Lambda: The Ultimate Sublanguage (Experience Report)

JEREMY YALLOP, University of Cambridge, UK
LEO WHITE, Jane Street Capital, UK

We describe our experience teaching an advanced typed functional programming course based around the use of System Fω as a programming language.

CCS Concepts: • **Social and professional topics** → **Computer science education**; • **Mathematics of computing** → *Lambda calculus*; • **Software and its engineering** → *Functional languages*.

Additional Key Words and Phrases: education, pedagogy, types, functional programming, lambda calculus, mental models, sublanguages

ACM Reference Format:
Jeremy Yallop and Leo White. 2019. Lambda: The Ultimate Sublanguage (Experience Report). *Proc. ACM Program. Lang.* 3, ICFP, Article 116 (August 2019), 17 pages. https://doi.org/10.1145/3342713

116

1 INTRODUCTION

Feature-rich programming languages can be challenging to learn. One source of difficulty is the tendency for advanced features to "leak out" unexpectedly in error messages. For example, a student of Haskell, having learned about function composition and `map`, might make the following erroneous attempt at a function that calculates the successors of a list of integers:

```
Prelude> let succlist = map . (1+)
```

and be baffled by the resulting error message:

```
Non type-variable argument in the constraint: Num (a -> b)
  (Use FlexibleContexts to permit this)
```

The error itself (composition in place of application) is elementary, but the diagnostic mentions several advanced concepts that are unlikely to help a beginner identify the problem. The root of this communication failure is a mismatch in *models*: the compiler describes the error in terms of the full language definition, but the beginner's mental model covers only a subset of the language.

One approach to overcoming the leaky error message problem is to use a *sublanguage* [Brusilovsky et al. 1994; Pagan 1980]: an implementation of a programming language that supports only a subset of its features. For example, the Helium sublanguage of Haskell [Heeren et al. 2003] excludes type classes, removing a large set of concepts (such as "constraints" and "FlexibleContexts") from the language, so that errors are more likely to be reported in terms that beginners can understand. Similarly, DrRacket supports several sublanguages (called "language levels") that remove macros, quasiquotation, higher-order functions, etc., improving the clarity of error messages for programs written by beginners [Marceau et al. 2011].

Authors' addresses: Jeremy Yallop, Department of Computer Science and Technology, University of Cambridge, UK, jeremy.yallop@cl.cam.ac.uk; Leo White, Jane Street Capital, UK, leo@lpw25.net.

2475-1421/2019/8-ART116
https://doi.org/10.1145/3342713

The problem that sublanguages tackle is central to learning. Learning typically involves building a mental model and incrementally refining it by integrating new facts. By bringing the language implemented by the compiler closer to the language learnt by a beginner, sublanguages make it much easier to integrate facts such as type mismatch reports into the learner's mental model.

Sublanguages are an effective way to learn the basics of a language. But how can students move from basic competency to mastery — that is, to a point where the intricate details of a language and sophisticated programming techniques can be readily understood? Here, too, the sublanguage approach can help. A strength of most functional languages is their definition in terms of simpler constructs: deep pattern matching may be translated to shallow cases; overloading may be translated away; syntactic sugar may be expanded into a tiny core. Treating this core as a sublanguage can help students to develop intuition — that is, a clear mental model — that provides a solid foundation for understanding advanced language features and techniques that would otherwise appear ad-hoc.

This paper describes our experience teaching a course based on this approach over the last four years. The first week of the course involves a typed lambda calculus, System Fω, that supports only abstraction and application. Using this tiny language students explore language features found in full-featured typed functional languages: nested types and GADTs, first-class polymorphism, higher-kinded types, existentials, higher-order modules, etc. Later, the course introduces these features directly, building on the intuition developed in the early parts. The intuitions developed using System Fω allow us to cover ground rapidly, and eight weeks later students are familiar with many of the topics beloved of ICFP participants and able to write and understand sophisticated typed functional programs involving indexed data, staging constructs, algebraic effects, and more.

1.1 Outline

Our thesis, then, is that System Fω is an effective sublanguage for teaching advanced functional programming, and the remainder of the paper presents our experience as supporting evidence.

Section 2 gives an overview of the course: its principles, structure and content (Section 2.1), its assessment (Section 2.2) and the tools students use to learn System Fω (Section 2.3).

Section 3 discusses a broad range of System Fω programming exercises, and their relation to the topics covered in the later sections of the course: data types and folds (Section 3.1), type equality (Section 3.2), non-regular types (Section 3.3), GADTs (Section 3.4), proofs (Section 3.5), semirings (Section 3.6), existentials (Section 3.7) and modules (Section 3.8).

Developing and teaching a new course is often an educational experience for instructors as well as students. Section 4 discusses some lessons we learnt from running the course.

Section 5 covers related work, focusing on books and courses that take a similar approach.

2 A COURSE IN ADVANCED FUNCTIONAL PROGRAMMING

Our course is a unit on a one-year masters course; it runs for eight weeks (Figure 1), with two lectures each week. The course focuses on OCaml, including extensions and future features, such as modular implicits [White et al. 2014] (week 6), algebraic effects [Dolan et al. 2015] (week 7) and multi-stage programming [Kiselyov 2014] (week 8).

The aim of the course is to develop familiarity with advanced typed functional programming constructs and techniques: students are expected to reach a level where they can readily grasp much of the research presented at ICFP.

We typically have between 10 and 25 registered students, and other students and staff from the department often join the lectures.

2.1 Course Structure and Content

The first week serves as a kind of précis of the course: many of the ideas that are studied in detail in OCaml later on appear in miniature using System Fω here. For example, week 5 covers programming with indexed data such as GADTs. However, all the components of GADTs — type equality witnesses (Section 3.2), polymorphic recursion (Section 3.3) and existential types (Section 3.7) appear in the first week, in the context of System Fω. In some years the students have even learned to program with encodings of GADTs directly in System Fω (Section 3.4) in the first assessed exercise (Section 2.2) before revisiting GADTs using OCaml's language support in week 5.

Topic	Week
Lambda Calculus	1
Type Inference / Curry-Howard	2
Polymorphism & Parametricity	3
Modules & Abstraction, Semirings	4
Indexed Data	5
Overloading, Generic Programming	6
Effects	7
Multi-Stage Programming	8

Fig. 1. Course structure

Why System Fω? The System Fω calculus has long played a central role in the theory and practice of typed functional programming. Compilers for ML and Haskell use intermediate languages based on System Fω [Morrisett et al. 1999; Sulzmann et al. 2007], and many features of functional languages — higher-rank and higher-kinded polymorphism [Jones 1993; Peyton Jones et al. 2007; Vytiniotis et al. 2008; Yallop and White 2014], recursion over non-regular types [Bird and Meertens 1998], type classes [Hall et al. 1996], higher-order modules [Rossberg et al. 2014], existential types (Section 3.7), and more — have a straightforward elaboration into the calculus. Further, System Fω can be used as the elaboration language for type inference [Pottier 2014], helping to elucidate the limitations of inference, relating to the value restriction [Wright 1995][1] and its relaxation in OCaml [Garrigue 2004], polymorphic recursion [Henglein 1993] and higher-rank types [Garrigue and Rémy 1999; Peyton Jones et al. 2007; Wells 1994], which we touch on in weeks 2 and 3.

Similarly, System Fω provides a useful working model for type equality (Section 3.2), with some limitations (e.g. the System Fω encodings do not support the injectivity principle.)

Most other topics found in the course can be similarly based around System Fω, which provides a useful basis for understanding the Curry-Howard correspondence (Section 3.5) and parametricity [Wadler 2003], can give a semantics to multi-stage programming [Kameyama et al. 2008], and offers features essential for generic programming [Siek and Lumsdaine 2005].

Finally, the remarkable expressive power of System Fω exposes some limitations of functional programming languages: for example, OCaml's type language supports only first-order applications, and Haskell's supports only higher-order applications; neither supports type abstractions (although it is often possible to simulate abstractions via lambda-lifting [Lindley 2012]); contrariwise, the limitations of System Fω, which does not support primitive effects or general recursion, help to make clear exactly what can be accomplished without those facilities (Cf. Turner [2004]).

[1]Elaborating Hindley-Milner implicit type schemes into System Fω-style type abstractions justifies Wright's value restriction: it is only safe to generalize when the body is a value, since in other cases inserting a type abstraction would change the evaluation and sharing behaviour. Kiselyov [2015] further explores connections between sharing and generalization.

type variables: α, β, ..., A, B, ... term variables: x, y kinds: κ ::= $*$ | $\kappa \Rightarrow \kappa$

$$\begin{array}{ll} \text{types} & \sigma,\ \tau\ ::=\ \alpha\ \mid\ \sigma \rightarrow \tau\ \mid\ \forall\alpha :: \kappa.\tau\ \mid\ \lambda\alpha :: \kappa.\tau\ \mid\ \sigma\ \tau \\ \text{terms} & \mathtt{L,M,N}\ ::=\ \mathtt{x}\ \mid\ \lambda\mathtt{x}\colon \tau.M\ \mid\ \mathtt{L}\ \mathtt{M}\ \mid\ \Lambda\alpha :: \tau.M\ \mid\ \mathtt{L}[\tau] \end{array}$$

Fig. 2. System Fω syntax

2.2 Exercises

The assignments seemed quite hard at first, but everything got clearer once I started reviewing the material. It was very rewarding to understand most of it in the end![2] (2017)

Without the assignments the lectures are too abstract to understand on anything but a high level. The assignments really tied things together well. (2018)

The course is assessed by three programming exercises (issued around weeks 2, 5 and 8), which students submit two weeks after they are issued. There is no final exam.

The exercises are a crucial part of the course; the lectures and notes present principles, but the exercises build the familiarity that underlies intuition, and ensure that students steadily master functional programming principles as the course progresses. The exercises follow a similar structure each year: the first, whose contents this paper describes in some detail, focuses on System Fω programming, establishing a foundation for the remainder of the course.

Balancing pedagogy with assessment is challenging with questions about System Fω and *typeful* programming more generally [Cardelli 1989]. Section 4.3 discusses our response to this challenge.

2.3 Tooling

We provide a primitive System Fω interpreter to support the exercises, based on the implementation by Pierce [2002]. Figure 2 shows the supported syntax: type variables (denoted by Greek letters or capitals), term variables (lowercase), kinds (formed from $*$ and $\Rightarrow$), type expressions (variables, function types, quantified types, abstractions and applications), and terms (variables, term abstractions and applications, and type abstractions and applications). Type binders (quantification, type functions, and type abstractions) can bind type variables of arbitrary kind.

There are a few concessions to usability: the interpreter also supports top-level definitions and signatures and some additional types (sums, products and existentials) which could be encoded but are more convenient to have as primitives. Furthermore, kind annotations may be omitted for binders of type $*$; all other binders must be type- or kind-annotated.

Besides a command-line tool, we also provide a simple web interface to the System Fω interpreter based on `js_of_ocaml` [Vouillon and Balat 2014].

Section 4.1 discusses the effectiveness of this toolchain.

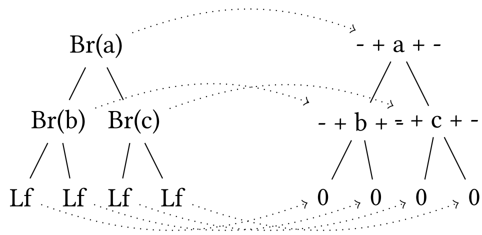
3 PROGRAMMING WITH SYSTEM Fω

3.1 Data Types and Folds in System Fω

Folds, over lists and over other data types, are central to functional programming. They are particularly useful in System Fω, where they can be used (as typed Church encodings) in place of data types, which are not available in the language. The first exercise on the course often involves a simple fold-based encoding of a tree type together with some functions (such as `sum` or `depth`) defined on trees (Figure 3).

As well as providing a useful warm-up, encodings of data types illustrate the usefulness of higher kinds, and the value of the extra expressive power offered by System Fω. In System F it is possible

[2] We include excerpts from students' anonymized feedback throughout the paper.




```
type 'a tr = Lf : 'a tr | Br : 'a tr * 'a * 'a tr -> 'a tr
Tree = λα.∀φ::*⇒*.(φ α) → (φ α → α → φ α → φ α) → φ α
```

Fig. 3. Folds for tree types. Note the correspondence between the OCaml and System Fω definitions

to encode the type `t list` for any given `t`, but not the type constructor `list` (which has kind $* \Rightarrow *$) itself. With System Fω it becomes possible to encode both `list` and a wide variety of other data types, including the nested (non-regular) types that appear in advanced functional programming techniques [Bird and Paterson 1999; Hinze 1999] (Section 3.3).

Further, data type encodings based on folds extend neatly to the so-called final tagless style [Carette et al. 2009] commonly used to define DSLs. At first, writing functions compositionally seems unnatural and sometimes inefficient; however, at a larger scale compositionality becomes a boon, supporting elaborate optimizations of DSLs [Kiselyov 2016] and even self-representations of typed languages [Brown and Palsberg 2017].

3.2 Type Equality in System Fω

Type equality — in particular, first-class equality witnesses — plays a significant role in typed functional programming, in meta-programming [Pasalic 2004], in intermediate language design [Sulzmann et al. 2007], and as the basis for GADTs [Johann and Ghani 2008].

Encoding type equalities in System Fω therefore serves as a first step towards implementing GADTs; indeed, decomposing GADTs into their constituent parts (type equality (Section 3.2), existential types (Section 3.7), and non-regularity (Section 3.3)) robs them of much of their mystery. In high-level languages such as OCaml and Haskell, type equalities are automatically introduced into the context by the compiler in GADT pattern matches [Garrigue and Rémy 2013]. However, it is instructive to deal with type equalities that must be explicitly constructed and applied.

Figure 4 presents the elements of an exercise from the 2016 instance of the course, where students were asked to construct a fold-based encoding of the equality GADT and show its equivalence to the more common Leibniz encoding. The so-called Leibniz encoding `Eq` (which is given in roughly this form by Church [1940], and appears more recently in various functional programming papers [Baars and Swierstra 2002; Weirich 2004; Yallop and Kiselyov 2010]) appears at the top left: it says that types α and β are equal if α can be turned into β in any context ϕ. The fold-based encoding `Equal` appears in the top right: it directly corresponds to the OCaml definition of `eq`: the polymorphic argument $(\forall\gamma.\phi\ \gamma\ \gamma)$ corresponds to the type of the `Refl` constructor `('x, 'x) eq`, and the result type corresponds to `('a, 'b) eq`. Both encodings fundamentally rely on higher-kinded polymorphism, and cannot be defined in System F.

The students' task was to provide conversions between `Eq` and `Equal`, and further to show that `Equal` represents an equivalence relation by defining values that encode reflexivity, symmetry and transitivity properties; sample solutions are given in Figure 4.

Two representations of $\sigma \equiv \tau$ in System Fω

```
Eq :: ∗⇒∗⇒∗                               Equal :: ∗⇒∗⇒∗
Eq = λα.λβ.∀ϕ::∗⇒∗.ϕ α → ϕ β              Equal = λα.λβ.∀ϕ::∗⇒∗⇒∗.(∀γ.ϕ γ γ)→ϕ α β
```

$$\longleftarrow \boxed{\forall\alpha.\forall\beta.\mathtt{Equal}\ \alpha\ \beta \rightarrow \mathtt{Eq}\ \alpha\ \beta} \longrightarrow$$

```
Λα.Λβ.λe:Equal α β.Λϕ::∗⇒∗.λh:ϕ α.e[λα.λβ.ϕ α→ϕ β] (Λγ.λx:ϕ γ.x) h
```

$$\longrightarrow \boxed{\forall\alpha.\forall\beta.\mathtt{Eq}\ \alpha\ \beta \rightarrow \mathtt{Equal}\ \alpha\ \beta} \longrightarrow$$

```
Λα.Λβ.λe:Eq α β.Λϕ::∗ ⇒ ∗ ⇒ ∗.λh:(∀γ.ϕ γ γ).e[ϕ α] (h[α])
```

```
refl : ∀α.Equal α α
refl = Λα. Λϕ::∗⇒∗⇒∗.λf:(∀γ.ϕ γ  γ).f[α]

symm : ∀α.∀β.Equal α β→Equal β α
symm = Λα.Λβ.λe:Equal α β.Λϕ::∗⇒∗⇒∗.λf:(∀γ.ϕ γ  γ).e[λα.λβ.ϕ β  α] f

trans : ∀α.∀β.∀γ.Equal α β→Equal β γ→Equal α γ
trans = Λα.Λβ.Λγ.λe1:Equal α β.λe2:Equal β γ.Λϕ::∗⇒∗⇒∗.
          λf:(∀γ.ϕ γ  γ).e2[λγ.λδ.ϕ α  γ→ϕ α  δ] (Λγ.λx:ϕ α  γ.x) (e1[ϕ] f)
```

Type equalities in OCaml

```
type ('a, 'b) eql = Refl : ('x, 'x) eql
let symm : type a b. (a, b) eql -> (b, a) eql =
  fun Refl -> Refl
let trans : type a b. (a, b) eql -> (b, c) eql -> (a, c) eql =
  fun Refl Refl -> Refl
```

Fig. 4. Type equalities

As well as improving familiarity with type equality, the exercise is also a useful lesson in the power of polymorphic code, since any correctly-typed implementation that converts between `Equal` and `Eq` is necessarily correct.

For example, consider the conversion from `Eq` to `Equal` which, after expanding the aliases (and renaming variables to avoid confusion) has the following type:

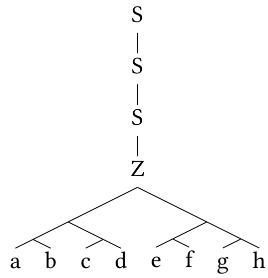
$$\forall\alpha.\forall\beta.(\forall\Psi::*\Rightarrow*.\psi\ \alpha\ \rightarrow\ \psi\ \beta) \rightarrow (\forall\phi::*\Rightarrow*\Rightarrow*.(\forall\gamma.\phi\ \gamma\ \gamma)\ \rightarrow\ \phi\ \alpha\ \beta)$$

The term may be built by following the types outside-in, introducing a Λ for each outer $\forall$ and a λ for each outer $\rightarrow$, giving the following partially-constructed term:

$$\Lambda\alpha.\Lambda\beta.\lambda\mathtt{e}.(\forall\Psi::*\Rightarrow*.\psi\ \alpha\ \rightarrow\ \psi\ \beta).\Lambda\phi::*\Rightarrow*\Rightarrow*.\lambda\mathtt{h}:(\forall\gamma.\phi\ \gamma\ \gamma).???$$

Then the remainder of the type requires that the body have type $\phi\ \alpha\ \beta$, and so we must construct such a term from α, β, `e`, ϕ and `h`. Only two of these, `e` and `h`, are term variables, and `h` is unsuitable (since it can only build terms where both arguments of ϕ are the same). In order to use `e` it is sufficient to find a suitable ψ such that $\psi\ \beta\ \equiv\ \phi\ \alpha\ \beta$. Fortunately, this equation has an obvious solution ($\psi\ \equiv\ \phi\ \alpha$), which can be used to instantiate `e` and start building the body:

$$\mathtt{e}[\phi\ \alpha]\ :\ \phi\ \alpha\ \alpha\ \rightarrow\ \phi\ \alpha\ \beta$$




```
type 'a perfect =
   ZeroP : 'a->'a perfect
 | SuccP : ('a * 'a) perfect->'a perfect
```

$$\texttt{Perfect} = \lambda\alpha.\forall\phi{::}*\Rightarrow*.(\forall\alpha.\alpha \to \phi\ \alpha)\to(\forall\alpha.\phi\ (\alpha\times\alpha)\to\phi\ \alpha)\to\phi\ \alpha$$

$$\texttt{zeroP} = \Lambda\alpha.\lambda \texttt{x}{:}\alpha.\Lambda\phi{::}*\Rightarrow*.\lambda \texttt{z}{:}\forall\alpha.\alpha\to\phi\ \alpha.\lambda \texttt{s}{:}(\forall\beta.\phi\ (\beta\times\beta)\to\phi\ \beta).\texttt{z}[\alpha]\ \texttt{x}$$

$$\begin{aligned}\texttt{succP} = \Lambda\alpha.\lambda \texttt{p}{:}\texttt{Perfect}\ (\alpha\times\alpha).\Lambda\phi{::}*\Rightarrow*.\lambda \texttt{z}{:}\forall\alpha.\alpha\to\phi\ \alpha.\\ \lambda \texttt{s}{:}(\forall\beta.\phi\ (\beta\times\beta)\to\phi\ \beta).\texttt{s}[\alpha]\ (\texttt{p}[\phi]\ \texttt{z}\ \texttt{s})\end{aligned}$$

Fig. 5. Perfect trees in System Fω and OCaml

Finally, completing the puzzle requires a term of type $\phi\ \ \alpha\ \ \alpha$ to pass to this function. We can construct a suitable term by instantiating `h`:

$\texttt{e}[\phi\ \alpha]\ (\texttt{h}[\alpha])$

which suffices to complete the exercise. Besides illustrating the utility of polymorphic types in reasoning about programs, exercises of this sort help to distinguish between straightforwardly mechanical programming (e.g. type-directed insertion of introduction forms) and steps that involve some insight (e.g. finding suitable substitutions).

Besides applications to GADTs, encodings of type equalities can also be used as a basis (or, at least, as an inspiration) for encodings of other relationships between types, such as subtyping [Yallop and Dolan 2017]. Equality proofs also plays a central role in many programs based on the Curry-Howard correspondence (Section 3.5).

3.3 Non-Regular Types in System Fω

Non-regular types [Bird and Meertens 1998] (often called *nested* types) — i.e. data types that are applied to something other than their parameters within their own definition — appear frequently in advanced functional programming applications [Bird and Paterson 1999; Hinze 1999, 2000; Okasaki 1998]. Figure 5 gives a typical example: the `perfect` type constructor appears as `('a,'a) perfect` (rather than `'a perfect`) in the argument to `SuccP`; every tree of that type therefore has the form illustrated in the figure, with a chain of n `SuccP` constructors and exactly 2^n values at the leaves.

Encodings of nested types arise naturally in System Fω using the same scheme used to define the regular tree types (Figure 3). Figure 5 gives a definition: the System Fω type `Perfect` closely follows the definition of the OCaml `perfect`, with the type variable ϕ in place of `perfect` and α in place of `'a`.

3.4 GADTs in System Fω

Generalized algebraic data types (GADTs), whose usefulness once appeared to be restricted to length-indexed vectors and well-typed evaluators, are now commonplace in functional programming, appearing in applications from parsers [Pottier and Régis-Gianas 2006] to foreign function interfaces [Yallop et al. 2018]. However, novices still find GADTs challenging to understand. Encodings of GADTs in System Fω are certainly less convenient than using the facilities provided by high-level functional languages directly but, like other verbose encodings, can serve as a significant aid to understanding. (We cannot recall a single instance from the past four years where a student has had trouble with GADTs after programming in System Fω.)

For example, given the standard equality GADT

```
type ('a, 'b) eql = Refl : ('a, 'a) eql
```

and a pattern

```
(Refl : (int list, string list) eql) -> ...
```

the OCaml compiler reasons as follows [Garrigue and Normand 2015]. First, the definition of `Refl` requires that the two arguments to `eql` are equivalent, so there should be a type equation `int list ≡ string list`, scoped under the pattern. Second, the type constructor `list` is injective, so there should be a second equation `int ≡ string`. Finally, since `int` and `string` are in fact known to be distinct types, the equation is unsatisfiable and the pattern is redundant.

In OCaml programs this reasoning happens automatically, but with a System Fω encoding, each step must be performed explicitly by manipulating values in the program. While verbosity is inconvenient, explicitness can be useful, both for programmers seeking to understand the types of the programs they write, and for language designers. For example, a slightly careless approach to reasoning about these kinds of interactions between injectivity and type equality led to a soundness bug in the OCaml compiler [Garrigue 2013]. Our aim in breaking the convenient GADT matching down into unwieldy explicit components is to help students understand, identify and avoid such problems.

Later parts of the course often make heavy use of GADTs: our exercises have involved building balanced trees (with insertion and deletion) using GADT constraints to ensure balancing, defining a DSL that represents only valid XHTML values [Elsman and Larsen 2004], implementing typed `printf` and `scanf` format specifiers [Vaugon 2013], and ensuring invariants for the classic pure functional two-list implementation of queues (e.g. that the length of `outq` never exceeds the length of `inq`).

With System Fω encodings of GADTs we have been a little less ambitious, given their verbosity and awkwardness. In 2017 we asked the students to define functions based on an encoding of vectors (length-indexed sequences) that approximately follows Atkey [2012]:

```
Vec :: * ⇒ * ⇒ *
Vec = λα.λM.∀ϕ::*⇒*.ϕ Z → (∀N.α → ϕ N → ϕ (S N)) → ϕ M
```

There are two arguments to the function that represents a vector: the first, of type `ϕ Z`, represents an empty vector (indexed by zero); the second, of type `∀N.α → ϕ N → ϕ (S N)`, represents a cons cell containing a head (type α) a tail (type `ϕ N`) and returning type `ϕ (S N)` — i.e., a vector indexed by the successor of the length of the tail.

The awkwardness of programming with encodings of GADTs provides a reminder of the various conveniences provided by an in-language implementation. For example, several auxiliary functions and values representing facts about numbers are needed, such as a proof that zero is not the successor of any number:

```
sz_distinct : ∀N.Not (Eql (S N) Z)
= ΛN.λeq:Eq (S N) Z. eq[λX.X] (inr[N] unit)
```

Although it is clunky, almost everything needed to program with vectors can be encoded in this way, except for a proof that the successor function is injective, which must be added as an axiom.

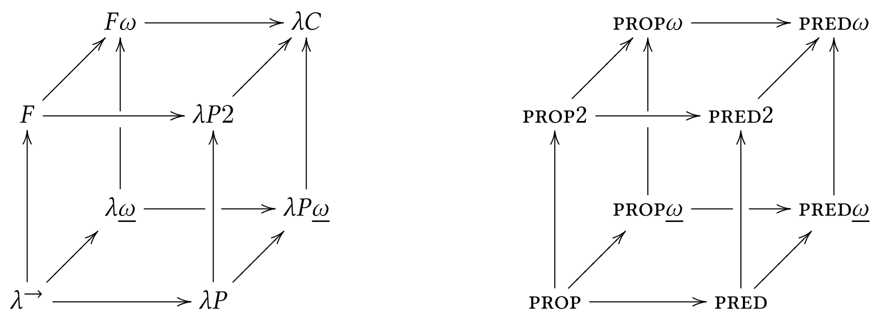


Fig. 6. The lambda and logic cubes

```
DM₁ = ∀α.∀β.Not (α × β)→(Not α + Not β)
DM₂ = ∀α.∀β.(Not α + Not β)→Not (α × β)
DM₃ = ∀α.∀β.Not (α + β)→(Not α × Not β)
DM₄ = ∀α.∀β.(Not α × Not β)→Not (α + β)
```

Fig. 7. De Morgan's laws in System Fω

```
DM₅ = ∀ϕ::∗⇒∗.Not (∀α.ϕ α)→(∃α.Not (ϕ α))
DM₆ = ∀ϕ::∗⇒∗.(∃α.Not (ϕ α))→Not (∀α.ϕ α)
DM₇ = ∀ϕ::∗⇒∗.Not (∃α.ϕ α)→(∀α.Not (ϕ α))
DM₈ = ∀ϕ::∗⇒∗.(∀α.Not (ϕ α))→Not (∃α.ϕ α)
```

Fig. 8. Generalization of De Morgan's laws in System Fω

(Similarly, Brown and Palsberg [2017] extended System Fω with a `typecase` construct to support injectivity in their self-representation of System Fω.)

A particularly tricky function to define is the function that takes the tail of a non-empty vector:

```
tail : ∀α.∀μ.Vec α (S μ)→Vec α μ
```

Readers familiar with the predecessor function for Church-encodings of natural numbers may recognise the difficulty. Here we have the additional challenge of programming with a much more richly typed structure. (We discuss this function further in Section 4.3)

3.5 Proofs in System Fω

The famous Curry-Howard correspondence [Wadler 2015] connects propositions with types, programs with proofs, and normalization with proof simplification. It is an interesting curiosity, and occasionally useful for language designers (e.g. [Davies 1996; Mitchell and Plotkin 1988]), but is it a worthwhile addition to a course which is intended to teach programming?

Sheard [2005] answers in the affirmative: the Curry-Howard correspondence can be put to work to build a variety of interesting programs, from composable pipelines to modular arithmetic. Like System Fω, the correspondence provides a mental model — a way of thinking about programs and programming languages that can be fruitful for reasoning.

Figure 6 gives one key to understanding the connection between functional programming and logic: for each calculus in the famous lambda cube on the left there is a corresponding logic on the less well known cube on the right. As the figure shows, System Fω and the other languages on the left face of the lambda cube all correspond to propositional logics, which can express propositions about sets but not about individual objects. (The predicate logics, which can express propositions about objects, correspond to dependently-typed languages, such as λC, the calculus of constructions.) By considering the differences in the expressiveness between logics, students can develop intuition for the differences in expressiveness between traditional functional programming languages and languages with dependent types. Grasping these differences also helps with understanding the "singletons" technique (i.e. turning individual values into sets) which is often used to make dependently-typed techniques (somewhat) accessible in languages without dependent types [Altenkirch et al. 2005; Eisenberg and Weirich 2012; Lindley and McBride 2013]. Exercises later in the course sometimes build on these encodings (perhaps taking them rather too far): for example, we ask students to define properties of data types such as a guarantee that the elements of a tree are kept in order (somewhat in the style of Kiselyov and Shan [2007]).

As with the other aspects of the course, programming in System Fω can help with developing intuitions about the Curry-Howard isomorphism. For example, it is sometimes useful to know whether a given type is inhabited; considering the type as a proposition can be a useful guide. (Viewed this way, the type $\forall\alpha\,.\,\alpha$ is obviously uninhabited, since the corresponding proposition is not provable; similarly $\forall\alpha\,.\,\alpha \to \alpha$ is obviously inhabited.)

Figures 7 and 8 illustrate a programming exercise from 2017: attempting to witness De Morgan's laws in System Fω, first using binary connectives, and then using an arbitrary unary predicate. A catch is that only three of the laws in each case are provable constructively; students are left to discover the "classical" law, and give proofs (as System Fω terms) for the others.

3.6 Semirings

Yet another useful mental model for programming is the *semiring* structure of types [Carette and Sabry 2016]. This is closely related to the *logical* structure given by the Curry-Howard isomorphism, but lacks idempotency. (For example, while $A \wedge A$ is equivalent to A in propositional logic, $A \times A$ is not typically equal to A in other semirings.) As with the Curry-Howard correspondence, the semiring view can guide the implementation of programs. To give a trivial example, in any semiring it is the case that $A \times 1 \equiv A$, and we can indeed easily give a pair of functions that convert between the types A and $A \times 1$.

```
f : ∀A.A×1 → A                          g : ∀A.A → A×1
  = ΛA.λp:A×1.fst p                       = ΛA.λx:A.⟨x,⟨⟩⟩
```

It is not possible to prove *within* System Fω that functions so defined do actually form an isomorphism — such a proof would need predicate logic. However, external proofs of properties like this are straightforward using techniques developed later in the course, such as equational reasoning or parametricity.

In 2018 we asked the students to define terms corresponding to several more complex semiring equations (such as distributivity), and later to build a variant of generalized functional tries [Elliott 2009; Hinze 2000] in OCaml using the semiring and exponent equations.

3.7 Existentials in System Fω

Existential types are common in typed functional programs, in the form of abstract types [Mitchell and Plotkin 1988], and as a way of hiding possibly-heterogeneous data behind a common interface.

Although our System Fω implementation provides existentials as a primitive, it is straightforward to define them in terms of universals using the well-known encoding

$\exists\alpha.\tau \;::=\; \forall\beta.(\forall\alpha.\tau \to \beta) \to \beta$

```
pack_ [Bools]                          λp:Exists Bools.
      [Unit + Unit]                      open_ [Bools] p [Unit]
       ⟨inr [Unit] unit,                  (Λβ.λbools:Bools β.
        inl [Unit] unit,                     (prj3 bools)
        λb:Unit+Unit.Λα.λr:α.λs:α.           (prj1 bools)
          case b of x.s | y.r ⟩             [Unit] unit unit)
```

Fig. 9. An abstract type of bools using encoded existentials

In fact, System Fω can encode *higher-kinded* existentials, such as $\exists\alpha::*\Rightarrow*.\tau$ in this way. (However, each kind requires a new encoding, since System Fω does not support kind polymorphism.)

In 2018 we asked the students to investigate encodings of existential types, along with operations corresponding to the standard **pack** and **open** introduction and elimination forms

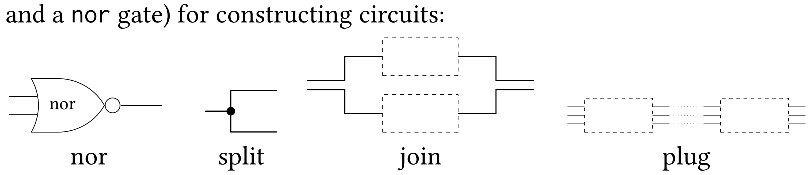
```
Exists :: (* ⇒ *) ⇒ *
pack_ : ∀ϕ::* ⇒*.∀β.ϕ β →Exists ϕ
open_ : ∀ ϕ ::* ⇒*.Exists ϕ →(∀α.(∀β. ϕ β →α) →α)
```

and to use the encodings to define various abstract types. Figure 9 gives an example: the code on the left uses existentials to hide the implementation of `Bools` as a sum of units, and the implementation on the right unpacks an implementation of `Bools` and performs computation.

3.8 Modules in System Fω

Modules play a crucial role in writing programs in ML-family languages, but beginners often make very limited use of the module system, shying away from apparently advanced features such as functors. System Fω can be used to give a semantics to large parts of ML module systems, and writing System Fω programs is a good way to develop intuition for modules.

In 2018 we asked students to encode a module with four primitive components (three plumbing operations, and a `nor` gate) for constructing circuits:



The following System Fω definition corresponds to such a module signature; the type parameter `T` corresponds to type component that can be instantiated by an implementation of the module. The remainder of the exercise involves using the module to build new gates (`And`, `Not`), implementing the signature with implementations that count and interpret gates, and encoding higher-kinded existentials (Section 3.7) to make the module type member abstract.

```
Gates = λT::*⇒*⇒*.(T (Bool × Bool) Bool) ×
                  (∀α.T α (α×α)) ×
                  (∀α.∀β.∀γ.∀δ.T α γ →T β δ →T (α × β) (γ × δ)) ×
                  (∀α.∀β.∀γ.T α β →T β γ →T α γ)
```

4 LESSONS AND CHALLENGES

I definitely learned a lot, but it was tough getting there (2018)

Planning and teaching the course involves substantial ongoing work; for example, we develop a fresh set of exercises each year, and review the course material, adding or removing lectures, based

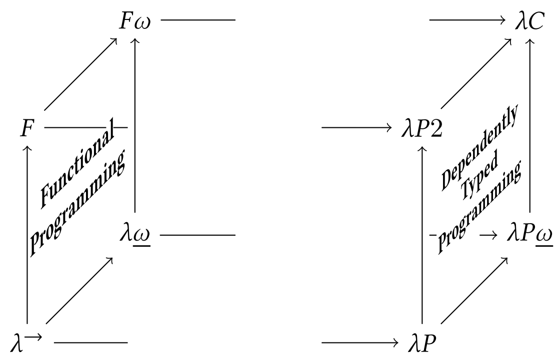


Fig. 10. Functional vs dependently-typed programming

on student feedback and other measures of effectiveness. One substantial change in 2018 was the addition of a few sections on dependently-typed programming. In retrospect, this turned out to be a mistake: shifting the focus away from System Fω-based programming led to a less coherent presentation; in future we plan to stay firmly on the left-hand side of the cube (Figure 10).

Overall, feedback from students has been positive, often extremely so:

> *It was probably my favourite course in the entire tripos*[3]*. The material was very cutting-edge, interesting and intellectually stimulating.* (2017)

Other students reported finding the course practical:

> *I could see how the material could be usefully applied in practice* (2017)

or inspiring:

> *I have started to really consider research in programming languages and category theory in part due to this course.* (2018)

although some students disagreed:

> *I think there are very few careers (research or applied) that this course would have any bearing on.* (2018)

4.1 Tooling

Section 2.3 described our simple batch-mode System Fω interpreter. In practice this is adequate, but it is likely that the students' learning experience could be significantly enhanced with some usability improvements.

The main shortcoming is the quality of error messages, which report type or kind mismatches, but do not describe the exact nature of the incompatibility. Producing good error messages for languages with type inference (such as ML or Haskell) can be challenging but, since every variable in a System Fω program is annotated, we have no such excuse! Good error messages are valuable; one student submitted some improvements to the interpreter to improve location reporting.

It would also be helpful to have a selection of modern conveniences found in more serious language implementations: a REPL, typed holes, syntax highlighting, and auto-completion, particularly where the context uniquely determines the possible inhabitants of a type (Section 3.2).

[3] *Tripos* is a Cambridge term for the degree course.

4.2 Preparation

> *Actually, the course was a lot more in-depth that I would've expected from reading the syllabus—a very pleasant surprise!* (2017)

The different backgrounds of the students provide an additional challenge, especially during the early weeks of the course. In practice, students are divided into two (roughly equal) camps: those who have continued from a bachelor's degree at our university, and those who join the masters course from elsewhere. The local students have usually taken a course on type systems that covers System Fω[4], and find little new in the early material:

> *For those having taken any Type Theory courses in the past, much of the first half of the course was completely familiar.* (2018)

In contrast, some students joining from elsewhere have little to no experience with type systems and functional programming. Although we ask that students have some typed functional programming experience before joining the course, some students do not heed the advice, and find the plunge into System Fω rather a shock.

Despite this variety in experience, most students complete the course successfully; indeed, in typical years, all students do so. We attribute this high success rate to several factors: a selective admissions process, a supportive environment with opportunities for questions and discussion (during lectures, in office hours, and via a class mailing list), comprehensive lecture notes, small class sizes and a course structure that means that students are often taking only other unit concurrently, allowing them to focus their efforts. Furthermore, assessment via compulsory exercises rather than a final exam makes it easier to discover difficulties in understanding at an early stage, and makes it possible for students who struggle with the material to overcome difficulties with prolonged effort.

However, although these factors make it possible to overcome the difficulties arising from the students' different backgrounds, we think that it would be even better to address the issue head-on. One way to do this would be to split the course into two parts: an initial optional part that teaches type theory up to System Fω, and a functional programming section that starts with programming in System Fω before moving on to standard functional programming languages such as OCaml or Haskell. At present our course covers both parts, but the initial part is rather fast-paced for students without much type theory experience, and mostly unnecessary for students who have taken a type theory course already. Unfortunately, administrative constraints have so far made it difficult for us to divide the course in this way.

4.3 Assessment

> *Because the compiler provides so much feedback, it seems most of the marking is check marks.* (2018)

Finding a suitable level of difficulty for the assessed exercises has often proved challenging. In languages with powerful type systems such as System Fω it is frequently the case that programs are either correct or ill-typed. Students therefore often know before submitting whether their solutions are correct and high scores are common.

> *Assignments were very well constructed, matched the course material extremely well, and were tricky yet satisfying* (2018)

From a pedagogical viewpoint this is ideal! An incentive structure that encourages students to persist until they succeed is likely to result in mastery of the course material. (In practice, this is exactly what happens, and the spread in marks is replaced by a spread in time taken: some students

[4] When we started teaching our course, the type systems course only covered System F

spend ten times as long to complete exercises as others, and there is little correlation between the time taken and the mark achieved.) Furthermore, teaching students to structure programs so that type checking ensures the absence of errors is a central aim of the course.

> *Assignments were generally really good. Just long enough to make you understand the content without dragging it out.* (2015)

However, from a practical viewpoint there is sometimes a need to differentiate students — for example, PhD programs often ask for student rankings. In order to ensure a spread of marks we consequently include some functions such as `tail` (Section 3.4) that are likely to prove challenging even for those who have mastered the linguistic aspects of System Fω.

5 RELATED WORK

We believe that the precise approach described here — using System Fω as a sublanguage for typed functional programming — is unusual. However, since System Fω lies at the core of several typed functional programming languages it is no surprise to find other advanced functional programming courses with a substantial System F or System Fω component. A complete survey is impractical, but we mention some representative examples.

François Pottier and colleagues at Inria teach *Functional programming and type systems* (MPRI 2-4), which focuses on the semantics, compilation and metatheory of functional languages based on System F, but has recently started to incorporate some broader programming ideas such as effectful programming, tagless interpreters and generic programming.

Kevin Hamlen at the University of Texas teaches *Advanced Programming Languages*, starting with functional languages and progressing to System F.

Tim Sheard's *Advanced Functional Programming* course at Portland covers several of the topics touched on here (e.g. higher-rank polymorphism, quasiquotation, and monads), but using Haskell as the main teaching language.

Several books use some form of lambda calculus as a means to teach functional programming. Nordström et al. [1990] teach programming directly in type theory — that is, programming on the right face of the cube (Figure 10). Similarly, Thompson [1991] focuses on type theory for programming, drawing connections (analogies and comparisons) with Miranda.

At the opposite end of the type spectrum, Michaelson [2011] introduces functional programming starting from untyped lambda calculus, before moving to simple types and on to Standard ML.

Felleisen et al. [2004] critically examine the introductory computer science curriculum, focusing on the shortcomings of the classic *Structure and Interpretation of Computer Programs*, and proposing an alternative based (in part) upon the use of sub-languages; one inspiration for our course is their advocacy of that approach.

ACKNOWLEDGMENTS

We are grateful to all those whose help and ideas contributed to the course, including Alan Mycroft, Stephen Dolan, Nada Amin, Neel Krishnaswami, KC Sivaramakrishnan, Dominic Mulligan, Anil Madhavapeddy, Heidi Howard and, of course, the Advanced Computer Science students who participated in the course over the past few years and provided valuable feedback. We thank Sally Fincher and Alan Blackwell for ethical guidance. Finally, we thank the ICFP'19 reviewers for helpful comments.

REFERENCES

Thorsten Altenkirch, Conor McBride, and James McKinna. 2005. Why Dependent Types Matter. (April 2005). http://www.cs.nott.ac.uk/~psztxa/publ/ydtm.pdf.

Robert Atkey. 2012. Relational Parametricity for Higher Kinds. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France (LIPIcs)*, Patrick Cégielski and Arnaud Durand (Eds.), Vol. 16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 46–61. https://doi.org/10.4230/LIPIcs.CSL.2012.46

Arthur I. Baars and S. Doaitse Swierstra. 2002. Typing Dynamic Typing. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM. https://doi.org/10.1145/581478.581494

Richard S. Bird and Lambert G. L. T. Meertens. 1998. Nested Datatypes. In *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings (Lecture Notes in Computer Science)*, Johan Jeuring (Ed.), Vol. 1422. Springer, 52–67. https://doi.org/10.1007/BFb0054285

Richard S. Bird and Ross Paterson. 1999. De Bruijn Notation as a Nested Datatype. *J. Funct. Program.* 9, 1 (Jan. 1999), 77–91. https://doi.org/10.1017/S0956796899003366

Matt Brown and Jens Palsberg. 2017. Typed Self-Evaluation via Intensional Type Functions. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 415–428. https://doi.org/10.1145/3009837

P. Brusilovsky, A. Kouchnirenko, P. Miller, and I Tomek. 1994. Teaching Programming to Novices: A Review of Approaches and Tools. In *Proceedings of ED-MEDIA 94–World Conference on Educational Multimedia and Hypermedia*.

Luca Cardelli. 1989. Typeful Programming. In *Formal Description of Programming Concepts, based on a seminar organized by IFIP Working Group 2.2 and held near Rio de Janeiro in April 1989 (IFIP State-of-the-Art Reports)*, Erich J. Neuhold and Manfred Paul (Eds.). Springer, 431. http://lucacardelli.name/Papers/TypefulProg.pdf

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (Sept. 2009), 509–543. https://doi.org/10.1017/S0956796809007205

Jacques Carette and Amr Sabry. 2016. Computing with Semirings and Weak Rig Groupoids. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.), Vol. 9632. Springer, 123–148. https://doi.org/10.1007/978-3-662-49498-1_6

Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic* 5, 2 (1940). http://www.jstor.org/stable/2266170

R. Davies. 1996. A Temporal-logic Approach to Binding-time Analysis. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*. IEEE Computer Society, Washington, DC, USA, 184–. http://dl.acm.org/citation.cfm?id=788018.788825

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. 2015. Effective Concurrency through Algebraic Effects. (September 2015). OCaml Users and Developers Workshop 2015.

Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently Typed Programming with Singletons. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 117–130. https://doi.org/10.1145/2364506.2364522

Conal Elliott. 2009. *Denotational Design with Type Class Morphisms (Extended Version)*. Technical Report 2009-01. LambdaPix. http://conal.net/papers/type-class-morphisms

Martin Elsman and Ken Friis Larsen. 2004. Typing XHTML Web Applications in ML. In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages (Lecture Notes in Computer Science)*, Vol. 3057. Springer International Publishing, 224–238. https://doi.org/10.1007/978-3-540-24836-1_16

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2004. The Structure and Interpretation of the Computer Science Curriculum. *J. Funct. Program.* 14, 4 (July 2004), 365–378. https://doi.org/10.1017/S0956796804005076

Jacques Garrigue. 2004. Relaxing the Value Restriction. In *Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings (Lecture Notes in Computer Science)*, Yukiyoshi Kameyama and Peter J. Stuckey (Eds.), Vol. 2998. Springer, 196–213. https://doi.org/10.1007/978-3-540-24754-8_15

Jacques Garrigue. 2013. On variance, injectivity, and abstraction. OCaml Meeting. (September 2013).

Jacques Garrigue and Jacques Le Normand. 2015. GADTs and Exhaustiveness: Looking for the Impossible, See [Yallop and Doligez 2017], 23–35. https://doi.org/10.4204/EPTCS.241.2

Jacques Garrigue and Didier Rémy. 1999. Semi-Explicit First-Class Polymorphism for ML. *Inf. Comput.* 155, 1-2 (1999), 134–169. https://doi.org/10.1006/inco.1999.2830

Jacques Garrigue and Didier Rémy. 2013. Ambivalent Types for Principal Type Inference with GADTs. In *11th Asian Symposium on Programming Languages and Systems*. Melbourne, Australia.

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. 1996. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (1996), 109–138. https://doi.org/10.1145/227699.227700

Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003. Helium, for Learning Haskell. In *Proceedings of the ACM SIGPLAN Haskell Workshop (Haskell'03), Uppsala, Sweden* (proceedings of the acm sigplan haskell workshop (haskell'03), uppsala, sweden ed.). ACM SIGPLAN, 62. https://www.microsoft.com/en-us/research/publication/helium-for-learning-haskell/

Fritz Henglein. 1993. Type Inference with Polymorphic Recursion. *ACM Trans. Program. Lang. Syst.* 15, 2 (1993), 253–289. https://doi.org/10.1145/169701.169692

Ralf Hinze. 1999. Polytypic Functions Over Nested Datatypes. *Discrete Mathematics & Theoretical Computer Science* 3, 4 (1999), 193–214. http://dmtcs.episciences.org/266

Ralf Hinze. 2000. Generalizing generalized tries. *J. Funct. Program.* 10, 4 (2000), 327–351. http://journals.cambridge.org/action/displayAbstract?aid=59745

Patricia Johann and Neil Ghani. 2008. Foundations for structured programming with GADTs. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 297–308. https://doi.org/10.1145/1328438.1328475

Mark P. Jones. 1993. A System of Constructor Classes: Overloading and Implicit Higher-order Polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. ACM, New York, NY, USA, 52–61. https://doi.org/10.1145/165180.165190

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2008. Closing the Stage: From Staged Code to Typed Closures. In *PEPM 2008*, Robert Glück and Oege de Moor (Eds.). ACM, 147–157. https://doi.org/10.1145/1328408.1328430

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Lecture Notes in Computer Science, Vol. 8475. Springer International Publishing, 86–102.

Oleg Kiselyov. 2015. Generating Code with Polymorphic let: A Ballad of Value Restriction, Copying and Sharing, See [Yallop and Doligez 2017], 1–22. https://doi.org/10.4204/EPTCS.241.1

Oleg Kiselyov. 2016. Modular, composable, typed optimizations in the tagless-final style. (August 2016). http://okmij.org/ftp/tagless-final/course/optimizations.html.

Oleg Kiselyov and Chung-chieh Shan. 2007. Lightweight Static Capabilities. *Electr. Notes Theor. Comput. Sci.* 174, 7 (2007), 79–104. https://doi.org/10.1016/j.entcs.2006.10.039

Sam Lindley. 2012. Embedding F. In *Proceedings of the 8th ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2012, Copenhagen, Denmark, September 9-15, 2012*, Andres Löh and Ronald Garcia (Eds.). ACM, 45–56. https://doi.org/10.1145/2364394.2364402

Sam Lindley and Conor McBride. 2013. Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell '13)*. ACM, New York, NY, USA, 81–92. https://doi.org/10.1145/2503778.2503786

Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind Your Language: On Novices' Interactions with Error Messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2011)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/2048237.2048241

Greg Michaelson. 2011. *An Introduction to Functional Programming Through Lambda Calculus*. Dover Publications.

John C. Mitchell and Gordon D. Plotkin. 1988. Abstract Types Have Existential Type. *ACM Trans. Program. Lang. Syst.* 10, 3 (July 1988), 470–502. https://doi.org/10.1145/44501.45065

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst.* 21, 3 (May 1999), 527–568. https://doi.org/10.1145/319301.319345

Bengt Nordström, Kent Petersson, and Jan M. Smith. 1990. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press. http://www.cse.chalmers.se/research/group/logic/book/.

Chris Okasaki. 1998. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA.

Frank G. Pagan. 1980. Nested Sublanguages of Algol 68 for Teaching Purposes. *SIGPLAN Not.* 15, 7 and 8 (July 1980), 72–81. https://doi.org/10.1145/947680.947687

Emir Pasalic. 2004. *The Role of Type Equality in Meta-programming*. Ph.D. Dissertation. Advisor(s) Sheard, Timothy E. AAI3151199.

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *J. Funct. Program.* 17, 1 (2007), 1–82. https://doi.org/10.1017/S0956796806006034

Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

François Pottier. 2014. Hindley-milner elaboration in applicative style: functional pearl. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 203–212. https://doi.org/10.1145/2628136.2628145

François Pottier and Yann Régis-Gianas. 2006. Towards Efficient, Typed LR Parsers. *Electr. Notes Theor. Comput. Sci.* 148, 2 (2006), 155–180. https://doi.org/10.1016/j.entcs.2005.11.044

Andreas Rossberg, Claudio Russo, and Derek Dreyer. 2014. F-ing Modules. *Journal of Functional Programming* 24, 5 (2014), 529–607. https://doi.org/10.1017/S0956796814000264

Tim Sheard. 2005. Putting Curry-howard to Work. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell (Haskell '05)*. ACM, New York, NY, USA, 74–85. https://doi.org/10.1145/1088348.1088356

Jeremy G. Siek and Andrew Lumsdaine. 2005. Essential Language Support for Generic Programming. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/1065010.1065021

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '07)*. ACM, New York, NY, USA, 53–66. https://doi.org/10.1145/1190315.1190324

Simon Thompson. 1991. *Type Theory and Functional Programming*. Addison-Wesley. https://www.cs.kent.ac.uk/people/staff/sjt/TTFP/.

D. A. Turner. 2004. Total Functional Programming. *Journal of Universal Computer Science* 10, 7 (Jul 2004), 751–768.

Benoît Vaugon. 2013. A New Implementation of OCaml Formats based on GADTs. OCaml 2013. (September 2013). https://ocaml.org/meetings/ocaml/2013/proposals/formats-as-gadts.pdf

Jérôme Vouillon and Vincent Balat. 2014. From bytecode to JavaScript: the Js_of_ocaml compiler. *Softw., Pract. Exper.* 44, 8 (2014), 951–972. https://doi.org/10.1002/spe.2187

Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2008. FPH: first-class polymorphism for Haskell. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 295–306. https://doi.org/10.1145/1411204.1411246

Philip Wadler. 2003. The Girard–Reynolds Isomorphism. *Information and Computation* 186, 2 (2003), 260 – 284. https://doi.org/10.1016/S0890-5401(03)00141-X Theoretical Aspects of Computer Software (TACS 2001).

Philip Wadler. 2015. Propositions As Types. *Commun. ACM* 58, 12 (Nov. 2015), 75–84. https://doi.org/10.1145/2699407

Stephanie Weirich. 2004. Functional Pearl: type-safe cast. *Journal of Functional Programming* 14 (2004). Issue 06. https://doi.org/10.1017/S0956796804005179

J. B. Wells. 1994. Typability and Type-Checking in the Second-Order lambda-Calculus are Equivalent and Undecidable. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*. IEEE Computer Society, 176–185. https://doi.org/10.1109/LICS.1994.316068

Leo White, Frédéric Bour, and Jeremy Yallop. 2014. Modular implicits. In *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014. (EPTCS)*, Oleg Kiselyov and Jacques Garrigue (Eds.), Vol. 198. 22–63. https://doi.org/10.4204/EPTCS.198.2

Andrew K. Wright. 1995. Simple Imperative Polymorphism. *LISP and Symbolic Computation* 8, 4 (01 Dec 1995), 343–355. https://doi.org/10.1007/BF01018828

Jeremy Yallop and Stephen Dolan. 2017. First-Class Subtypes. In *Proceedings ML Family / OCaml Users and Developers workshops, ML/OCaml 2017, Oxford, UK, 7th September 2017. (EPTCS)*, Sam Lindley and Gabriel Scherer (Eds.), Vol. 294. 74–85. https://doi.org/10.4204/EPTCS.294.4

Jeremy Yallop and Damien Doligez (Eds.). 2017. *Proceedings ML Family / OCaml Users and Developers workshops, ML Family/OCaml 2015, Vancouver, Canada, 3rd & 4th September 2015*. EPTCS, Vol. 241. https://doi.org/10.4204/EPTCS.241

Jeremy Yallop and Oleg Kiselyov. 2010. First-class modules: hidden power and tantalizing promises. ACM SIGPLAN Workshop on ML. (September 2010). Baltimore, Maryland, United States.

Jeremy Yallop, David Sheets, and Anil Madhavapeddy. 2018. A modular foreign function interface. *Sci. Comput. Program.* 164 (2018), 82–97. https://doi.org/10.1016/j.scico.2017.04.002

Jeremy Yallop and Leo White. 2014. Lightweight Higher-Kinded Polymorphism. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science)*, Michael Codish and Eijiro Sumii (Eds.), Vol. 8475. Springer, 119–135. https://doi.org/10.1007/978-3-319-07151-0_8