Master thesis project proposal

Samuel Klumpers
6057314

April 19, 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 summarizes and contains the preliminary work done for this project.

Section 4 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 5 proposes a planning for the described project.

1 Program Vivisection

1.1 Program Verfication

What defines good programming? If there was a definitive correct answer to this question, it would be the holy grail of programming standards. However, we can at least say that a good program necessarily "does what it is supposed to"; which can mean one of many things, from "my user interface does not hang" to "my arithmetic logic unit design computes correctly".

Write here about: "Something about a bit of certainty through types, segue into verification"

Broadly speaking, program verification concerns itself with establishing program specifications, and verifying that programs behave according to their specifications. One can be confident that a program complies with its specification by testing it against a number of testcases. Simple testing may run the program on a set of inputs with known correct outputs, which is conventionally used to test the program on edge cases: the inputs which one finds most likely to produce incorrect outputs. In more intricate testing, the specification is typically separated from the inputs and is given as a function that computes whether an input-output pair is correct. The inputs can then still be taken from a set, or can be randomly drawn from generators constructed to produce sorts of input specific to the specification. Most of the work organizing these tests is best left to unit testing frameworks, such as QuickCheck, so that the program to be tested, the generators, and the specifications can be written in the same language; the framework can then assemble these into groups of testcases, combining the functionality of randomly generating the inputs, running the tests, and reporting the outcomes.

However, unit testing does not give absolute certainty. Dijkstra (1970): "Program testing can be used to show the presence of bugs, but never to show their absence!". If the program fails a test, then we know the program is flawed; if it does not fail, then we have simply not yet found an input on which it fails. When unit testing does not give enough certainty, we can instead try to prove the program to be correct. While not without pitfalls, if we view a program as a composition of mathematical functions, then we may be able to show its correctness by applying sequences of accepted rewriting rules. For example, a program computing an integer function is certainly correct if it reduces to this function after applying arithmetical rules. While a proof promises a lot about the correctness of a program, it will certainly take more effort to produce than the corresponding unit test. Furthermore, the correctness of the proof itself, which typically lives "on paper", would now have to be verified as well.

1.2 Proof Assistants

To ensure correctness of proofs, we can turn to proof assistants: 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.

Write here about: "Expand here about List"

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.

1.3 Initial Semantics

Write here about: "Expand here about initial algebras"

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.

1.4 Representation Independence

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 `List`s, 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.

Write here about: "Expand here about interfaces and normal representation independence"

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

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

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 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,

Write here about: "Expand here about Cubical Agda"

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] 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.

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 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 Preliminary work

3.1 Proof Transport via the Structure Identity Principle

To give an understanding of the basics of Cubical Agda [VMA19] and the Structure Identity Principle (SIP), we walk through the steps to transport proofs about addition on Peano naturals to Leibniz naturals. We give an overview of some features of Cubical Agda, such as that paths give the primitive notion of equality, until the simplified statement of univalence. We do note that Cubical Agda has two downsides relating to termination checking and universe levels, which we encounter in later sections.

Starting by defining the unary Peano naturals and the binary Leibniz naturals, we prove that they are isomorphic by interpreting them into eachother. We explain that these interpretations are easily seen to be mutual inverses by proving lemmas stating that both interpretations "respect the constructors" of the types. Next, we demonstrate how this isomorphism can be promoted into an equivalence or an equality, and remark that this is sufficient to transport intrinsic properties, such as having decidable equality, from one natural to the other.

Noting that transporting unary addition to binary addition is possible but not efficient, we define binary addition while ensuring that it corresponds to unary addition. We present a variant on refinement types as a syntax to recover definition from chains of equality reasoning, allowing one to rewrite definitions while preserving equalities.

We clarify that to transport proofs referring to addition from unary to binary naturals, we indeed require that these are meaningfully related. Then, we observe that in this instance, the pairs of "type and operation" are actually equated as magmas, and explain that this is an instance of the SIP.

Finally, we describe the use case of the SIP, how it generalizes our observation about magmas, and how it can calculate the minimal requirements to equate to implementations of an interface. This is demonstrated by transporting associativity from unary addition to binary addition, noting that this would save many lines of code provided there is much to be transported.

3.2 Types from Specifications: Ornamentation and Calculation

Using an example where we try to safely refactor a piece of code to use trees rather than lists, we motivate the need for a framework to organize different container types under a similar description. We explain that for indexed types, we can use representability, e.g., vectors correspond to functions out of finite types.

We describe how we can also derive these datastructures from functions [HS22], starting from a number system, yielding a numerical representation [Oka98]; this is demonstrated by an example deriving of vectors from unary naturals [HS22]. The vector type is computed by chains of equality reasoning like in the previous section, giving the correspondence to functions out of the finite type.

We illustrate how both (the functions out of a type and the concrete vectors) can implement an array interface, such as two-sided flexible arrays. We remark that the laws for such interfaces can be more easily proven on the function-based implementations, so that they can be transported to the concrete implementation.

Reflecting on this derivation, we note that the computation for binary naturals would be analogous, amending constructors constructors with fields holding appropriate number of elements. We relate this to ornamentation [KG16], which lets us relate types by recursive structure. After that, we give a short overview of the capabilities of descriptions and ornaments, and demonstrate these by deriving the list datatype from the unary natural type using an ornamental description.

We remark that this approach needs to be adjusted to use indices before it can be applied to binary naturals; we clarify how "metaphorical" this construction of binary trees is to binary numbers by letting the weight of the "digits" control the numbers of elements in each constructor. We explain that in doing so, the shape of the binary tree seen as a binary number then corresponds to the number of elements it contains.

After that, we give a completely different application of ornaments; we recall that the construction of heterogeneous lists, lists which contain elements with different types, is rather mechanical. Observing that a "heterogeneous X" is expressed as an "X-indexed X", we assert that this self-indexing can be captured as an ornament.

To define this ornament, we needed to include a parameter field in the definition of descriptions and ornaments. Then we define "heterogeneization" as an "ornament-computing function", which takes a description and produces an ornamental description. We demonstrate how we can heterogeneize lists and maybes, allowing us to produce a natural implementation of the heterogeneous head operation, and relate this to earlier work deriving heterogeneous random access lists [SWI20].

3.3 More equivalences for less effort

Noting that constructing equivalences directly or from isomorphisms as in Subsection 3.1 can quickly become challenging when one of the sides is complicated, we work out a different approach making use of the initial semantics of W-types instead. We claim that the functions in the isomorphism of Subsection 3.1 were partially forced, but this fact was not made use of.

First, we explain that if we assume that one of the two sides of the equivalence is a fixpoint or initial algebra of a polynomial functor (that is, the μ of a Desc′), this simplifies giving an equivalence to showing that the other side is also initial.

We describe how we altered the original ornaments [KG16] to ensure that μ remains initial for its base functor in Cubical Agda, explaining why this fails otherwise, and how defining base functors as datatypes avoids this issue.

In a subsection focussing on the categorical point of view, we show how we can describe initial algebras (and truncate the appropriate parts) in such a way that the construction both applies to general types (rather than only sets), and still produces an equivalence at the end. We explain how this definition, like the usual definition, makes sure that a pair of initial objects always induces a pair of conversion functions, which automatically become inverses. Finally, we explain that we can escape our earlier truncation by appealing to the fact that "being an equivalence" is a proposition.

Next, we describe some theory, using which other types can be shown to be initial for a given algebra. This is compared to the construction in Subsection 3.1, observing that intuitively, initiality follows because the interpretation of the zero constructor is forced by the square defining algebra maps, and the other values are forced by repeatedly applying similar squares. This is clarified as an instance of recursion over a polynomial functor.

To characterize when this recursion is allowed, we define accessibility with respect to polynomial functors as a mutually recursive datatype as follows. This datatype is constructed using the fibers of the algebra map, defining accessibility of elements of these fibers by cases over the description of the algebra. Then we remark that this construction is an atypical instance of well-founded recursion, and define a type as well-founded for an algebra when all its elements are accessible.

We interpret well-foundedness as an upper bound on the size of a type, leading us to claim that injectivity of the algebra map gives a lower bound, which is sufficient to induce the isomorphism. We sketch the proof of the theorem, relating part of this construction to similar concepts in the formalization of well-founded recursion in the Standard Library. In particular, we prove an irrelevance and an unfolding lemma, which lets us show that the map into any other algebra induced by recursion is indeed an algebra map. By showing that it is also unique, we conclude initiality, and get the isomorphism as a corrolary.

The theorem is applied and demonstrated to the example of binary naturals. We remark that the construction of well-foundedness looks similar to view-patterns. After this, we conclude that this example takes more lines that the direct deriviation in Subsection 3.1, but we argue that most of this code can likely be automated.

4 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.

5 Planning

In the planning of the project, we identify four main topics.

5.1 Finger trees

In the context of numerical representations, we will define and test variants of finger trees. Due to the 2-3 tree structure of the original finger trees [HP06], finger trees are not readily rendered as numerical representations, leading us to the following subexperiments.

First, we will attempt to simplify the definition of finger trees, and test how this changes the performance bounds on their two-sided flexible array operations. Then, we can compute the "trivial numerical representation" on the original finger trees, and check to what extent the arising representation simplifies the proofs of the two-sided flexible array laws. Finally, we may try to forget about finger trees for a moment, and try to construct different numerical representations, achieving a subset of, or ideally all of, the performance bounds of finger trees.

Furthermore, the numerical representation of any "symmetric array" like finger trees seems to have a redundant associated number system. We know that for most operations, we can either simply ignore this, or place the type in a quotient (quotienting over the fibers of a map directly, or applying quasi-equivalence relations). However, indexing a quotiented type remains challenging; so as further work, it may be interesting to find a non-redundant symmetric numerical representation, or investigate index types for quotient types further.

5.2 Enumeration

To characterize numerical representations, we first have to describe number systems; from one point of view, we can accept any type with a surjective interpretation into naturals as a number system. This description also allows for redundant number systems. The other point of view is that a number system must be countably infinite, which ensures that the system can be made non-redundant by enumerating it.

We can approach the problem from the other side, and look at enumerations first, investigating how large classes of W-types can always be equipped with an enumeration structure. As side-questions to this, we can look at applications of enumeration to random testing, where not only the existence of the enumeration matters, but also the "fairness" and the memory usage. We will investigate if and how enumerations can return unbalanced shapes, and how we can ameliorate this; also keeping in mind how the memory usage of enumerations can be reduced by avoiding the replication of identical subtrees.

5.3 Ornaments

In our preliminary work we apply ornaments to describe numerical representations, express heterogeneization, and we use descriptions to characterize equivalences to initial algebras. One downside is that each result uses a different definition of description or ornament, which all have their advantages and drawbacks. We identify the following interesting directions to further research ornaments:

Heterogeneization uses a variant of descriptions allowing parameter introduction, but this does not allow treating the parameter as a variable, nor supports nested types, which may be fruitful to generalize by changing descriptions to allow for higher order functors [JG07].

We can also restrict descriptions and ornaments to a closed universe, allowing us to avoid increasing the levels in the cubical compatible setup.

Furthermore, we have not yet investigated the applicability of patches [DM14] to our experiments; these could be interesting when lifting flexible two-sided array operations from a number system to custom finger tree, but may also need adjustments to be able to deal with our modifications to descriptions.

Finally, we think that, like heterogeneization, there are more common and intuitive transformations of types which can be captured as ornament-computing functions.

5.4 SIP

The SIP as described earlier allows us to concisely prove the equivalences of implementations of structures. However, by definition of the SIP, this limited to structures over unindexed types, while in the context of vectors we may want to express a structure over an indexed type, in which case the indexes themselves may also be only equivalent rather than definitionally equal. Furthermore, the implementation we will use [Ang+20] restricts the basic structure formers; while in our scenarios we do not need much more complicated structures, we do expect the SIP to apply to structures containing most W-types with a free parameter. Solutions to both problems might be applicable to our research, so both may be interesting as further work.

Date	Target
2023-04-24	Finger trees
2023-05-01	”
2023-05-08	”
2023-05-15	Enumerations
2023-05-22	”
2023-05-29	”
2023-06-05	”
2023-06-12	Ornamentation
2023-06-19	”
2023-06-26	”
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	SIP
2023-09-18	”
2023-09-25	TBD[3]
2023-10-02	”
2023-10-09	Write
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`.

[3]This slot is flexible, and can be filled by one of the earlier experiments if I find that one of them requires more time, or may be filled by another experiment should I encounter new interesting and relevant questions

[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.

[JG07] Patricia Johann and Neil Ghani. "Initial Algebra Semantics Is Enough!" In: *Typed Lambda Calculi and Applications*. Ed. by Simona Ronchi Della Rocca. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 207–222. ISBN: 978-3-540-73228-0.

[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`.