Heapless Functional Programming

Ellis Kesterton[0009−0006−9369−8672] and
Edwin Brady[0000−0002−9734−367X]

University of St Andrews, UK
{erk4,ecb10}@st-andrews.ac.uk

Abstract. Typed functional programming languages like Haskell and OCaml make heavy use of the heap at run-time. This makes them largely unsuitable for *systems programming*, where resources are limited and programs are often expected to run on bare-metal. This paper demonstrates how a (slightly restricted) high-level, pure, functional language can be compiled to machine code which does not use the heap at all. Despite usually requiring a heap at run-time, features such as higher-order functions, polymorphism and typeclasses are all supported by the surface language. This is made possible through *partial evaluation* [11]: by carefully reducing the program at compile-time, we can eliminate these high-level features entirely, resulting in a residual program which is trivial to compile. This paper describes the operation of this partial evaluator in detail, and introduces a novel type system which guarantees that the partial evaluator will always succeed in removing all heap-using features.

1 Introduction

At the core of most functional languages is a set of high-level features including higher-order functions, polymorphism and algebraic data types. Most of the time these features are a boon: they allow the programmer to write concise, maintainable, readable code. However, all of these features typically require a heap at run-time. This is problematic if we want to use functional languages for *systems programming*, where it is common to target environments with extremely limited resources. Consider the following Haskell program:

```
main :: IO ()
main = do
    n :: Int <- readInt
    m :: Int <- readInt
    printInt (n + m)
```

While this program may appear simple, it implicitly makes use of a wide variety of high-level (heap-using) features. Alone, using `do`-notation requires typeclasses (to resolve the `Monad` instance), higher-order functions (for the implicit calls to `>>=`) and polymorphism (also for `>>=`)! It is clear that every non-trivial

Haskell program will use the heap in some capacity – features such as higher-order functions are too ubiquitous to avoid. Does this mean that high-level functional programming is entirely reliant on the heap? This paper demonstrates that this is not the case.

Reconsider our example from a computational perspective: it is unclear why a program which adds two integers should need a heap at all! In this case, features such as typeclasses and higher-order functions are only abstractions for the programmer; they are not adding any real computational value. We show that most of the time we can actually use these abstractions for free. Through careful manipulation at compile-time, it is possible to transform a large class of high-level functional programs into equivalent low-level programs which do not require the heap at all. We make use of the type system to restrict input programs, ensuring that this transformation is always possible. Compiling a desugared version of the earlier example with our prototype implementation produces identical machine code to the following Rust program[1].

```
fn main() {
    let n = readInt();
    let m = readInt();
    printInt(n + m);
}
```

2 Contributions

The goal of this paper is to describe a method for designing and implementing high-level functional languages which do not use the heap at run-time. We believe this goal is worthwhile as not every low-level environment can feasibly support the high amount of heap allocations typically required by a functional language – by only using the stack, it becomes possible to use functional programming in these restricted environments. We hope that this work will help facilitate the design of functional languages which are suitable for systems programming on low-resource devices such as microcontrollers.

We focus on the design and implementation of a core language $\mathcal{C}$, which aims to be an elaboration target for high-level strict functional languages. Specifically, we make the following contributions:

- We define a polymorphic, higher-order core language $\mathcal{C}$ which serves as an elaboration target for surface languages. $\mathcal{C}$ has a novel type system which guarantees that it can be compiled to stack-based code.
- We describe a pipeline which compiles $\mathcal{C}$ programs down to machine code. We primarily focus on how partial evaluation can be used eliminate $\mathcal{C}$'s high-level features (higher-order functions, polymorphism) at compile-time.

[1] We are assuming that `readInt` and `readInt` are externally defined primitives in both code snippets.

– We describe how one could build a practical surface language on top of $\mathcal{C}$. We cover how common features such as typeclasses (incl. monads), implicit polymorphism and data types can be implemented.

We have implemented a prototype for the approach described in this paper, whose source code is publicly available[2].

3 Core Language

3.1 Outline

This section will define the core language $\mathcal{C}$ and describe how it can be compiled to stack-based machine code. The compilation process from $\mathcal{C}$ to machine code is broken down into several distinct steps:

1. Partial Evaluation
2. Uncurrying
3. Lambda Lifting
4. Code Generation

At the core of our approach lies a *partial evaluator*. By carefully performing reductions on the input program at compile-time, we can completely eliminate high-level features such as higher-order functions (HOFs) and polymorphism. By eliminating these features at compile-time, we no longer need to deal with the problem of representing them at run-time. The result of partial evaluation is therefore a program written in a very restricted subset of $\mathcal{C}$, greatly simplifying the rest of the compilation process. So that we can guarantee that partial evaluation will eliminate all occurences of these high-level features, we slightly restrict $\mathcal{C}$ programs through the type system. These extra/augmented rules will be justified in detail throughout the remainder of this section.

After partial evaluation, we follow up with two auxiliary source-to-source transformations: uncurrying and lambda lifting. These steps remove curried and nested functions respectively, and are implemented using relatively standard algorithms. Unlike standard approaches to uncurrying [8], the type system and partial evaluator guarantee that the input is in a restricted form where function definitions are maximally η-expanded and there are no partial applications. This greatly simplifies the process, resulting in a trivial algorithm. Our approach to lambda lifting is entirely standard, and ultimately the choice of algorithm is irrelevant. Quadratic-time algorithms have been specified in detail in existing literature [20].

Finally, the result of lambda lifting is compiled to machine code. At this point in the pipeline the language is extremely restricted, with only first-order top-level function definitions remaining. Compilation is therefore trivial using standard techniques for first-order languages – our sample implementation uses a straightforward conversion to Rust [13] to complete the compilation process.

[2] https://github.com/eayus/heapless

3.2 Syntax

The core language $\mathcal{C}$ is an extension of System F_ω with let-bindings (both regular and recursive), products, base types, and primitive operations. We assume the obvious strict semantics. The full syntax is given in Figure 1. In general, we use a single colon for annotating expressions with their type (e.g. $e : \tau$), and we use a double colon for annotating types with their kind (e.g. $\tau :: \kappa$).

Order	o	$\in$	$\{1, 2, 3\}$	
Kind	κ	$::=$	$\star_o$	type
		$\mid$	$\kappa \to \kappa$	type constructor
Base Type	ι	$::=$	`Unit`	unit
		$\mid$	`Int`	integer
		$\mid$	`Bool`	boolean
		$\mid$	`World`	world token
Type	τ	$::=$	ι	base type
		$\mid$	$\tau \to \tau$	function type
		$\mid$	$\forall \alpha :: \kappa . \tau$	quantification
		$\mid$	α	type variable
		$\mid$	$\lambda \alpha :: \kappa . \tau$	type abstraction
		$\mid$	$\tau\ \tau$	type application
		$\mid$	$\tau \times \tau$	product
Expr	e	$::=$	x	variable
		$\mid$	$\lambda x : \tau . e$	application
		$\mid$	$e\ e$	application
		$\mid$	$\Lambda \alpha :: \kappa . e$	type abstraction
		$\mid$	$e\ \tau$	type application
		$\mid$	$\langle e, e \rangle$	pair
		$\mid$	`let` $\langle x, y \rangle$ `=` e `in` e	pair destructuring
		$\mid$	`let` x `=` e `in` e	let binding
		$\mid$	`let rec` x `=` e `in` e	recursive let binding
		$\mid$	`if` e `then` e `else` e	branching
		$\mid$	$\langle\rangle$	unit
		$\mid$	n	numeral
		$\mid$	`true` $\mid$ `false`	truth
		$\mid$	op $\overrightarrow{e}$	primitive
Context	Γ	$::=$	$\emptyset$	empty context
		$\mid$	$\Gamma, \alpha :: \kappa$	bind type
		$\mid$	$\Gamma, x : \tau$	bind term

Fig. 1. Syntax of the core language $\mathcal{C}$.

The reason for choosing System F_ω as the base for our core language is that it is expressive enough to support all of the high-level features we are interested in via trivial elaboration. Notably, the type-level evaluation facilities of System F_ω are used to implement typeclasses, discussed in detail in Section 6.3.

As a further syntactic convention, throughout the paper we will often give examples in Haskell-like syntax rather than $\mathcal{C}$. This is to aid readability, as programming directly in a core language is very terse. We assume the natural elaboration from the Haskell-like syntax to $\mathcal{C}$, which mostly amounts to inferring things like explicit type applications and type binders; inferring whether a definition is bound with `let` or `let rec`; and ASCII rather than unicode syntax (i.e. `Type1` rather than $\star_1$).

3.3 Type System

The kinding and typing rules for $\mathcal{C}$ are given in Figures 2 and 3 respectively. The typing rules for primitive operations (including constants and arithmetic/logic operations) are omitted for brevity, assuming the usual types.

The most unusual part of the type system is the kind $\star_o$, indexed by an *order* o. The full justification for this system is given in Section 4.5, but we give an intuitive overview here. The basic premise is that we want to differentiate first-order and higher-order terms. Types are categorised as follows.

- $\star_1$ Basic types, such as `Int`, `Bool`, etc.
- $\star_2$ First-order function types, modulo currying. `Int -> Int -> Bool` is included in this kind, but `(Int -> Int) -> Int` is not.
- $\star_3$ Everything else, including higher-order function types and types using universal quantification.

Intuitively, the kinds form a hierarchy where kinds with a higher number are more lenient in the types they allow. This hierachy is realised through the sub-typing rule for kinds KIND-SUB, meaning each kind is included in the one above.

$$\textsc{KIND-BASE}\ \frac{}{\Gamma \vdash \iota :: \star_1} \qquad \textsc{KIND-FORALL}\ \frac{\Gamma, \alpha :: \kappa \vdash \tau :: \star_i}{\Gamma \vdash (\forall \alpha :: \kappa . \tau) :: \star_3} \qquad \textsc{KIND-VAR}\ \frac{(\alpha :: \kappa) \in \Gamma}{\Gamma \vdash \alpha :: \kappa}$$

$$\textsc{KIND-ARROW}\ \frac{\Gamma \vdash \tau_1 :: \star_i \qquad \Gamma \vdash \tau_2 :: \star_j \qquad k = max(min(i+1, 3), j)}{\Gamma \vdash \tau_1 \to \tau_2 :: \star_k}$$

$$\textsc{KIND-ABS}\ \frac{\Gamma, \alpha :: \kappa_1 \vdash \tau :: \kappa_2}{\Gamma \vdash (\lambda \alpha :: \kappa_1 . \tau) :: \kappa_1 \to \kappa_2} \qquad \textsc{KIND-APP}\ \frac{\Gamma \vdash \tau_1 :: \kappa_1 \to \kappa_2 \qquad \Gamma \vdash \tau_2 :: \kappa_1}{\Gamma \vdash \tau_2 :: \kappa_1}$$

$$\textsc{KIND-PROD}\ \frac{\Gamma \vdash \tau_1 :: \star_i \qquad \Gamma \vdash \tau_2 :: \star_j \qquad k = max(i, j)}{\Gamma \vdash \tau_1 \times \tau_2 :: \star_k} \qquad \textsc{KIND-SUB}\ \frac{\Gamma \vdash \tau :: \star_i \qquad i < j}{\Gamma \vdash \tau :: \star_j}$$

Fig. 2. Kinding rules for $\mathcal{C}$.

$$\textsc{type-var}\ \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{type-abs}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \textsc{type-app}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\textsc{type-forall}\ \frac{\Gamma, \alpha :: \kappa \vdash e : \tau}{\Gamma \vdash (\Lambda\alpha :: \kappa.e) : (\forall \alpha :: \kappa.\tau)} \qquad \textsc{type-inst}\ \frac{\Gamma \vdash e : \forall \alpha :: \kappa.\tau_1 \qquad \Gamma \vdash \tau_2 :: \kappa}{\Gamma \vdash e\ \tau_2 : \tau_1[\alpha \mapsto \tau_2]}$$

$$\textsc{type-prod}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad \textsc{type-let-prod}\ \frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2 \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash e_2 : \tau_3}{\Gamma \vdash (\texttt{let}\ \langle x, y \rangle = e_1\ \texttt{in}\ e_2\ ) : \tau_3}$$

$$\textsc{type-let}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \qquad \Gamma \vdash \tau_1 :: \star_2}{\Gamma \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ ) : \tau_2}$$

$$\textsc{type-let-rec}\ \frac{\Gamma, x : \tau_1 \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash (\texttt{let rec}\ x = e_1\ \texttt{in}\ e_2\ ) : \tau_2} \qquad \textsc{type-if}\ \frac{\Gamma \vdash e_b : \texttt{Bool} \qquad \Gamma \vdash e_t : \tau \qquad \Gamma \vdash e_f : \tau}{\Gamma \vdash \texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f : \tau}$$

Fig. 3. Typing rules for $\mathcal{C}$.

Most of the rules are straightforward, however some of the rules require some special attention. In general, types with a high order should be *contagious*, meaning that if they are used as a component of a composite type, then the composite type should also have a kind with a high order. For example, a product type where one of the elements is a higher order function (of kind $\star_3$) must also be considered higher order, and hence be given the kind $\star_3$. Formally, we represent this in the typing rules as taking the maximum of two orders (see *max* in KIND-PROD and KIND-ARROW). The typing rule TYPE-LET-REC makes use of the kinding information attached to types to enforce that recursive functions must not be higher-order – the justification behind this rule is described in detail in 4.4.

3.4 Handling IO

IO functions are an important part of any practical language. In general, impurity does not mix well with program manipulation and partial evaluation – even something as simple as inlining a let-binding can become problematic if the bound expression has a side-effect. The most popular solution in modern functional programming is the IO monad, but it isn't trivial to represent at run-time without a heap. Instead, we thread a linear[3] `World` token throughout the program, similar to the approach used by Clean [14]. At run-time the `World` token

[3] In this context, the distinction between linear and unique types is irrelevant, as only one `World` will ever be present.

can be treated as the unit type. In Section 6.4 we show how to reintroduce the IO monad in the surface language userspace.

Note that the typing rules presented in this paper do not enforce the linearity of the world token – this is done intentionally to keep the presentation of the type system concise. However, real implementations should enforce this property properly, as we do in our sample implementation. We use an approach which supports linearity "on the arrow" using multiplicities, almost identical to the approach used in Linear Haskell [4].

4 Partial Evaluator

4.1 Rules

We characterise the partial evaluator by a set of reduction rules. The notation $t \rightsquigarrow u$ means that subterms matching the pattern t should be replaced with u during partial evaluation. Unless specified otherwise, there are no restrictions on where these reductions can take place, meaning that we reduce under binders by default. There is no stipulation on the order that reductions should take place, as the rules are always designed to preserve confluence. Occasionally, rules may have an attached side-condition which restricts when the rule should be applied – this means the rules may not form a true *rewriting system* [9], but we borrow the notation and terminology for conciseness.

Figures 4 and 5 detail the partial evaluator's reduction rules for types and expressions respectively.

$$(\lambda\alpha :: \kappa.\tau_1)\ \tau_2 \rightsquigarrow \tau_1[\alpha \mapsto \tau_2] \tag{1}$$

Fig. 4. Reduction rules for partially evaluating types.

$$e \rightsquigarrow \lambda x.e\ x \quad \text{if } e : \tau_1 \to \tau_2 \quad \textit{(}\eta\textit{)}$$
$$(\lambda x.e_1)e_2 \rightsquigarrow e_1[x \mapsto e_2] \quad \textit{(}\beta\textit{)}$$
$$\texttt{(if } e_1 \texttt{ then } e_2 \texttt{ else } e_3\texttt{)}\ e_4 \rightsquigarrow \texttt{if } e_1 \texttt{ then } e_2\ e_4 \texttt{ else } e_3\ e_4 \quad \textit{(if-distr)}$$
$$\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \rightsquigarrow e_2[x \mapsto e_1] \quad \text{if } e_1 \text{ is higher-order} \quad \textit{(let-inl)}$$
$$\texttt{(let } x \texttt{ = } e_1 \texttt{ in } e_2\texttt{)}\ e_3 \rightsquigarrow \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2\ e_3 \quad \textit{(let-distr)}$$
$$\texttt{(let rec } x \texttt{ = } e_1 \texttt{ in } e_2\texttt{)}\ e_3 \rightsquigarrow \texttt{let rec } x \texttt{ = } e_1 \texttt{ in } e_2\ e_3 \quad \textit{(let-rec-distr)}$$
$$\texttt{(let } \langle x, y\rangle \texttt{ = } e_1 \texttt{ in } e_2\texttt{)}\ e_3 \rightsquigarrow \texttt{let } \langle x, y\rangle \texttt{ = } e_1 \texttt{ in } e_2\ e_3 \quad \textit{(let-prod-distr)}$$
$$(\Lambda\alpha.e)\tau \rightsquigarrow e[\alpha \mapsto \tau] \quad \textit{(mono)}$$

Fig. 5. Reduction rules for partially evaluating expressions.

4.2 Elimination

Ultimately, the partial evaluator removes high-level features by applying the appropriate elimination rule. For example, higher-order functions are eliminated through β-reduction; polymorphic functions are eliminated through monomorphisation. Almost all of the other rules are there to facilitate these eliminators by rewriting expressions into a form which is more amenable to further reduction. The most straightforward example of a "facilitating rule" is let-inlining (*let-inl*). Consider the following expression (assume g is a free variable of the appropriate type):

$$\begin{array}{l}\texttt{let } \mathit{twice} = \lambda f : \texttt{Int} \rightarrow \texttt{Int}.\ \lambda x : \texttt{Int}.\ f\ (f\ x) \\ \texttt{in } \mathit{twice}\ g\ n\end{array}$$

At the moment there are no possible β-reductions, yet a higher-order function remains. We must therefore inline the higher-order function bound by `let` so that β-reduction can take place. After running the above program through the partial evaluator we are left with a first-order residual program:

$$g\ (g\ n)$$

The same idea applies to polymorphic functions which quantify over type variables – they must be inlined and monomorphised. This process is closely related to approaches which specialize higher-order functions to their functional arguments [6]. A deeper comparison with specialization is given in Section 7.2.

4.3 Distribution

We must be careful that other constructs in the core language do not interfere with the elimination of high-level features such as higher-order functions and polymorphism. Specifically, language constructs satisfying all of the following criteria must be paid extra attention:

- The construct is supported at run-time and may appear in the residual program.
- The construct can have a higher-order type if a higher-order function is used as direct sub-expression of the construct.

Intuitively, the reasoning behind this criteria is that a higher-order function may get "stuck" as a sub-expression of the construct – if the construct may appear in the residual program, it is possible that the higher-order function will never get reduced and eliminated.

There are two constructs satisfying this criteria: if-expressions and let-bindings (of all forms). The rules for distribution are explained in the following subsections.

If-Expressions If-expressions allow any type of data in their branches as long as both branches have the same type. We therefore should consider what happens if both branches contain higher-order functions, like in the following expression:

$$\texttt{(if } e_0 \texttt{ then } \lambda x.e_1 \texttt{ else } \lambda y.e_2\texttt{)} \ \lambda z.e_3$$

With just β-reduction, there is no way to make progress, yet higher-order functions still remain in the residual program! One might try adding extra reduction rules for `if`:

$$\texttt{if true then } e_t \texttt{ else } e_f \rightsquigarrow e_t$$
$$\texttt{if false then } e_t \texttt{ else } e_f \rightsquigarrow e_f$$

However, if e_0 is a free variable (whose value may not be known until runtime), then we are still stuck. Instead, we add the rule *if-distr* which distributes applications into the branches of the `if`. The higher-order functions can then be β-reduced and eliminated, leaving the following residual expression:

$$\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2$$

Let Bindings Our rule for inlining let-bindings covers the case when the bound expression is a higher-order function, but what about when the body of the `let` is a higher-order function? The following expression shows that there is a similar problem as with if-expressions:

$$\texttt{(let } n \texttt{ = 0 in } \lambda f.e_1\texttt{)} \ \lambda x.e_2$$

The partial evaluator will not inline the `let` as it does not bind a higher-order term to avoid code duplication. We are once again stuck, with no way to β-reduce the higher-order function bound in the body of the `let`. The solution is analogous to the one for if-expressions: introduce a rule which distributes applications inside the `let`'s body. The rules *let-distr*, *let-rec-distr* and *let-prod-distr* implement this behaviour for the various types of let-bindings.

4.4 Recursion

Practical functional programming languages almost always support general recursion, usually in the form of recursive definitions. General recursion is a particularly interesting feature as it makes the language turing-complete, and means that the language is no longer strongly-normalizing. This can cause problems for a partial evaluator, as we can no longer blindly reduce expressions without risking non-termination (this is true even if the input program is total).

In general, any rule which indiscriminantly unfolds recursion (i.e. `let rec` bindings) will be problematic. However, if we do not unfold recursion, then we run into problems when recursion is used in combination with higher-order

functions. If we define a function which is both higher-order and recursive, then it will remain in the residual program (which should be first-order). Unlike non-recursive higher-order functions, we cannot inline and reduce the definition due to the `let rec` blocking further evaluation.

To resolve this, we introduce the following simple restriction: *recursive functions must not be higher-order*. This is realised through the kind system, where the typing rule TYPE-LET-REC requires that the type of the recursive definition must have kind $\star_2$. The reason why we use the kind system to enforce this rather than a syntactic check is explained in Section 4.5.

With this set up, no problematic reduction rules are needed for unfolding recursion, as we can only bind first-order functions (which are easy to compile). However, this restrction does seem impractical: plenty of useful functions are both higher-order and recursive! Fortunately there is a systematic transformation that we can make to the majority of these functions, which will allow them to be accepted by the type system. More precisely, we can try to extract out the recursive part of the function into a local definition which is first-order. As an example, suppose we wanted to write the following function:

```
nTimes :: Int -> (Int -> Int) -> (Int -> Int)
nTimes 0 f x = x
nTimes n f x = f (nTimes (n - 1) f x)
```

In its current form, the function is both higher order and recursive, and so would not be permitted by our type system. However, we can refactor it into the following equivalent definition which would be permitted:

```
nTimes :: Int -> (Int -> Int) -> (Int -> Int)
nTimes n f = go n
  where
    go :: Int -> Int -> Int
    go 0 x = x
    go m x = f (go (m - 1) x)
```

The local definition `go` is recursive but not higher-order, while the exposed outer definition `nTimes` is higher-order but not recursive.

It is possible to do this refactoring for the majority of higher-order functions. Precisely, as long as the functional argument (in this case, `f :: a -> b`) remains constant throughout all recursive calls, the refactoring is possible and straightforward[4].

4.5 Polymorphism

Another trademark feature of modern functional programming languages is *polymorphism*. So that one function can operate on many different types of data, most functional languages use a uniform representation for all data – a pointer

[4] A practical compiler might choose to perform this refactoring automatically.

to a heap allocated cell. Polymorphic functions can then indiscriminantly treat all inputs in the same way, regardless of type. Since this is not an option without a heap, we use a variant of the other predominant method: *monomoprhisation*. This is realised through the *mono* rule in the partial evaluator.

Recursive Functions Like higher-order functions, we are relying on polymorphism being eliminated at compile-time by the partial evaluator. However, once again, recursive function definitions can cause issues when used in conjunction with polymorphism. As already discussed, recursive functions cannot be unfolded due to issues with termination. Unfortunately, this means that any function that is both recursive and polymorphic will remain in the residual program, as the partial evaluator cannot monomorphise without inlining first. Similar to higher-order functions, the typing rule TYPE-LET-REC prevents this from happening as the kind of any type which uses universal quantification will be $\star_3$.

To implement standard functions which are both polymorphic and recursive we can employ a similar trick to the one for higher-order functions: abstract out the recursive part into an inner auxiliary definition. This is always possible provided recursive calls are made with the same type variables to what the function was given originally (i.e. no *polymorphic recursion*). For example, we can improve our earlier example `nTimes` by making it polymorphic:

```
nTimes :: forall (a :: Type1). Int -> (a -> a) -> (a -> a)
nTimes n f = go n
  where
    go :: Int -> a -> a
    go 0 x = x
    go m x = f (go (m - 1) x)
```

Note how, even though the function `go` uses type variables, because it does not use universal quantification, it is still considered a first-order function (reliant on the fact that the type variable `a` has kind $\star_1$).

Kind System In addition to the direct interaction between polymorphism and recursion, there is a more subtle nuance. When the typing rule for `let rec` was introduced, you may have wondered why a kind system is used to ensure the function is first-order (modulo currying), as opposed to a syntactic condition. There are a few reasons for this.

Primarily, the problem with a syntactic check relates to polymorphic type variables. It raises the question – should a type variable be considered a first-order or higher-order type? Consider a recursive function which takes an argument whose type is a type variable[5]:

[5] Note that in its current form, the function `choose` is invalid as it does not give a kind for the type variable `a`. This is done intentionally to demonstrate the necessity of the kind system.

```
choose :: forall a. Int -> a -> a -> a
choose = go
  where
    go :: Int -> a -> a -> a
    go 0 x y = x
    go n x y = go (n - 1) y x
```

Depending on whether the variable `a` is later instantiated with a basic type or function type, the type of the monomorphised function may be either first-order (modulo currying) or higher-order. Because of this, we cannot treat all type variables as being basic types: doing so would allow the programmer to circumvent the restrictions on higher-order functions by hiding functions types behind a type variable. By separating type variables based on whether they represent functions or not, we can reason about whether a polymorphic function definition is truly first-order for every possible type instantiation.

4.6 Semantics

An obvious consequence of introducing general recursion is that input programs are no longer guaranteed to always terminate. Not only does this cause issues with the termination of the partial evaluator itself, but it can also pose problems related to the preservation of program semantics. In general, we would expect the program resulting from partial evaluation to have the same semantics as the input program. However, since general recursion introduces the implicit side effect of non-termination, this is not always the case. Consider the following program:

$$\begin{aligned}&\texttt{let rec } f = \lambda(x : \texttt{Int}).fx\\&\texttt{let } \omega = f\ 0\\&\texttt{in } 0\end{aligned}$$

The function f will never terminate when it is called. We then call f and bind it to variable using `let`.

This program will have different semantics before and after partial evaluation. Before, the program would not terminate: we would be forever stuck trying to evaluate the local variable ω (as the language is strict). However, partial evaluation will inline the unused variable ω, resulting in a residual expression which clearly terminates.

The root of the problem is that non-termination is an untracked side effect, making the language impure. We propose two ways of handling this problem:

- Modify the language so it is pure and total by introducing a termination checker.
- Only provide the guarantee of semantics preservation for programs that terminate.

For the purpose of this paper we choose the latter, though a practical implementation may prefer to use a termination checker so that the compiler is more predictable. If the programmer wishes to intentionally perform an infinite loop, then this is still possible by using an `IO` primitive.

4.7 Further Optimisation

In addition to eliminating high-level constructs, partial evaluation can also be used as a general optimisation method. The partial evaluator presented in this paper uses only the minimal rules necessary to eliminate high-level constructs, but a practical implementation may wish to add additional rules for optimisation purposes.

A simple example is to add rules which implement *constant folding*, a process which reduces primitive operations at compile-time if all operands are known. For example, $3 * (2 + 2)$ should be reduced to 12 at compile-time. Integrating this optimisation into the partial evaluator could remove the need for a separate pass later on in the compilation process. Implementing this extension is trivial, and so we do not elaborate on the details.

5 Implementation

5.1 Partial Evaluator

The most straightforward way to implement the partial evaluator is to continuously traverse the input expression, identifying and reducing expressions which match a reduction rule. For β-redexes, reduction will involve substitution, which is another traversal of the expression. It is well-known that this method of implementing normalizers/partial evaluators is inefficient due to the repeated term traversals, even if optimizations such as De-Bruijn indices [3] are used to speed up substitution.

A modern approach to implementing normalizers is *normalization by evaluation* (NbE), where expressions are first evaluated into semantic values and then later *reified* back into syntactic terms [1,2]. Expensive substitutions are avoided entirely, with environments and variable lookups being used instead. NbE has been widely employed in the implementation of type checkers for dependently typed languages such as Idris and Agda. Since normalizers and partial evaluators are fundamentally very similar, it is relatively straightforward to adapt NbE algorithms to implement our partial evaluator efficiently – this is the approach we use in our sample implementation.

There are a few ways in which our implementation differs from a conventional NbE algorithm. First, the definition of our normal and neutral terms is more lenient than usual, as redexes like let-bindings should be permitted in the result (provided the let-binding does not bind a higher-order function of course). Second, because we are working in a language based off System F_ω, we must perform both term-level and type-level reductions. This requires two distinct evaluation

environments. Third, core language terms must carry thorough typing annotations as we are compiling to a typed language backend – the algorithm must be ensure that these typing annotations are preserved during evaluation. The complete algorithm is verbose and relatively uninteresting, and so we refer the reader to our sample implementation for the details.

5.2 Uncurrying

The goal of this step in the compilation pipeline is to remove curried functions by transforming them to ones which operate on tuples instead. Due to the η rule in our partial evaluator, functions will always be maximally η-expanded and maximally applied. This makes this process extremely trivial, and we implement this as two simple syntactic transformations:

$$f t_1 t_2 \dots t_n \rightsquigarrow f(t_1, t_2, \dots t_n)$$
$$\lambda x_1 . \lambda x_2 . \dots \lambda x_n . t \rightsquigarrow \lambda(x_1, x_2, \dots x_n).t$$

Fig. 6. Reduction rules for uncurrying.

5.3 Lambda Lifting

At this stage the program may still have local function definitions. We remove these using the standard method of lambda lifting. This has been discussed in detail in existing literature [7, 20], and so we do not repeat the details here.

5.4 Backend

At this point in the compilation pipeline, we have a relatively simple first-order language. We offload the remainder of the compilation process to a backend – there are many possible choices, but in our sample implementation we generate Rust code[6]. This would be an unusual choice for a production language as the Rust compiler will also perform type checking, which is unnecessary and inefficient. However, it is well suited to a prototype implementation as Rust supports first class tuples and nested let-bindings, of which both are utilized in $\mathcal{C}$. Compared to an alternative backend like LLVM [17], this simplifies the compilation process significantly as we do not need extra passes to handle these features.

[6] Note that we do not depend on any code in the standard library which uses the heap.

6 Elaboration

Rather than continuously extend our partial evaluator with new features, we use a common architectural pattern: keep the core language small, but have a more expressive surface language which *elaborates* down to the core language. This section details several common features which can be implemented this way which cumulatively form a Haskell-like language.

6.1 Implicit Polymorphism

An easy extension to our core language which significantly improves practical usability is *implicit polymorphism*, where type applications and abstractions may be omitted and left to be inferred by the compiler. This is a well understood idea and so we do not discuss this in detail. One concrete approach is to use a modification of the Hindley Milner algorithm [18] which produces type annotated terms (elaboration).

6.2 Algebraic Data Types

Non-Recursive Non-recursive algebraic datatypes are trivial to implement via a simple sum-of-products representation. Record and product types can be represented by nested product types, and sum types may be implemented using tagged unions.

Recursive Implementing recursive algebraic datatypes is significantly more challenging. Clearly, it is not possible to represent these types at runtime as they have no fixed size and we do not have access to a heap[7]. However, like many other features such as higher-order functions, we can permit recursive datatypes in the surface language provided we can guarantee that they will be eliminated at compile time. A remarkably simple way of achieving this is to use Church-encoding to represent recursive datatypes as functions (à la System F), transformed during the elaboration step. We already know how to deal with higher-order functions in the core language, and so this can be implemented with little complexity. For example, after elaboration, the standard definition of a linked list would be represented by the following type:

$List \equiv \lambda(A :: \star_3).\forall X.X \to (A \to X \to X) \to X$

There are a few important technical details. First, in the surface language any recursive algebraic data type must have kind $\star_3$, as it will eventually be elaborated into a polymorphic higher-order function. Second, elimination of recursive algebraic datatypes through recursive functions should be disallowed, and instead a primitive `fold` operator should be introduced. The fold operator can be elaborated down into a simple application of the church-encoded function.

This idea of elaborating data types into their church encoding so that they will be eliminated at compile time is essentially a form of *deforestation* [22].

[7] The lack of a heap means that we cannot use the conventional linked representation.

Our approach is much more simplistic than many existing approaches to deforestation, mainly because we do not allow arbitrary recursive definitions (instead exposing a primitive `fold`).

6.3 Typeclasses

Typeclasses are another common feature in surface-level languages which we can support via elaboration. Typeclass constraints can be viewed as implicit function arguments, where the typeclass's functions are stored as a record – this technique is commonly referred to as "dictionary passing". For example, suppose we had a `Number` class:

```
class Number (a :: Type1) where
  add :: a -> a -> a
  sub :: a -> a -> a
```

This could be elaborated to the following record type:
$\texttt{Number} \equiv \lambda(a :: \star_1).(a \to a \to a) \times (a \to a \to a)$

As with all features introduced via elaboration, we must be careful to ensure that the elaborated code is always valid with respect to the core language's typing rules. Since a typeclass constraint may eventually be elaborated into a higher-order type, then it means that in the surface language types using typeclasses should live in $\star_3$. To write a recursive function which uses typeclasses, we must apply our classic transformation of factoring out the recursive part of the function into a local definition.

6.4 Monadic Example

We now culminate these features into a simple high-level example program. Suppose we want to write a program which reads in two numbers from the command line, n and k, and then prints out k for n repetitions. Since this program heavily interacts with IO, it would make sense to write our program within the IO monad. Fortunately, despite not being a primitive in the language, we can still use the IO monad by building it in userspace:

```
newtype IO (a :: Type1) = IO { runIO :: World -> (World, a) }
```

The `IO` type has trivial instances for `Functor`, `Applicative` and `Monad`. There is some nuance in choosing the right kind for these typeclasses – we discuss this in more detail in Section 8.1, but for now we take a pragmatic approach and assume that all monads have kind $\star_1 \to \star_2$. See the appendix for the complete example code, including the definitions for the typeclasses and their instances.

```
instance Functor IO where ...
instance Applicative IO where ...
instance Monad IO where ...
```

Primitive functions should be wrapped in a format which is compatible with the IO monad.

```
readInt :: IO Int
readInt = IO $ \w -> primReadInt w

printInt :: Int -> IO ()
printInt n = IO $ \w -> (primPrintInt n w, ())
```

We can define the familiar `replicateM`, despite it being both higher-order and recursive:

```
replicateM_ :: forall m a. Monad m => Int -> m a -> m ()
replicateM_ n x = go n
  where
    go :: Int -> m ()
    go 0 = pure ()
    go m = do
      x
      go (m - 1)
```

Finally, we can write our program in idiomatic functional style, with no worries that it will be using the heap at runtime. Note that up until now, all of the code that we have written would likely be part of a standard library, and so the programmer would directly skip to this last step. Our program looks identical to regular Haskell code, which we believe is a large benefit of our approach.

```
printK :: IO ()
printK = do
  n <- readInt
  k <- readInt
  replicateM_ n (printInt k)
```

7 Related Work

7.1 Multi-Stage Programming

Partial evaluators and multi-stage languages have always been closely related. While the former lets the compiler automatically manipulate the program [10], the latter gives full control to the programmer and allows them to choose which reductions will happen at compile-time [21]. Kovács's recent work [16] has shown how a heterogenous multi-stage language can be used to generate programs which contain no closures, a goal shared by this paper. We believe there are advantages to both Kovács's and our approaches, inherited from the fundamental differences between multi-stage programming and partial evaluation respectively – our approach has less syntactic overhead as we do not require extra staging annotations, but the programmer no longer has full control over the reductions that happen at compile-time.

7.2 Higher-Order Removal

The removal of higher-order functions by translating to a first-order representation has been very well-studied. The usual motivation is to simplify the compilation process as first-order programs are much simpler to compile to machine-code. Unfortunately, existing methods for removing higher-order functions are not applicable to our use-case of heapless programming, as in the process of removing higher-order functions they introduce other language constructs which also require the heap at run-time.

Defunctionalization Defunctionalization observes that there are only a finite amount of lambda abstractions in a program, and so to represent a function we can enumerate all of the possibilities encoded in an algebraic data type. If a lambda captures variables from the surrounding scope, parameters should be added to the corresponding ADT constructor representing the captured data.

For example, if a program contained two lambda abstractions:

$$\Gamma \vdash \lambda x : \texttt{Int}.x + x \tag{2}$$

$$\Gamma, y : \texttt{Int} \vdash \lambda x : \texttt{Int}.x + y \tag{3}$$

Then we encode functions by the following ADT:

```
data Func
  = Lam1      -- captures nothing
  | Lam2 Int  -- captures y : Int
```

The problem with this method for our use-case occurs when one function captures another function from the surrouding scope. The captured function must be added as a parameter to the data type, making the data type recursive. This would require a heap at runtime. More recent work such as *lambda set specialization* [5] does not change anything in this regard.

Closure Conversion The predominant method for compiling higher-order functions is closure-conversion, where functions are represented by a function pointer and values for any captured variables [19]. Since different functions (with the same type) may capture different amounts of variables, closures are variably sized, and so the usual approach is to allocate them on the heap behind a uniformly sized pointer.

While heap-allocated closures are obviously not an option without a heap, some low-level languages like Rust and C++ support stack-allocated closures instead. The basic idea is to allocate closures on the stack when they are created, and then pass them by reference so that they have a uniform representation. There are a few undesirable complications introduced with this method:

- When one closure captures another, the first closure is stored by reference in the second. This means that the second closure must not outlive the first, or we will access a dangling reference. Rust solves this with its signature "lifetimes", but we believe such an approach would be quite unwieldy given the ubiquity of closures in functional programming.
- Since different closures of with same type may be different sizes, every closure is assumed to have a unique (mutually incompatible) type. For example, an if-expression which returns different closures in different branches will be rejected by the type checker, since the types of branches will be different too. The approach in this paper does not suffer from this problem due to distribution rules (Section 4.3).

Specialization Our approach is closely related to *specialization*, which generates new definitions for higher-order functions every time they are used with a new higher-order argument [6]. Unlike existing approaches to specialization, our type system ensures complete removal of higher-order functions, rather than a "best effort" approach which aims to remove as much as possible. Specializing methods are better at avoiding duplication of code, as they do not require inlining of every higher-order definition. This difference becomes important when the same function is called with the same higher-order argument in many different places: our approach will inline the definition many times, while specialization will produce a single top-level definition which is re-used. While this may seem like a significant disadvantage, in practice most non-recursive higher-order functions are only a few operations long, and so are prime candidates for inlining anyway.

On the other hand, there are other areas in which our approach appears to improve over specializing approaches. For example, our method can eliminate higher-order functions which appear in data structures such as products, which is vital for implementing typeclasses in a surface language (Section 6.3). We also cover how to handle other language features that may usually use the heap, such as the IO monad. We believe our approach is simpler to implement than most specializing algorithms, as it is a simple modification of normalization rather than an entirely bespoke algorithm.

8 Future Work

8.1 Order Polymorphism

In the current formalization of the core language, a polymorphic term must fix a specific kind for the type variables it quantifies over (e.g. $\star_1$, $\star_2$ etc.). This can be undesirable as often we want to use the same function or datatype for many different kinds. For example, consider the reader monad:

$Reader \equiv \lambda R.\lambda A.R \to A$

What kind should we give this type? There are many different possibilities: $\star_1 \to \star_1 \to \star_2$ and $\star_2 \to \star_2 \to \star_3$ are just two examples. It is undesirable to

repeat ourselves and write multiple versions of the same type that only differ by kind, and so it would be better if the core language provided a way to write a single polymorphic definition.

Looking at the typing rules for the $\rightarrow$ type constructor, the most general kind for this would be polymorphic over *orders* (i.e. the number that the kind $\star$ is indexed by). We propose a system very similar to the universe level polymorphism [15] we see in dependently type languages like Agda. Constructs such as $\uparrow$ (increment order) and $\sqcup$ (maximum of two orders) could be exposed in the syntax for orders, along with type constructors which quantify orders. The kind for the reader monad could then be specified as $\forall ij.\star_i \rightarrow \star_j \rightarrow \star_{(\uparrow i)\sqcup j}$, which is maximally general. The details of such a system are yet to be worked out, however given the stark similarity to universe levels, we believe that such a system is feasible.

8.2 Heap Effect

So far we have discussed the design of a language which avoids using the heap entirely, so that it can be ran in environments where a heap would be inefficient or impractical. However, real-world applications are unlikely to be so clear-cut, and sometimes we may actually have (partial) access to the heap. To better support this use-case, we could allow finer grained control over what parts of the program use the heap.

Rather than banning heap usage across the entire program, a monad or algebraic effect [12] could be used to track heap usage in the type system. Functions which are disallowed in our core language (i.e. recursive and higher-order definitions) could be allowed on the condition that their type indicates that they use the heap. A handler for the heap effect could be implemented in different ways depending on the run-time environment, potentially after some stack-based initialisation code. For example, an operating system might first query the system memory map, identify a free location, set up a heap, and finally handle the heap effect so that subsequent code can use the heap. This is only possible in a language which supports modularity over where the heap is used.

9 Conclusion

We have introduced a novel approach for designing pure functional languages which do not require the heap at runtime. We have shown how, supported by an appropriate type system, partial evaluation can be used to entirely eliminate the high-level features of a language at compile time. We hope that this work will help facilitate more research into functional languages suitable for systems programming.

References

1. Abel, A., Chapman, J.: Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types. arXiv preprint arXiv:1406.2059 (2014)
2. Berger, U., Eberl, M., Schwichtenberg, H.: Normalization by evaluation. Prospects for Hardware Foundations: ESPRIT Working Group 8533 NADA—New Hardware Design Methods Survey Chapters pp. 117–137 (1998)
3. Berghofer, S., Urban, C.: A head-to-head comparison of de bruijn indices and names. Electronic Notes in Theoretical Computer Science **174**(5), 53–67 (2007)
4. Bernardy, J.P., Boespflug, M., Newton, R.R., Peyton Jones, S., Spiwack, A.: Linear haskell: Practical linearity in a higher-order polymorphic language. Proceedings of the ACM on Programming Languages **2**(POPL), 1–29 (2017)
5. Brandon, W., Driscoll, B., Dai, F., Berkow, W., Milano, M.: Better defunctionalization through lambda set specialization. Proceedings of the ACM on Programming Languages **7**(PLDI), 977–1000 (2023)
6. Chin, W.N., Darlington, J.: A higher-order removal method. Lisp and Symbolic Computation **9**, 287–322 (1996)
7. Danvy, O., Schultz, U.P.: Lambda-lifting in quadratic time. In: International Symposium on Functional and Logic Programming. pp. 134–151. Springer (2002)
8. Hannan, J., Hicks, P.: Higher-order uncurrying. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 1–11 (1998)
9. Jay, C.B.: Long bn normal forms and confluence. Tech. rep., Tech. Rep. LFCS-91-183 (and its revised version) (1991)
10. Jones, N.D.: An introduction to partial evaluation. ACM Computing Surveys (CSUR) **28**(3), 480–503 (1996)
11. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Peter Sestoft (1993)
12. Karachalias, G., Koprivec, F., Pretnar, M., Schrijvers, T.: Efficient compilation of algebraic effect handlers. Proceedings of the ACM on Programming Languages **5**(OOPSLA), 1–28 (2021)
13. Klabnik, S., Nichols, C.: The Rust Programming Language. No Starch Press (2023)
14. Koopman, P., Plasmeijer, R., van Eekelen, M., Smetsers, S.: Functional programming in clean (2002)
15. Kovács, A.: Generalized universe hierarchies and first-class universe levels. arXiv preprint arXiv:2103.00223 (2021)
16. Kovács, A.: Closure-free functional programming in a two-level type theory. Proceedings of the ACM on Programming Languages **8**(ICFP), 659–692 (2024)
17. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: International symposium on code generation and optimization, 2004. CGO 2004. pp. 75–86. IEEE (2004)
18. Milner, R.: A theory of type polymorphism in programming. Journal of computer and system sciences **17**(3), 348–375 (1978)
19. Minamide, Y., Morrisett, G., Harper, R.: Typed closure conversion. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on principles of programming languages. pp. 271–283 (1996)
20. Morazán, M.T., Schultz, U.P.: Optimal lambda lifting in quadratic time. In: Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers 19. pp. 37–56. Springer (2008)

21. Taha, W.: A gentle introduction to multi-stage programming. In: Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers. pp. 30–50. Springer (2004)
22. Wadler, P.: Deforestation: Transforming programs to eliminate trees. In: European Symposium on Programming. pp. 344–358. Springer (1988)

Appendix

```
newtype IO (a :: Type1) = IO { runIO :: World -> (World, a) }

class Functor (f :: Type1 -> Type2) where
  fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative (f :: Type1 -> Type2) where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

class Applicative f => Monad (f :: Type1 -> Type2) where
  (>>=) :: f a -> (a -> f b) -> f b

instance Functor IO where
  fmap f (IO g) = IO $ \w -> let (w', x) = g w in (w', f x)

instance Applicative IO where
  pure x = IO $ \w -> (w, x)
  IO f <*> IO x = IO $ \w ->
    let (w', f') = f w
        (w'', x') = x w'
    in (w'', f' x')

instance Monad IO where
  (IO g) >>= f = IO $ \w -> let (w', x) = g w in runIO (f x) w'

readInt :: IO Int
readInt = IO $ \w -> primReadInt w

printInt :: Int -> IO ()
printInt n = IO $ \w -> (primPrintInt n w, ())

replicateM_ :: forall m a. Monad m => Int -> m a -> m ()
replicateM_ n x = go n
  where
    go :: Int -> m ()
    go 0 = pure ()
    go m = do
      x
```

```
      go (m - 1)

printK :: IO ()
printK = do
  n <- readInt
  k <- readInt
  replicateM_ n (printInt k)

main :: World -> World
main w = let (w', ()) = runIO printK w in w'
```