Stitch: The Sound Type-Indexed Type Checker (Functional Pearl)

(Author's Cut)

Richard A. Eisenberg
Tweag I/O
Paris, France
Bryn Mawr College
Bryn Mawr, PA, USA
rae@richarde.dev

Abstract

A classic example of the power of generalized algebraic datatypes (GADTs) to verify a delicate implementation is the type-indexed expression AST. This functional pearl refreshes this example, casting it in modern Haskell using many of GHC's bells and whistles. The Stitch interpreter is a full executable interpreter, with a parser, type checker, common-subexpression elimination, and a REPL. Making heavy use of GADTs and type indices, the Stitch implementation is clean Haskell code and serves as an existence proof that Haskell's type system is advanced enough for the use of fancy types in a practical setting. The paper focuses on guiding the reader through these advanced topics, enabling them to adopt the techniques demonstrated here.

Keywords: Haskell, GADTs, fancy types

1 A Siren from the Folklore

A major focus of modern functional programming research is to push the boundaries of type systems. The fancy types born of this effort allow programmers not only to specify the shape of their data—types have done *that* for decades—but also the meaning and correctness conditions of their data. In other words, while well typed programs do not go wrong, fancy typed programs always go right. By leveraging a type system to finely specify the format of their data, programmers can hook into the declarative specifications inherent in type systems to be able to reason about their programs in a compositional and familiar manner.

Though fancy types come in a great many varieties, this work focuses on an entry-level fancy type, the generalized algebraic data type, or GADT. GADTs, originally called first-class phantom types [13] or guarded recursive datatypes [71], exhibit one of the most basic ways to use fancy types. Pattern-matching on a GADT value provides information about the type of that value. Accordingly, different branches of a GADT pattern match have access to different typing assumptions. In this way, a term-level, runtime operation (the pattern-match) informs the type-level, compile-time type-checking—one of the hallmarks of dependently typed programming. Indeed, GADTs, in concert with other features, can be used to effectively mimic dependent types, even without full-spectrum support [19, 41].

It is high time for an example:[1]

```
data G :: Type → Type where
  BoolCon :: G Bool
  IntCon  :: G Int

match :: ∀a. G a → a
match BoolCon = True
match IntCon  = 42
```

The GADT *G* has two constructors. One constrains *G*'s index to be *Bool*, the other *Int*. The *match* function matches on a value of type *G a*. If the value is *BoolCon*, then we learn that *a* is *Bool*; our function can thus return *True* :: *a*. Otherwise, *match*'s argument is *IntCon*, and thus *a* is *Int*; we return 42 :: *Int*. The runtime pattern-match informs the compile-time type, allowing the branches to have *different* types. In contrast, a simple pattern-match requires every branch to have the *same* type.

1.1 Stitch

This paper presents the design and implementation of Stitch, a simple extension of the simply typed λ-calculus (STLC), including integers, Booleans, basic arithmetic, conditionals, a fixpoint operator, and `let`-bindings. ("Stitch" refers both to the language and its implementation.) The expression abstract syntax tree (AST) type in Stitch is a GADT such that only well typed Stitch expressions can be formed. That is, there is simply no representation for the expression `true 5`, as that expression is ill typed. The AST type, *Exp*, is *indexed* by the type of the expression represented, so that if *exp* :: *Exp ctx ty*, then the Stitch expression encoded in *exp* has the type *ty* in a typing context *ctx*.

Haskell '20, August 27, 2020, Virtual Event, USA
2020. ACM ISBN 978-1-4503-8050-8/20/08...$15.00
https://doi.org/10.1145/3406088.3409015

[1] The examples in this paper are type-checked in GHC 8.10 during the typesetting process, with gratitude to lhs2TeX [37].

The example of a λ-calculus implementation using a GADT in this way is common in the folklore, and it has been explored in previous published work (see Section 10.4). However, the goal of this current work is not to present an type-indexed AST as a novel invention, but instead to methodically explore the usage of one. It is my hope that, through this example, readers can gain an appreciation for the power and versatility of fancy types and learn techniques applicable in their own projects.

It can be easy to dismiss the example of well typed λ-calculus terms as too introspective: Can't PL researchers come up with a better example to tout their wares than a PL implementation? However, I wish to turn this argument on its head. A PL implementation is a fantastic example, as most programmers in a functional language will quickly grasp the goal of the example, allowing them to focus on the implementation aspects instead of trying to understand the program's behavior. Furthermore, implementing a language *is* practical. Many systems require PL implementations, including web browsers, database servers, editors, spreadsheets, shells, and even many games.

This paper will focus on the version of Haskell implemented in GHC 8.10, making critical use of GHC's support for using GADT constructors at the type level [66, 73], higher-rank type inference [49], and, of course, GADT type inference [50, 64].[2] Accordingly, this paper can serve as an extended example of how recent innovations in GHC can power a more richly typed programming style.

1.2 Contributions

While this functional pearl does not offer new *technical* contributions, it illuminates recent innovations in Haskell and invites intermediate programmers to use advanced PL techniques in their programs. It makes the following contributions:

- Stitch is a full executable interpreter of the STLC, available online,[3] and suitable for classroom use as a demonstration of a λ-calculus.
- Section 3 is an accessible primer on Haskell's advanced features, as used in the examples in this paper.
- This work offers many settings for the use of fancy types. For example, parser output is guaranteed to be well-scoped.
- Section 9 describes aspects of the common-subexpression elimination pass implemented in Stitch, offered as proof that the use of an indexed AST scales to the more complex analyses inherent in real compilers.
- The development described here serves as an existence proof that Haskell—even without full dependent types—is a suitable language in which to use practical fancy types.

[2]The full set of extensions used somewhere in the codebase is as follows: AllowAmbiguousTypes, BangPatterns, ConstraintKinds [42], CPP, DataKinds [73], DefaultSignatures, DeriveAnyClass, DeriveDataTypeable [32, 33, 52], DeriveGeneric [38, 39], DeriveTraversable, EmptyCase, ExistentialQuantification, ExplicitForAll, FlexibleContexts, FlexibleInstances, FunctionalDependencies [28], GADTs [29, 50, 64], GeneralizedNewtypeDeriving [8], InstanceSigs, KindSignatures, LambdaCase, MagicHash, MultiParamTypeClasses [45], NondecreasingIndentation, PatternGuards [21], PatternSynonyms [55], PolyKinds [66, 73], QuantifiedConstraints [6], RankNTypes [49], RoleAnnotations [8], Safe [61], ScopedTypeVariables [47], StandaloneDeriving, Trustworthy [61], TupleSections, TypeApplications [20], TypeFamilies [10, 18], TypeFamilyDependencies [59], TypeOperators, UnboxedSums, UnboxedTuples [46], UndecidableInstances, UndecidableSuperClasses, ViewPatterns [65].

[3]Some more general definitions have been monomorphized in this presentation to aid in understanding. The executable code is at https://gitlab.com/goldfirere/stitch/-/tree/hs2020.

2 Introducing Stitch

Stitch is an implementation of the simply typed λ-calculus, so we will start off with a review of this little language, including the Stitch extensions. See Figure 1.[4]

We see that Stitch is quite a standard implementation of the STLC [e.g., 56, Chapter 9] with modest extensions. It has a call-by-value semantics, and the value of a `let`-bound variable is computed before entering the body of the `let`. Stitch supports general recursion by way of its (standard) `fix` operator, which evaluates to a fixpoint.

Stitch comes with both a small-step and big-step operational semantics, though the small-step semantics is elided here. Users of Stitch may find it interesting to compare its behavior with respect to the choice of semantics; commands at the Stitch REPL allow the user to choose how they wish to reduce an expression to a value, allowing users to witness that big-step semantics tell you nothing about a term during evaluation, while the small-step semantics can show you the steps the expression takes as it reduces.

2.1 The Stitch REPL

Before we jump into the implementation, it is helpful to look at the user's experience of Stitch. The Stitch REPL allows the user to enter in expressions for evaluation and to query aspects of an expression. An example is illustrative:

```
Welcome to the Stitch interpreter, version 1.0.
λ> 1 + 1
2 : Int
λ> \x:Int->Int.\y:Int.x y
λ#:Int -> Int.λ#:Int.#1 #0 : (Int -> Int) -> Int -> Int
λ> expr = (\x:Int->Int.\y:Int.x y) (\z:Int.z + 3) 5
(λ#:Int -> Int.λ#:Int.#1 #0) (λ#:Int.#0 + 3) 5 : Int
λ> expr
8 : Int
λ> :step expr
(λ#:Int -> Int.λ#:Int.#1 #0) (λ#:Int.#0 + 3) 5 : Int
--> (λ#:Int.(λ#:Int.#0 + 3) #0) 5 : Int
```

[4]The formalization is type-checked and typeset with the help of ott [58].

Metavariables:

x	term vars

Grammar:

$\tau ::= \tau_1 \to \tau_2 \mid \mathbf{Int} \mid \mathbf{Bool}$	types
$op ::= + \mid - \mid * \mid / \mid \% \mid < \mid \leq \mid > \mid \geq \mid \equiv$	operators
$\mathbb{Z} ::= \ldots$	integers
$\mathbb{B} ::= \mathbf{true} \mid \mathbf{false}$	Booleans
$e ::= x \mid \lambda x{:}\tau.e \mid e_1\ e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid e_1\ op\ e_2$ $\mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \mid \mathbf{fix}\ e \mid \mathbb{Z} \mid \mathbb{B}$	expressions
$v ::= \lambda x{:}\tau.e \mid \mathbb{Z} \mid \mathbb{B}$	values
$\Gamma ::= \emptyset \mid \Gamma, x{:}\tau$	contexts

result(op) is the result type of an operator: $\mathbf{Int}$ for $\{+, -, *, /, \%\}$ and $\mathbf{Bool}$ for $\{<, \leq, >, \leq, \equiv\}$

apply(op, v_1, v_2) is the result of applying op to v_1 and v_2

$e_1[e_2/x]$ denotes substitution of e_2 for x in e_1

$\boxed{\Gamma \vdash e : \tau}$ Typing rules

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{T\_Var} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}\ \textsc{T\_Lam}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{T\_App}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}\ \textsc{T\_Let} \qquad \frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \mathbf{fix}\ e : \tau}\ \textsc{T\_Fix}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{Int} \qquad \Gamma \vdash e_2 : \mathbf{Int}}{\Gamma \vdash e_1\ op\ e_2 : \mathrm{result}(op)}\ \textsc{T\_Arith}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}\ \textsc{T\_Cond}$$

$$\frac{}{\Gamma \vdash \mathbb{Z} : \mathbf{Int}}\ \textsc{T\_Int} \qquad \frac{}{\Gamma \vdash \mathbb{B} : \mathbf{Bool}}\ \textsc{T\_Bool}$$

$\boxed{e \Downarrow v}$ Big-step operational semantics

$$\frac{}{v \Downarrow v}\ \textsc{E\_Value} \qquad \frac{\begin{array}{c} e_1 \Downarrow \lambda x{:}\tau.e \qquad e_2 \Downarrow v_2 \\ e[v_2/x] \Downarrow v \end{array}}{e_1\ e_2 \Downarrow v}\ \textsc{E\_App}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2[v_1/x] \Downarrow v}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow v}\ \textsc{E\_Let}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1\ op\ e_2 \Downarrow \mathrm{apply}(op, v_1, v_2)}\ \textsc{E\_Arith}$$

$$\frac{e \Downarrow \lambda x{:}\tau.e' \qquad e'[\mathbf{fix}\,(\lambda x{:}\tau.e')/x] \Downarrow v}{\mathbf{fix}\ e \Downarrow v}\ \textsc{E\_Fix}$$

$$\frac{e_1 \Downarrow \mathbf{true} \qquad e_2 \Downarrow v}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow v}\ \textsc{E\_IfTrue}$$

$$\frac{e_1 \Downarrow \mathbf{false} \qquad e_3 \Downarrow v}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow v}\ \textsc{E\_IfFalse}$$

Figure 1. The simply typed λ-calculus, as embodied in Stitch.

```
--> (λ#:Int.#0 + 3) 5 : Int
--> 5 + 3 : Int
--> 8 : Int
```

We see here that the syntax is straightforward and familiar, though Stitch requires a type annotation at every λ-abstraction. The most distinctive aspect of this session is Stitch's approach to variable binding, which we explore next.

2.2 De Bruijn Indices

Every implementor of a programming language must make a choice of representation of variable binding. The key challenge is that, no matter which representation we choose, we must be sure that $\lambda x{:}\tau.x$ and $\lambda y{:}\tau.y$ are treated identically in all contexts. There are *many* possible choices out there: named binders [57], locally nameless binders [23], using higher-order abstract syntax [53], *parametric* higher-order abstract syntax [14], Unbound [69], bound [30], among others. The interested reader is referred to Weirich et al. [69], where even more possibilities lie in wait. In this work, however, I choose trusty, old de Bruijn indices [16], as these serve two design goals of Stitch well: de Bruijn indices work easily with an indexed AST, and they can easily arise when teaching implementations of the λ-calculus [e.g., 56, Chapter 6].

A de Bruijn index is a number used in the place of a variable name; it counts the number of binders that intervene between a variable occurrence and its binding site. We see above that the expression `\x:Int->Int. \y:Int x y` desugars to `λ#:Int -> Int. λ#:Int. #1 #0`, where the `#1` refers to the outer binder (1 intervening binding site) and the `#0` refers to the inner binder (0 intervening binding sites). De Bruijn indices have the enviable property of making α-equivalence utterly trivial: because variables no longer have names, we need not worry about renaming. However, they make other aspects of implementation harder. Specifically, two challenges come to the fore:

1. De Bruijn indices are hard for programmers to understand and work with.
2. As an expression moves into a new context, the indices may have to be shifted (increased or decreased) in order to preserve their identity, as the number of intervening binding sites might have changed. It is very easy for an implementor to make a mistake when doing these shifts.

As a partial remedy to the first problem, Stitch color-codes its output (as can be seen in this typeset document). A variable occurrence and its binding site are assigned the same color, so that a reader no longer has to count binding sites. Though only a modest innovation, this color-coding has proved to be wildly successful in practice; not only has it been helpful in my own debugging, but working functional programmers who see it have gasped, "I finally understand de Bruijn indices now!" more than once. Note that programmers never have to *write* using de Bruijn indices (the parser converts their names to indices quite handily) and so this simple reading aid goes a long way toward fixing the first drawback.

The second drawback can be more troublesome. The reason we have such a plethora of approaches to variable binding must be, in part, that implementors have been unhappy with the approaches available—they thus invent a new one. One reason for this unhappiness is that capture-avoiding substitution is a real challenge. Pierce [56, Section 5.3] gives an instructive account of the pitfalls an implementor encounters. And it is not just substitution. As a language grows in complexity, dealing with name clashes and renaming crops up in a variety of places. Indeed, the venerable GHC implementation only relatively recently (January, 2016) added checks to make sure its handling of variable naming is bug-free; I count 33 call sites within the GHC source code (as of March, 2020) that still use the "unchecked" variant of substitution because using the checked version fails on certain test cases. Each of these call sites is perhaps a lurking bug, waiting for a pathological program to induce an unexpected name clash that could cause GHC to go wrong.

However, a solution to this conundrum is at hand: because Stitch's expression AST type is indexed by the type of the expression represented, an erroneous or forgotten shifting of a de Bruijn index leads to a straightforward error, caught as Stitch itself is being compiled. Indeed, I shudder to think about the challenge in getting all the shifts correct without the aid of an indexed AST. Thus, using an indexed AST fully remedies the second drawback.

One twist on the second drawback remains, however: all this shifting can slow the interpreter down. A variable shift requires a full traversal and rebuild of the AST, costing precious time and allocations. Though I have not done it in my implementation, it would be possible to add a *Shift* constructor to the AST type to allow these shifts to be lazily evaluated; the design and implementation of other opportunities for optimization are left as future work.

2.3 A Slightly Longer Example: Primality Checking

As a final example of a user's interaction with Stitch, I present the program in Figure 2. It implements a primality checker in Stitch. The file `prime.stitch`,[5] can be loaded into the Stitch REPL with `:load prime.stitch`.

```
λ> :load prime.stitch
...
λ> isPrime 7
true : Bool
λ> isPrime 9
```

Stitch source, `prime.stitch`:

```
noDivisorsAbove =
 fix \nda: Int -> Int -> Bool.
  \tester:Int. \scrutinee:Int.
   if tester * tester > scrutinee
   then true
   else if scrutinee % tester == 0
    then false
    else nda (tester+1) scrutinee ;

isPrime = noDivisorsAbove 2
```

After parsing and type checking:

```
noDivisorsAbove =
 fix λ#:Int -> Int -> Bool.
  λ#:Int. λ#:Int.
   if #1 * #1 > #0
   then true
   else if #0 % #1 == 0
    then false
    else #2 (#1 + 1) #0
     : Int -> Int -> Bool

isPrime = fix ... 2 : Int -> Bool
```

Figure 2. A primality checker in Stitch.

```
false : Bool
```

In the right half of the figure, we see Stitch's parsed and type-checked representation of the original program. This AST cannot store global variables (all variables are de Bruijn indices), so Stitch inlines `noDivisorsAbove` in the definition of `isPrime`, above.

We are now almost ready to start seeing the fancy types, but first, we need to install some necessary infrastructure.

3 Fancy-Typed Utilities

Every great edifice necessarily requires some plumbing. What is fun in this case is that even the plumbing needs some fancy types in order to support what comes ahead. The definitions in this section are standard, and readers familiar with dependently typed programming may wish to skim this section quickly or skip to the next section. The utilities described here are useful beyond just Stitch, and some have implementations released separately. However, I have included them within the Stitch package in order to keep it self-contained. These modules, too, are prefixed with `Language.Stitch.` so as not to pollute the module namespace. This section introduces Peano natural numbers (useful for tracking the number of bound variables), length-indexed vectors (useful for tracking the types of in-scope variables), and singletons (useful

[5] https://gitlab.com/goldfirere/stitch/-/blob/hs2020/tests/prime.stitch

during type checking, when we must connect a type-level context with term-level type representations).

3.1 Length-Indexed Vectors

No exploration of fancy types would be complete without the staple of length-indexed vectors, a ubiquitous example because of their perspicuity and usefulness. A length-indexed vector is simply a linked list, where the list type includes the length of the list; thus, a list of length 2 is a distinct type from a list of length 3. Here is the type definition:

```
data Nat = Zero | Succ Nat
data Vec :: Type → Nat → Type where
  VNil :: Vec a Zero
  (:>) :: a → Vec a n → Vec a (Succ n)
```

We will take this line-by-line. First is the declaration of unary natural numbers. This type is terribly inefficient at run-time, but we use it only at compile-time [73], where it gives us nice inductive reasoning principles. We next see that *Vec* is parameterized by an element type of kind *Type* and a length index of kind *Nat*. The declaration for *VNil* states that *VNil* is always a *Vec* of length *Zero*, but it can have any element type *a*. The cons operator :> takes an element (of type *a*), the tail of the vector (of type *Vec a n*) and produces a vector that is one longer than the tail (of type *Vec a* (*Succ n*)).

Note the use of *Nat* as a kind and *Zero* and *Succ* as types. When GHC is resolving names used in a type, it first looks in the type-level namespace, where definitions like *Vec* and *Nat* live. Failing that lookup (for capitalized identifiers), it looks in the term-level namespace; this is what happens in the case of *Zero* and *Succ*.[6] Finding these constructors, GHC has no trouble using them in types, where they keep their usual meaning.[7]

3.1.1 Appending. We will need to append vectors, and the two vectors may be of different lengths. Clearly, the append function should take arguments of type *Vec a n* and *Vec a m*, where the element type *a* is the same but the length indices *n* and *m* are different. However, what should the result type of appending be? Of course, the length of the concatenation of two vectors is the sum of the lengths of the vectors: the result should be *Vec a* (*n* + *m*). We thus need to define + on *Nat*s. What is unusual here is that we need to use + in *types*, not in terms. GHC's approach here is to use a *type family* [10, 18], which is essentially a function that works on types and type-level data. Here are the definitions:

```
type family n + m where
  Zero   + m = m
  Succ n + m = Succ (n + m)
```

[6]If the identifier exists in both namespaces, it can be prefixed with ' to tell GHC to look only in the term-level namespace.

[7]In my experience, these hand-written unary naturals work better than GHC's built-in naturals for defining vectors, owing to their inductive structure.

```
(⧺) :: Vec a n → Vec a m → Vec a (n + m)
VNil      ⧺ ys = ys
(x :> xs) ⧺ ys = x :> (xs ⧺ ys)
```

Already, the fancy types are working for us, making sure our code is correct. In the first clause of ⧺, we pattern-match on *VNil*. This match tells us both that the first vector is empty, and also that the type variable *n* equals *Zero*. This second fact comes from the declared type of *VNil* in the definition of *Vec*. All *VNil*s have a type index of *Zero*, and thus we know that if *VNil* :: *Vec a n*, then *n* must be *Zero*. The type checker uses this fact to accept the right-hand side of that equation: it must be convinced that *ys* :: *Vec a* (*n* + *m*), the declared return type of ⧺. Because the type checker knows that *n* is *Zero*, however, it can use the definition of the type family + to reduce *Zero* + *m* to *m*, and then it simply uses the fact that *ys* :: *Vec a m*, as *ys* is the second argument to ⧺. The second equation is similar, except that it uses the second equation of + to check the equation's right-hand side. If we forgot to cons *x* onto *xs* ⧺ *ys* in this right-hand side, the definition of ⧺ would be rejected as ill typed.

3.1.2 Indexing. How should we look up a value in a vector? We could use an operator like Haskell's standard !! operator that looks up a value in a list. However, this is unsatisfactory, because the !! throws an exception when its index is out of range. Given that we know a vector's length at compile-time, we can do better.

The key step is to have a type that represents natural numbers less than some known bound. The type *Fin* (short for "finite set"), common in dependently typed programming, does the job:

```
data Fin :: Nat → Type where
  FZ :: Fin (Succ n)
  FS :: Fin n → Fin (Succ n)
```

The *Fin* type is indexed by a natural number *n*. The type *Fin n* contains exactly *n* values, corresponding to the numbers 0 through $n - 1$. This GADT tends to be a bit harder to understand than *Vec* because (unlike *Vec*), you cannot tell the type of a *Fin* just from the value. For example, the value *FS FZ* can have both type *Fin* 2 and *Fin* 10 (taking liberty to use decimal notation instead of unary notation for *Nat*s), but not *Fin* 1. Let us understand this type better by tracing how we can assign a type to *FS FZ*:

- Suppose we are checking to see whether *FS FZ* :: *Fin* 1. We see that *FS* :: *Fin n* → *Fin* (*Succ n*). Thus, for *FS FZ* :: *Fin* 1, we must instantiate *FS* to have type *Fin Zero* → *Fin* (*Succ Zero*). We must now check *FZ* :: *Fin Zero*. However, this fails, because *FZ* :: *Fin* (*Succ n*)—that is, *FZ*'s type index must not be *Zero*. We accordingly reject *FS FZ* :: *Fin* 1.
- Now say we are checking *FS FZ* :: *Fin* 5. This proceeds as above, but in the end, we must check *FZ* :: *Fin* 4. The

number 4 is indeed the successor of another natural, and so *FZ* :: *Fin* 4 is accepted, and thus so is *FS FZ* :: *Fin* 5.

Following this logic, we can see how *Fin n* really has precisely *n* values.

As a type whose values range from 0 to $n - 1$, *Fin n* is the perfect index into a vector of length *n*:

```
(!!!) :: Vec a n → Fin n → a
vec !!! fin = case (fin, vec) of     -- reversed due to laziness
  (FZ,   x :> _)  → x
  (FS n, _ :> xs) → xs !!! n
```

GHC comes with a pattern-match completeness checker [29] that marks this **case** as complete, even without an error case. To understand why, we follow the types. After matching *fin* against either *FZ* or *FS n*, the type checker learns that *n* must not be zero—the types of both *FZ* and *FS* end with a *Succ* index. Since *n* is not zero, then it cannot be the case that *vec* is *VNil*. Even though the pattern match includes only :>, that is enough to be complete.

Now, we can explore this match reversal. Haskell is a lazy language [44], which means that variables can be bound to diverging computations (denoted with ⊥). When matching a compound pattern, Haskell matches the patterns left-to-right, meaning that the left-most scrutinee (*fin*, in our case) is evaluated to a value and then inspected before evaluating later scrutinees, such as *vec*. Imagine matching against *vec* first. In this case, it is conceivable that *vec* would be *VNil* while *fin* would be ⊥. This is not just theoretical; witness the following function:

```
lazinessBites :: Vec a n → Fin n → String
lazinessBites VNil _ = "empty vector"
lazinessBites _    _ = "non-empty vector"
```

If we try to evaluate *lazinessBites VNil undefined*, that expression is accepted by the type checker and evaluates handily to `"empty vector"`. If we scrutinize *vec* first, then, the completeness checker correctly tells us that we must handle the *VNil* case. On the other hand, in the implementation of !!! with the pattern match reversed, we ensure that *fin* is not ⊥ before ever looking at *vec* and can thus be sure that *vec* cannot be *VNil*.

3.2 Singletons

The technique of *singletons* is a well worn and well studied [41] way to simulate dependent types in a non-dependent language. Though at least two libraries exist for automatically generating singletons in Haskell [19, 40], Stitch does not depend on these libraries, in order to maintain some simplicity and be self-contained. However, the design of these libraries is the direct inspiration for the definitions in Stitch.

To motivate singletons, consider writing *replicate* for vectors. The *replicate* function takes a natural number *n* and an element *elt* and creates a vector of length *n* consisting of *n* copies of *elt*. Despite this simple specification, there is no easy way to write a type signature for *replicate*; you might try *replicate* :: *Nat* → *a* → *Vec a* ?, but you'd be stuck at the ?. The problem is that the choice of the *type* index for the return type must be the *value* of the first parameter. This is the hallmark of dependent types. However, because Haskell does not yet support dependent types, singletons will have to do. Here is the definition of a singleton *Nat* (or, more precisely the family of singleton *Nat*s):

```
data SNat :: Nat → Type where
  SZero :: SNat Zero
  SSucc :: SNat n → SNat (Succ n)
```

The type *SNat* is indexed by a *Nat* that corresponds to the value of the *SNat*. That is, the type of *SSucc* (*SSucc SZero*) is *SNat* (*Succ* (*Succ Zero*)). Conversely, the *only* value of the type *SNat* (*Succ* (*Succ Zero*)) is *SSucc* (*SSucc SZero*). This last fact is why singleton types are so named: a singleton type has precisely one value. Because of the correspondence between types and terms with singleton types, matching on the values of a singleton inform the type index—exactly what we need here.

Here is the definition for *replicate*:

```
replicate :: SNat n → a → Vec a n
replicate SZero      _   = VNil
replicate (SSucc n′) elt = elt :> replicate n′ elt
```

The GADT pattern match against *SZero* tells the type checker that *n* is *Zero* in the first equation, making *VNil* an appropriate result. Similarly, the match tells the type checker that *n* is *Succ n′* (for some *n′*) in the second equation, and thus a vector one longer than *n′* is an appropriate result. Essentially, the *n* in the type signature for *replicate* *is* the value of the first parameter, exactly as desired.

Because a singleton value is uniquely determined by its type, it is convenient to be able to pass singletons implicitly. We can take advantage of Haskell's type class mechanism to do this, via the following type class and instances:

```
class                SNatI (n :: Nat) where snat :: SNat n
instance             SNatI Zero       where snat = SZero
instance SNatI n ⇒ SNatI (Succ n)    where snat = SSucc snat
```

Any function with a *SNatI n* constraint can gain access to the singleton for *n* simply by calling the *snat* method.

Singletons are not the final word for dependent types in Haskell. They can be unwieldy [36] and conversions between singleton types and unrefined base types (such as converting from *SNat n* to *Nat*) are potentially costly. Work is under way [17, 25, 68, 72] to add full dependent types to Haskell. However, for our present purposes, the singletons work quite nicely, and their drawbacks do not bite.

```
  -- Stitch types, and their singletons
data Ty = TInt | TBool | Ty :→ Ty
data STy :: Ty → Type where
  SInt  :: STy TInt
  SBool :: STy TBool
  (::→) :: STy arg → STy res → STy (arg :→ res)
toSTy :: Ty → (∀t. STy t → r) → r
toSTy TInt      k = k SInt
toSTy TBool     k = k SBool
toSTy (a :→ b) k = toSTy a $ λsa → toSTy b $ λsb →
                     k (sa ::→ sb)
  -- Propositional equality
data (a :: k) :∼:(b :: k) where
  Refl :: a :∼: a
  -- Informative equality comparison
class TestEquality (t :: k → Type) where
  testEquality :: t a → t b → Maybe (a :∼: b)
instance TestEquality STy where ...
```

Figure 3. Stitch types and singletons

4 Stitch Types

We start our exploration of the Stitch implementation by looking at its representation for types, in Figure 3. The type definition, *Ty* is uninteresting, defining integers, Booleans, and functions between these. However, *Ty* is not enough: in order to build our indexed AST, we will need to reason about Stitch types both at runtime *and* at compile time. We thus need the *singleton* type *STy*, indexed by *Ty*.

When processing a λ-abstraction, Stitch needs to parse the type annotation on the argument, producing a *Ty*. During type-checking, however, Stitch needs an *STy*; we thus must be able to convert from *Ty* to *STy*. This is done in the *toSTy* function. However, we cannot give this function a type such as *Ty* → *STy t*: there is no way to choose what the output *t* should be. What we would like to write, ideally, is *toSTy* :: *Ty* → ∃*t*. *STy ty*. However, Haskell does not support such a convenient construct. While Haskell's support for existential variables in datatypes could work here, I found that continuation-passing style (CPS), as seen in the higher-rank type of *toSTy* in Figure 3, was easier and made for code with a better flow. With CPS, we can easily pass the type index *t* to the continuation *k*.

A critical job of any type-checker is comparing types for equality. In Stitch, though, we must compare the singleton *STy*s, not the unrefined *Ty*s. The usual (==) operator will not work for us, because the two *STy*s we are comparing might have different type indices. Instead, we want to be able to compare *STy a* with *STy b*. Furthermore, if *STy a* equals *STy b*, we need to be able to tell GHC's type-checker that *a* equals *b*: this will allow GHC to accept the implementation of Stitch's type checker. (See the case for type-checking function applications in Section 7 for an illustrative example.) Because this general pattern—testing an indexed type for equality in order to get a type equality usable by GHC—comes up with some regularity when doing fancy-typed programming, GHC includes the (:∼:) type and *TestEquality* class in its `Data.Type.Equality` module. These also appear in Figure 3.

The type (:∼:) encodes *propositional equality*. That is, if you have a value of type *a* :∼: *b*, then you can pattern-match on this value to learn that *a* equals *b*. Types that are indexed by *a* will now be equal to types indexed by *b*. The *testEquality* method in *TestEquality* thus optionally returns *a* :∼: *b*; this way, if the test succeeds, we can pattern-match on the result to learn that two types should be considered equivalent. We call this an *informative* equality comparison, because it informs GHC's type checker of the equality. In contrast, a *Bool* return type would not.

Having seen how types are represented throughout Stitch, we are now ready to start exploring the Stitch pipeline, beginning with the parser.

5 Scope-Checked Parsing

Though Stitch's hallmark is its indexed AST for expressions, we cannot parse into that AST directly. Type-checking can produce better error messages and is more easily engineered independent from the left-to-right nature of parsing. We thus must define an unchecked (un-indexed) AST for the result of parsing the user's program.

However, even here there is a role for fancy types. While type-checking during parsing is a challenge, name resolution during parsing works nicely. We can thus parse into an AST that can express only well-scoped terms. The AST type definition follows:

```
  -- Unchecked expression, indexed by the
  -- number of variables in scope
data UExp (n :: Nat)
  = UVar (Fin n)    -- de Bruijn index for a variable
  | ULam Ty (UExp (Succ n))
  | UApp (UExp n) (UExp n)
  | UIntE Int
    ...
```

The type *UExp* ("unchecked expression") is indexed by a *Nat* that denotes the number of local variables in scope. So, a *UExp* 0 is a closed expression, while a *UExp* 2 denotes an expression with up to two free variables. Note that *ULam* increments this index for the body of the λ-abstraction.

Variables are naturally stored in a *Fin n*—precisely the right type to store de Bruijn indices. If an expression has only 2 variables in scope, then we must make sure that a variable has an index of either 0 or 1, never more. Using *Fin* gives us this guarantee nicely.

Lambda-abstractions store a *Ty*, the type of the bound variable. Types are further explored in Section 4. Note that there is no explicit place in the AST for the bound variable, as the bound variable always has a de Bruijn index of 0.

The main novelty in working with *UExp* is, of course, the *Fin n* type for de Bruijn indices. Supporting this design requires accommodations in the parser. Stitch's parser is a monadic parser built on the Parsec library [35]. Its input is the series of tokens, each annotated with location information, produced by the entirely unremarkable lexer (also built using Parsec). It can parse either statements or expressions.

The most interesting aspect of the parser is that the parser type must be indexed by number of in-scope variables—this is what will set the index of any parsed *Fin* de Bruijn indices. We thus have this definition for the parser monad:

```
type Parser n a
  = ParsecT [LToken] () (Reader (Vec String n)) a
```

The *ParsecT* monad transformer [27] is indexed by (1) the type of the input stream, which in our case is [*LToken*]; (2) the state carried by the monad, which in our case is trivial; (3) an underlying monad, which in our case is *Reader* (*Vec String n*), where the environment is a vector of the names of the in-scope variables; and (4) the return type of computations, *a*. Thus, a computation of type *Parser n a* parses a list of located tokens into something of type *a* in an environment with access to the names of *n* in-scope local variables.

5.1 A Heterogeneous Reader Monad

The only small difficulty in working with *Parser*, as defined above, is around variable binding (naturally). Here is the relevant combinator:

```
bind :: String → Parser (Succ n) a → Parser n a
bind bound_var thing_inside
  = hlocal (bound_var :>) thing_inside
```

Given a bound variable name, *bind* parses some type *a* in an extended environment (with *Succ n* bound variables) and then returns the result in an environment with only *n* bound variables. Note that *bind* does *not* do any kind of shifting or type-change of the result: if the inner parser is of type, say, *Parser* (*Succ n*) (*Fin* (*Succ n*)), then the outer result will have type *Parser n* (*Fin* (*Succ n*)). Note that the index to the *Fin* does not change.

The *bind* function is implemented using a new combinator *hlocal*, inspired by the *local* method of the *MonadReader* class from the mtl (monad transformer library). The relevant part of this class is

```
class Monad m ⇒ MonadReader r m | m → r where
  local :: (r → r) → m a → m a
  ...
```

The *local* method allows a computation to assume a local value of the environment for some smaller computation. This is exactly what we want here. The only problem is that the type of the local environment is *different* than the type of the outer environment: the outer environment has type *Vec String n* while the local one has type *Vec String* (*Succ n*).

We must accordingly define a heterogeneous reader monad, which allows a type change for the local environment. Here is the class definition:

```
class Monad m ⇒ MonadHReader r₁ m | m → r₁ where
  type SetEnv r₂ m :: Type → Type
  hlocal :: (r₁ → r₂)
         → (Monad (SetEnv r₂ m) ⇒ SetEnv r₂ m a)
         → m a
```

The *MonadHReader* class allows for the possibility that the environment (denoted with the *r* variables here) in a local computation is different than the environment in the outer computation. Because there may be many types that have *MonadHReader* instances, we must use the associated type family *SetEnv* to update the monad type with the new environment type.

In the inner computation, we need to know that the underlying monad, with the updated environment, is still a member of the *Monad* type class. This fact is assumed by putting the constraint *Monad* (*SetEnv* r_2 *m*) on the inner computation, leveraging Haskell's support for higher-rank types [49].[8]

Returning to our indexed parser, we need these two instances:

```
instance Monad m ⇒ MonadHReader r₁ (ReaderT r₁ m) where
  type SetEnv r₂ (ReaderT r₁ m) = ReaderT r₂ m
  hlocal f thing_inside = ...
instance MonadHReader r₁ m
  ⇒ MonadHReader r₁ (ParsecT s u m) where
  type SetEnv r₂ (ParsecT s u m) = ParsecT s u (SetEnv r₂ m)
  hlocal f thing_inside = ...
```

Here, *ReaderT* is the monad-transformer form of the *Reader* monad we saw earlier in the definition of *Parser*. (*Reader* is just defined to be a *ReaderT* based on the *Identity* monad.) The first instance says that the environment associated with a *ReaderT* r_1 *m* is r_1; that is why the r_1 is the first parameter in the *MonadHReader* instance. It then describes that to update the environment from r_1 to r_2, we just replace the type parameter to *ReaderT*. The implementation is straightforward and elided here.

[8] A reader informed about recent updates to GHC might wonder why we do not use *quantified constraints* [6] here. While this approach would seem to work, the current implementation fails us, because the head of a quantified constraint cannot be a type family, as described at https://ghc.haskell.org/trac/ghc/ticket/14860.

```
type Ctx n = Vec Ty n
data Exp :: ∀n. Ctx n → Ty → Type where
  Var  :: Elem ctx ty → Exp ctx ty
  Lam  :: STy arg → Exp (arg :> ctx) res
          → Exp ctx (arg :→ res)
  App  :: Exp ctx (arg :→ res) → Exp ctx arg → Exp ctx res
  IntE :: Int → Exp ctx TInt
  ...
  -- An encoding of (\x:Int. x) 5, as an example
example :: Exp VNil TInt
example = App (Lam SInt (Var EZ)) (IntE 5)
```

Figure 4. The type-indexed Exp expression AST

The ParsecT instance lifts a MonadHReader instance through the ParsecT monad transformer, propagating the action of SetEnv. The implementation requires the usual type chasing characteristic of monad-transformer code, but offered no particular coding challenge.

With all this in place, it is straightforward to use the hlocal method in the bind function, giving us exactly the behavior that we want.

6 The Type-Indexed Expression AST

We now are ready to greet the Exp type, the type-indexed AST for expressions. Its definition appears in Figure 4. The Exp type is indexed by two parameters: a typing context of kind Ctx n, where n is the number of bound variables; and a the expression's type, a Ty.

Compare the definition of Exp with the typing rules in Figure 1. Each constructor corresponds with precisely one rule. The types of the constructor arguments correspond precisely with the premises of the rule, and the type of the constructor result corresponds precisely with the rule conclusion. Take function application as an example. The T_App rule has two premises: one gives expression e_1 type $\tau_1 \to \tau_2$, and the other checks to see that e_2 has the argument type τ_1. In the same way, the first argument to the constructor App takes an expression in some context ctx and with some type arg :→ res. The second argument to App then has type arg. Furthermore, just as the conclusion to the T_App rule says that the overall $e_1\ e_2$ expression has type τ_2, the result type of the App constructor is an expression of type res. An easier example is for the constructor IntE, where the resulting type is simply TInt, regardless of the context.

Note the Lam constructor for building λ-abstractions. The first argument is STy arg. This argument contains both a Stitch type, suitable for runtime comparisons and pretty-printing, and also a compile-time type index arg, used later in the type of Lam. Like replicate, this is a place where a dependent type is called for. Happily, the STy singleton works well here.

The definition of Exp shows us why modeling a typed language is such a perfect fit for GADTs—the information in the typing rules is directly expressed in the AST type definition.

Perhaps the most distinctive aspect of Exp—other than its indices—is the choice of representation for variables. Exp continues our use of de Bruijn indices, but we must be careful here: we need the type of a variable to be expressed in the return index to the Var constructor. While it is conceivable to do this via some Lookup type family, the Elem type is a much more direct approach:

```
data Elem :: ∀a n. Vec a n → a → Type where
  EZ :: Elem (x :> xs) x
  ES :: Elem xs x → Elem (y :> xs) x
```

The Elem type is indexed by a vector (of any element type a) and a distinguished element of that vector. An Elem value, when viewed as a Peano natural number, is simply the index into the vector that selects that distinguished element. Equivalently, a value of type Elem xs x is a proof that x is an element of the vector xs; the computational content of the proof is x's location in xs.

The definitions of the two constructors support this description. The EZ constructor has type Elem (x :> xs) x—we can see plainly that the distinguished element x is the first element in the vector. The ES constructor takes a proof that x is in a vector xs and produces a proof that x is in the vector y :> xs (for any y). Naturally, x's index in y :> xs is one greater than x's index in xs, thus underpinning the interpretation of ES as a Peano successor operator.

In the case of our use of Elem within the Exp type, the vectors at hand are contexts (vectors of Tys) and the elements are types of Stitch variables. The Elem type gives us exactly what we need: a type-level relationship between a context and a type, along with the term-level information (the de Bruijn index) to locate that type within that context.

7 The Sound Type-Indexed Type Checker

We are ready now for the part we have all been waiting for: the sound type-indexed type checker. The core cases appear in Figure 5; these cases illustrate the points of interest.

The check function takes an unchecked expression of type UExp and converts it into a checked expression of type Exp. For the same reasons that toSTy was written using CPS in Section 4, we use CPS here. We also must pass STy t to the continuation, so that runtime comparisons can be performed.

The check function works over closed expressions, as we always call it on a top-level expression. However, it must recur into open expressions, and so we define the more-general go local helper function. The go function's type mimics that

```
check :: MonadError Doc m
      ⇒ UExp Zero
      → (∀(t :: Ty). STy t → Exp VNil t → m r)
      → m r
check = go SCNil where
  go :: MonadError Doc m
     ⇒ SCtx (ctx :: Ctx n) → UExp n
     → (∀t. STy t → Exp ctx t → m r) → m r

  go ctx (UVar n) k = check_var n ctx $ λty elem →
                          k ty (Var elem) where
    check_var :: Fin n → SCtx (ctx :: Ctx n)
              → (∀t. STy t → Elem ctx t → m r) → m r
    check_var FZ      (ty :%> _)     k0 = k0 ty EZ
    check_var (FS n0) (_ :%> ctx0) k0 =
      check_var n0 ctx0 $ λty elem → k0 ty (ES elem)

  go ctx (ULam ty body) k =
    toSTy ty $ λsty →
    go (sty :%> ctx) body $ λres_ty body′ →
    k (sty ::→ res_ty) (Lam sty body′)

  go ctx e@(UApp e1 e2) k =
    go ctx e1 $ λfun_ty e1′ →
    go ctx e2 $ λarg_ty e2′ →
    case fun_ty of
      arg_ty′ ::→ res_ty
          | Just Refl ← testEquality arg_ty arg_ty′
         → k res_ty (App e1′ e2′)
      _  → typeError e ...

  go _ (UIntE n) k = k SInt (IntE n)
```

Figure 5. The sound type-indexed type checker (excerpted)

of *check* but allows for the possibility of open expressions, quantifying over the context length, *n*, and context *ctx*. Because we will need to look up variable types at runtime, we need the context to be available both at compile-time (to use as an index to *Exp*) and at runtime. This means that we need a singleton for the context, as embodied by this definition:

```
data SCtx :: ∀n. Ctx n → Type where
  SCNil :: SCtx VNil
  (:%>) :: STy t → SCtx ts → SCtx (t :> ts)
```

Checking variables. The variable case is handled by the helper function *check_var*. The *check_var* function uses the *Fin n* stored by the *UVar* constructor to index into the typing context, stored as the singleton *SCtx*. When *check_var* finds the type it is looking for, it passes that type to the continuation, along with an *Elem* value which will store the de Bruijn index in the *Exp* type. GHC's type checker is working hard here to make sure this function definition is correct, using the definition of *Fin* to ensure that our pattern-match is complete,[9] and that the *Elem* we build really does show that the type *t* is in the context *ctx*. Note that there is no possibility of errors here: the use of *Fin* in the *UExp* type guarantees that the variable is in scope.

Checking a λ-abstraction. The *Lam* case is where we use the *toSTy* function introduced in Section 4. After converting the input *Ty* into an *STy* named *sty*, we check the abstraction body, learning its result type *res_ty* and getting the type-checked expression *body′*. We then continue with a function type composed from *sty* and *res_ty*, using the ::→ constructor of *STy*.

Checking an application. Checking function applications is really the heart of any type checker: this is the principal place where two types may be in conflict. In our case, we check the two expressions separately, getting their types and type-checked expression trees. We then must ensure that *fun_ty*, the type of the applied function, is indeed a function type. This is done by a **case**-match, looking for a ::→ constructor. We then must ensure that the actual argument type *arg_ty* matches the function's expected argument type *arg_ty′*. We use the *testEquality* function, explained in Section 4. If successful, this function returns a proof to the type checker that *arg_ty* equals *arg_ty′*, and we are then allowed to build the application with *App*. If either check fails, we issue an error.

The type discipline in Stitch works to keep us correct here. If we skipped the type checks, the *App* application would be ill-typed, as *App* expects its first argument to be a function and its second argument to have the argument type of that function. The checks ensure this to GHC, which then allows our use of *App* to succeed.

There are several more cases in the type checker, all similar to those presented here. In all, this type checker was remarkably easy to write, given the groundwork in setting up the types correctly. GHC's type checker stops us from making mistakes here—the whole point of using an indexed expression AST. Furthermore, the type errors I encountered during implementation were indeed helpful, pointing out any missing type equality checks.

Beyond these observations, I wish to note simply that such a type checker is possible to write at all. In conversations with experienced functional programmers, some have been surprised that the type-indexed expression AST has any practical use, despite the fact that this technique is not new [e.g., 43]. After all, how could you guarantee that expressions are well typed? The answer is: check them first, as *check* does for us here.

[9]Note that we match the *Fin* before the vector, as we did in Section 3.1.2.

```
data Length :: ∀a n. Vec a n → Type where
  LZ :: Length VNil
  LS :: Length xs → Length (x :> xs)
subst :: ∀ctx s t. Exp ctx s → Exp (s :> ctx) t → Exp ctx t
subst e = go LZ where
  go  ::  Length (lcls :: Ctx n) → Exp (lcls ++ s :> ctx) t₀
    → Exp (lcls ++ ctx) t₀
  go len (Var v)            = var len v
  go len (Lam ty body) = Lam ty (go (LS len) body)
  ...    -- other forms are treated homomorphically
  var  ::  Length (lcls :: Ctx n) → Elem (lcls ++ s :> ctx) t₀
    → Exp (lcls ++ ctx) t₀
  var LZ        EZ      = e            -- no locals; substitute
  var LZ        (ES v) = Var v        -- no locals; decrement
  var (LS _)   EZ      = Var EZ    -- var is local; no change
  var (LS len) (ES v) = shift (var len v)    -- recur
```

Figure 6. Indexed substitution

8 Evaluation with an Indexed AST

Writing evaluators is where the indexed AST really shines: we essentially can not get it wrong.

A type-indexed AST allows us to easily write a *tagless* interpreter, where a value does not need to be stored with a runtime tag that indicates the value's type. To see the problem, imagine an unindexed AST and a function *eval*::*Exp* ⟶ *Value*. The *Value* type would have to be a sum type with several constructors, say, for integer, Boolean, and function values. This means that every time we extract a value, we have to check the tag, a potentially costly step at runtime. However, with our indexed expression type, we can evaluate to a type *Value ty*, where *Value* is this type family:

```
type family Value t where
  Value TInt      = Int
  Value TBool     = Bool
  Value (a :→ b) = Exp VNil a → Exp VNil b
```

Values are accordingly tagless—no runtime check needs to be performed when inspecting one. Tagless interpreters have been studied at some length [9, 43, 60], and we will not explore this aspect of Stitch further.

Evaluation is as one might expect. The interesting part is about substitution, which we focus on next.

8.1 Substitution

Substitution is the bane of implementors using de Bruijn indices. Once again, the type indices save us from making errors—there seems to be no real way to go wrong, and the type errors that we encounter gently guide us to the right answer. The final result is in Figure 6.

The *subst* function takes an expression *e* of type *s* and another expression with a free variable of type *s* and substitutes *e* into the latter expression. The *subst* function's type requires that the variable to be substituted have a de Bruijn index of 0, as is needed during β-reduction. However, as anyone who has proved a substitution lemma knows, we must generalize this type to get a powerful enough recursive function to do the job.

Note that the type of *subst* is precisely the shape of a substitution lemma: that if $\Gamma \vdash e_1 : \sigma$ and $\Gamma, x{:}\sigma \vdash e_2 : \tau$, then $\Gamma \vdash e_2[e_1/x] : \tau$. A proof of this lemma must strengthen the induction hypothesis to allow bound local variables, leading to a proof of this stronger claim: if $\Gamma \vdash e_1 : \sigma$ and $\Gamma, x{:}\sigma, \Gamma' \vdash e_2 : \tau$, then $\Gamma, \Gamma' \vdash e_2[e_1/x] : \tau$. If we call Γ' *lcls* and Γ *ctx*, this strengthened induction hypothesis matches up with the type of the helper function *go*. (Recall that contexts in the implementation are in reverse order to those in the formalism.) As one implements such a function, this correspondence is a strong hint that the function type is correct.

The *go* function takes one additional argument: a value of type *Length lcls*. The *Length* type is included in Figure 6; values are Peano naturals that describe the length of a vector.[10] This extra piece is necessary as local variables get treated differently in a substitution than do variables from the outer context. The number of locals informs the *var* function when to substitute, when to shift, and when to leave well enough alone. Pierce [56, Chapter 6] offers an accessible introduction to the delicate operation of substitution in the presence of de Bruijn indices, and a full exploration of this algorithm would take us too far afield; suffice it to say that any misstep in *var* would be caught by GHC's type checker.

For an example of a plausible mistake and its error message, imagine we forgot to call *shift* (explained below) in the last equation of the *var* helper function. GHC produces an error saying it

```
Could not deduce: (xs ++ ctx) ~~ (x :> (xs ++ ctx))
from the context: (..., lcls ~~ (x :> xs))
...
Expected type: Exp (lcls ++ ctx) t₀
Actual type:    Exp (xs ++ ctx) t₀
```

We can see here that the actual type of *var len v* does not account for adding the new variable, *x*, to the context. This must mean we need to add that variable; the way to do so is via a shifting operation, which we cover next.

8.2 Shifting

As hinted at previously, substitution with de Bruijn indices is subtle not only because it is hard to keep track of which

[10] Although vectors are indexed by their length, that index is a compile-time natural only. To get the length of a vector at runtime, it is still necessary to recur down the length of the vector.

```
class Shiftable (a :: ∀n. Ctx n → Ty → Type) where
  shifts :: Length prefix → a ctx ty → a (prefix +++ ctx) ty
  shifts0 :: a VNil ty → a prefix ty    -- closed exprs only
  unshifts :: Length prefix → a (prefix +++ ctx) ty
           → Maybe (a ctx ty)    -- needed for CSE
instance Shiftable Exp  where ...
instance Shiftable Elem where ...
  -- Common case: shifting by one
shift :: ∀(a :: ∀n. Ctx n → Ty → Type) ctx t ty.
      Shiftable a ⇒ a ctx ty → a (t :> ctx) ty
shift = shifts (LS LZ)
```

Figure 7. De Bruijn index shifting

variable one is substituting, but also because the expression being substituted suddenly appears in a new context and accordingly may require adjustments to its indices. This process is called *shifting*. If we have an expression `#1 #0` (where both variables are free) and wish to substitute into an expression with an additional bound variable, we must shift to `#2 #1`. I have intentionally kept the colors consistent during the shift, as the identity of these variables does *not* change—just the index does.

Shifting is an operation that makes sense both on full expressions *Exp* and also on indices *Elem* directly. We will discover that both of these are sometimes necessary when performing common-subexpression elimination (CSE, Section 9), and so we generalize the notion of shifting by introducing a type class, presented in Figure 7.

The first detail to notice here is that *Shiftable* classifies a polykinded type variable *a*—note the ∀*n* in *a*'s kind. This gives *Shiftable* a *higher-rank kind*. GHC deals with this exotic species in stride; the only challenge is that GHC will never infer a variable to have a polykind, and so all introductions of *a* must be written with a kind annotation. The polymorphism in the kind of *a* is essential here because, as a stand-in for *Exp* or *Elem*, *a* must be able to be applied to contexts of any length. Without this polymorphism, it would be impossible to write the *Shiftable* class.

As before, the implementation of these instances is straightforward, once we have written down the types and can be guided by GHC's type checker.

8.3 Shifting Closed Expressions

The *Shiftable* class includes a method *shifts0*, specializing *shifts* to work over closed expressions. Closed expressions are a special case for shifting, because we can prove that no variables need to be shifted. And yet, shifting also changes the *type* of the expression (from *Exp VNil ty* to *Exp ctx ty*), so we can omit the call to *shifts0*. This method is needed in the processing of user-defined globals, a feature Stitch supports, but a full description of which would distract from our main goal.

```
shifts0Exp :: ∀prefix ty. Exp VNil ty → Exp prefix ty
shifts0Exp = ...
  -- Short-circuit the no-op shifts0Exp:
{-# noinline shifts0Exp #-}
{-# rules shifts0Exp shifts0Exp = unsafeCoerce #-}
```

Figure 8. Shifting closed expressions should be trivial

See Figure 8, which defines *shifts0Exp*, the definition of *shifts0* in the *Shiftable* instance for *Exp*. This function must tiresomely walk the entire structure of its argument in order to do nothing. The problem is the change in type; the only way to convince GHC that no action needs to be taken is a full recursive traversal.

This is disappointing. We want our types to help prevent errors, not require extra runtime work. It is conceivable that a language with full dependent types would support a proof that *shifts0Exp* has no runtime effect, but this is still hard to imagine, given that the output of *shifts0Exp* has a different type than its input.

The fullness of GHC's feature set comes to the rescue here. GHC supports *rewrite rules* [48], which allow a programmer to provide arbitrary term rewriting rules that GHC applies during its optimization passes. These rules are type-checked to make sure both sides have the same type, but no checking is done for semantic consistency. It is just the ticket for us here: we can fix the types up with an *unsafeCoerce* and trust our by-hand analysis that *shifts0Exp* really does nothing at runtime. The **noinline** is necessary to force GHC not to inline the function, so that the rewrite rule can trigger.

Is this design a win or a loss? I am not sure. It surely has aspects of a loss because the compiler can not figure out that *shifts0Exp* is pointless. On the other hand, the workaround is very easy and fully effective. And, even in a language with a richer type system than GHC's Haskell, it is not clear we can do better.

9 Common-Subexpression Elimination

Having covered the basic necessities of an interpreter, we now explore an extension, as evidence that we can still implement non-trivial transformations over an indexed AST. Common-subexpression elimination is a standard optimization pass, which identifies expressions with common subexpressions, transforming these to use a `let`-bound variable instead. A full description of the CSE algorithm is unnecessary here but is well documented in the Stitch's `CSE` module; instead, we will focus on the (indexed) data structures used to power the CSE algorithm.

The key data structure needed for CSE is a finite map that uses expressions as keys. Using such a map, we can

```
class IHashable (t :: k → Type) where
  ihashWithSalt :: Int → t a → Int
instance TestEquality (Exp ctx) where ...
instance ... ⇒ IHashable (Exp (ctx :: Ctx n)) where ...
data IHashMap :: ∀k. (k → Type) → (k → Type) → Type
insert  :: (TestEquality k, IHashable k)
        ⇒ k i → v i → IHashMap k v → IHashMap k v
lookup  :: (TestEquality k, IHashable k)
        ⇒ k i → IHashMap k v → Maybe (v i)
map :: (∀i. v1 i → v2 i) → IHashMap k v1 → IHashMap k v2
type ExpMap ctx a = IHashMap (Exp ctx) a
```

Figure 9. Key definitions for indexed *HashMap*s

store what expressions we have seen so far in order to find duplicates, and we can map expressions to fresh `let`-bound variables. The challenge here is that we need to make sure an expression of type *ty* maps to a variable of type *ty*; failing to do so would lead the CSE algorithm not to pass GHC's type checker.

Naturally, we want the CSE algorithm to be reasonably efficient. Instead of creating our own mapping structure, we would like to use the existing optimized *HashMap* structure from the unordered-containers library, a widely-used containers implementation. However, a *HashMap* requires that all the keys in the map have the same type. This is usually a desired property, but not in our case here: the different keys will all be *Exp*s, but they may have different type indices. The solution is to alter *HashMap* to work with indexed types. To implement this idea, I took the source code from unordered-containers, made a few small changes to the types, and then simply fixed the errors that GHC reported. Some key definitions are in Figure 9.

9.1 Indexed Maps

Just as a traditional mapping structure must depend on a key's *Eq* instance, an indexed mapping structure must depend on a key's *TestEquality* instance. Our *Exp* type naturally is a member of the *TestEquality* class: if two expressions are equal (in a shared context *ctx*), their types are, too.

We also must generalize the *Hashable* class used for traditional *HashMap*s so that we can state that *Exp* has a hash, no matter its type. This is straightforward to do; see *IHashable*.

In the definition of *IHashMap*, we must index the map by the type constructors, not the concrete types. Note that in the definition for *ExpMap*, the key is *Exp ctx*, *not Exp ctx ty*. In this way, a map can contain expressions of many types. Accordingly, the *insert* and *lookup* functions work by applying the key type *k* and value type *v* to an index *i*. (Note: the *k* in the definition of *IHashMap* is the *k*ind of the index, not the *k*ey.) The magic here is that *IHashMap* is not itself indexed by *i*, so we can look up *k i*, for any *i*, in a *IHashMap k v*, retrieving (perhaps) a *v i*.

Though not used in CSE, I have included here the type of the *map* function. Its function argument must be polymorphic in the index *i*. This is because the function must work over all values stored in the map; these values, of course, may have different indices. With a higher-rank type, however, *map* (and other functions) are straightforward to adapt to the indexed setting.

9.2 Experience Report

The adaptation of *HashMap* into an indexed setting was shockingly easy. Once I had committed to adapting the existing implementation, it took me roughly 2 hours to update the 2.5k lines of code implementing lazy *HashMap*s and *HashSet*s. The process flowed as we all imagine typed refactoring should: I changed the datatype definitions and just followed the errors. It all worked splendidly once it compiled. I was aided by the fact that *TestEquality* is already exported from GHC's set of libraries and that this class has just the right shape for usage in a finite map structure.

Many functions, such as *map*, require higher-rank types. Interestingly, several class instance definitions also require a higher rank, but these require a higher-rank *constraint*, also known as a quantified constraint [6]. For example, here are the instance heads for two instances of *IHashMap*:

```
instance (TestEquality k, IHashable k
  , ∀i. Read (k i), ∀i. Read (v i)) ⇒ Read (IHashMap k v)
instance
  (∀i. Show (k i), ∀i. Show (v i)) ⇒ Show (IHashMap k v)
```

In order to parse the contents of a *IHashMap k v*, we need to be able to read elements of type *k i* and *v i*, for any *i*, and similarly for pretty-printing. With quantified constraints, we can express this fact directly, and type-checking proceeds without a hiccup.

The CSE implementation overall was also agreeably easy. While the design of the algorithm took some careful thought, working with indexed types was an aid to the process, not an obstacle. The way *Exp*'s indices track contexts, in particular, was critical, because any recursive algorithm over *Exp*s must occasionally change contexts; it would have been very easy to forget a shift or unshift during this process without GHC's type checker helping me get it right.

10 Discussion

10.1 **let** *Should* Sometimes be Generalized

Type inference in the presence of GADTs is hard [12, 50, 51, 64]. One of the confounding effects of GADTs is that GHC does not generalize local **let**-bound variables in a module with the `MonoLocalBinds` language flag enabled, which is implied by the GADTs extension [63].[11] However, this lack of generalization stymied my implementation.

In the adaptation of *HashMap* to *IHashMap*, it was necessary to make many traversal functions have higher-rank types, like *map* in Section 9.1. Other functions in the *HashMap* library use these traversals with locally defined helper functions, which generally lacked type signatures. However, because **let**s were not generalized in the module, the type of the **let**-bound function was not polymorphic enough to be used as the argument to the higher-rank traversal function. While adding the type signatures to the local functions was not terribly difficult, it was tedious, and I opted instead to specify NoMonoLocalBinds, to good effect.

10.2 Dependent Types

To my surprise, this project did *not* strongly want for full dependent types. As we have seen, we needed a few singletons. A language with support for dependent types would naturally not need these singletons. However, one of the real pain points for singletons—costly runtime conversions between singletons and unrefined types—arose in only one place: the calculation of what color is used to render a de Bruijn index. Another big pain point is code duplication, but that problem, too, was almost entirely absent from Stitch. Despite being the author of the singletons library [19] that automates working with them, I was not tempted to use it.

10.3 Type Errors and Editor Integration

One aspect in which GHC/Haskell lags behind other dependently typed languages is in its editor integration. Idris, for example, supports interactive type errors, allowing a user to explore typing contexts and other auxiliary information in reading an error [15]. Idris, Agda, and Coq all allow a programmer to focus on one goal at a time. The closest feature in GHC is its support for typed holes [22], where a programmer can replace an expression with an underscore and GHC will tell you the desired type of the expression and suggest type-correct replacements.

The extra features in other language systems would have been helpful, but their lack did not bite in this development. I used typed holes a few times, and I had to comment out code in order to focus on smaller sections, but these were not burdens. Type errors were often screen-filling, but it was easy enough to discern the key details without being overwhelmed. So, while I agree that GHC has room to improve in this regard, its current state is still quite usable.

10.4 Related Work

The basic idea embodied in Stitch is not new. Though written before the invention of indexed data types, Pfenning and Lee [54] consider an encoding of System F in a third-order polymorphic λ-calculus (F_3); only well-typed programs are representable. Their encoding is very much a foreshadowing of more recent papers. Perhaps the first elucidation of the technique of restricting evaluation only to well-typed ASTs is by Augustsson and Carlsson [4], who implemented their interpreter in Cayenne [3]. The idea was picked up by Pašalić et al. [43], who use a similar example to power the introduction of Meta-D, a language useful for writing indexed ASTs. Other work principally focusing on an indexed AST includes that by Chen and Xi [11], which includes an indexed CPS transform, implemented in ATS [70]. An implementation of this idea in Haskell is described by Guillemette and Monnier [24], who embed System F; their encoding is limited by the lack of, e.g., rich kinds in Haskell at the time, and their focus is more on compiler transformations than on type checking. More recently, an indexed AST has been encoded in Agda [1, 2]; the authors' focus in both works cited is in generating correct definitions and proofs without boilerplate. Kokke et al. [31] also uses this example; notable there is the way an executable interpreter is extracted from the type safety proof of the language. Going beyond just embedding the λ-calculus, Weirich [67] embeds a richly typed AST for regular expressions in Haskell. The indexed AST idea comes up, in passing or with focus, in many more works beyond these, both in the folklore and in published literature.

The real focus of this paper is not an indexed AST, however; it is to serve as a tutorial to the advanced features of Haskell. In this space, this paper's contribution is indeed novel: to my knowledge, this is the first formally peer-reviewed work aiming to teach these techniques. There is educational material in the folklore and posted online [26, 34]. A tutorial focusing on an indexed AST embedding in Idris [7] is part of that language's online documentation [62], and Benton et al. [5] use an indexed AST to explore intrinsic-verification features of Coq. In contrast to those materials, this paper is set in the context of a complete software artifact that is a practical tool for teaching the operation of the λ-calculus, with a user-oriented executable. The goal in doing so is to demonstrate that it is indeed possible to build relatively mundane software components, such as a REPL or parser, using fancy types in Haskell—a fact not necessarily yet appreciated by the broader programming language community.

10.5 Conclusion

I have presented Stitch, a simply typed λ-calculus interpreter, amenable for pedagogic use and implemented using an indexed AST. This paper has explored the implementation and described the features of Haskell that power the encoding and enable Stitch to be written. I have reported on Haskell's support for richly typed work such as Stitch, concluding that Haskell is ready for serious work with fancy types.

[11] More precisely, GHC does not generalize local **let**-bound variables whose right-hand side mentions a variable bound from an outer scope. In other words, if the local definition can be easily lifted out to top-level, GHC still *does* generalize it.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1704041. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A Type and Scope Safe Universe of Syntaxes with Binding: Their Semantics and Proofs. *Proc. ACM Program. Lang.* 2, ICFP, Article 90 (July 2018), 30 pages. https://doi.org/10.1145/3236785

[2] Guillaume Allais, James Chapman, Conor McBride, and James McKinna. 2017. Type-and-scope Safe Programs and Their Proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*. ACM, New York, NY, USA, 195–207. https://doi.org/10.1145/3018610.3018613

[3] Lennart Augustsson. 1998. Cayenne—a language with dependent types. In *Proc. ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. ACM, 239–250.

[4] Lennart Augustsson and Magnus Carlsson. 1999. An exercise in dependent types: A well-typed interpreter. (1999). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.39.2895&rep=rep1&type=pdf Unpublished manuscript.

[5] Nick Benton, Chung-Kil Hur, Andrew J. Kennedy, and Conor McBride. 2012. Strongly Typed Term Representations in Coq. *Journal of Automated Reasoning* 49, 2 (01 Aug 2012), 141–159. https://doi.org/10.1007/s10817-011-9219-0

[6] Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified Class Constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA, 148–161. https://doi.org/10.1145/3122955.3122967

[7] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Prog.* 23 (2013).

[8] Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2016. Safe Zero-cost Coercions for Haskell. *J. Funct. Program.* 26 (2016), 1–79.

[9] 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.

[10] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyon Jones. 2005. Associated Type Synonyms. In *International Conference on Functional Programming (ICFP '05)*. ACM.

[11] Chiyan Chen and Hongwei Xi. 2003. Implementing Typeful Program Transformations. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM '03)*. ACM, New York, NY, USA, 20–28. https://doi.org/10.1145/777388.777392

[12] Sheng Chen and Martin Erwig. 2016. Principal Type Inference for GADTs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 416–428. https://doi.org/10.1145/2837614.2837665

[13] James Cheney and Ralf Hinze. 2003. *First-Class Phantom Types*. Technical Report. Cornell University.

[14] Adam Chlipala. 2008. Parametric Higher-order Abstract Syntax for Mechanized Semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. ACM, New York, NY, USA, 143–156. https://doi.org/10.1145/1411204.1411226

[15] David Raymond Christiansen. 2015. A Pretty Printer that Says What it Means. Talk, Haskell Implementors Workshop, Vancouver, BC, Canada. https://www.youtube.com/watch?v=m7BBCcIDXSg

[16] Nicolaas Govert de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

[17] Richard A. Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice*. Ph.D. Dissertation. University of Pennsylvania.

[18] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *Principles of Programming Languages (POPL '14)*. ACM.

[19] Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently Typed Programming with Singletons. In *ACM SIGPLAN Haskell Symposium*.

[20] Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan Ahmed. 2016. Visible Type Application. In *European Symposium on Programming (ESOP) (LNCS)*. Springer-Verlag.

[21] Martin Erwig and Simon Peyton Jones. 2000. Pattern Guards and Transformational Patterns. In *Haskell Workshop 2000* (haskell workshop 2000 ed.). https://www.microsoft.com/en-us/research/publication/pattern-guards-and-transformational-patterns/

[22] Matthías Páll Gissurarson. 2018. *Suggesting Valid Hole Fits for Typed-Holes in Haskell*. Master's thesis. Chalmers University of Technology, University of Gothenburg. https://mpg.is/papers/gissurarson2018suggesting-msc.pdf

[23] Andrew D. Gordon. 1994. A mechanisation of name-carrying syntax up to alpha-conversion. In *Higher Order Logic Theorem Proving and Its Applications*, Jeffrey J. Joyce and Carl-Johan H. Seger (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 413–425.

[24] Louis-Julien Guillemette and Stefan Monnier. 2008. A type-preserving compiler in Haskell. In *Proc. 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. ACM, 75–86.

[25] Adam Gundry. 2013. *Type Inference, Haskell and Dependent Types*. Ph.D. Dissertation. University of Strathclyde.

[26] Hiromi Ishii. 2014. Dependent Types in Haskell. School of Haskell blog. https://www.schoolofhaskell.com/user/konn/prove-your-haskell-for-great-safety/dependent-types-in-haskell

[27] Mark P. Jones. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming*, Johan Jeuring and Erik Meijer (Eds.). LNCS, Vol. 925. Springer Verlag.

[28] Mark P. Jones. 2000. Type Classes with Functional Dependencies. In *European Symposium on Programming*.

[29] Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. 2015. GADTs meet their match. In *International Conference on Functional Programming (ICFP '15)*. ACM.

[30] Edward Kmett. 2012. bound. Haskell package. https://github.com/ekmett/bound/

[31] Wen Kokke, Jeremy G. Siek, and Philip Wadler. 2020. Programming Language Foundations in Agda. *Science of Computer Programming* 194 (2020), 102440. https://doi.org/10.1016/j.scico.2020.102440

[32] Ralf Lämmel and Simon Peyton Jones. 2003. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Workshop on Types in Languages Design and Implementation*. ACM.

[33] Ralf Lämmel and Simon Peyton Jones. 2005. Scrap Your Boilerplate With Class: Extensible Generic Functions. In *ICFP*.

[34] Justin Le. 2018. Introduction to Singletons. (2018). https://blog.jle.im/entry/introduction-to-singletons-3.html

[35] Daan Leijen. 2001. *Parsec: a fast combinator parser*. Technical Report UU-CS-2001-26. University of Utrecht.

[36] Sam Lindley and Conor McBride. 2013. Hasochism: the pleasure and pain of dependently typed Haskell programming. In *ACM SIGPLAN Haskell Symposium*.

[37] Andres Löh. 2012. lhs2TeX. Haskell package. https://www.andres-loeh.de/lhs2tex/

[38] José Pedro Magalhães. 2012. The Right Kind of Generic Programming. (2012). To appear at WGP.

[39] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. A generic deriving mechanism for Haskell. In *Proceedings of the 3rd ACM Haskell Symposium on Haskell (Haskell '10)*. ACM.

[40] Conor McBride. 2011. The Strathclyde Haskell Enhancement. https://personal.cis.strath.ac.uk/conor.mcbride/pub/she/.

[41] Stefan Monnier and David Haguenauer. 2010. Singleton types here, singleton types there, singleton types everywhere. In *Programming languages meets program verification (PLPV '10)*. ACM.

[42] Dominic Orchard and Tom Schrijvers. 2010. Haskell Type Constraints Unleashed. In *Functional and Logic Programming*, Matthias Blume, Naoki Kobayashi, and Germán Vidal (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 56–71.

[43] Emir Pašalić, Walid Taha, and Tim Sheard. 2002. Tagless Staged Interpreters for Typed Languages. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 218–229. https://doi.org/10.1145/581478.581499

[44] Simon Peyton Jones. 2003. Wearing the Hair Shirt: A Retrospective on Haskell. Invited talk at POPL.

[45] Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: an exploration of the design space. In *Haskell Workshop*, John Launchbury (Ed.). Amsterdam, Netherlands.

[46] Simon Peyton Jones and John Launchbury. 1991. Unboxed values as first class citizens. In *FPCA (LNCS)*, Vol. 523. 636–666.

[47] Simon Peyton Jones and Mark Shields. 2004. Lexically-scoped type variables. (2004). http://research.microsoft.com/en-us/um/people/simonpj/papers/scoped-tyvars/ Draft.

[48] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the Rules: Rewriting as a practical optimisation technique in GHC. In *Proceedings of the Haskell Workshop*.

[49] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming* 17, 1 (Jan. 2007).

[50] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming (ICFP '06)*. ACM.

[51] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. 2004. *Wobbly types: type inference for generalised algebraic data types*. Technical Report MS-CIS-05-26. University of Pennsylvania.

[52] Simon Peyton Jones, Stephanie Weirich, Richard A. Eisenberg, and Dimitrios Vytiniotis. 2016. A reflection on types. In *A list of successes that can change the world*. Springer. A festschrift in honor of Phil Wadler.

[53] Frank Pfenning and Conal Elliott. 1988. Higher-order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. ACM, New York, NY, USA, 199–208. https://doi.org/10.1145/53990.54010

[54] Frank Pfenning and Peter Lee. 1989. LEAP: A language with eval and polymorphism. In *TAPSOFT '89*, J. Díaz and F. Orejas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 345–359.

[55] Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern Synonyms. In *ACM SIGPLAN Haskell Symposium*. ACM.

[56] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA.

[57] Andrew M. Pitts. 2003. Nominal logic, a first order theory of names and binding. *Information and Computation* 186, 2 (2003), 165 – 193. https://doi.org/10.1016/S0890-5401(03)00138-X Theoretical Aspects of Computer Software (TACS 2001).

[58] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2010. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming* 20, 1 (Jan. 2010).

[59] Jan Stolarek, Simon Peyon Jones, and Richard A. Eisenberg. 2015. Injective Type Families for Haskell. In *Haskell Symposium (Haskell '15)*. ACM.

[60] Walid Taha, Henning Makholm, and John Hughes. 2001. Tag elimination and Jones-optimality. In *Programs as Data Objects*, Olivier Danvy and Andrzej Filinski (Eds.). LNCS, Vol. 2053. Springer Verlag.

[61] David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. 2012. Safe Haskell. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 137–148. https://doi.org/10.1145/2364506.2364524

[62] The Idris Team. 2017. Example: The Well-Typed Interpreter. The Idris Tutorial. http://docs.idris-lang.org/en/latest/tutorial/interp.html

[63] Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. 2010. Let Should Not Be Generalized. In *Types in Language Design and Implementation (TLDI '10)*. ACM.

[64] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X): Modular Type Inference with Local Assumptions. *Journal of Functional Programming* 21, 4-5 (Sept. 2011).

[65] Philip Wadler. 1987. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. Association for Computing Machinery, New York, NY, USA, 307–313. https://doi.org/10.1145/41625.41653

[66] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with Explicit Kind Equality. In *International Conference on Functional Programming (ICFP '13)*. ACM.

[67] Stephanie Weirich. 2017. The Influence of Dependent Types. Keynote, POPL '17. https://www.youtube.com/watch?v=rflCw9bT4_0

[68] Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP, Article 31 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110275

[69] Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. 2011. Binders Unbound. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 333–345. https://doi.org/10.1145/2034773.2034818

[70] Hongwei Xi. 2004. Applied Type System. In *Types for Proofs and Programs*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 394–408.

[71] Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *Principles of Programming Languages (POPL '03)*. ACM.

[72] Ningning Xie and Richard A. Eisenberg. 2018. Coercion Quantification. In *Haskell Implementors' Workshop*.

[73] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a promotion. In *Types in Language Design and Implementation (TLDI '12)*. ACM.