Unraveling recursion: compiling an IR with recursion to System F

Michael Peyton Jones[1][0000−0003−0602−1657], Vasilis Gkoumas[1], Roman Kireev[1][0000−0003−4687−2739], Kenneth MacKenzie[1], Chad Nester[2], and Philip Wadler[2][0000−0001−7619−6378]

[1] IOHK
{roman.kireev,michael.peyton-jones,vasilis.gkoumas,kenneth.mackenzie}@iohk.io
[2] University of Edinburgh
{cnester,wadler}@inf.ed.ac.uk

Abstract. Lambda calculi are often used as intermediate representations for compilers. However, they require extensions to handle higher-level features of programming languages. In this paper we show how to construct an IR based on System F_ω^μ which supports recursive functions and datatypes, and describe how to compile it to System F_ω^μ. Our IR was developed for commercial use at the IOHK company, where it is used as part of a compilation pipeline for smart contracts running on a blockchain.

1 Introduction

Many compilers make use of *intermediate representations* (IRs) as stepping stones between their source language and their eventual target language. Lambda calculi are tempting choices as IRs for functional programming languages. They are simple, well-studied, and easy to analyze.

However, lambda calculi also have several features that make them poor IRs.

- They are hard to read and write. Although they are mostly read and written by computers, this complicates writing compilers and debugging their output.
- They can be hard to optimize. Some optimizations are much easier to write on a higher-level language. For example, dead-binding elimination is much easier with explicit let-bindings.
- They make the initial compilation step "too big". Compiling all the way from a high-level surface language to a lambda calculus can involve many complex transformations, and it is often advantageous from an engineering standpoint to break it into smaller steps.

Hence it is common to design an IR by extending a lambda calculus with additional features which make the IR more legible, easier to optimize, or closer to the source language (e.g. GHC Core [26], Henk [25], Idris' TT [4], and OCaml's Lambda [20]). However, given that such IRs are desirable, there is little material on implementing or compiling them.

In this paper we construct an IR suitable for a powerful functional programming language like Haskell. We take as our lambda calculus System F^{μ}_{ω} (System F_{ω} with indexed fixpoints: see [27, Chapter 30], formalized recently in [8]), which allows us to talk about higher-kinded recursive types, and extend it to an IR called FIR which adds the following features:

- Let-binding of non-recursive terms, types, and datatypes.
- Let-binding of recursive terms and datatypes.

This is a small, but common, subset of the higher-level features that functional programming languages usually have, so this provides a reusable IR for compiler writers targeting System F^{μ}_{ω}.

Moreover, all of the compilation passes that we provide are *local* in the sense that they do not access more than one level of the syntax tree, and they do not require any type information that is not present in type annotations. So while we provide typing rules for FIR, it is not necessary to perform type synthesis in order to compile it.

Encoding recursive terms has traditionally been done with fixpoint combinators. However, the textbook accounts typically do not cover mutual recursion, and where it *is* handled it is often assumed that the calculus is non-strict. We construct a generalized, polyvariadic fixpoint combinator that works in both strict and non-strict base calculi, which we use to compile recursive terms.

In order to compile datatypes, we need to encode them and their accompanying constructors and destructors using the limited set of types and terms we have available in our base calculus. The Church encoding ([27, Chapter 5.2, Chapter 23.4]) is a well-known method of doing this in plain System F. With it, we can encode even recursive datatypes, so long as the recursion occurs only in positive positions.

However, some aspects of the Church encoding are not ideal, for example, it requires time proportional to the size of a list to extract its tail. We use a different encoding, the Scott encoding [1], which can encode any recursive datatype, but requires adding a fixpoint operator to System F in order to handle arbitrary type-level recursion.

To handle mutually recursive datatypes we borrow some techniques from the generic programming community, in particular indexed fixpoints, and the use of type-level tags to combine a family of mutually recursive datatypes into a single recursive datatype. While this technique is well-known (see e.g. [32]), the details of our approach are different, and we face some additional constraints because we are targeting System F^{μ}_{ω} rather than a full dependently-typed calculus.

We have used FIR as an IR in developing Plutus [16], a platform for developing smart contracts targeting the Cardano blockchain. Users write programs in Haskell, which are compiled by a GHC compiler plugin into Plutus Core, a small functional programming language. Plutus Core is an extension of System F^{μ}_{ω}, so in order to easily compile Haskell's high-level language features we developed FIR as an IR above Plutus Core. We have used this compiler to write substantial programs in Haskell and compile them to Plutus Core, showing that the tech-

niques in this paper are usable in practice. The compiler is available for public use at [17].

Contributions. We make the following contributions.

- We give syntax and typing rules for FIR, a typed IR extending System F^{μ}_{ω}.
- We define a series of local compilation passes which collectively compile FIR into System F^{μ}_{ω}.
- We provide a reference implementation of the syntax, type system, and several of the compilation passes in Agda [24], a powerful dependently typed programming language.
- We have written a complete compiler implementation in Haskell as part of a production system for the Plutus platform.

Our techniques for encoding datatypes are not novel [32][21]. However, we know of no complete presentation that handles mutual recursion and parameterized datatypes, and targets a calculus as small as System F^{μ}_{ω}.

We believe our techniques for encoding mutually recursive functions are novel.

While the Agda compiler implementation is incomplete, and does not include soundness proofs, we believe that the very difficulty of doing this makes our partial implementation valuable. We discuss the difficulties further in Section 5.

Note on the use of Agda. Although System F^{μ}_{ω} is a complete programming language in its own right, it is are somewhat verbose and clumsy to use for the *exposition* of the techniques we are presenting.

Consequently we will use:

- Agda code, typeset colourfully, for exposition.
- System F^{μ}_{ω} code, typeset plainly, for the formal descriptions.

We have chosen to use $*$ for the kind of types, whereas Agda normally uses Set. To avoid confusion we have aliased Set to $*$ in our Agda code. Readers should recall that Agda uses $\rightarrow$ following binders rather than a . character.

The Agda code in this paper and the Agda compiler code are available in the Plutus repository.

Notational conventions. We will omit kind signatures in System F^{μ}_{ω} when they are $*$, and any other signatures when they are obvious from context or repetition.

We will be working with a number of constructs that have sequences of elements. We will adopt the metalanguage conventions suggested by Guy Steele [29], in particular:

- $t[x := v]$ is a substitution of v for x in t.
- $\overline{t}$ is expanded to any number of (optionally separated) copies of t. Any underlined portions of t must be expanded the same way in each copy. Where we require access to the index, the overline is superscripted with the index. For example:

  - $\overline{x : T}$ is expanded to $x_1 : T_1 \ldots x_n : T_n$
  - $\overline{\Gamma \vdash J}$ is expanded to $\Gamma \vdash J_1 \ldots \Gamma : J_n$
  - $\overline{x_j : T_{j+1}}^{j}$ is expanded to $x_1 : T_2 \ldots x_n : T_{n+1}$
- $\overline{t \to}\ u$ is expanded to $t_1 \to \ \ldots \ \to t_n \to u$, similarly for $\Rightarrow$.

2 Datatype encodings

The Scott encoding represents a datatype as the type of the pattern-matching functions on it. For example, the type of booleans, Bool, is encoded as

$$\forall R.R \to R \to R$$

That is, for any output type R you like, if you provide an R for the case where the value is false and an R for the case where the value is true, then you have given a method to construct an R from all possible booleans, thus performing a sort of pattern-matching. In general the arguments to the encoded datatype value are functions which transform the arguments of each constructor into an R.

The type of naturals, Nat, is encoded as

$$\forall R.R \to (\mathsf{Nat} \to R) \to R$$

Here we see an occurrence of Nat in the definition, which corresponds to recursive use in the "successor" constructor. We will need type-level recursion to deal with recursive references.

The Church encoding of Bool is the same as the Scott encoding. This is true for all non-recursive datatypes, but not for recursive datatypes. The Church encoding of Nat is:

$$\forall R.R \to (R \to R) \to R$$

Here the recursive occurrence of Nat has disappeared, replaced by an R. This is because while the Scott encoding corresponds to a pattern-match on a type, the Church encoding corresponds to a *fold*, so recursive occurrences have already been folded into the output type.

This highlights the tradeoffs between the two encodings (see [19] for further discussion):

- To operate on a Church encoded value we must perform a fold on the entire structure, which is frequently inefficient. For a Scott encoded value, we only have to inspect the surface level of the term, which is inexpensive.
- Since recursive occurrences of the type are already "folded" in the Church encoding, there is no need for a type-level recursion operator. Contrast this with the situation with the Scott encoding, in which additional type-level machinery (fixed points) is needed to define type-level recursion.

In this paper we will use the Scott encoding to encode datatypes.

3 Syntax and type system of System F^{μ}_{ω} and FIR

FIR is an extension of System F^{μ}_{ω}, which is itself an extension of the well-known System F_{ω}. In the following figures we give

- Syntax (Figure 1)
- Kinding (Figure 2)
- Well-formedness of constructors and bindings (Figure 4)
- Type equivalence (Figure 5)
- Type synthesis (Figure 6)

for full FIR. Cases without highlighting are for System F_{ω}, while we highlight additions for System F^{μ}_{ω} and FIR.

There are a number of auxiliary definitions in Figure 3 for dealing with datatypes and bindings. These define kinds and types for the various bindings produced by datatype bindings. We will go through examples of how they work in Section 4.3.

3.1 Recursive types

System F_{ω} is very powerful, but does not allow us to define (non-positive) recursive types. Adding a type-level fixed point operator enables us to do this (see e.g. [27, Chapter 20]). However, we must make a number of choices about the precise nature of our type-level fixed points.

Isorecursive and equirecursive types. The first choice we have is between two approaches to exposing the fixpoint property of our recursive types. Systems with *equirecursive* types identify $(\texttt{fix}\, f)$ and $f(\texttt{fix}\, f)$; whereas systems with *isorecursive* types provide an isomorphism between the two, using a term `unwrap` to convert the first into the second, and a term `wrap` for the other direction.

The tradeoff is that equirecursive types add no additional terms to the language, but have a more complicated metatheory. Indeed, typechecking System F^{μ}_{ω} with equirecursive types is not known to be decidable in general ([11,7]). Isorecursive types, on the other hand, have a simpler metatheory, but require additional terms. It is not too important for an IR to be easy to program by hand, so we opt for isorecursive types, with our witness terms being `wrap` and `unwrap`.

Choosing an appropriate fixpoint operator. We also have a number of options for *which* fixpoint operator to add. The most obvious choice is a fixpoint operator `fix` which takes fixpoints of type-level endofunctions at any kind K (i.e. it has signature $\texttt{fix} : (K \Rightarrow K) \Rightarrow K$). In contrast, our language System F^{μ}_{ω} has a fixpoint operator `ifix` ("indexed fix") which allows us to take fixpoints only at kinds $K \Rightarrow *$.

The key advantage of `ifix` over `fix` is that it is much easier to give fully-synthesizing type rules for `ifix`. To see this, suppose we had a `fix` operator

terms	t, u	$::=$	x	variable
			$\lambda x : T.t$	lambda abstraction
			$t\ t$	function application
			$\Lambda X :: K.t$	type abstraction
			$t\ \{T\}$	type application
			$\texttt{wrap}\ T\ U\ t$	wrap
			$\texttt{unwrap}\ t$	unwrap
			$\texttt{let}\ [\texttt{rec}]\ \overline{b}\ \texttt{in}\ t$	let
bindings	b	$::=$	$x : T = t$	term binding
			$X :: K = T$	type binding
			$\texttt{data}\ X\ (\overline{Y :: K}) = \overline{c}\ \texttt{with}\ x$	datatype binding
constructors	c	$::=$	$x\ (\overline{T})$	
values	v	$::=$	$\lambda x : T.t$	lambda abstraction
			$\Lambda X :: K.t$	type abstraction
			$\texttt{wrap}\ T\ U\ v$	wrap
types	T, U	$::=$	X	type variable
			$T \to U$	arrow type
			$\forall X :: K.T$	universal type
			$\lambda X :: K.T$	function type
			$T\ U$	function application
			$\texttt{ifix}\ T\ U$	fixpoint type
contexts	Γ	$::=$	$\varnothing$	empty
			$\Gamma, x : T$	term variable binding
			$\Gamma, X :: K$	type variable binding
kind	K	$::=$	$*$	type kind
			$K \Rightarrow K$	arrow kind

Fig. 1: Syntax of FIR

in our language, with corresponding `wrap` and `unwrap` terms. We now want to write typing rules for `wrap`. However, `fix` allows us to take fixpoints at *arbitrary* kinds, whereas `wrap` and `unwrap` are terms, which always have types of kind $*$. Thus, the best we can hope for is to use `wrap` and `unwrap` with *fully applied* fixed points, i.e.:

$\texttt{wrap}_0\ f_0$	$t : \texttt{fix}\ f_0$	where $t : f_0\ (\texttt{fix}\ f_0)$
$\texttt{wrap}_1\ f_1\ a1$	$t : \texttt{fix}\ f_1\ a1$	where $t : f_1\ (\texttt{fix}\ f_1)\ a1$
$\texttt{wrap}_2\ f_2\ a1\ a2$	$t : \texttt{fix}\ f_2\ a1\ a2$	where $t : f_2\ (\texttt{fix}\ f_2)\ a1\ a2$
...		

$$\text{K-TVar}\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \qquad \text{K-Abs}\frac{\Gamma, X :: K_1 \vdash T :: K_2}{\Gamma \vdash (\lambda X :: K_1.T) :: K_1 \Rightarrow K_2}$$

$$\text{K-App}\frac{\Gamma \vdash T_1 :: K_1 \Rightarrow K_2 \quad \Gamma \vdash T_2 :: K_1}{\Gamma \vdash (T_1\ T_2) :: K_2} \qquad \text{K-Arrow}\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash (T_1 \to T_2) :: *}$$

$$\text{K-All}\frac{\Gamma, X :: K \vdash T :: *}{\Gamma \vdash (\forall X :: K.T) :: *} \qquad \text{K-Ifix}\frac{\Gamma \vdash T :: K \quad \Gamma \vdash F :: (K \Rightarrow *) \Rightarrow (K \Rightarrow *)}{\Gamma \vdash (\texttt{ifix}\ F\ T) :: *}$$

Fig. 2: Kinding for FIR

Throughout this figure when d or c is an argument
$d = \texttt{data}\ X\ (\overline{Y :: K}) = (\overline{c})\ \texttt{with}\ x$
$c = x(\overline{T})$

Auxiliary functions

$$\begin{aligned}
\mathsf{branchTy}(c, R) &= \overline{T \to}\ R \\
\mathsf{dataTy}(d) &= \lambda(\overline{Y :: K}).\forall R.(\overline{\mathsf{branchTy}(c, \underline{R})}) \to R \\
\mathsf{dataKind}(d) &= \overline{K \Rightarrow}\ * \\
\mathsf{constrTy}(d, c) &= \forall(\overline{Y :: K}).\overline{T \to}\ X\ \overline{Y} \\
\mathsf{matchTy}(d) &= \forall(\overline{Y :: K}).(X\ \overline{Y}) \to (\mathsf{dataTy}(d)\ \overline{Y})
\end{aligned}$$

Binder functions

$$\begin{aligned}
\mathsf{dataBind}(d) &= X :: \mathsf{dataKind}(d) \\
\mathsf{constrBind}(d, c) &= c : \mathsf{constrTy}(d, c) \\
\mathsf{constrBinds}(d) &= \overline{\mathsf{constrBind}(\underline{d}, c)} \\
\mathsf{matchBind}(d) &= x : \mathsf{matchTy}(d) \\
\mathsf{binds}(x : T = t) &= x : T \\
\mathsf{binds}(X : K = T) &= X : K \\
\mathsf{binds}(d) &= \mathsf{dataBind}(d), \mathsf{constrBinds}(d), \mathsf{matchBind}(d), x : \mathsf{matchTy}(d)
\end{aligned}$$

Fig. 3: Auxiliary definitions

This must be accounted for in our typing rules for fixed points.

It is possible to give typing rules for `wrap` that will do the right thing regardless of how the fixpoint type is applied. One approach is to use *elimination contexts*, which represent the context in which a type will be eliminated (i.e. applied). This is the approach taken in [10]. However, this incurs a cost, since we cannot *guess* the elimination context (since type synthesis is bottom-up), so we must attach elimination contexts to our terms somehow.

An alternative approach is to pick a more restricted fixpoint operator. Using `ifix` avoids the problems of `fix`: it always produces fixpoints at kind $K \Rightarrow *$,

$$\text{W-Con}\frac{c = x(\overline{T}) \qquad \overline{\Gamma \vdash T :: *}}{\Gamma \vdash_{\mathsf{ok}} c}$$

$$\text{W-Term}\frac{\Gamma \vdash T :: * \qquad \Gamma \vdash t : T}{\Gamma \vdash_{\mathsf{ok}} x : T = t} \quad \text{W-Type}\frac{\Gamma \vdash T :: K}{\Gamma \vdash_{\mathsf{ok}} X : K = T}$$

$$\text{W-Data}\frac{\begin{array}{c} d = \texttt{data}\ X\ (\overline{Y :: K}) = (\overline{c})\ \texttt{with}\ x \\ \Gamma' = \Gamma, \overline{Y :: K} \qquad \overline{\Gamma' \vdash_{\mathsf{ok}} c} \end{array}}{\Gamma \vdash_{\mathsf{ok}} d}$$

Fig. 4: Well-formedness of constructors and bindings

$$\text{Q-Refl}\frac{}{T \equiv T} \qquad \text{Q-Symm}\frac{T \equiv S}{S \equiv T}$$

$$\text{Q-Trans}\frac{S \equiv U \qquad U \equiv T}{S \equiv T} \qquad \text{Q-Arrow}\frac{S_1 \equiv S_2 \qquad T_1 \equiv T_2}{(S_1 \to T_1) \equiv (S_2 \to T_2)}$$

$$\text{Q-All}\frac{S \equiv T}{(\forall X :: K.S) \equiv (\forall X :: K.T)} \quad \text{Q-Abs}\frac{S \equiv T}{(\lambda X :: K.S) \equiv (\lambda X :: K.T)}$$

$$\text{Q-App}\frac{S_1 \equiv S_2 \qquad T_1 \equiv T_2}{S_1 T_1 \equiv S_2 T_2} \qquad \text{Q-Beta}\frac{}{(\lambda X :: K.T_1)T_2 \equiv T_1[X := T_2]}$$

Fig. 5: Type equivalence for FIR

which must be applied to precisely one argument of kind K before producing a type of kind $*$. This means we can give relatively straightforward typing rules as shown in Figure 6.

Adequacy of `ifix`. Perhaps surprisingly, `ifix` is powerful enough to give us fixpoints at any kind K. We give a semantic argument here, but the idea is simply stated: we can "CPS-transform" a kind K into $(K \Rightarrow *) \Rightarrow *$, which then has the correct shape for `ifix`.

Definition 1. *Let J and K be kinds. Then J is a* retract *of K if there exist functions $\phi : J \Rightarrow K$ and $\psi : K \Rightarrow J$ such that $\psi \circ \phi = id$.*

Proposition 1. *Suppose J is a retract of K and there is a fixpoint operator $\texttt{fix}_K$ on K. Then there is fixpoint operator $\texttt{fix}_J$ on J.*

Proof. Take $\texttt{fix}_J(f) = \psi(\texttt{fix}_K(\phi \circ f \circ \psi))$.

Proposition 2. *Let K be a kind in System F^{μ}_{ω}. Then there is a unique (possibly empty) sequence of kinds $(K_0, \ldots, K_n)$ such that $K = \overline{K \Rightarrow}\ *$.*

Proof. Simple structural induction.

$$\text{T-Var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{T-Abs}\ \frac{\Gamma, x : T_1 \vdash t : T_2 \quad \Gamma \vdash T_1 :: *}{\Gamma \vdash (\lambda x : T_1.t) : T_1 \to T_2}$$

$$\text{T-App}\ \frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash (t_1\ t_2) : T_2} \qquad \text{T-TAbs}\ \frac{\Gamma, X :: K \vdash t : T}{\Gamma \vdash (\Lambda X :: K.t) : (\forall X :: K.T)}$$

$$\text{T-TApp}\ \frac{\Gamma \vdash t_1 : \forall X :: K_2.T_1 \quad \Gamma \vdash T_2 :: K_2}{\Gamma \vdash (t_1\ \{T_2\}) : T_1[X := T_2]} \qquad \text{T-Eq}\ \frac{\Gamma \vdash t : S \quad S \equiv T}{\Gamma \vdash t : T}$$

$$\text{T-Wrap}\ \frac{\Gamma \vdash M : (F\ (\lambda(X :: K).\,\texttt{ifix}\,F\ X))\ T \quad \Gamma \vdash T :: K \quad \Gamma \vdash F :: (K \Rightarrow *) \Rightarrow (K \Rightarrow *)}{\Gamma \vdash \texttt{wrap}\ \ F\ T\ M : \texttt{ifix}\,F\ T}$$

$$\text{T-Unwrap}\ \frac{\Gamma \vdash M : \texttt{ifix}\,F\ T \quad \Gamma \vdash T :: K}{\Gamma \vdash \texttt{unwrap}\,M : (F\ (\lambda(X :: K).\,\texttt{ifix}\,F\ X))\ T}$$

$$\text{T-Let}\ \frac{\Gamma \vdash T :: * \quad \overline{\Gamma \vdash_{\mathsf{ok}} b} \quad \Gamma, \overline{\mathsf{binds}(b)} \vdash t : T}{\Gamma \vdash (\texttt{let}\,\overline{b}\,\texttt{in}\,t) : T}$$

$$\text{T-LetRec}\ \frac{\Gamma \vdash T :: * \quad \Gamma' = \Gamma, \overline{\mathsf{binds}(b)} \quad \overline{\Gamma' \vdash_{\mathsf{ok}} b} \quad \Gamma' \vdash t : T}{\Gamma \vdash (\texttt{let rec}\,\overline{b}\,\texttt{in}\,t) : T}$$

Fig. 6: Type synthesis for FIR

Proposition 3. *For any kind K in System F^{μ}_{ω}, K is a retract of $(K \Rightarrow *) \Rightarrow *$.*

Proof. Let $K = \overline{K \Rightarrow}\ *$ (by Proposition 2), and take

$$\begin{aligned}
&\phi : K \Rightarrow (K \Rightarrow *) \Rightarrow * \\
&\phi = \lambda(x :: K).\lambda(f :: K \Rightarrow *).f\ x \\
&\psi : ((K \Rightarrow *) \Rightarrow *) \Rightarrow K \\
&\psi = \lambda(w :: (K \Rightarrow *) \Rightarrow *).\lambda(\overline{a :: K}).w(\lambda(o :: K).o\ \overline{a})
\end{aligned}$$

Corollary 1. *If there is a fixpoint operator at kind $(K \Rightarrow *) \Rightarrow *$ then there is a fixpoint operator at any kind K.*

We can instantiate `ifix` with $K \Rightarrow *$ to get fixpoints at $(K \Rightarrow *) \Rightarrow *$, so `ifix` is sufficient to get fixpoints at any kind.

Note that since our proof relies on Proposition 2, it will not go through for arbitrary kinds when there are additional kind forms beyond $*$ and $\Rightarrow$. However, it will still be true for all kinds of the structure shown in Proposition 2.

The fact that retractions preserve the fixed point property is well-known in the context of algebraic topology: see [12, Exercise 4.7] or [5, Proposition

23.9] for example. While retractions between datatypes are a common tool in theoretical computer science (see e.g. [30]), we have been unable to find a version of Proposition 1 in the computer science literature. Nonetheless, we suspect this to be widely known.

3.2 Datatypes

FIR includes *datatypes*. A FIR datatype defines a type with a kind, parameterized by several type variables. The right-hand side declares of a list of constructors with type arguments, and the name of a matching function.[3] They thus are similar to the familiar style of defining datatypes in languages such as Haskell.

For example,

$$\texttt{data}\ \mathsf{Maybe}\ (A :: *) = (\mathsf{Nothing}(), \mathsf{Just}(A))\ \texttt{with}\ \mathsf{matchMaybe}$$

defines the familiar Maybe datatype, with constructors Nothing and Just, and matching function `matchMaybe`.

The type of `matchMaybe` is $\mathsf{Maybe}\, A \to \forall R.R \to (A \to R) \to R$. This acts as a pattern-matching function on Maybe — exactly as we saw the Scott encoding behave in Section 2. The matcher converts the abstract datatype into the raw, Scott-encoded type which can be used as a pattern matcher. We will see the full details in Section 4.3, and the type is given by $\mathsf{matchTy}(\mathsf{Maybe})$ as defined in Figure 10.

Since FIR includes recursive datatypes, we could have removed `ifix`, `wrap` and `unwrap` from FIR. However, in practice it is useful for the target language (System F^{μ}_{ω}) to be a true subset of the source language (FIR), as this allows us to implement compilation as a series of FIR-to-FIR passes.

3.3 Let

FIR also features let terms. These have a number of bindings in them which bind additional names which are in scope inside the body of the let, and inside the right-hand-sides of the bindings in the case of a recursive let.

FIR supports let-binding terms, (opaque) types, and datatypes.

The typing rules for let are somewhat complex, but are crucially responsible for managing the scopes of the bindings defined in the let. In particular:

- The bindings defined in the let are *not* in scope when checking the right-hand sides of the bindings if the let is non-recursive, but *are* in scope if it is recursive.

[3] Why declare a matching function explicitly, rather than using case expressions? The answer is that we want to be *local*: matching functions can be defined and put into scope when processing the datatype binding, whereas case expressions require additional program analysis to mach up the expression with the corresponding datatype.

– The bindings defined in the let are *not* in scope when checking the type of the entire binding.[4]

The behaviour of type-let is also worth explaining. Type-lets are more like *polymorphism* than type aliases in a language like Haskell. That is, they are opaque inside the body of the let, whereas a type alias would be transparent. This may make them seem like a useless feature, but this is not so. Term-lets are useful for binding sub-expressions of term-level computations to reusable names; type-lets are similarly useful for binding sub-expressions of *type-level* computations.

4 Compilation

We will show how to compile FIR by defining a compilation scheme for each feature in FIR:

– Non-recursive bindings of terms ($\mathbb{C}_{\text{term}}$, Section 4.1) and types ($\mathbb{C}_{\text{type}}$, Section 4.1)
– Recursive bindings of terms ($\mathbb{C}_{\text{termrec}}$, Section 4.2)
– Non-recursive bindings of datatypes ($\mathbb{C}_{\text{data}}$, Section 4.3)
– Recursive bindings of datatypes ($\mathbb{C}_{\text{datarec}}$, Section 4.4)

We do not consider recursive bindings of types, since the case of recursive datatypes is much more interesting and subsumes it.

Although our goal is to compile to System F^{μ}_{ω}, since it is a subset of FIR we can treat each pass as targeting FIR, by eliminating one feature from the language until we are left with precisely the subset that corresponds to System F^{μ}_{ω}. This has the advantage that we can continue to features of FIR until the point in the pipeline in which they are eliminated.[5]

In particular, we will use non-recursive let-bindings in $\mathbb{C}_{\text{termrec}}$ and $\mathbb{C}_{\text{datarec}}$, which imposes some ordering constraints on our passes.

Homogeneous let-bindings. We have said that we are going to compile e.g. term and type bindings separately, but our syntax (and typing rules) allows for let terms with many bindings of both sorts. While this is technically true, it is an easy problem to avoid.

Non-recursive bindings do not interfere with each other, since the newly-defined variables cannot occur in the right-hand sides of other bindings. That means that we can always decompose a single term with n bindings into n separate terms, one for each binding. Hence we can consider each sort of binding (and indeed, each individual binding) in isolation.

[4] This is the same device usually employed when giving typing rules for existential types to ensure that the inner type does not escape.

[5] An elegant extension of this approach would be to define an indexed *family* of languages with gradually fewer features. However, this would be a distraction from the main point of this paper, so we have not adopted it.

$$\mathbb{C}_{\text{term}}(\texttt{let}\ x : t = b\ \texttt{in}\ v) = (\lambda(x : t).v)\ b$$
$$\mathbb{C}_{\text{type}}(\texttt{let}\ t :: k = b\ \texttt{in}\ v) = (\Lambda(t :: k).v)\ \{b\}$$

Fig. 7: Compilation of non-recursive term and type bindings

The same is not true for recursive bindings. To simplify the presentation we add a restriction to the programs that we compile: we require recursive lets to be *homogeneous*, in that they must only contain one sort of binding (term, type, or datatype). This means that we can similarly consider each sort of binding in isolation, although we will of course need to consider *multiple* bindings of the same sort.

This restriction is not too serious in practice. Given a recursive let term with arbitrary bindings:

- Types cannot depend on terms, so there are no dependencies from types or datatypes to terms.
- We do not support recursive type bindings, so there are no dependencies from types or datatypes to types.

So we can always pull out the term and type bindings into separate (recursive) let terms. The situation would be more complicated if we wanted to support recursive types or dependent types.

4.1 Non-recursive term and type bindings

Non-recursive term and type bindings are easy to compile. They are encoded as immediately-applied lambda- and type-abstractions, respectively. We define the compilation scheme in Figure 7.

4.2 Recursive term bindings

Self-reference and standard combinators. It is well-known that we cannot encode the Y combinator in the polymorphic lambda calculus, but that we *can* encode it if we have recursive types [14, Section 20.3].[6] We need the following types:

```
Fix₀ : (* → *) → *
Fix₀ = IFix (λ Rec F → F (Rec F))

- Type for values which can be applied to themselves
Self : * → *
```

[6] We here mean *arbitrary* recursive types, not merely strictly positive types. We cannot encode the Y combinator in Agda, for example, without disabling the positivity check.

```
Self a = Fix₀ (λ Rec → Rec → a)

self : ∀ {A} → (Self A → A) → Self A
self f = wrap f

unself : ∀ {A} → Self A → Self A → A
unself s = unwrap s

selfApply : ∀ {A} → Self A → A
selfApply s = unself s s
```

The first thing we defined was $\mathsf{Fix}_0 : (* \Rightarrow *) \Rightarrow *$, which is a fixpoint operator that only works at kind $*$. We won't need the full power of `ifix` for this section, so the techniques here should be applicable for other recursive variants of System F_ω, provided they are able to define Fix_0.

Now we can define the Y combinator and its η-expanded version, the Z combinator.

```
y : ∀ {A} → (A → A) → A
y f = (λ z → f (selfApply z))
      (self (λ z → f (selfApply z)))

z : ∀ {A B : *} → ((A → B) → (A → B)) → (A → B)
z f = (λ z → f (λ a → (selfApply z) a))
      (self (λ z → f (λ a → (selfApply z) a)))
```

In strict lambda calculi the Y combinator does not terminate, and we need to use the Z combinator, which has a more restricted type (it only allows us to take the fixpoint of things of type $A \to B$).

Mutual recursion. The Y and Z combinators allow us to define singly recursive functions, but we also want to define *mutually* recursive functions.

This is easy in a non-strict lambda calculus: we have the Y combinator, and we know how to encode tuples, so we can simply define a recursive *tuple* of functions. However, this is still easy to get wrong, as we must be careful not to force the recursive tuple too soon.

Moreover, this approach does not work with the Z combinator, since a tuple is not a function (the Scott-encoded version is a function, but a *polymorphic* function).

We can instead construct a more generic fixpoint combinator which will be usable in both a non-strict and strict setting. We will present the steps using recursive definitions for clarity, but all of these can be implemented with the Z combinator.

Let us start with the function fix_2 which takes the fixpoint of a function of 2-tuples.

```
fix₂ : ∀ {A B} → (A × B → A × B) → A × B
fix₂ f = f (fix₂ f)
```

We can transform this as follows: first we curry f.

```
fix₂-uncurry : ∀ {A B} → (A → B → A × B) → A × B
fix₂-uncurry f = (uncurry f) (fix₂-uncurry f)
```

Now, we replace both the remaining tuple types with Scott-encoded versions, using the corresponding version of uncurry for Scott-encoded 2-tuples.

```
uncurry₂-scott
  : ∀ {A B R : ∗}
  → (A → B → R)
  → ((∀ {Q} → (A → B → Q) → Q) → R)
uncurry₂-scott f g = g f

fix₂-scott
  : ∀ {A B}
  → (A → B → ∀ {Q} → (A → B → Q) → Q)
  → ∀ {Q} → (A → B → Q) → Q
fix₂-scott f = (uncurry₂-scott f) (fix₂-scott f)
```

Finally, we reorder the arguments to f to make it look as regular as possible.

```
fix₂-rearrange
  : ∀ {A B}
  → (∀ {Q} → (A → B → Q) → A → B → Q)
  → ∀ {Q} → (A → B → Q) → Q
fix₂-rearrange f k = (uncurry₂-scott (f k)) (fix₂-rearrange f)
```

This gives us a fixpoint function pairs of mutually recursive values, but we want to handle *arbitrary* sets of recursive values. At this point, however, we notice that all we need to do to handle, say, triples, is to replace $A \to B$ with $A \to B \to C$ and the binary uncurry with the ternary uncurry. And we can abstract over this pattern.

```
fixBy
  : ∀ {F : ∗ → ∗}
  → ((∀ {Q} → F Q → Q) → ∀ {Q} → F Q → Q)
  → (∀ {Q} → F Q → F Q) → ∀ {Q} → F Q → Q
fixBy by f = by (fixBy by f) ∘ f
```

To get the behaviour we had before, we instantiate by appropriately:

```
by₂
  : ∀ {A B}
  → (∀ {Q} → (A → B → Q) → Q)
  → ∀ {Q} → (A → B → Q) → Q
by₂ r k = (uncurry₂-scott k) r
```

```
fixBy₂
  : ∀ {A B}
  → (∀ {Q} → (A → B → Q) → A → B → Q)
  → ∀ {Q} → (A → B → Q) → Q
fixBy₂ = fixBy by₂
```

How do we interpret by? Inlining uncurry into our definition of by_2 we find that it is in fact the identity function! However, by choosing the exact definition we can tweak the termination properties of our fixpoint combinator. Indeed, our current definition does not terminate even in a non-strict language like Agda, since it evaluates the components of the recursive tuple before feeding them into f. However, we can avoid this by "repacking" the tuple so that accessing one of its components will no longer force the other.[7]

```
- Repacking tuples.
repack₂ : ∀ {A B} → A × B → A × B
repack₂ tup = (proj₁ tup , proj₂ tup)

- Repacking Scott-encoded tuples.
by₂-repack
  : ∀ {A B : ∗}
  → (∀ {Q : ∗} → (A → B → Q) → Q)
  → ∀ {Q : ∗} → (A → B → Q) → Q
by₂-repack r k = k (r (λ x y → x)) (r (λ x y → y))
```

Passing `by₂-repack` to `fixBy` gives us a fixpoint combinator that terminates in a non-strict language like Agda or Haskell.

Can we write one that terminates in a strict language? We can, but we incur the same restriction that we have when using the Z combinator: the recursive values must all be functions. This is because we use exactly the same trick, namely η-expanding the values.

```
- with tuples
repack₂-strict
  : ∀ {A₁ B₁ A₂ B₂ : ∗}
  → (A₁ → B₁) × (A₂ → B₂)
  → (A₁ → B₁) × (A₂ → B₂)
repack₂-strict tup = ((λ x → proj₁ tup x) , (λ x → proj₂ tup x))

- with Scott-encoded tuples
by₂-strict
  : ∀ {A₁ B₁ A₂ B₂ : ∗}
  → (∀ {Q : ∗} → ((A₁ → B₁) → (A₂ → B₂) → Q) → Q)
```

[7] We have defined × as a simple datatype, rather than using the more sophisticated version in the Agda standard library. The standard library version has different strictness properties — indeed, for that version `repack₂` is precisely the identity.

Auxiliary functions

$$
\begin{aligned}
\mathsf{Id} &= \lambda(X :: *).X \\
F \rightsquigarrow G &= \forall Q.F\ Q \to G\ Q \\
\mathsf{fixBy} &= \Lambda(F :: * \Rightarrow *).\lambda(by : (F \rightsquigarrow \mathsf{Id}) \to (F \rightsquigarrow \mathsf{Id})). \\
&\quad\ z\ (\lambda(r : (F \rightsquigarrow F) \to (F \rightsquigarrow \mathsf{Id})).\lambda(f : F \rightsquigarrow F).by(\Lambda Q.\lambda(k : F\ Q).r\ f\ \{Q\}\ (f\ \{Q\}\ k))) \\
\mathsf{sel}_k(\overline{T}) &= \lambda(\overline{x : T}).x_k \\
\mathsf{by}(\overline{T}) &= \lambda(r : \forall Q.(\overline{T \to} Q) \to Q).\Lambda Q.\lambda(k : \overline{T \to} Q).k\ \overline{\underline{r}\ \{\underline{Q}\}\ (\mathsf{sel}_j\ \underline{(\overline{T})})}^{j} \\
\mathsf{fix}(\overline{T}) &= \mathsf{fixBy}\ \{\lambda Q.\overline{T \to} Q\}\ \ \mathsf{by}(\overline{T})
\end{aligned}
$$

Compilation function

$$
\begin{aligned}
&\mathbb{C}_{\text{termrec}}(\texttt{let rec}\ \overline{x : T = t}\ \texttt{in}\ u) \\
&\qquad = \texttt{let}\ r = \mathsf{fix}(\overline{T})\ \Lambda Q.\lambda(k : \overline{T \to} Q)\ (\overline{x : T}).k\ \overline{t} \\
&\qquad\quad\ \texttt{in let}\ \overline{x = \underline{r}\ \{T\}\ (\mathsf{sel}_j\ \underline{(\overline{T})})}^{j}\ \texttt{in}\ u
\end{aligned}
$$

Fig. 8: Compilation of recursive let-bindings

$$
\begin{aligned}
&\to \forall\ \{Q : *\} \to ((A_1 \to B_1) \to (A_2 \to B_2) \to Q) \to Q \\
&\mathsf{by}_2\text{-strict}\ r\ k = k\ (\lambda\ x \to r\ (\lambda\ f_1\ f_2 \to f_1\ x))\ (\lambda\ x \to r\ (\lambda\ f_1\ f_2 \to f_2\ x))
\end{aligned}
$$

This gives us general, n-ary fixpoint combinators in System F^{μ}_{ω}.

Formal encoding of recursive let-bindings. We define the compilation scheme for recursive term bindings in Figure 8, along with a number of auxiliary functions.
The definitions of fixBy, by, and fix are as in our Agda presentation. The function sel_k is what we pass to a Scott-encoded tuple to select the kth element. The Z combinator is defined as in the previous section (we do not repeat the definition here). We have given the lazy version of by, but it is straightforward to define the strict version, in exchange for the corresponding restriction on the types of the recursive bindings.

The compilation function is a little indirect: we create a recursive tuple of values, then we let-bind each component of the tuple again! Why not just pass a single function to the tuple that consumes all the components and produces t? The answer is that in order to use the Scott-encoded tuple we need to give it the *type* of the value that we are producing, which in this case would be the type of t. But we do not know this type without doing type inference on FIR. This way we instead extract each of the components, whose types we *do* know, since they are in the original let-binding.

Polymorphic recursion with the Z combinator. Neither the simple Z combinator nor our strict fixBy allow us to define recursive values which are not of function type. This might not seem too onerous, but this also forbids defining *polymorphic* values, such as polymorphic functions. For example, we cannot define a polymorphic map function this way.

```
let rec map : ∀A B.(A → B) → (List A → List B) =
          ΛA B. λ(f : A → B) (l : List A)
          matchList {A} l {List B}
          (Nil {B})
          (λ(h : A)(t : List A). Cons {B} (f h) (map {A} {B} f t))
in t

⇒

let rec map′ : Unit → ∀A B.(A → B) → (List A → List B) =
          λ(u : Unit). ΛA B. λ(f : A → B) (l : List A)
          matchList {A} l {List B}
          (Nil {B})
          (λ(h : A)(t : List A). Cons {B} (f h) (map′ () {A} {B} f t))
in let  map : ∀A B.(A → B) → (List A → List B) = map′ ()
in t
```

Fig. 9: Example of transforming polymorphic recursion

Sometimes we can get around this problem by floating the type abstraction out of the recursion. This will work in many cases, but fails in any instance of polymorphic recursion, which includes most recursive functions over irregular datatypes.

However, we can work around this restriction if we are willing to transform our program. The *thunking* transformation is a variant of the well-known transformation for simulating call-by-name evaluation in a call-by-value language [9][28]. Conveniently, this also has the property that it transforms the "thunked" parameters into values of *function* type, thus making them computable with the Z combinator.

The thunking transformation takes a set of recursive definitions $f_i : T_i = b_i$ and transforms it by:

- Defining the Unit datatype with a single, no-argument constructor ().
- Creating new (recursive) definitions $f'_i : \mathsf{Unit} \to T_i = \lambda(u : \mathsf{Unit}).b_i$.
- Replacing all uses of f_i in the b_is with $f'_i\ ()$,
- Creating new (non-recursive) definitions $f_i : T_i = f'_i\ ()$ to replace the originals.

Now our recursive value is truly of function type, rather than universal type, so we can compile it as normal.

An example is given in Figure 9 of transforming a polymorphic map function.

4.3 Non-recursive datatype bindings

Non-recursive datatypes are fairly easy to compile. We will generalize the Scott-encoding approach described in Section 2.

Throughout this figure when d or c is an argument
$d = \texttt{data}\ X\ (\overline{Y :: K}) = (\overline{c})\ \texttt{with}\ x$
$c = x(\overline{T})$

Auxiliary functions

$$\begin{aligned}
\mathsf{unveil}(d, t) &= t[X := \mathsf{dataTy}(d)] \\
\mathsf{constr}_k(d, c) &= \mathsf{unveil}(d, \Lambda(\overline{Y :: K}).\lambda(\overline{a : T}).\Lambda R.\lambda(\overline{b : \mathsf{branchTy}(c, \underline{R})})\ b_k\ \overline{a}) \\
\mathsf{constrs}(d) &= \overline{\mathsf{constr}_j(\underline{d}, c_j)}^{\,j} \\
\mathsf{match}(d) &= \Lambda(\overline{Y :: K}).\lambda(x : (\mathsf{dataTy}(d)\ \overline{Y}).x
\end{aligned}$$

Compilation function

$$\begin{aligned}
&\mathbb{C}_{\mathrm{data}}(\texttt{let}\ d\ \texttt{in}\ t) \\
&\qquad = (\Lambda(\mathsf{dataBind}(d)).\lambda(\mathsf{constrBinds}(d)).\lambda(\mathsf{matchBind}(d)).t) \\
&\qquad\quad \{\mathsf{dataTy}(d)\} \\
&\qquad\quad \mathsf{constrs}(d) \\
&\qquad\quad \mathsf{match}(d)
\end{aligned}$$

Fig. 10: Compilation of non-recursive datatype bindings

We define the compilation scheme for non-recursive datatype bindings in Figure 10, along with a number of auxiliary functions in addition to those in Figure 3.
Let's go through the auxiliary functions in turn (both those in Figure 3 and Figure 10), using the Maybe datatype as an example.

$$d := \texttt{data}\ \mathsf{Maybe}\ \ A = (\mathsf{Nothing}(), \mathsf{Just}(A))\ \texttt{with}\ \mathsf{match}$$

– $\mathsf{branchTy}(c, R)$ computes the type of a function which consumes all the arguments of the given constructor, producing a value of type R.

$$\begin{aligned}
\mathsf{branchTy}(\mathsf{Nothing}(), R) &= R \\
\mathsf{branchTy}(\mathsf{Just}\ A, R) &= A \to R
\end{aligned}$$

– $\mathsf{dataKind}(d)$ computes the kind of the datatype type. This is a kind arrow from the kinds of all the type arguments to $*$.

$$\mathsf{dataKind}(\mathsf{Maybe}) = * \Rightarrow *$$

– $\mathsf{dataTy}(d)$ computes the Scott-encoded datatype. This binds the type variables with a lambda and then constructs the pattern matching function type using the branch types.

$$\mathsf{dataTy}(d) = \lambda A.\forall R.R \to (A \to R) \to R$$

– $\mathsf{constrTy}(c, T)$ computes the type of a constructor of the datatype d.

$$\begin{aligned}
\mathsf{constrTy}(\mathsf{Nothing}(), \mathsf{Maybe}) &= \forall A.\ \mathsf{Maybe}\ A \\
\mathsf{constrTy}(\mathsf{Just}\ A, \mathsf{Maybe}) &= \forall A.A \to \mathsf{Maybe}\ A
\end{aligned}$$

– $\mathsf{unveil}(d, t)$ "unveils" the datatype inside a type or term, replacing the abstract definition with the concrete, Scott-encoded one. We apply this to the *definition* of the constructors, for a reason we will see shortly. This makes no difference for non-recursive datatypes, but will matter for recursive ones.

$$\mathsf{unveil}(d, t) = t[\mathsf{Maybe} := \lambda A.\forall R.R \to (A \to R) \to R]$$

– $\mathsf{constr}_k(d, c)$ computes the definition of the kth constructor of a datatype. To match the signature of the constructor, this is type abstracted over the type variables and takes arguments corresponding to each of the constructor arguments. Then it constructs a pattern matching function which takes branches for each alternative and uses the kth branch on the constructor arguments.

$$\mathsf{constr}_1(d, \mathsf{Nothing}()) = \Lambda A.\Lambda R.\lambda(b_1 : R)(b_2 : A \to R).b_1$$
$$\mathsf{constr}_2(d, \mathsf{Just}(A)) = \Lambda A.\lambda(v : A).\Lambda R.\lambda(b_1 : R)(b_2 : A \to R).b_2\ v$$

– $\mathsf{matchTy}(d)$ computes the type of the datatype matcher, which converts from the abstract datatype to a pattern-matching function — that is, precisely the Scott-encoded type.

$$\mathsf{matchTy}(d) = \forall A.\, \mathsf{Maybe}\, A \to (\forall R.R \to (A \to R) \to R)$$

– $\mathsf{match}(d)$ computes the definition of the matcher of the datatype, which is the identity.

$$\mathsf{match}(d) = \Lambda A.\lambda(v : \mathsf{Maybe}\, A).v$$

The basic idea of the compilation scheme itself is straightforward: use type abstraction and lambda abstraction to bind names for the type itself, its constructors, and its match function.

There is one quirk: usually when encoding let-bindings we create an *immediately* applied type- or lambda-abstraction, but here they are *interleaved*. The reason for this is that the datatype must be abstract inside the *signature* of the constructors and the match function, since otherwise any uses of those functions inside the body will not typecheck. But inside the *definitions* the datatype must be concrete, since the definitions make use of the concrete structure of the type. This explains why we needed to use $\mathsf{unveil}(d, t)$ on the definitions of the constructors, since they appear outside the scope in which we define the abstract type. Note that this means we really must perform a substitution rather than creating a let-binding, since that would simply create another abstract type.[8]

[8] It is well-known that abstract datatypes can be encoded with existential types ([22]). The presentation we give here is equivalent to using a value of existential type which is immediately unpacked, and where existential types are given the typical encoding using universal types.

Consider the following example:

$$
\begin{aligned}
&\mathbb{C}_{\text{data}}(\texttt{let data}\ \mathsf{Maybe}\ A = (\mathsf{Nothing}(), \mathsf{Just}(A))\ \texttt{with match} \\
&\qquad \texttt{in}\ \mathsf{match}\ \{\mathsf{Int}\}\ (\mathsf{Just}\{\mathsf{Int}\}1)\ 0\ (\lambda x : \mathsf{Int}.x + 1)) \\
&= (\Lambda(\mathsf{Maybe} :: * \Rightarrow *). && \text{(signature of Maybe)} \\
&\quad \lambda(\mathsf{Nothing} : \forall A.\, \mathsf{Maybe}\, A). && \text{(signature of Nothing)} \\
&\quad \lambda(\mathsf{Just} : \forall A. A \to \mathsf{Maybe}\, A). && \text{(siganture of Just)} \\
&\quad \lambda(\mathsf{match} : \forall A.\, \mathsf{Maybe}\, A \to \forall R. R \to (A \to R) \to R). && \text{(signature of match)} \\
&\quad \mathsf{match}\ \{\mathsf{Int}\}\ (\mathsf{Just}\{\mathsf{Int}\}1)\ 0\ (\lambda x : \mathsf{Int}.x + 1)) && \text{(body of the let)} \\
&\quad (\lambda A. \forall R. R \to (A \to R) \to R) && \text{(definition of Maybe)} \\
&\quad (\Lambda A. \Lambda R. \lambda(b_1 : R)\ (b_2 : A \to R).b_1) && \text{(definition of Nothing)} \\
&\quad (\Lambda A. \lambda(v_1 : A). \Lambda R. \lambda(b_1 : R)\ (b_2 : A \to R).b_2\ v_1) && \text{(definition of Just)} \\
&\quad (\Lambda A. \lambda(v : \forall R. R \to (A \to R) \to R).v) && \text{(definition of match)}
\end{aligned}
$$

Here we can see that:

- Just needs to produce the abstract type inside the body of the let, otherwise the application of match will be ill-typed.
- The definition of Just produces the Scott-encoded type.
- match maps from the abstract type to the Scott-encoded type inside the body of the let.
- The definition of match is the identity on the Scott-encoded type.

4.4 Recursive datatype bindings

Adding singly recursive types is comparatively straightforward. We can write our datatype as a type-level function (often called a "pattern functor" [3]) with a parameter for the recursive use of the type, and then use our fixpoint operator to produce the final datatype.[9]

```
ListF : (* → *) → (* → *)
ListF List A =
  - This is the normal Scott-encoding, using the
  - recursive 'List' provided by the pattern functor.
  ∀ {R : *} → R → (A → List A → R) → R

List : * → *
List A = IFix ListF A
```

However, it is not immediately apparent how to use this to define *mutually* recursive datatypes. The type of `ifix` is quite restrictive: we can only produce something of kind $k \Rightarrow *$.

[9] This is where the Scott encoding really departs from the Church encoding: the recursive datatype itself appears in our encoding, since we are only doing a "one-level" fold whereas the Church encoding gives us a full recursive fold over the entire datastructure.

If we had kind-level products and an appropriate fixpoint operator, then we could do this relatively easily by defining a singly recursive product of our datatypes. However, we do not have products in our kind system.

But we can *encode* type-level products. In [32] the authors use the fact that an n-tuple can be encoded as a function from an index to a value, and thus type-level naturals can be used as the index of a type-level function to encode a tuple of types. We take a similar approach except that we will not use a natural to index our type, but rather a richer datatype. This will prove fruitful when encoding parameterized types.

Let's consider an example: the mutually recursive types of trees and forests.

```
mutual
  data Tree₀ (A : *) : * where
    node₀ : A → Forest₀ A → Tree₀ A

  data Forest₀ (A : *) : * where
    nil₀ : Forest₀ A
    cons₀ : Tree₀ A → Forest₀ A → Forest₀ A
```

First of all, we can rewrite this with a "tag" datatype indicating which of the two cases in our datatype we want to use. That allows us to use a single data declaration to cover both of the types. Moreover, the tag can include the type parameters of the datatype, which is important in the case that they differ between the different datatypes.

```
data TreeForest^t : * where
  - 'Tree^t A' tags the type 'Tree A'
  Tree^t : * → TreeForest^t
  - 'Forest^t A' tags the type 'Forest A'
  Forest^t : * → TreeForest^t

module Single where
  - This mutual recursion is not strictly necessary,
  - and is only there so we can define the 'Tree'
  - and 'Forest' aliases for legibility.
  mutual
    - Type alias for the application of the main
    - datatype to the 'Tree' tag
    Tree : * → *
    Tree A = TreeForest (Tree^t A)

    - Type alias for the application of the main
    - datatype to the 'Forest' tag
    Forest : * → *
    Forest A = TreeForest (Forest^t A)

    data TreeForest : TreeForest^t → * where
```

```
node : ∀ {A} → A → Tree A → Tree A
nil : ∀ {A} → Forest A
cons : ∀ {A} → Tree A → Forest A → Forest A
```

That has eliminated the mutual recursion, but we still have a number of problems:

- We are relying on Agda's data declarations to handle recursion, rather than our fixpoint combinator.
- We are using inductive families, which we don't have a way to encode.
- `TreeForest`^{t} is being used at the kind level, but we don't have a way to encode datatypes at the kind level.

Fortunately, we can get past all of these problems. Firstly we need to make our handling of the different constructors more uniform by encoding them as sums of products.

```
module Constructors where
  mutual
    Tree : ∗ → ∗
    Tree A = TreeForest (Tree^t A)

    Forest : ∗ → ∗
    Forest A = TreeForest (Forest^t A)

    - This chooses the type of the constructor
    - given the tag
    TreeForestF : TreeForest^t → ∗
    - The 'Tree' constructor takes a pair of
    - an 'A' and a 'Forest A'
    TreeForestF (Tree^t A) = A × Forest A
    - The 'Forest' constructor takes either nothing,
    - or a pair of a 'Tree A' and a 'Forest A'
    TreeForestF (Forest^t A) = ⊤ ⊎ Tree A × Forest A

    {-# NO_POSITIVITY_CHECK #-}
    data TreeForest (tag : TreeForest^t) : ∗ where
      treeForest : TreeForestF tag → TreeForest tag
```

If we now rewrite `TreeForestF` to take the recursive type as a parameter instead of using it directly, we can write this with `ifix`.

```
module IFixed where
  TreeForestF : (TreeForest^t → ∗) → (TreeForest^t → ∗)
  TreeForestF Rec (Tree^t A) = A × Rec (Forest^t A)
  TreeForestF Rec (Forest^t A) = ⊤ ⊎ Rec (Tree^t A) × Rec (Forest^t A)

  TreeForest : TreeForest^t → ∗
  TreeForest = IFix TreeForestF
```

Finally, we need to encode the remaining datatypes that we have used. The sums and products in the right-hand-side of `TreeForestF` should be Scott-encoded as usual, since they represent the constructors of the datatype.

The tag type is more problematic. The Scott encoding of the tag type we have been using would be:

$$\mathsf{Scott\text{-}tag} = \forall\ \{R : *\} \to (* \to R) \to (* \to R) \to R$$

However, we do not have polymorphism at the kind level! But if we look at how we use the tag we see that we only ever match on it to produce something of kind $*$, and so we can get away with immediately instantiating this to $*$.

```
module Encoded where
  - Tag type instantiated to '*'
  TreeForest^e : *
  TreeForest^e = (* → *) → (* → *) → *

  - Encoded 'Tree^t' tag
  Tree^e : * → TreeForest^e
  Tree^e A = λ T F → T A

  - Encoded 'Forest^t' tag
  Forest^e : * → TreeForest^e
  Forest^e A = λ T F → F A

  TreeForestF : (TreeForest^e → *) → (TreeForest^e → *)
  TreeForestF Rec tag =
    - Pattern matching has been replaced by application
    tag
      - The encoded 'Tree' constructor
      (λ A → ∀ {R} → (A → Rec (Forest^e A) → R) → R)
      - The encoded 'Forest' constructor
      (λ A → ∀ {R} → R → (Rec (Tree^e A) → Rec (Forest^e A) → R) → R)

  TreeForest : TreeForest^e → *
  TreeForest = IFix TreeForestF
```

This, finally, gives us a completely System F^{μ}_{ω}-compatible encoding of our mutually recursive datatypes.

Formal encoding of recursive datatypes. We define the compilation scheme for recursive datatype bindings in Figure 11, along with a number of auxiliary functions. We will reuse some of the functions from Figure 10, but many of them need variants for the recursive case, which are denoted with a `rec` superscript. Let's go through the functions again, this time using Tree and Forest as examples:

$$d_1 := \texttt{data}\ \ \mathsf{Tree}\ \ A = (\mathsf{Node}(A, \mathsf{Forest}\ A))\ \texttt{with}\ \mathsf{matchTree}$$
$$d_2 := \texttt{data}\ \ \mathsf{Forest}\ \ A = (\mathsf{Nil}(), \mathsf{Cons}(\mathsf{Tree}\ A, \mathsf{Forest}\ A))\ \texttt{with}\ \mathsf{matchForest}$$

Throughout this figure when l, d, or c is an argument
$l = \texttt{let rec } \overline{d} \texttt{ in } t$
$d = \texttt{data } X\ (\overline{Y :: K}) = (\overline{c})\ \texttt{with } x$
$c = x(\overline{T})$

Auxiliary functions

$$
\begin{array}{ll}
\mathsf{tagKind}(l) & = \overline{\mathsf{dataKind}(d) \Rightarrow *} \\
\mathsf{tag}_k(l,d) & = \lambda(\overline{Y :: K}).\lambda(\overline{X :: \mathsf{dataKind}(d)}).X_k\ \overline{Y} \\
\mathsf{inst}_k(f,l,d) & = \lambda(\overline{Y :: K}).f\ (\mathsf{tag}_k(l,d)\ \overline{Y}) \\
\mathsf{family}(l) & = \lambda(r :: \overline{\mathsf{dataKind}(d) \Rightarrow *}).\lambda(t :: \mathsf{tagKind}(l)).\texttt{let } \overline{X = \mathsf{inst}_j(\underline{r}, \underline{l}, d_j)}^{\,j} \texttt{ in } t\ \overline{\mathsf{dataTy}(d)} \\
\mathsf{instFamily}_k(l,d) & = \lambda(\overline{Y :: K}).\texttt{ifix } \mathsf{family}(l)\ (\mathsf{tag}_k(l,d)\ \overline{Y}) \\
\mathsf{unveil}^{\mathtt{rec}}(l,t) & = t[\overline{X := \mathsf{instFamily}_j(\underline{l}, d_j)}]^j \\
\mathsf{constr}^{\mathtt{rec}}_{k,m}(l,d,c) & = \Lambda(\overline{Y :: K}).\lambda(\overline{a : T}).\texttt{wrap } \mathsf{family}(l)\ (\mathsf{tag}_k(l,d)\ \overline{Y})\ (\Lambda R.\lambda(\overline{b : \mathsf{branchTy}(c, \underline{R})}).\ b_m\ \overline{a}) \\
\mathsf{constrs}^{\mathtt{rec}}_k(l,d) & = \overline{\mathsf{constr}^{\mathtt{rec}}_{k,j}(\underline{l}, \underline{d}, c_j)}^{\,j} \\
\mathsf{match}^{\mathtt{rec}}_k(l,d) & = \Lambda(\overline{Y :: K}).\lambda(x : \mathsf{instFamily}_k(l,d)\ \overline{Y}).\texttt{unwrap}\, x
\end{array}
$$

Compilation function

$$
\begin{array}{ll}
\mathbb{C}_{\mathrm{datarec}}(l) & = (\Lambda(\overline{\mathsf{dataBind}(d)}).\lambda(\overline{\mathsf{constrBinds}(d)}).\lambda(\overline{\mathsf{matchBind}(d)}).t) \\
& \quad \{\overline{\mathsf{instFamily}_j(l,d_j)}^{\,j}\} \\
& \quad \overline{\mathsf{constrs}^{\mathtt{rec}}_j(\underline{l}, d_j)}^{\,j} \\
& \quad \overline{\mathsf{match}^{\mathtt{rec}}_j(\underline{l}, d_j)}^{\,j}
\end{array}
$$

Fig. 11: Compilation of recursive datatype bindings

– $\mathsf{tagKind}(l)$ defines the kind of the type-level tags for our datatype family, which is a Scott-encoded tuple of types.

$$\mathsf{tagKind}(l) = (* \Rightarrow *) \Rightarrow (* \Rightarrow *) \Rightarrow *$$

– $\mathsf{tag}_k(l,d)$ defines the tag type for the datatype d in the family.

$$
\begin{aligned}
\mathsf{tag}_1(l, \mathsf{Tree}) &= \lambda A.\lambda(v_1 :: * \Rightarrow *)(v_2 :: * \Rightarrow *).v_1\ A \\
\mathsf{tag}_2(l, \mathsf{Forest}) &= \lambda A.\lambda(v_1 :: * \Rightarrow *)(v_2 :: * \Rightarrow *).v_2\ A
\end{aligned}
$$

– $\mathsf{inst}_k(f,l,d)$ instantiates the family type f for the datatype d in the family by applying it to the datatype tag.

$$
\begin{aligned}
\mathsf{inst}_1(f, l, \mathsf{Tree}) &= \lambda A.f\ (\mathsf{tag}_1(l, \mathsf{Tree})\ A) \\
\mathsf{inst}_2(f, l, \mathsf{Forest}) &= \lambda A.f\ (\mathsf{tag}_2(l, \mathsf{Forest})\ A)
\end{aligned}
$$

– $\mathsf{family}(l)$ defines the datatype family itself. This takes a recursive argument and a tag argument, and applies the tag to the Scott-encoded types of the datatype components, where the types themselves are instantiated using the

recursive argument.

$$\begin{aligned}
\mathsf{family}(l) &= \lambda r\ t.\, \texttt{let} \\
&\quad \mathsf{Tree} = \mathsf{inst}_1(r, l, \mathsf{Tree}) \\
&\quad \mathsf{Forest} = \mathsf{inst}_2(r, l, \mathsf{Forest}) \\
&\quad \texttt{in}\ t\ \mathsf{dataTy}(d_1)\ \mathsf{dataTy}(d_2) \\
\mathsf{dataTy}(d_1) &= \lambda A.\forall R.(A \to \mathsf{Forest}\, A \to R) \to R \\
\mathsf{dataTy}(d_2) &= \lambda A.\forall R.R \to (\mathsf{Tree}\, A \to \mathsf{Forest}\, A \to R) \to R
\end{aligned}$$

- $\mathsf{instFamily}_k(l, d)$ is the full recursive datatype family instantiated for the datatype d, much like $\mathsf{inst}_k(f, l, d)$, but with the full datatype family.

$$\mathsf{instFamily}_1(l, \mathsf{Tree}) = \lambda A.\, \texttt{ifix}\ (\mathsf{family}(l))\ (\mathsf{tag}_1(l, \mathsf{Tree})\ A)$$

- $\mathsf{unveil}^{\mathtt{rec}}(l, t)$ "unveils" the datatypes as before, but unveils all the datatypes and replaces them with the full recursive definition instead of just the Scott-encoded type.
- $\mathsf{constr}^{\mathtt{rec}}_{k,m}(l, d, c)$ defines the constructor c of the datatype d in the family. It is similar to before, but includes a use of `wrap`.

$$\begin{aligned}
\mathsf{constr}^{\mathtt{rec}}_{1,1}(l, \mathsf{Tree}, \mathsf{Node}) &= \Lambda A.\lambda(v_1 : A)(v_2 : \mathsf{Forest}\, A). \\
&\quad \texttt{wrap}\ (\mathsf{instFamily}_1(l, \mathsf{Tree}))\ A \\
&\quad (\Lambda R.\lambda(b_1 : A \to \mathsf{Forest}\, A \to R).b_1\ v_1\ v_2) \\
\mathsf{constr}^{\mathtt{rec}}_{2,1}(l, \mathsf{Forest}, \mathsf{Nil}) &= \Lambda A. \\
&\quad \texttt{wrap}\ (\mathsf{instFamily}_2(l, \mathsf{Forest}))\ A \\
&\quad (\Lambda R.\lambda(b_1 : R)(b_2 : \mathsf{Tree}\, A \to \mathsf{Forest}\, A \to R).b_1) \\
\mathsf{constr}^{\mathtt{rec}}_{2,2}(l, \mathsf{Forest}, \mathsf{Cons}) &= \Lambda A.\lambda(v_1 : \mathsf{Tree}\, A)(v_2 : \mathsf{Forest}\, A). \\
&\quad \texttt{wrap}\ (\mathsf{instFamily}_2(l, \mathsf{Forest}))\ A \\
&\quad (\Lambda R.\lambda(b_1 : R)(b_2 : \mathsf{Tree}\, A \to \mathsf{Forest}\, A \to R).b_2\ v_1\ v_2)
\end{aligned}$$

- $\mathsf{match}^{\mathtt{rec}}_k(l, d)$ defines the matcher of the datatype d as before, but includes a use of `unwrap`.

$$\begin{aligned}
\mathsf{match}^{\mathtt{rec}}_1(l, \mathsf{Tree}) &= \Lambda A.\lambda(v : \mathsf{Tree}\, A).\, \texttt{unwrap}\ v \\
\mathsf{match}^{\mathtt{rec}}_2(l, \mathsf{Forest}) &= \Lambda A.\lambda(v : \mathsf{Forest}\, A).\, \texttt{unwrap}\ v
\end{aligned}$$

5 Compiler implementation in Agda

As a supplement to the presentation in this paper, we have written a formalisation of a FIR compiler in Agda.[10] The compiler includes the syntax, the type system (the syntax is intrinsically typed, so there is no need for a typechecker), and implementations of several of the passes. In particular, we have implemented:

[10] The complete source can be found in the Plutus repository.

– Type-level compilation of mutually recursive datatypes into System F^{μ}_{ω} types.
– Term-level compilation of mutually recursive terms into System F^{μ}_{ω} terms.

The Agda presentation uses an intrinsically-typed syntax, where terms are identified with their typing derivations [2]. This means that the compilation process is provably kind- and type-preserving.

However, the implementation is incomplete. The formalization is quite involved since the term-level parts of datatypes (constructors) must exactly line up with the type-level parts. Moreover, we have not proved any soundness results beyond type preservation. The complexity of the encodings makes it very hard to prove soundness. The artifact contains some further notes on the difficulties in the implementation.

6 Optimization

FIR has the virtue that it is significantly easier to optimize than System F^{μ}_{ω}. Here are two examples.

6.1 Dead binding elimination

Languages with let terms admit a simple form of dead code elimination: any bindings in let terms which are unused can be removed. A dead binding in a FIR term can be easily identified by constructing a dependency graph over the variables in the term, and eliminating any bindings for unreachable variables.

We can certainly do something with the compiled form of simple, non-recursive let bindings in System F^{μ}_{ω}. These are compiled to immediately-applied lambda abstractions, which is an easy pattern to identify, and it is also easy to work out whether the bound variable is used.

Recursive let bindings are much trickier. Here the compiled structure is obscured by the fixpoint combinator and the construction and deconstruction of the encoded tuple, which makes the pattern much harder to spot. Datatype bindings are similarly complex.

The upshot is that it is much easier to perform transformations based on the structure of variable bindings when those bindings are still present in their original form.

6.2 Case-of-known-constructor

The case-of-known-constructor optimization is very important for functional programming languages with datatypes (see e.g. [26, section 5]). When we perform a pattern-match on a term which we know is precisely a constructor invocation, we can collapse the immediate construction and deconstruction.

For example, we should be able to perform the following transformation:

$$\mathsf{match}\ \{\mathsf{Int}\}\ (\mathsf{Just}\ \{\mathsf{Int}\}\ 1)\ 0\ (\lambda x.x + 1) \Longrightarrow (\lambda x.x + 1)\ 1$$

This is easy to implement in FIR, since we still have the knowledge of which constructors and destructors belong to the same datatype. But once we have compiled to System F^{μ}_{ω} we lose this information. A destructor-constructor pair is just an inner redex of the term, which happens to reduce nicely. But reducing arbitrary redexes is risky (since we have no guarantee that it will not grow the program), and we do not know of a general approach which would identify these redexes as worth reducing.

7 Why not support these features natively?

The techniques in this paper cause a significant amount of runtime overhead. The combinator-based approach to defining recursive functions requires many more reductions than a direct implementation which could implement recursive calls by jumping directly to the code pointer for the recursive function.

Similarly, representing datatype values as functions is much less efficient than representing them as tagged data.

However, there are tradeoffs here for the language designer. If the language is intended to be a competitive general-purpose programming language like Haskell, then these performance losses may be unacceptable. On the other hand, if we care less about performance and more about correctness, then the benefits of having a minimal, well-studied core may dominate.

Moreover, even if a language has a final target language which provides these features natively, a naive but higher-assurance backend can provide a useful alternative code generator to test against.

Of course, the proof is in the pudding, and we have practical experience using these techniques in the Plutus platform [16]. Experience shows that the overhead proves not to be prohibitive: the compiler is able to compile and run substantial real-world Haskell programs, and is available for public use at [17].

8 Related work

8.1 Encoding recursive datatypes

Different approaches to encoding datatypes are compared in [19]. The authors provide a schematic formal description of Scott encoding, but ours is more thorough and includes complete handling of recursive types.

Indexed fixpoints are used in [32] to encode regular and mutually recursive datatypes as fixpoints of pattern functors. We use the same fixpoint operator — they call it "hfix", while we call it "ifix". They also use the trick of encoding products with a tag, but they use the natural numbers as an index, and they do not handle parameterized types. Later work in [21] does handle parameterized types, but our technique of putting the parameters into the tag type appears to be novel. Neither paper handles non-regular datatypes.

There are other implementations of System F_{ω} with recursive types. Brown and Palsberg [6] use isorecursive types, and includes an indexed fixpoint operator

as well as a typecase operator. However, the index for the fixpoint must be of kind $*$, whereas ours may be of any kind. Cai et al. [7] differ from this paper both in using equirecursive types and in that their fixpoint operator only works at kind $*$. Moreover, algebraic datatypes are supported directly, rather than via encoding.

8.2 Encoding recursive terms

There is very little existing material on compiling multiple mutually recursive functions, especially in a strict language. Some literature targets lower-level or specialized languages ([15,31,23]), whereas ours is a much more standard calculus. There are some examples which use fixpoint combinators (such as [18], extending [13] for typed languages) which use different fixpoint combinators.

8.3 Intermediate representations

GHC Haskell is well-known for using a fairly small lambda-calculus-based IR ("Core") for almost all of its intermediary work [26]. FIR is very inspired by GHC Core, but supports far fewer features and is aimed at eliminating constructs like datatypes and recursion, whereas they are native features of GHC Core.

A more dependently-typed IR is described in [25]. We have not yet found the need to generalize our base calculus to a dependently-typed one like Henk, but all the techniques in this paper should still apply in such a setting. Extensions to Henk that handle let-binding and datatypes are discussed, but it appears that these are intended as additional native features rather than being compiled away into a base calculus.

9 Conclusion

We have presented FIR, a reusable, typed IR which provides several typical functional programming language features. We have shown how to compile it into System F^{μ}_{ω} via a series of local compilation passes, and given a reference implementation for the compiler.

There is more work to do on the theory and formalisation of FIR. We have not given a direct semantics, in terms of reduction rules or otherwise. We would also like to prove our compilation correct, in that it commutes with reduction. A presentation of a complete compiler written in Agda with accompanying proofs would be desirable.

We could also remove some of the restrictions present in this paper: in particular the lack of mutually recursive type bindings, and the requirement that recursive let terms be homogeneous.

Acknowledgments. The authors would like to thank Mario Alvarez-Picallo and Manuel Chakravarty for their comments on the manuscript, as well as IOHK for funding the research.

References

1. Abadi, M., Cardelli, L., Plotkin, G.: Types for the Scott numerals (1993)
2. Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. In: Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings. pp. 453–468 (1999). https://doi.org/10.1007/3-540-48168-0_32, https://doi.org/10.1007/3-540-48168-0_32
3. Backhouse, R., Jansson, P., Jeuring, J., Meertens, L.: Generic programming — an introduction. In: LNCS. vol. 1608, pp. 28–115. Springer-Verlag (1999)
4. Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming **23**(5), 552–593 (2013). https://doi.org/10.1017/S095679681300018X, https://doi.org/10.1017/S095679681300018X
5. Bredon, G.E.: Topology and Geometry, Graduate Texts in Mathematics, vol. 139. Springer–Verlag (1993)
6. Brown, M., Palsberg, J.: Typed self-evaluation via intensional type functions. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 415–428. POPL 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009853, http://doi.acm.org/10.1145/3009837.3009853
7. Cai, Y., Giarrusso, P.G., Ostermann, K.: System F-omega with equirecursive types for datatype-generic programming. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 30–43. POPL '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837660, http://doi.acm.org/10.1145/2837614.2837660
8. Chapman, J., Kireev, R., Nester, C., Wadler, P.: System F in Agda, for fun and profit. In: Mathematics of Program Construction. LNCS (2019). https://doi.org/10.1007/978-3-030-33636-3_10
9. Danvy, O., Hatcliff, J.: Thunks (continued). In: Actes WSA'92 Workshop on Static Analysis (Bordeaux, France), September 1992, Laboratoire Bordelais de Recherche en Informatique (LaBRI), Proceedings. pp. 3–11 (1992)
10. Dreyer, D.: Understanding and Evolving the ML Module System. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (2005), aAI3166274
11. Dreyer, D.: A type system for recursive modules. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming. pp. 289–302. ICFP '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1291151.1291196, http://doi.acm.org/10.1145/1291151.1291196
12. Fulton, W.: Algebraic Topology: A First Course, Graduate Texts in Mathematics, vol. 153. Springer–Verlag (1995)
13. Goldberg, M.: A variadic extension of curry's fixed-point combinator. Higher Order Symbol. Comput. **18**(3-4), 371–388 (Dec 2005). https://doi.org/10.1007/s10990-005-4881-8, http://dx.doi.org/10.1007/s10990-005-4881-8
14. Harper, R.: Practical Foundations for Programming Languages. Cambridge University Press, New York, NY, USA (2012)
15. Hirschowitz, T., Leroy, X., Wells, J.B.: Compilation of extended recursion in call-by-value functional languages. In: Proceedings of the 5th ACM

SIGPLAN International Conference on Principles and Practice of Declarative Programming. pp. 160–171. PPDP '03, ACM, New York, NY, USA (2003). https://doi.org/10.1145/888251.888267, http://doi.acm.org/10.1145/888251.888267
16. IOHK: Plutus. https://github.com/input-output-hk/plutus (May 2019), accessed: 2019-05-01
17. IOHK: Plutus playground. https://prod.playground.plutus.iohkdev.io/ (May 2019), accessed: 2019-05-01
18. Kiselyov, O.: Many faces of the fixed-point combinator. http://okmij.org/ftp/Computation/fixed-point-combinators.html (Aug 2013), accessed: 2019-02-21
19. Koopman, P., Plasmeijer, R., Jansen, J.M.: Church encoding of data types considered harmful for implementations: Functional pearl. In: Proceedings of the 26th 2014 International Symposium on Implementation and Application of Functional Languages. pp. 4:1–4:12. IFL '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2746325.2746330, http://doi.acm.org/10.1145/2746325.2746330
20. Leroy, X.: The ZINC experiment: an economical implementation of the ML language. Ph.D. thesis, INRIA (1990)
21. Löh, A., Magalhães, J.P.: Generic programming with indexed functors. In: Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming. pp. 1–12. WGP '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2036918.2036920, http://doi.acm.org/10.1145/2036918.2036920
22. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential types. In: Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 37–51. POPL '85, ACM, New York, NY, USA (1985). https://doi.org/10.1145/318593.318606, http://doi.acm.org/10.1145/318593.318606
23. Nordlander, J., Carlsson, M., Gill, A.J.: Unrestricted pure call-by-value recursion. In: Proceedings of the 2008 ACM SIGPLAN Workshop on ML. pp. 23–34. ML '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1411304.1411309, http://doi.acm.org/10.1145/1411304.1411309
24. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology (2007)
25. Peyton Jones, S., Meijer, E.: Henk: A typed intermediate language. In: In Proc. First Int'l Workshop on Types in Compilation (1997)
26. Peyton Jones, S.L., Santos, A.L.M.: A transformation-based optimiser for haskell. Sci. Comput. Program. **32**(1-3), 3–47 (Sep 1998). https://doi.org/10.1016/S0167-6423(97)00029-4, http://dx.doi.org/10.1016/S0167-6423(97)00029-4
27. Pierce, B.C.: Types and Programming Languages. MIT press (2002)
28. Steele, G.L., Sussman, G.J.: Lambda: The ultimate imperative. Tech. rep., Cambridge, MA, USA (1976)
29. Steele Jr, G.L.: It's time for a new old language. In: PPOPP. p. 1 (2017)
30. Stirling, C.: Proof systems for retracts in simply typed lambda calculus. In: Proceedings of the 40th International Conference on Automata, Languages, and Programming - Volume Part II. pp. 398–409. ICALP'13, Springer-Verlag, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39212-2_36, http://dx.doi.org/10.1007/978-3-642-39212-2{_}36
31. Syme, D.: Initializing mutually referential abstract objects: The value recursion challenge. Electron. Notes Theor. Comput. Sci. **148**(2), 3–25 (Mar

2006). https://doi.org/10.1016/j.entcs.2005.11.038, http://dx.doi.org/10.1016/j.entcs.2005.11.038
32. Yakushev, A.R., Holdermans, S., Löh, A., Jeuring, J.: Generic programming with fixed points for mutually recursive datatypes. SIGPLAN Not. **44**(9), 233–244 (Aug 2009). https://doi.org/10.1145/1631687.1596585, http://doi.acm.org/10.1145/1631687.1596585