3 (Apr. 2014), pp. 316–383. DOI: 10.1017/s0956796814000069. URL: https://doi.org/10.1017%2Fs0956796814000069.

[DS16] Larry Diehl and Tim Sheard. “Generic Lookup and Update for Infinitary Inductive-Recursive Types”. In: *Proceedings of the 1st International Workshop on Type-Driven Development*. TyDe 2016. Nara, Japan: Association for Computing Machinery, 2016, pp. 1–12. ISBN: 9781450344357. DOI: 10.1145/2976022.2976031. URL: https://doi.org/10.1145/2976022.2976031.

[EC22] Lucas Escot and Jesper Cockx. “Practical Generic Programming over a Universe of Native Datatypes”. In: *Proc. ACM Program. Lang.* 6.ICFP (Aug. 2022). DOI: 10.1145/3547644. URL: https://doi.org/10.1145/3547644.

[HP06] RALF HINZE and ROSS PATERSON. “Finger trees: a simple general-purpose data structure”. In: *Journal of Functional Programming* 16.2 (2006), pp. 197–217. DOI: 10.1017/S0956796805005769.

[HS22] Ralf Hinze and Wouter Swierstra. “Calculating Datastructures”. In: *Mathematics of Program Construction*. Ed. by Ekaterina Komendantskaya. Cham: Springer International Publishing, 2022, pp. 62–101. ISBN: 978-3-031-16912-0.

[Kap23] Kevin Kappelmann. *Transport via Partial Galois Connections and Equivalences*. 2023. arXiv: 2303.05244 [cs.PL].

[KG16] HSIANG-SHANG KO and JEREMY GIBBONS. “Programming with ornaments”. In: *Journal of Functional Programming* 27 (2016), e2. DOI: 10.1017/S0956796816000307.

[McB14] Conor McBride. “Ornamental Algebras, Algebraic Ornaments”. In: 2014.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. USA: Cambridge University Press, 1998. ISBN: 0521631246.

[SWI20] WOUTER SWIERSTRA. “Heterogeneous binary random-access lists”. In: *Journal of Functional Programming* 30 (2020), e10. DOI: 10.1017/S0956796820000064.

[Tea23] Agda Development Team. *Agda*. 2023. URL: https://agda.readthedocs.io/en/v2.6.3/.

[VMA19] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. “Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types”. In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341691. URL: https://doi.org/10.1145/3341691.