Stitch: The Sound Type-Indexed Type Checker (Author's Cut)

A Functional Pearl

RICHARD A. EISENBERG, Bryn Mawr College, USA

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, idiomatic Haskell and serves as an existence proof that Haskell's is advanced enough for the use of fancy types in a practical setting.

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 don't 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 [Cheney and Hinze 2003] or guarded recursive datatypes [Xi et al. 2003], exhibit one of the most basic ways to use fancy types. When you pattern-match on a GADT value, you learn information about the type of that value. Accordingly, different branches of a GADT pattern match have access to different typing information and can make effective use of that information. 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 [Eisenberg and Weirich 2012; Monnier and Haguenauer 2010].

It's high time for an example of what we're talking about:[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 does a GADT pattern-match on a value of type *G a*. If the value is *BoolCon*, then

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

Author's address: Richard A. Eisenberg, Computer Science Department, Bryn Mawr College, 101 N. Merion Ave, Bryn Mawr, PA, 19010, USA, rae@cs.brynmawr.edu.

2018. 2475-1421/2018/9-ART1 $15.00
https://doi.org/

we learn that *a* is in fact *Bool*; our function can thus return *True* :: *a*. In the other branch, the value of type *G a* is *IntCon*, and thus *a* must be *Int*; we can return 42 :: *Int*. The runtime pattern-match tells us the compile-time type, allowing the branches to have *different* types. In contrast, a simple pattern-match always 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. (I use "Stitch" to refer 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*. (Here, *ctx* is the typing context for any free variables in the expression.)

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 10.6). 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 some techniques for how they can apply this technology 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* a practical example. Many significant 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 8.4 (the Glasgow Haskell Compiler),[2] making critical use of GHC's recent support for using GADT constructors at the type level [Weirich et al. 2013; Yorgey et al. 2012], type reflection (i.e. *Typeable*) [Peyton Jones et al. 2016], higher-rank type inference [Peyton Jones et al. 2007], and, of course, GADT type inference [Peyton Jones et al. 2006; Vytiniotis et al. 2011]. 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 Highlights

While a functional pearl does not offer novel contributions, I list these highlights of this work:

- Stitch is a full executable interpreter of the STLC, suitable for classroom use.
- This paper is intended to be educational, including Section 3 that introduces fancy types to help the intermediate Haskeller.
- Fancy types are used liberally. 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.

[2]The development of Stitch revealed a few unremarkable bugs in GHC. Please use the HEAD build of the compiler; see https://ghc.haskell.org/trac/ghc/wiki/Building.

- The implementation is done in a practical, real-world language: Haskell. As an application using many modern features of the language, this pearl serves as an assessment of these features and declares that Haskell is an appropriate implementation language for a program with fancy types, despite the lack of support for full dependent types.

Stitch is available for download at http://cs.brynmawr.edu/~rae/papers/2018/stitch/stitch.tar.gz.

2 INTRODUCING STITCH

2.1 The Simply Typed λ-Calculus

Stitch is an implementation of the simply typed λ-calculus, so let's start off with a review of this little language, including the Stitch extensions. See Figure 1.

We see that Stitch is quite a standard implementation of the STLC with modest extensions. (Those unfamiliar with the STLC are recommended to consult Pierce [2002, Chapters 9 & 11] for an introduction.) 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. All λ-abstractions are annotated with the type of the argument.

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 two presentations 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 looping term, while the small-step semantics can show you the recurring steps the expression takes.

2.2 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, to bind new global variables, and to query aspects of an expression. An example is worth at least several hundred words here:

```
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
expr = (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
--> (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 REPL allows the user to create new global variables, like `expr`. These are unevaluated. The syntax `expr = ...` is called a *statement*, as included in Figure 1.

Metavariables:

$$\begin{array}{lll} x & & \text{term variables} \end{array}$$

Grammar:

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

Other notation:

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

$\mathrm{apply}(op, v_1, v_2)$ computes the result of using op with operands v_1 and v_2

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

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

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{T\_Var} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}\ \text{T\_Lam} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{T\_App}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\ \text{T\_Let} \qquad \frac{\Gamma \vdash e_1 : \textbf{Int} \qquad \Gamma \vdash e_2 : \textbf{Int}}{\Gamma \vdash e_1\ op\ e_2 : \mathrm{result}(op)}\ \text{T\_Arith}$$

$$\frac{\Gamma \vdash e_1 : \textbf{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau}\ \text{T\_Cond} \qquad \frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \textbf{fix}\ e : \tau}\ \text{T\_Fix}$$

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

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

$$\frac{}{v \Downarrow v}\ \text{E\_Value} \qquad \frac{\begin{array}{c} e_1 \Downarrow \lambda x{:}\tau.e \qquad e_2 \Downarrow v_2 \\ e[v_2/x] \Downarrow v \end{array}}{e_1\ e_2 \Downarrow v}\ \text{E\_App} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2[v_1/x] \Downarrow v}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Downarrow v}\ \text{E\_Let}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1\ op\ e_2 \Downarrow \mathrm{apply}(op, v_1, v_2)}\ \text{E\_Arith} \qquad \frac{e \Downarrow \lambda x{:}\tau.e' \qquad e'[\textbf{fix}\ (\lambda x{:}\tau.e')/x] \Downarrow v}{\textbf{fix}\ e \Downarrow v}\ \text{E\_Fix}$$

$$\frac{e_1 \Downarrow \textbf{true} \qquad e_2 \Downarrow v}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \Downarrow v}\ \text{E\_IfTrue} \qquad \frac{e_1 \Downarrow \textbf{false} \qquad e_3 \Downarrow v}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \Downarrow v}\ \text{E\_IfFalse}$$

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

We can then input the global by itself or as part of a larger expression to evaluate it. However, the most distinctive aspect of this session is Stitch's approach to variable binding, which we explore next.

2.3 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 [Pitts 2003], locally nameless binders [Gordon 1994], using higher-order abstract syntax [Pfenning and Elliott 1988], *parametric* higher-order abstract syntax [Chlipala 2008], Unbound [Weirich et al. 2011], bound[3], among others. The interested reader is referred to Weirich et al. [2011], where even more possibilities lie in wait. In this work, however, I choose trusty, old de Bruijn indices [de Bruijn 1972].

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 do not need to 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 [2002, Section 5.3] gives an instructive account of the pitfalls an implementor encounters. And it’s 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 recently (January, 2016) added checks to make sure its handling of variable naming is bug-free; I count 29 call sites within the GHC source code (as of February, 2018) 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

[3] http://hackage.haskell.org/package/bound

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
```

Fig. 2. A primality checker in Stitch.

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 is left as future work.

2.4 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`, included in the Stitch tarball, can be loaded into the Stitch REPL with `:load prime.stitch`.

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

2.5 An overview of Stitch

Before we get mired in the details, let's review the overall architecture of the Stitch interpreter. Throughout the rest of this paper, I will refer to individual modules in the package; these references are intended to help the reader who is following along in the actual codebase, though the text of this paper is self-contained and does not require doing so. The map of modules is in Figure 3.

A Stitch program travels through the interpreter in the usual fashion. The REPL module defines an interactive prompt which reads a string from the user. This string is then lexed into a series of

Modules that principally define datatypes:

- `Type`: Stitch types, (§4)
- `Op`: Binary operators (§6.3)
- `Token`: Lexer tokens
- `Unchecked`: The AST for parsed, but not type checked, expressions (§5)
- `Exp`: Expressions AST (§6)
- `Globals`: Global variables (§7.2)
- `Statement`: Statements (§2.2)

Modules that principally define algorithms:

- `Lex`: Lexer
- `Parse`: Parser (§5)
- `Check`: Type checker (§7)
- `Shift`: de Bruijn index shifting (§8.2)
- `Eval`: Operational semantics (§8)
- `CSE`: Common-subexpression elimination (§9)
- `Pretty`: Pretty-printing
- `Repl`: The user-facing REPL (§2.2)

Fig. 3. Principal modules in Stitch. All module names are prefixed with `Language.Stitch`.

tokens and then parsed into an expression AST that is *not* checked for type safety (defined in the `Unchecked` module). This expression type is then run through the type checker to be transformed into the checked AST (defined in `Exp`). The checked AST is optionally optimized (by performing CSE) and then evaluated according to the semantics the user chooses. A pretty-printer [Wadler 2003] renders the result back to the user.

We're 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's 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), existentials (useful for storing the values of global variables, perhaps of different types), and singletons (useful during type checking, when we must connect a type-level context with term-level type representations).

3.1 Natural numbers

The `Data.Nat` module defines routine Peano unary natural numbers:

```
data Nat = Zero | Succ Nat
```

This datatype is used in Stitch solely in types, via Haskell's datatype promotion mechanism [Yorgey et al. 2012]. For the last several years, GHC has allowed programmers to use data constructors (*Zero* and *Succ* in this case) in types; correspondingly, *Nat* is not only a type classifying terms, but also a kind classifying types. Indeed, recent improvements in GHC have eliminated the distinction between types and kinds [Weirich et al. 2013], and I have come to view the usage of *Zero* and *Succ* in types more as a namespace issue (Haskell maintains separate "type-level" and "term-level" namespaces) than as promotion, per se. We will soon see an example of these type-level constructors in action (§3.2). Because *Nat* is used solely in types, the inefficiency of storying a unary number does not bite at runtime, slowing down only the compilation process of the Stitch interpreter, not the compiled executable.

One might ask: Why use unary *Nat*s instead of GHC's built-in support for type-level natural numbers?[4] Unary naturals have an inherent inductive structure, making for easy definitions and proofs. While GHC cannot know, say, that $n + m$ is the same as $m + n$, the type-level arithmetic used in Stitch is quite simple and no arithmetic reasoning is necessary. In my experience, these hand-written unary naturals work better than the built-in naturals for defining vectors.

3.2 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 Vec :: Type → Nat → Type where
  VNil :: Vec a Zero
  (:>) :: a → Vec a n → Vec a (Succ n)
```

Let's take this line-by-line. We see here 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's 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*.[5] Finding these constructors, GHC has no trouble using them in types, where they keep their usual meaning.

3.2.1 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's unusual here is that we need to use + in *types*, not in terms. GHC's approach here is to use a *type family* [Chakravarty et al. 2005; Eisenberg et al. 2014], which is essentially a function that works on types and type-level data. Here is the definition:

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

We are now ready to define appending two vectors:

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

[4] https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#type-level-literals

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

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.

3.2.2 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 and declared in `Data.Fin`, 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 (where I take liberty to use decimal notation instead of unary notation for *Nat*s), but not *Fin* 1. Let's understand this type better by tracing how we can assign a type to *FS FZ*:

- Suppose we're 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're 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    -- reverse order due to laziness
  (FZ,   x :> _)  → x
  (FS n, _ :> xs) → xs !!! n
```

GHC comes with a pattern-match completeness checker [Karachalias et al. 2015] 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's enough to be complete.

Now, let's explore this match reversal. Haskell is a lazy language [Peyton Jones 2003], 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*. Let's 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*.

3.3 Existentials

Suppose we want a ragged two-dimensional vector. We might be tempted to use *Vec* (*Vec a n*) *m*, but this type requires that all inner vectors have length *n*, going against our desire for a ragged collection. Of course, we could use lists, but let's stick with *Vec* for the sake of example—we won't have the easy escape of lists when we encounter this problem later.

What we want is a way to hide the *n* index from the type of a vector; we want a collection of vectors where every vector has *some* length, but not necessarily the same one. This is what an *existential type* does: it essentially hides a type index, allowing us to recover it only through pattern matching. Here is the quintessential existential type, defined in `Data.Exists`:

```
data Ex :: (k → Type) → Type where
  Ex :: a i → Ex a
```

The *Ex* type is parameterized over the indexed type constructor *a* of the data it holds; the index itself can be of any kind *k*. Thus, *a* has kind *k* → *Type*. The *Ex* data constructor takes one argument of type *a i* for any *i*—note that *i* is *not* mentioned in the return type *Ex a*. This makes *i existentially bound*.

Let's understand this better through an example:

```
exVecSum :: Ex (Vec Int) → Int
exVecSum (Ex v) = go v
  where go :: Vec Int n → Int
        go VNil      = 0
        go (x :> xs) = x + go xs
```

The pattern match in *exVecSum* unpacks the existential to reveal a vector *v*. Naturally, *v* has type *Vec* and stores *Int*s; but, what is *v*'s length index? It is impossible to know: there exists a length, but we do not know it. Essentially, the length index is stored by the *Ex* constructor along with *v*. When we pattern-match against the *Ex* constructor, we get both the index and the term. When we call the *go* helper method, the type of that method is instantiated to the unknown (and unnamed) index and executes as expected.

Now that we have *Ex*, we can make our ragged two-dimensional vector type: *Vec* (*Ex* (*Vec a*)) *m*. We know a value of this type has *m* rows, but each row has a different (and unknown) length.

3.4 Singletons

The technique of *singletons* is a well worn and well studied [Monnier and Haguenauer 2010] way to simulate dependent types in a non-dependent language. Though at least two libraries exist for automatically generating singletons in Haskell [Eisenberg and Weirich 2012; McBride 2011], 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 a version of *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 precisely 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.

The `Data.Singletons` module contains several more definitions in order to support polymorphic singletons. A full treatment of these definitions would take us too far afield, and the approach roughly mimics that taken by Eisenberg and Weirich [2012]. In this text, I avoid using these definitions; readers following along in the actual implementation may notice a few insignificant differences in the use of singletons, but these are inessential for our topics of interest.

```
type Ty = Ex (TypeRep :: Type → Type)
pattern Ty t = Ex t
{-# complete Ty #-}
  -- Decompose a function type
pattern (:→) :: () ⇒ (fun ∼ (arg → res)) ⇒ TypeRep arg → TypeRep res → TypeRep fun
isTypeRep :: ∀a b. Typeable a ⇒ TypeRep b → Maybe (a :≈: b)
isTypeRep = eqTypeRep (typeRep @a)
```

Fig. 4. Some definitions supporting Stitch types.

Singletons are not the final word for dependent types in Haskell. They can be unwieldy [Lindley and McBride 2013] and the conversions are potentially costly at runtime. Work is under way [Eisenberg 2016; Gundry 2013] to add full dependent types to Haskell. However, for our present purposes, the singletons work quite nicely, and their drawbacks do not get in our way.

4 A STITCH TYPE IS A HASKELL TYPE

An early choice in designing an interpreter for a typed language is how one will represent types. The Stitch language's type system is very simple, as portrayed in Figure 1: it contains *Int*s, *Bool*s, and functions among these. Conveniently, the Haskell type system also contains these types, and GHC's type reflection mechanism [Peyton Jones et al. 2016] allows a programmer access to type representations.

A key aspect of GHC's reflection mechanism is that it provides a *type-indexed* type representation, *TypeRep*. The type *TypeRep* has kind $\forall k.\ k \rightarrow Type$, allowing for a representation of a type of any kind. The representation for *Int* has type *TypeRep Int*; the representation for *Bool* has type *TypeRep Bool*. As such, *TypeRep* is actually the singleton type for the kind *Type*.[6] GHC also provides a number of facilities for inspecting and building type representations, exported through its `Type.Reflection` module. By using *TypeRep* to represent Stitch types, we hook into the existing mechanism for efficient comparison of types, generation of hashes (used in Section 9), and singleton support. An excerpt of Stitch's `Type` module appears in Figure 4.

Along with re-exporting *Type* itself, the module defines *Ty*, a type synonym for an existential package (Section 3.3) containing a *TypeRep*. The *Ty* type is used when we wish to refer to a type without doing any compile-time reasoning—for example, in the unchecked, parsed expression AST (Section 5). In order to make usage of *Ty* easier throughout Stitch, a pattern synonym [Pickering et al. 2016] is introduced. This pattern synonym, also named *Ty* (but in the term-level namespace), comes with a {-# **complete** *Ty* #-} pragma; this compiler directive instructs GHC that the *Ty* pattern, all by itself, is a complete pattern match against the *Ty* type. This pragma silences pattern-match completeness warnings, which do not yet work with pattern synonyms.

4.1 Decomposing functions

Next, we see the definition of the :→ pattern synonym, which allows for decomposition of function types. For example, if we want to check whether *fun* :: *TypeRep ty* is a function type, we could say

[6] *TypeRep* can be viewed as a universal singleton type, because it works at all kinds. However, working with *TypeRep*s for non-*Type* singletons is even more unwieldy than singletons usually are, and so I use *TypeRep* only at kind *Type* → *Type* and write custom singleton types for other singletons.

```
case fun of arg :→ res → ...
            _other     → ...
```

A careful reader will note the unusual type assigned to the pattern :→, with *two* constraints offset by ⇒. (The first is empty, ().) While a full explanation of pattern synonym types would be a digression—and Pickering et al. [2016, Section 6] gives an accessible introduction with many examples—suffice it to say that this type indicates that a successful pattern match tells you that the scrutinee's type index (denoted with *fun* in the type signature) will be refined to *arg* → *res* in the body of the match. This is exactly what we will need in the type checker.

4.2 Comparing *TypeRep*s using propositional equality

Following :→ is *isTypeRep*, a convenient way to check whether a *TypeRep* matches a desired type. For example, this is used in the type checker when checking to see that the condition in an **if** is indeed of type *Bool*. If we are checking *rep* :: *TypeRep b*, then we would query *isTypeRep* @*Bool rep*. The @*Bool* argument is a *visible type application* [Eisenberg et al. 2016], which allows a caller of *isTypeRep* to choose the instantiation for the type variable *a*. Note that the signature for *isTypeRep* lists *a* first, meaning that the first usage of a visible type application would instantiate *a*. The body of *isTypeRep* also uses visible type application to extract an explicit *TypeRep* from the implicit *Typeable*, where we have *typeRep* :: *Typeable a* ⇒ *TypeRep a*.

Curiouser still is the return type of *isTypeRep*, *Maybe* (*a* :≈: *b*). The type :≈: is exported from GHC's `Data.Type.Equality` and has this definition:

```
data (a :: k1) :≈: (b :: k2) where HRefl :: a :≈: a
```

The type :≈: is *heterogeneous propositional equality*. It is heterogeneous because the two types related might not have the same kind. It is propositional because we must match against a value in *a* :≈: *b* (that is, *HRefl*) to convince the type checker that *a* is, in fact, the same as *b*. If *a* :: k_1 and *b* :: k_2, then matching something of type *a* :≈: *b* against *HRefl* convinces the type checker that *a* equals *b and* k_1 equals k_2 through the usual behavior of GADT pattern-matching.

This is the appropriate return type provided by GHC's *eqTypeRep* :: *TypeRep a* → *TypeRep b* → *Maybe* (*a* :≈: *b*), and therefore Stitch's *isTypeRep*. The *eqTypeRep* function is used to compare two type representations. If they are in fact equal, then it is often necessary to reflect this equality back to the type checker. Here is an example:

```
castTo :: ∀a b. Typeable a ⇒ a → TypeRep b → Maybe b
castTo x repB = case isTypeRep @a repB of
  Just HRefl → Just x
  Nothing    → Nothing
```

The idea here is that we have a value *x* of type *a*, but we wish for it to have some other type *b*. We also have the type representations of both; *a* is implicit (*Typeable*) while *b* is explicit (*TypeRep*). If the type representations are equal—that is, if we can discover at runtime that both *a* and *b* are, in fact, the same—then we can return *x* at type *b*. In the *Just* case, we match against *HRefl*, a proof that *a* equals *b*. This then allows GHC to accept *Just x* as having the return type of *Maybe b*. Without the match against *HRefl*, *Just x* :: *Maybe b* would be rejected.

The *eqTypeRep* function must use *heterogeneous* equality (instead of the homogeneous version :∼:, which is otherwise similar) because *TypeRep* is polykinded: we might be comparing types of different kinds. Not only do we need to know the types equal, but we need to know the kinds equal as well. This heterogeneous equality is available in GHC only since version 8.0, powered by recent advances in the theory [Weirich et al. 2013].

```
  -- Unchecked expression, indexed by the number of variables in scope
data UExp (n :: Nat) = UVar (Fin n)    -- de Bruijn index for a variable
                     | UGlobal String
                     | ULam Ty (UExp (Succ n))
                     | UApp (UExp n) (UExp n)
                     | UArith (UExp n) UArithOp (UExp n)
                     | UIntE Int
                       ...
  -- An encoding of (\x:Int. x + 1) 5, as an example
uexample :: UExp Zero    -- Zero because the expression is closed
uexample = UApp (ULam (Ty (typeRep @Int)) (UArith (UVar FZ) (UArithOp Plus) (UIntE 1)))
                (UIntE 5)
```

Fig. 5. The AST for parsed expressions, from the Unchecked module.

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 appears in Figure 5.

The type *UExp* ("unchecked expression") is indexed by a *Nat* that denotes the number of local variables in scope in the expression. 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 constructs.

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.

You will see in the definition of *UExp* a few other small details:

- Occurrences of global variables are stored as strings. These will then be interpreted during type-checking to inline the stored value of the global.
- Lambda-abstractions store a *Ty*—the existential wrapper around *TypeRep*—to denote the argument type of the function. 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 *UArith* constructor stores a *UArithOp*, which is an existential wrapper around the indexed *ArithOp* type, explored in more depth in Section 6.3.

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 [Leijen 2001]. 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 [Jones 1995] 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*); 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 *MonadReader* 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. For the purposes of 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 = ...
```

```
type Ctx n = Vec Type n
data Exp :: ∀n. Ctx n → Type → Type where
   Var  :: Elem ctx ty → Exp ctx ty
   Lam  :: TypeRep arg → Exp (arg :> ctx) res → Exp ctx (arg → res)
   App  :: Exp ctx (arg → res) → Exp ctx arg → Exp ctx res
   Arith :: Exp ctx Int → ArithOp ty → Exp ctx Int → Exp ctx ty
   IntE  :: Int → Exp ctx Int
   ...
  -- An encoding of (\x:Int. x + 1) 5, as an example
example ::  Exp VNil Int
example = App (Lam (typeRep @Int) (Arith (Var EZ) Plus (IntE 1))) (IntE 5)
```

Fig. 6. The type-indexed *Exp* expression AST

```
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's 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.

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 6. 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 type of kind *Type*.

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. Let's take function application as an example. The T_App rule has two premises: one gives expression e_1 type $\tau_1 \rightarrow \tau_2$, 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 *Int*, regardless of the context.

It is for this reason that 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.

6.1 The *Elem* type and type-indexed de Bruijn indices

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 *Type*s) 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.

6.2 *Lam* requires the indexed *TypeRep*

Note the *Lam* constructor for building λ-abstractions. The first argument is *TypeRep arg*. This argument contains both a runtime type representation, 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 *TypeRep* singleton works well here.

It may be interesting to note that this *TypeRep* argument was actually not required in an early (but fully working) version of Stitch. Lacking the *TypeRep* meant that the pretty-printer was unable to annotate type-checked λ-expressions, but that was the only drawback. The *arg* type index was (and still is) an existential type, packed by the *Lam* constructor. Because the choice of *arg* was never needed at runtime, no runtime witness was necessary. The addition of *TypeRep* was forced, however, when implementing common-subexpression elimination, as the argument is necessary in order to write *Exp*'s *TestEquality* instance. See Section 9.1.

6.3 Arithmetic operators

The *Arith* constructor contains two subexpressions and the choice of arithmetic operator. All binary operators in Stitch operate on two *Int*s, so the subexpressions are constrained each to have type *Int*. The return type, on the other hand, varies with the operator. For example, + produces an *Int* while < produces a *Bool*. We thus need another indexed type, *ArithOp*, indexed by the return type of the operation.[7] The definition appears in Figure 7.

Given the introduction above, this definition should be very unsurprising. Additionally, Figure 7 includes definitions for *UArithOp*, the unindexed variant of *ArithOp*, used before type checking. A *UArithOp* must store the singleton associated with the existentially bound type index so that the Stitch type checker can compare this type with the expected type of an expression.

[7] It would be easy to generalize this also to be indexed by argument types, if they varied among operators.

```
data ArithOp ty where
  Plus, Minus, Times, Divide, Mod       :: ArithOp Int
  Less, LessE, Greater, GreaterE, Equals :: ArithOp Bool

  -- Like Ex, but includes a Typeable constraint for the existentially bound index
  -- This is declared in the Data.Exists module with the Ex type
data TypeableEx :: (k → Type) → Type where
  TypeableEx :: Typeable i ⇒ a i → TypeableEx a

  -- UArithOp ("unchecked ArithOp") is an existential package for an ArithOp
type UArithOp = TypeableEx ArithOp

pattern UArithOp op = TypeableEx op
{-# complete UArithOp #-}
```

Fig. 7. Arithmetic operators, from the Op module.

7 THE SOUND TYPE-INDEXED TYPE CHECKER

We're ready now for the part we've all been waiting for: the sound type-indexed type checker. Many cases appear in Figure 8; these cases illustrate the points of interest.

At its core, the *check* function takes an unchecked expression of type *UExp* and converts it into a checked expression of type *Exp*. Already we see an unexpected twist in the type of *check*: it is written in continuation-passing style (CPS). The reason for this is that there is naturally no way to know what indices should be placed on the output *Exp*. What we'd like to write, ideally, is *check* :: *UExp Zero* → ∃*ty*. *Exp VNil ty* (ignoring the monadic context). However, Haskell does not support such a convenient construct. While we could use the *Ex* existential package here quite profitably, I found that CPS was easier and made for code with a better flow. With CPS, we can pass the type index *t* to the continuation using a higher-rank type for *check*. We also must pass *TypeRep t* to the continuation, so that runtime comparisons can be performed.

The *check* function works over closed expressions, as we'll 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 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
  (:%>) :: TypeRep t → SCtx ts → SCtx (t :> ts)
```

An *SCtx* operates analogously to a *SNat*, forcing the runtime value to match exactly the compile-time type index.

The other small curiosity in the type of *go* is that it adds a *SNatI n* constraint, where *n* is the length of the typing context. This constraint is not needed for type checking but instead is needed only for pretty-printing. In the text produced for type errors (elided here), we often want to print parts of expressions. Recall that the pretty-printer colors the de Bruijn indices in the output to indicate the indices' provenance (i.e., which binder they refer to). While the numeral to output can be read directly from the *Fin* or *Elem* datatype, the color cannot—the color is computed by

```
check :: (MonadError Doc m, MonadReader Globals m)
      ⇒ UExp Zero → (∀(t :: Type). TypeRep t → Exp VNil t → m r) → m r
check = go SCNil
  where
  go :: (MonadError Doc m, MonadReader Globals m, SNatI n)
     ⇒ SCtx (ctx :: Ctx n) → UExp n → (∀t. TypeRep 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. TypeRep 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 _ (UGlobal n) k = do globals ← ask
                          lookupGlobal globals n $ λty exp →
                            k ty (shifts0 exp)
  go ctx (ULam (Ty arg_ty) body) k = go (arg_ty :%> ctx) body $ λres_ty body' →
                                     k (arg_ty :→ res_ty) (Lam arg_ty 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 HRefl ← eqTypeRep arg_ty arg_ty'
                                             → k res_ty (App e1' e2')
                                           _ → typeError e ...
  go ctx e@(UArith e1 (UArithOp op) e2) k = go ctx e1 $ λty1 e1' →
                                            go ctx e2 $ λty2 e2' →
                                            case (isTypeRep @Int ty1, isTypeRep @Int ty2) of
                                              (Just HRefl, Just HRefl)
                                                → k typeRep (Arith e1' op e2')
                                              _ → typeError e ...
  go _ (UIntE n) k = k typeRep (IntE n)
```

Fig. 8. The sound type-indexed type checker (excerpts)

subtracting the value of the de Bruijn index from the number of in-scope variables. For example, suppose the second bound variable is rendered in purple (as it is in the examples in Section 2.2). When two variables are in scope, index 0 should be purple. But if three variables are in scope, index 1 should be purple: the invariant here is that the index two less than the number of in-scope variables is purple. Accordingly, the pretty-printer needs to know the number of in-scope variables at runtime. This number is the type index n, and thus we need the singleton for n; in this case, it is convenient to pass it implicitly, leading to the *SNatI n* constraint.

```
newtype Globals = Globals (M.Map String (TypeableEx (Exp VNil)))
lookupGlobal :: MonadError Doc m
             ⇒ Globals → String → (∀ty. TypeRep ty → Exp VNil ty → m r) → m r
lookupGlobal (Globals globals) var k = case M.lookup var globals of
                                          Just exp → unpackTypeRepEx exp k
                                          Nothing → throwError ...

  -- From Data.Exists; unpacks a TypeableEx, providing an explicit TypeRep
unpackTypeRepEx :: TypeableEx a → (∀i. TypeRep i → a i → r) → r
unpackTypeRepEx (TypeableEx x) k = k typeRep x
```

Fig. 9. Storing and retrieving global variables; module *M* refers to *Data.Map* from the containers package

7.1 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 a singleton context. When *check_var* finds the type it's 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,[8] 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.

7.2 Inlining globals

Stitch allows its users to declare global variables in the REPL, as demonstrated in Section 2.2. Expressions to be stored in globals are parsed and type-checked, with the type-checked *Exp* stored for later retrieval. Of course, a global can have any type, and so the data structure used to store the globals must use an existential. All globals are closed, and so we already know that the context must be empty. The definition of the *Globals* datatype appears in Figure 9.

Globals is a newtype wrapping a finite map from strings (global variable names) to existential-packed expressions. We pack these expressions along with a *Typeable* constraint containing the expressions' types, for retrieval during type checking. As the type checking algorithm uses an explicit *TypeRep* for types, we use the *unpackTypeRepEx* function, which unpacks a *TypeableEx* existential package, converting the implicit *Typeable* type representation to an explicit *TypeRep*. (Recall that the *typeRep* function used in *unpackTypeRepEx* has type *Typeable a* ⇒ *TypeRep a*.)

A further complication arises in the fact that we inline the value of a global variable into an expression with potentially a non-empty context. Globals have an empty context, and so we must be careful to shift de Bruijn indices when inlining the global. I defer discussion of the *shifts0* function until we have talked about evaluation, where de Bruijn shifting is more at home. See Section 8.3. Regardless of the details, however, we can see already that the strongly typed discipline within Stitch prevents us from forgetting about this shifting: the continuation in *go* expects a *Exp ctx t*, where the context is provided as a parameter to *go*. That is, the context is known ahead of time. Since *lookupGlobal* passes an expression in an empty context to its continuation, that expression cannot be directly passed to the continuation of *go*: GHC would issue an error saying that it cannot

[8]Note that we match the *Fin* before the vector, as we did in Section 3.2.2.

prove that *ctx* is *VNil*. This error is spot on, pointing out that we've confused an expression in an empty context with one in a potentially non-empty one, necessitating shifting.

7.3 Checking a λ-abstraction

The *Lam* case is remarkably straightforward. 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 *arg_ty* (as unpacked from the *Ty* stored by the *ULam* constructor) and *res_ty*, using our :→ pattern synonym (Section 4.1). Note that if we did not store the *arg_ty* indexed *TypeRep* in the *ULam*, we would be stuck here.

7.4 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 matching against the :→ pattern synonym. We then must ensure that the actual argument type *arg_ty* matches the function's expected argument type *arg_ty′*. We use the *eqTypeRep* function, exported from GHC's `Type.Reflection` module and explained in Section 4.2. 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 is working hard 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.

7.5 Arithmetic expressions

Arithmetic expressions are straightforward to check, following broadly the pattern we saw in the function application case: simply check all the *TypeRep*s. We make use here of the *isTypeRep* function we defined in Section 4.2 to check that both arguments are indeed *Int*s. Upon success, we can retrieve the result type of the expression by using *typeRep*; recall that the *UArithOp* type (Section 6.3) stores a *Typeable* constraint for the operation type via its definition in terms of *TypeableEx*. Type inference figures out that the use of *typeRep* here should correspond to the result type of *Arith*, in turn set by the use of *op* as an argument to *Arith*.

We conclude with the case for integer literals. In the call of the continuation, we can once again use *typeRep*, as the use of *IntE* tells us we need the representation for the type *Int*.

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—that 's the whole point of using an indexed expression AST—and GHC's type inference allows us the convenience to pass type representations implicitly. 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 at all, despite the fact that this technique is not new [e.g., Pašalić et al. 2002]. After all, how could you guarantee that expressions are well typed? The answer is, of course, by checking them, as *check* does for us here.

```
data ValuePair ty = ValuePair {expr :: Exp VNil ty, val :: Value ty}
eval :: Exp VNil t → ValuePair t
eval (Var v)            = impossibleVar v
eval e@(Lam _ body) = ValuePair e $ λarg → subst arg body
eval (App e₁ e₂)        = eval (apply (eval e₁) (eval e₂))
eval (Arith e₁ op e₂)   = eval (arith (val $ eval e₁) op (val $ eval e₂))
eval e@(IntE n)         = ValuePair e n
...
impossibleVar :: Elem VNil x → a
impossibleVar = λcase { }
```

Fig. 10. Implementation of big-step operational semantics

8 EVALUATION WITH AN INDEXED AST

Writing evaluators is where the indexed AST really shines: we essentially can't 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. 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 Int     = Int
  Value Bool    = 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 [Carette et al. 2009; Pašalić et al. 2002; Taha et al. 2001], and we will not explore this aspect of Stitch further.

The two evaluators for Stitch are straightforward transcriptions of Stitch's operational semantics (Figure 1). There is only one small hitch: encoding values. We sometimes need to translate a value back into an expression—for example, when we substitute that value in for a variable during β-reduction. We thus define a type *ValuePair* :: *Type* → *Type* that stores closed expressions along with the untagged values. As there is only one constructor for the *ValuePair* type, its tag need not be checked at runtime. Its definition, along with the big-step evaluator, appear in Figure 10.

The helper functions *apply* and *arith* are routine and elided. Note, however, the *impossibleVar* function, which eliminates the possibility of encountering a variable in an empty context. It is implemented via an empty **case** expression. Empty **case** expressions are strict in Haskell, in contrast to non-empty **case**s. When the *Elem VNil x* is evaluated, it must be *ES* or *EZ*, both of which cannot be indexed by an empty context. GHC thus discovers that *Elem VNil x* is an empty type, and the empty **case** is accepted as a complete pattern match.

```
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 (locals :: Ctx n) → Exp (locals ++ s :> ctx) t0 → Exp (locals ++ ctx) t0
    go len (Var v)          = subst_var len v
    go len (Lam ty body) = Lam ty (go (LS len) body)
    ...   -- other forms are treated homomorphically
    subst_var :: Length (locals :: Ctx n) → Elem (locals ++ s :> ctx) t0 → Exp (locals ++ ctx) t0
    subst_var LZ        EZ       = e                          -- no locals; substitute
    subst_var LZ        (ES v) = Var v                      -- no locals; decrement index
    subst_var (LS _)  EZ       = Var EZ                    -- variable is local; no change
    subst_var (LS len) (ES v) = shift (subst_var len v)   -- recur
```

Fig. 11. Indexed substitution, from the `Eval` module

8.1 Substitution

We are left to discuss the bane of implementors using de Bruijn indices: substitution. 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 11.

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' *locals* 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 locals*. The *Length* type is included in Figure 11; values are Peano naturals that describe the length of a vector.[9] 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 *subst_var* function when to substitute, when to shift, and when to leave well enough alone. Pierce [2002, Chapter 6] offers an accessible introduction to the delicate operation of substitution in the presence of de Bruijn indices, and length concerns prohibit me from adequately explaining the subtleties here; suffice it to say that any misstep in *subst_var* would be caught by GHC's type checker.

[9] 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 → Type → Type) where
  shifts   :: Length prefix → a ctx ty → a (prefix ++ ctx) ty   -- multishifts are needed in CSE
  shifts0  :: a VNil ty → a prefix ty
  unshifts :: Length prefix → a (prefix ++ ctx) ty → Maybe (a ctx ty)   -- needed for CSE
instance Shiftable Exp where
  shifts   = shiftsExp
  shifts0  = shifts0Exp    -- see Section 8.3
  unshifts = unshiftsExp   -- elided
instance Shiftable Elem where ...
  -- Convenient abbreviation for the common case of shifting by only one index
shift :: ∀(a :: ∀n. Ctx n → Type → Type) ctx t ty. Shiftable a ⇒ a ctx ty → a (t :> ctx) ty
shift = shifts (LS LZ)
shiftsExp :: ∀prefix ctx ty. Length prefix → Exp ctx ty → Exp (prefix ++ ctx) ty
shiftsExp prefix = go LZ
  where
  go :: Length (locals :: Ctx n) → Exp (locals ++ ctx) ty0 → Exp (locals ++ prefix ++ ctx) ty0
  go len (Var v)          = Var (shifts_var len v)
  go len (Lam ty body) = Lam ty (go (LS len) body)
   ...   -- other forms are treated homomorphically
  shifts_var :: Length (locs :: Ctx n) → Elem (locs ++ ctx) ty0 → Elem (locs ++ prefix ++ ctx) ty0
  shifts_var LZ       v      = weakenElem prefix v
  shifts_var (LS _) EZ     = EZ
  shifts_var (LS l)  (ES e) = ES (shifts_var l e)
  -- Weaken an Elem to work against a larger vector.
weakenElem :: Length prefix → Elem xs x → Elem (prefix ++ xs) x
weakenElem LZ        e = e
weakenElem (LS len) e = ES (weakenElem len e)
```

Fig. 12. De Bruijn index shifting, from the `Shift` module

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 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*.[10] 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've 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

[10]In a call-by-value λ-calculus, this shifting will never affect a substituted expression, as all such expressions are closed. However, the definition of substitution is general and must take this shifting into account.

```
shifts0Exp :: ∀prefix ty. Exp VNil ty → Exp prefix ty
shifts0Exp = go LZ
  where
    go :: Length (locals :: Ctx n) → Exp locals ty0 → Exp (locals +++ prefix) ty0
    go len (Var v)          = Var (shifts0_var v len)
    go len (Lam ty body) = Lam ty (go (LS len) body)
    ...   -- other forms are treated homomorphically
    shifts0_var :: Elem locals ty0 → Length (locals :: Ctx n) → Elem (locals +++ prefix) ty0
    shifts0_var EZ      _        = EZ
    shifts0_var (ES v) (LS len) = ES (shifts0_var v len)
  -- Because shifts0Exp provably does nothing, we can short-circuit it:
{-# noinline shifts0Exp #-}
{-# rules "shifts0Exp" shifts0Exp = unsafeCoerce #-}
```

Fig. 13. Shifting closed expressions should be trivial

common-subexpression elimination (CSE, Section 9), and so we generalize the notion of shifting by introducing a type class. The relevant definitions are in Figure 12.

The first detail to notice here is that *Shiftable* classifies a polykinded type variable *a*—note the $\forall n$ 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. We see this in the type of *shift*. 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 functions is straightforward, once we have written down the types and can be guided by GHC's type checker. The types themselves come straight from standard type theory, where they correspond to the weakening and strengthening lemmas.

8.3 Using *shifts0* in the type checker

Part of the discussion about the *UGlobal* case in the type checker (Section 7.2) was deferred until after we have introduced shifting. We return to this case here. The code is in Figure 8.

The challenge is that globals all refer to *closed* expressions, and yet the global might be used in a context with several bound variables. We must, therefore, adjust the context of the expression stored in the global. However, the usual shifting logic surely is overkill here: a global variable expression is closed, after all. There's no way shifting can possibly make a difference!

While we *could* use the general shifting mechanism, we instead prefer to use a specialization of shifting, tailored for closed expressions, *shifts0*. See Figure 13, which defines *shifts0Exp*, the definition of *shifts0* in the *Shiftable* instance for *Exp*. This function tiresomely walks the entire structure of its argument in order to do nothing. The problem is that the type of the output really is different than the type of the input; 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*

```
  -- from GHC's Data.Type.Equality module
class TestEquality (t :: k → Type) where testEquality :: t a → t b → Maybe (a :~: b)
class IHashable (t :: k → Type) where ihashWithSalt :: Int → t a → Int   -- in Data.IHashable
instance TestEquality (Elem xs) where ...                                 -- in Data.Vec
  -- in Exp
type KnownLength (ctx :: Ctx n) = SNatI n   -- "a context's length is available at runtime"
instance TestEquality (Exp ctx) where ...
instance KnownLength ctx ⇒ IHashable (Exp ctx) where ...
instance KnownLength ctx ⇒ IHashable (Elem ctx) where ...
  -- In Data.IHashMap.Base:
data IHashMap :: ∀k. (k → Type) → (k → Type) → Type where ...
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                                  -- In CSE
```

Fig. 14. Key definitions for indexed *HashMap*s

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* [Peyton Jones et al. 2001], 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's 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 because GHC might observe that *shifts0Exp* is a short function (because it's defined almost immediately in terms of *go*) and decide to inline it. The **noinline** tells GHC not to, and that way the rewrite rule can trigger.

Is this design a win or a loss? I'm not sure. It surely has aspects of a loss because the compiler can't 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's not clear we can do better.

9 COMMON-SUBEXPRESSION ELIMINATION

Having covered the basic necessities of an interpreter, I was curious to see how using an indexed AST would scale to more complex applications. I chose to implement a common-subexpression elimination pass, optimizing expressions with common subexpressions to use a `let`-bound variable instead. A full description of the CSE algorithm would take us too far afield here and is well documented in the `CSE` module; instead, I 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 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, I wanted the CSE algorithm to be reasonably efficient. Instead of creating my own mapping structure, I wanted to use the existing *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 14.

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. The *TestEquality* class includes indexed types where an equality test can inform the equality of the indices. In our case, this clearly includes *Exp ctx*, because we can compare two expressions; if they equal (in the shared context), then surely their types are the same. As *Exp* is indexed by its type, a comparison between the values gives us an equality between their type indices—exactly the contract *TestEquality* requires.

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.

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.

The CSE implementation overall was also agreeably easy. While the design of the overall 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 Polymorphic recursion in types

It is well known that polymorphic recursion is impossible with Damas-Milner type inference [Henglein 1993; Mycroft 1984]. If we want to write a polymorphic recursive function, we must supply a type signature.

However, what if a *type* is polymorphic recursive? That is, a recursive occurrence in a type definition might have a parameter of a different kind than the outer definition. A handy example is the *Length* type, repeated here:

```
data Length :: ∀a n. Vec a n → Type where
  LZ :: Length VNil
  LS :: Length xs → Length (x :> xs)
```

This type is polymorphic recursive because the recursive occurrence in the *LS* constructor takes a parameter *xs* which has a different kind (*Vec a n*) than the kind of the parameter of the return type of *LS*, which is *Vec a* (*Succ n*). When should GHC accept such a definition? In other words, when does a type have a *kind signature*?

Given the syntax of GHC, this is not an easy question to answer. For example, the *Length* type as written above still requires a small amount of kind inference: I have not written the kinds of *a* or *n*. Other forms of type declarations have other confounding details. Worse, the decision whether or not a type has a kind signature must be made very early, before doing any kind inference on the type: the signal must be purely syntactic.

Accordingly, GHC defines a set of rules describing when types have a so-called *complete user-specified kind signature*, or CUSK. These rules, as documented in the GHC manual, say that a datatype declaration has a CUSK when any kind variables mentioned in its explicit kind are explicitly quantified (among other rules). This means that the $\forall a\ n$ above is compulsory—if I omit this, the type does not have a CUSK and thus cannot be polymorphic recursive.

This leads to an unpleasant user experience. Leaving out the explicit quantification induces an error message about mismatched kinds. It is not hard to work out that GHC is struggling to infer polymorphic recursion from this message, but nothing suggests to add explicit quantification to solve the problem. Instead, the programmer has to already be familiar with the vagaries of CUSKs to figure out what to do.

Happily, there is already a proposal written[11] to fix this problem by allowing users to write kind signatures distinct from type declarations, much as we do with term-level functions.

10.2 let *should* sometimes be generalized

Type inference in the presence of GADTs is hard [Chen and Erwig 2016; Peyton Jones et al. 2006, 2004; Vytiniotis et al. 2011]. 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 [Vytiniotis et al. 2010].[12] However, in two separate places, this lack of generalization tripped up my implementation:

Generalizing type signatures. If a function's type signature can be kind-generalized, GHC will automatically generalize it. For example, if we declare *typeRepShow* :: *TypeRep a* → *String*, GHC will infer that we really mean *typeRepShow* :: ∀ *k* (*a* :: *k*). *TypeRep a* → *String*. This implicit generalization is useful and rarely gets in the way.

[11] https://github.com/ghc-proposals/ghc-proposals/pull/54

[12] 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.

However, if I am declaring a local function whose type mentions in-scope variables from an outer scope, GHC does not kind-generalize, for exactly the same reasons that it does not generalize term-level **let**-definitions. (Vytiniotis et al. [2010] lay out these motivations in great detail.) This means that my type signature must explicitly mention any kind variables I wish to generalize over. This restriction bit me in the *go* helper functions to *subst* and *shiftsExp*, where the functions must be generalized over the length of the local context. I had not explicitly done so at first, and it took me some time to figure out what was going wrong. It might be helpful for GHC to alert a user when a **let** or type signature has been prevented from generalization.

Generalizing polymorphic traversal functions. 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.

10.3 Quantified constraints

The official version of the *HashMap* library contains a plethora of class instances for *HashMap*. Sadly, most of these instances could not be preserved in my adaptation. An illustrative example is for *Show*. Here is the original instance head:

instance (*Show k*, *Show v*) ⇒ *Show* (*HashMap k v*) **where** ...

We need instead an instance for *Show* (*IHashMap k v*), but now *k* and *v* are indexed types. We would like to assert *Show* (*k i*) and *Show* (*v i*), but *i* is not in scope. A knowledgable Haskeller might think of the class *Show1*, which classifies type constructors like [ ] and *Maybe*, but *Show1* lifts a *show* function on an element type into a functor-like type and requires its argument to have kind *Type* → *Type*. So, *Show1* does not work for us here.

What we need is this instance:

instance ((∀*i*. *Show* (*k i*)), ∀*i*. *Show* (*v i*)) ⇒ *Show* (*IHashMap k v*) **where** ...

The index *i* is now quantified in the constraint itself. For our *Exp* and *Elem* types, these constraints would be satisfiable.

Quantified class constraints are a recent innovation in the Haskell world, described by Bottu et al. [2017] and already proposed (with a working prototype implementation) as a future extension to GHC.[13] I have not tried my instance above against the prototype implementation, but I am confident that a correct implementation would allow instances such as this *Show* instance to be written.

10.4 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 familiar with the singletons library [Eisenberg and Weirich 2012] that automates working with them, I was not tempted to use it here.

[13] https://github.com/ghc-proposals/ghc-proposals/pull/109

10.5 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 [Christiansen 2015]. 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, where a programmer can replace an expression with an underscore and GHC will tell you the desired type of the expression.

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.

10.6 Related work

The basic idea embodied in Stitch is not new. Perhaps the first elucidation of the technique of using an indexed AST is by Augustsson and Carlsson [1999], who implemented their interpreter in Cayenne [Augustsson 1998]. The idea was picked up by Pašalić et al. [2002], who use the example of an indexed AST to power the introduction of Meta-D, a language useful for writing indexed ASTs. Other work principally focusing on an index AST includes that by Chen and Xi [2003], which includes an indexed CPS transform, implemented in ATS [Xi 2004]. An implementation of this idea in Haskell is described by Guillemette and Monnier [2008], 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. In contrast to the works cited here, the current paper does not seek to push the envelope in what is possible via this encoding. Instead, my goal is both to clarify the technique via a presentation available to intermediate Haskellers and also to assess the current state of GHC/Haskell for richly typed work.

10.7 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 modern 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 as a host language for serious work with fancy types.

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

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

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.

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

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.

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

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

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

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

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

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

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

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

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.

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

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

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.

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.

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

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

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.

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

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

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

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

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

Alan Mycroft. 1984. *Polymorphic type schemes and recursive definitions*. Springer, Berlin, Heidelberg.

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

Simon Peyton Jones. 2003. Wearing the Hair Shirt: A Retrospective on Haskell. Invited talk at POPL. (2003).

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*.

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).

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.

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.

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.

F. Pfenning and C. 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

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

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

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).

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.

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

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).

Philip Wadler. 2003. A Prettier Printer. In *The Fun of Programming: Essays in Honor of Richard Bird*, Jeremy Gibbons and Oege De Moor (Eds.). Palgrave.

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

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

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.

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

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.