Do-it-yourself Module Systems

Extending Dependently-Typed Languages to Implement Module System Features In The Core Language

Department of Computing and Software

McMaster University

Musa Al-hassy

April 6, 2020

PhD Thesis


```
-- Supervisors                -- Emails
Jacques Carette               carette@mcmaster.ca
Wolfram Kahl                  kahl@cas.mcmaster.ca
```

Abstract

Structuring-mechanisms, such as Java's `package` and Haskell's `module`, are often afterthought secondary citizens whose primary purpose is to act as namespace delimiters, while relatively more effort is given to their abstraction encapsulation counterparts, e.g., Java's classes and Haskell's typeclasses. A *dependently-typed language* (DTL) is a typed language where we can write *types* that depend on *terms*; thereby blurring conventional distinctions between a variety of concepts. In contrast, languages with non-dependent type systems tend to distinguish *external vs. internal* structuring-mechanisms —as in Java's `package` for namespacing vs. `class` for abstraction encapsulation— with more dedicated attention and power for the internal case —as it is expressible within the type theory.

To our knowledge, relatively few languages —such as Ocaml, Maude, and the B Method— allow for the manipulation of external structuring-mechanisms as they do for internal ones. Sufficiently expressive type systems, such as those of dependently typed languages, allow for the internalisation of many concepts thereby conflating a number of traditional programming notions. Since DTLs permit types that depend on terms, the types may require non-trivial term calculation in order to be determined. Languages without such expressive type systems necessitate certain constraints on its constructs according to their intended usage. It is not clear whether such constraints have been brought to more expressive languages out of necessity or out of convention. Hence we propose a systematic exploration of the structuring-mechanism design space for dependently typed languages to understand *what are the module systems for DTLs?*

First-class structuring-mechanisms have values and types of their own which need to be subject to manipulation by the user, so it is reasonable to consider manipulation combinators for them from the beginning. Such combinators would correspond to the many generic operations that one naturally wants to perform on structuring-mechanisms —e.g., combining them, hiding components, renaming components— some of which, in the external case, are impossible to perform in any DTL without resorting to third-party tools for pre-processing. Our aim is to provide a sound footing for systems of structuring-mechanisms so that structuring-mechanisms become another common feature in dependently typed languages. An important contribution of this work is an Agda implementation of our module combinators —which we hope to be accepted into a future release of the Agda standard library.

If anything, our aim is practical —to save developers from ad hoc copy-paste preprocessing hacks.

—Source: `https://github.com/alhassy/next-700-module-systems`—

Contents

1 Introduction —The Thesis' "Story" 2

1.1 A Language Has Many Tongues . . . 3

1.2 Needless Distinctions for Containers . . . 4

1.3 Novel Contributions . . . 5

1.4 Overview of the Remaining Chapters . . . 8

2 TODO Sections not yet written 10

References 11

Chapter 1

Introduction —The Thesis' "Story"

In this chapter we aim to present the narrative that demonstrates the distinction between what can currently be accomplished and what is desired when working with composition of software units. We arrive at the observation that packaging concepts differ only in their use —for example, a typeclass and a record are both sequences of declarations that only differ in that the former is used for polymorphism with instance search whereas the latter is used as a structure, grouping related items together. In turn, we are led to propose that the various packaging concepts ought to have a uniform syntax. Moreover, since records are a particular notion of packaging, the commitment to syntactic similarity gives rise to a homoiconic nature to the host language.

Within this work we refer to a *simple type theory* as a language that contains typed lambda terms for terms and formuale; if in addition it contains lambda terms whose types are indexed by values then we say it is a *dependently-typed language*, or 'DTL' for short — depending on intent, value-indexed types could be interpreted as *propositions* and their terms as *proofs*. With the exception of declarations and ephemeral notions, nearly everything in a DTL is a typed lambda term. Just as Lisp's homoiconic nature blurs data and code leaving it not as a language with primitives but rather a language with meta-primitives, so too the lack of distinction between term and type lends itself to generic and uniform concepts in DTLs thereby leaving no syntactic distinction between a constructive proof and an algorithm.

The sections below explore our primary observation. Section 1 demonstrates the variety of 'tongues' present in a single language which are conflated in a DTL, section 2 discusses that such conflation should by necessity apply to notions of packaging, section 3 contains contributed work to ensure that happens. Finally, section 4 concludes by outlining the remainder of the thesis.

1.1 A Language Has Many Tongues

A programming language is actually many languages working together.

The most basic of imperative languages comes with a notion of 'statement' that is executed by the computer to alter 'state' and a notion of 'value' that can be assigned to memory locations. Statements may be sequenced or looped, whereas values may be added or multiplied, for example. In general, the operations on one linguistic category cannot be applied to the other. Unfortunately, a rigid separation between the two sub-languages means that binary choice, for example, conventionally invites two notations with identical semantics —e.g.; in `C` one writes `if (cond) clause`$_1$ `else clause`$_2$ for statements but must use the notation `cond ? term`$_1$ `: term`$_2$ for values. Hence, there are value and statement languages.

Let us continue using the `C` language for our examples since it is so ubiquitous and has influenced many languages. Such a choice has the benefit of referring to a concrete language, rather than speaking in vague generalities. Besides Agda —our language of choice— we shall also refer to Haskell as a representative of the functional side of programming. For example, in Haskell there is no distinction between values and statements —the latter being a particular instance of the former— and so it uses the same notation `if` $\boxed{\ldots}$ `then` $\boxed{\ldots}$ `else` $\boxed{\ldots}$ for both. However, in practice, statements in Haskell are more pragmatically used as a body of a `do` block for which the rules of conditionals and local variables change —hence, Haskell is not as uniform as it initially appears.

In `C`, one declares an integer value by `int x;` but a value of a user-defined type `T` is declared `struct T x;` since, for simplicity, one may think of `C` having an array named `struct` that contains the definitions of user-defined types `T` and the notation `struct T` acts as an array access. Since this is a clunky notation, we can provide an alias using the declaration `typedef existing-name new-name;`. Unfortunately, the existing name must necessarily be a type, such as `struct T` or `int`, and cannot be an arbitrary term. One must use `#define` to produce term aliases, which are handled by the `C` preprocessor, which also provides `#include` to 'copy-paste import' existing libraries. Hence, the type language is distinct from the libraries language, which is part of the preprocessor language.

In contrast, Haskell has a pragma language for enabling certain features of the compiler. Unlike `C`, it has an interface language using type-`class`-es which differs from its `module` language [DJH; SHH01; She] since the former's names may be qualified by the names of the latter but not the other way around. In turn, type-`class` names may be used as constraints on types, but not so with `module` names. It may be argued that this interface language is part of the type language, but it is sufficiently different that it could be thought of as its own language [Ler00] —for example, it comes with keywords `class`, `instance`, `=>` that can only appear in special phrases. In addition, by default, variable declarations are the same for built-in and user-defined types —whereas `C` requires using `typedef` to mimic such behaviour. However, Haskell distinguishes between term and type aliases. In contrast, Agda treats aliasing as nothing more than a normal definition.

Certain application domains require high degrees of confidence in the correctness of software. Such program verification settings may thus have an additional specification language. For `C`, perhaps the most popular is the ANSI C Specification Language, ACSL [BP10]. Besides the `C` types, ACSL provides a type `integer` for specifications referring to unbounded integers as well as numerous other notions and notations not part of the `C` language. Hence, the specification language generally differs from the implementation language. In contrast, Haskell's specifications are generally [Hal+] in comments but its relative Agda allows specifications to occur at the type level.

Whether programs actually meet their specifications ultimately requires a proof language. For example, using the Frama-C tool [VME18], ACSL specifications can be supported by Isabelle or Coq proofs. In contrast, being dependently-typed, Agda allows us to use the implementation language also as a proof language —*the only distinction is a shift in our perspective; the syntax is the same.* Tools such as Idris and Coq come with 'tactics' — algorithms which one may invoke to produce proofs— and may combine them using specific operations that only act on tactics, whence yet another tongue.

Hence, even the simplest of programming languages contain the first three of the following sub-languages —types may be treated at runtime.

1. Expression language;
2. Statement, or control flow, language;
3. Type language;
4. Specification language;
5. Proof language;
6. Module language;
7. Meta-programming languages —including Coq tactics, C preprocessor, Haskell pragmas, Template Haskell's various quotation brackets `[x| ... ]`, Idris directives, etc.

As briefly discussed, the first five languages telescope down into one uniform language within the dependently-typed language Agda. So why not the module language?

1.2 Needless Distinctions for Containers

Computing is compositionality. Large mind-bending software developments are formed by composing smaller, much more manageable, pieces together. How? In the previous section we outlined a number of languages equipped with term constructors, yet we did not indicate which were more primitive and which could be derived.

The methods currently utilised are ad hoc, e.g., "dump the contents of packages into a new über package". What about when the packages contain conflicting names? "Make an über package with field names for each package's contents". What about viewing the new über package as a hierarchy of its packages? "Make conversion methods between the two representations." These tedious and error-prone operations *should be* mechanically derivable.

In general, there are special-purpose constructs specifically for working with packages of "usual", or "day-to-day" expression- or statement-level code. That is, a language for working with containers whose contents live in another language. This forces the users to think of these constructs as rare notions that are seldom needed —since they belong to an ephemeral language. They are only useful when connecting packages together and otherwise need not be learned.

When working with mutually dependent modules, a simple workaround to cyclic type-checking and loading is to create an interface file containing the declarations that dependents require. To mitigate such error-prone duplication of declarations, one may utilise literate programming [Knu84] to tangle the declarations to multiple files —the actual parent module and the interface module. This was the situation with Haskell before its recent module signature mechanism [Kil+14]. Being a purely functional language, it is unsurprising that Haskell treats nested record field updates awkwardly: Where a C-like language may have `a.b.c := d`, Haskell requires `a { b = b a {c = d}}` which necessarily has field names `b`, `c` polluting the global function namespace as field projections. Since a record is a possibly deeply nested list of declarations, it is trivial to flatten such a list to mechanically generate the names ``a-b-c'' —since the dot is reserved— unfortunately this is not possible in the core language thereby forcing users to employ 'lenses' [Rom20] to generate such accessors by compile-time meta-programming. In the setting of DTLs, records in the form of nested Σ-types tend to have tremendously poor performance —in existing implementations of Coq [GCS14] and Agda [Per17], the culprit generally being projections. More generally, what if we wanted to do something with packages that the host language does not support? "Use a pre-processor, approximate packaging at a different language level, or simply settle with what you have."

Main Observation Packages, modules, theories, contexts, traits, typeclasses, interfaces, what have you all boil down to dependent records at the end of the day and *really differ* in *how* they are used or implemented. At the end of section 2 we demonstrate various distinct presentations of such notions of packaging arising from a single package declaration.

1.3 Novel Contributions

The thesis investigates the current state of the art of grouping mechanisms —sometimes referred to as modules or packages—, their shortcomings, and implementing candidate solutions based upon a dependently-typed language.

The introduction of first-class structuring mechanisms drastically changes the situation by allowing the composition and manipulation of structuring mechanisms within the language itself. Granted, languages providing combinators for structuring mechanisms are not new; e.g., such notions already exist for Full Maude [DM07] and B [BGL06]. The former is closer in spirit to our work, but it differs from ours in that it is based on a *reflective logic*: A logic where certain aspects of its metatheory can be faithfully represented within the logic itself. Not only does the meta-theory of our effort not involve reflection, but our distinctive attribute is that our aim is to form powerful module system features for Dependently-Typed Languages (DTLs).

To the uninitiated, the shift to DTLs may not appear useful, or at least would not differ much from existing approaches. We believe otherwise; indeed, in programming and, more generally, in mathematics, there are three —below: 1, 2a, 2b— essentially equivalent perspectives to understanding a concept. Even though they are equivalent, each perspective has prompted numerous programming languages; as such, the equivalence does not make the selection of a perspective irrelevant. The perspectives are below, and examples in the subsequent table.

1. "Point-wise" or "Constituent-Based": A concept is understood by studying the concepts it is "made out of".

   Common examples include:

   ◇ *Extensionality*: A mathematical set is determined by the elements it contains.

   ◇ A method is determined by the sequence of statements or expressions it is composed from.

   ◇ A package —such as a record or data declaration— is determined by its components, which may be *thought of* as fields or constructors.

   Object-oriented programming is based on the notion of inheritance which is founded on the "has a" and "is a" relationships.

2. "Point-free" or Relationship Based: A concept is understood by its relationship to other concepts in the domain of discourse.

   This approach comes into two sub-classifications:

   (a) "First Class Citizen" or "Concept as Data": The concept is treated as a static entity and is identified by applying operations *onto it* in order to observe its nature.

   Common examples include:

   ◇ A singleton set is a set whose cardinality is 1.

   ◇ A method, in any coding language, is a value with the ability to act on other values of a particular type.

   ◇ A renaming scheme to provide different names for a given package; more generally, applicative modules.

(b) “Second Class Citizen” or “Concept as Method”: The concept is treated as a dynamic entity that is fed input stimuli and is understood by its emitted observational output.

Common examples include:

- ◇ A singleton set is a set for which there is a unique mapping to it from any other set. Input any set, obtain a map from it to the singleton set.
- ◇ A method, in any coding language, is unique up to observational equality: Feed it arguments, check its behaviour. Realistically, one may want to also consider efficiency matters.
- ◇ Generative modules as in the `new` keyword from object-oriented programming: Basic construction arguments are provided and a container object is produced.

Observing such a sub-classification as distinct led to traditional structural programming languages, whereas blurring the distinction somewhat led to functional programming.

Table 1.1: Four ways to percieve ‘the’ empty collection $\emptyset$, and associated theory

(1)	Extensional	$\texttt{X} = \emptyset \equiv (\forall\ \texttt{e} \bullet \texttt{e} \in \texttt{X} \equiv \texttt{false})$	Predicate Logic
(2)	Intensional	$\texttt{X} = \emptyset \equiv (\forall\ \texttt{Y} \bullet \texttt{X} \subseteq \texttt{Y})$	Set Theory
(2a)	Data	$\texttt{X} = \emptyset \equiv \#\texttt{X} = 0$	Numbers-as-Sets
(2b)	Method	$\texttt{X} = \emptyset \equiv (\forall\ \texttt{Y} \bullet \exists_1\ \texttt{f} \bullet \texttt{f} \in (\texttt{X} \to \texttt{Y}))$	Function Theory

A simple selection of equivalent perspectives leads to wholly distinct paradigms of thought. It is with this idea that we seek to implement first-class grouping mechanisms in a dependently typed language —theories have been proposed, on paper, but as just discussed *actual design decisions may have challenging impacts on the overall system*. Most importantly, this is a *requirements driven* approach to coherent modularisation constructs in dependently typed languages.

Later on, we shall demonstrate that with a sufficiently expressive type system, a number of traditional programming notions regarding ‘packaging up data’ become conflated —in particular: Records and modules; which for the most part can all be thought of as “dependent products with named components”. Languages without such expressive type systems necessitate certain constraints on these concepts according to their intended usage —e.g., no multiple inheritance for Java’s classes and only one instance for Haskell’s typeclasses. It is not clear whether such constraints have been brought to more expressive languages out of necessity, convention, or convenience. Hence, in chapter 2, we perform a systematic exploration of the structuring-mechanism design space for DTLs as a starting point for the design of an appropriate dependently-typed module system (§ 2). Along the way, we intend to provide a set of atomic combinators that suffice as building blocks for generally desirable features of grouping mechanisms, and moreover we intend to provide an analyses of their interactions.

That is, we want to look at the edge cases of the design space for structuring-mechanism *systems*, not only what is considered convenient or conventional. Along the way, we will undoubtedly encounter useless or non-feasible approaches. The systems we intend to consider would account for, say, module structures with intrinsic types —hence treating them as first class concepts— so that our examination is based on sound principles.

Understandably, some of the traditional constraints have to do with implementations. For example, a Haskell typeclass is generally implemented as a dictionary that can, for the most part, be inlined whereas a record is, in some languages, a contiguous memory block: They can be identified in a DTL, but their uses force different implementation methodologies and consequently they are segregated under different names.

In summary, our research builds upon the existing state of module systems [DCH03] in a dependently-typed setting [Mac86] which is substantiated by developing practical and pragmatic tools. Our outcomes include:

1. A clean module system for DTLs that treats modules uniformly as any other value type.
2. A variety of use-cases contrasting the resulting system with previous approaches.
   - ◇ We solve the so-called unbundling problem and demonstrate —using our implemented tools— how pushout and homomorphisms constructions, among many others, can be *mechanically* obtained.
3. A module system that enables rather than inhibits efficiency.
4. Demonstrate that module features traditionally handled using meta-programming can be brought to the data-value level; thereby not actually requiring the immense power and complexity of meta-programming.

Most importantly, we have implemented our theory thereby obtaining validation that it 'works'. We provide an extensible Emacs interface as well as an Agda library for forming module constructions.

1.4 Overview of the Remaining Chapters

When a programming languages does not provide sufficiently expressive primitives for a concept —such as typeclass derivation [BLS18]— users use some form of pre-processing to accomplish their tasks. In our case, the insufficient primitives are regarding the creation and manipulation of theories —i.e., records, classes, packages, modules. In section 2 , we will demonstrate an prototype that clarified the requirements of our envisioned system. Even though the prototype appears to be metaprogramming, the aim is not to force users interested in manipulating packages to worry about the intricacies of representations; that is, the end goal is to avoid metaprogramming —which is an over-glorified form of preprocessing. The goal is to *use a dependently-typed language to implement the 'missing' module system features directly inside the language.*

An important design decision is whether the resulting development is intended to be reasoned about or not. If reasoning is important, then a language that better supports it is ideal. That is why we are using Agda —using a simpler language and maintaining data invariants eventually becomes much harder [LM13].

The remainder of the thesis is organised as follows.

- ◇ **§2 Examples from the wild**

  There are a host of repeated module patterns since modules are not a first-class construct. We look at three Agda libraries and extract "module design patterns for dependently-typed programming". To the best of our knowledge, we are the first to formalise such design patterns for dependently-typed languages. Three other, non-module, design patterns are discussed in [OS08].

- ◇ **§2 Metaprogramming Module Meta-primitives**

  To show that first-class modules are *reasonable*, we begin by providing `PackageFormer` [ACK19]: A specfication and manipulation language for modules, for Agda. To show that the approach is promising, we demonstrate how some problems from §2 can be tackled.

  - The tool is a **practical** sandbox for exploring do-it-yourself grouping mechanisms: From pushouts and pullbacks, to forming homomorphism types over a given theory.

- ◇ **§2 Module Meta-primitives as Library Methods**

  The ideas learned from making the powerful `PackageFormer` prototype lead us to form the less-powerful `Context` framework, which has the orthogonal benefit of being an Agda library rather than an external pre-processing tool.

  - Along the way, we solve the **unbundling problem**: Features of a structure may be exposed at the type level as-needed.

- ◇ **§2 Conclusion: The lingua franca dream as reality**

  We compare the external `PackageFormer` tool with the `Context` library, and discuss how the latter has brought us closer to our original goal of having a single language for expressing values, types, and modules.

It has been an exciting journey, I hope you enjoy the ride!

Chapter 2

TODO Sections not yet written

Glossary

homoiconic The lack of distinction between 'data' and 'method'. E.g., `'(+ 1 2)` is considered a list of symbols, whereas the *unquoted* term `(+ 1 2)` is considered a function call that reduces to 3. 2

record Rather than holding a bunch of items in our hands and running around with them, we can put them in a bag and run around with it. That is, a record type bundles up related concepts so that may be treated as one coherent entity. If record types can 'inherit' from one another, then we have the notion of an 'object'. 2

typeclass Essentially a dictionary that associates types with a particular list of methods which define the typeclass. Whenever such a method is invoked, the dictionary is accessed for the inferred type and the appropriate definition is used, if possible. This provides a form of ad-hoc polymorphism: We have a list of methods that appear polymorphic, but in-fact their definitions depend on a particular parent type. 2

Bibliography

[ACK19] Musa Al-hassy, Jacques Carette, and Wolfram Kahl. "A language feature to unbundle data at will (short paper)". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, Athens, Greece, October 21-22, 2019*. Ed. by Ina Schaefer, Christoph Reichenbach, and Tijs van der Storm. ACM, 2019, pp. 14–19. ISBN: 978-1-4503-6980-0. DOI: `10.1145/3357765.3359523`. URL: `https://doi.org/10.1145/3357765.3359523` (cit. on p. 9).

[BGL06] Sandrine Blazy, Frédéric Gervais, and Régine Laleau. "Reuse of Specification Patterns with the B Method". In: *CoRR* abs/cs/0610097 (2006). arXiv: `cs/0610097`. URL: `http://arxiv.org/abs/cs/0610097` (cit. on p. 6).

[BLS18] Baldur Blöndal, Andres Löh, and Ryan Scott. "Deriving via: or, how to turn hand-written instances into an anti-pattern". In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*. 2018, pp. 55–67. DOI: `10.1145/3242744.3242746`. URL: `https://doi.org/10.1145/3242744.3242746` (cit. on p. 8).

[BP10] Eduardo Brito and Jorge Sousa Pinto. "Program Verification in SPARK and ACSL: A Comparative Case Study". In: *Reliable Software Technologiey - Ada-Europe 2010, 15th Ada-Europe International Conference on Reliable Software Technologies, Valencia, Spain, June 14-18, 2010. Proceedings*. 2010, pp. 97–110. DOI: `10.1007/978-3-642-13550-7\_7`. URL: `https://doi.org/10.1007/978-3-642-13550-7%5C_7` (cit. on p. 4).

[DCH03] Derek Dreyer, Karl Crary, and Robert Harper. "A type system for higher-order modules". In: *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*. 2003, pp. 236–249. DOI: `10.1145/640128.604151`. URL: `https://doi.org/10.1145/640128.604151` (cit. on p. 8).

[DJH] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. "A formal specification of the Haskell 98 module system". In: pp. 17–28. URL: `http://doi.acm.org/10.1145/581690.581692` (cit. on p. 3).

[DM07] Francisco Durán and José Meseguer. "Maude's module algebra". In: *Sci. Comput. Program.* 66.2 (2007), pp. 125–153. DOI: `10.1016/j.scico.2006.07.002`. URL: `https://doi.org/10.1016/j.scico.2006.07.002` (cit. on p. 6).

[GCS14] Jason Gross, Adam Chlipala, and David I. Spivak. *Experience Implementing a Performant Category-Theory Library in Coq.* 2014. arXiv: 1401.7694v2 [math.CT] (cit. on p. 5).

[Hal+] Thomas Hallgren et al. "An Overview of the Programatica Toolset". In: *HCSS '04*. URL: http://www.cse.ogi.edu/PacSoft/projects/programatica/ (cit. on p. 4).

[Kil+14] Scott Kilpatrick et al. "Backpack: retrofitting Haskell with interfaces". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 2014, pp. 19–32. DOI: 10.1145/2535838.2535884. URL: https://doi.org/10.1145/2535838.2535884 (cit. on p. 5).

[Knu84] Donald E. Knuth. "Literate Programming". In: *Comput. J.* 27.2 (1984), pp. 97–111. DOI: 10.1093/comjnl/27.2.97. URL: https://doi.org/10.1093/comjnl/27.2.97 (cit. on p. 5).

[Ler00] Xavier Leroy. "A modular module system". In: *J. Funct. Program.* 10.3 (2000), pp. 269–303. DOI: 10.1017/S0956796800003683 (cit. on p. 3).

[LM13] Sam Lindley and Conor McBride. "Hasochism: the pleasure and pain of dependently typed haskell programming". In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*. 2013, pp. 81–92. DOI: 10.1145/2503778.2503786. URL: https://doi.org/10.1145/2503778.2503786 (cit. on p. 9).

[Mac86] David B. MacQueen. "Using Dependent Types to Express Modular Structure". In: *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*. 1986, pp. 277–286. DOI: 10.1145/512644.512670. URL: https://doi.org/10.1145/512644.512670 (cit. on p. 8).

[OS08] Nicolas Oury and Wouter Swierstra. "The Power of Pi". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 39–50. ISBN: 9781595939197. DOI: 10.1145/1411204.1411213. URL: https://doi.org/10.1145/1411204.1411213 (cit. on p. 9).

[Per17] Natalie Perna. *(Re-)Creating sharing in Agda's GHC backend*. Jan. 2017. URL: https://macsphere.mcmaster.ca/handle/11375/22177 (cit. on p. 5).

[Rom20] Mario Román. *Profunctor optics and traversals*. 2020. arXiv: 2001.08045v1 [cs.PL] (cit. on p. 5).

[She] Tim Sheard. "Generic Unification via Two-Level Types and Parameterized Modules". In: *ICFP 2001*. to appear. acm press (cit. on p. 3).

[SHH01] Tim Sheard, William Harrison, and James Hook. "Modeling the Fine Control of Demand in Haskell." (submitted to Haskell workshop 2001). 2001 (cit. on p. 3).

[VME18] Grigoriy Volkov, Mikhail U. Mandrykin, and Denis Efremov. "Lemma Functions for Frama-C: C Programs as Proofs". In: *CoRR* abs/1811.05879 (2018). arXiv: 1811.05879. URL: http://arxiv.org/abs/1811.05879 (cit. on p. 4).