Making a fast curry
Push/enter vs eval/apply for higher-order languages

March 27, 2003

Simon Marlow and Simon Peyton Jones
Microsoft Research, Cambridge

Abstract

Higher-order languages that encourage currying are implemented using one of two basic evaluation models: push/enter or eval/apply. Implementors use their intitution and qualitative judgements to choose one model or the other.

Our goal in this paper is to provide, for the first time, a more substantial basis for this choice, based on our qualitative and quantitative experience of implementing both models in a state-of-the-art compiler for Haskell.

Our conclusion is simple, and contradicts our initial intuition: compiled implementations should use eval/apply.

1 Introduction

There are two basic ways to implement function application in a higher-order language: the *push/enter* model or the *eval/apply* model [11]. To illustrate the difference, consider the higer-order function `zipWith`, which zips together two lists, using a function `k` to combine corresponding list elements:

```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith k []     []     = []
zipWith k (x:xs) (y:ys) = k x y : zipWith xs ys
```

How should the compiler deal with the call `(k x y)` in the body of `zipWith`? It can't blithely apply `k` to two arguments, because `k` might in reality take just one argument and compute for a while before returning a function that consumes the next argument; or `k` might take three arguments, so that the result of the `zipWith` is a list of functions.

In the push/enter model, the call proceeds by *pushing* the arguments `x` and `y` on the stack, and *entering* the code for `k`. Every function's entry code is required to check how many arguments are on the stack, and behave appropriately: if there are too few arguments, the function must construct a partial application and return. If there are too many arguments, then only the required arguments are consumed, the rest of the arguments are left on the stack to be consumed later, presumably by the function that will be the result of this call.

In the eval/apply approach, the caller first *evaluates* the function `k`, and then *applies* it to the correct number of arguments. The latter step involves some run-time case analysis, based on information extracted from the closure for `k`. If `k` takes two arguments, we can call it straightforwardly. If it takes only one, we must call it passing `x`, and then call the function it returns passing `y`; if it takes more than two, we must build a closure for the partial application `(k x y)` and return that closure.

Which of the two is best in practice? The trouble is that the evaluation model has a pervasive effect on the implementation, so it is too much work to implement both and pick the best. Historically, compilers for strict languages (using call-by-value) have tended to use eval/apply, while those for lazy languages (using call-by-need) have often used push/enter, but this is 90% historical accident — either approach will work in both settings. In practice, implementors choose one of the two approaches based on a qualitative assessment of the trade-offs. In this paper we put the choice on a firmer basis:

- We explain precisely what the two models are, in a common notational framework (Section 4). Surprisingly, this has not been done before.
- The choice of evaluation model imposes a range of trade-offs, many of which are not immediately apparent. We identify and discuss these tradeoffs in Sections 5 and 6.
- We give the first detailed quantitative measurements that contrast the two approaches, based on a credible, optimising compiler (the Glasgow Haskell Compiler, GHC). These measurements, given in Section 8, give both bottom-line results such as total instruction count and allocation, and more insightful numbers such as breakdowns of call patterns.

Our conclusions are these. First, on efficiency grounds there is little to choose between the two. Second, while eval/apply has similar implementation complexity to push/enter, it is more concentrated in one place, and therefore easier to work with. We have now adopted eval/apply for GHC.

There are lots of nitty-gritty details in this paper, for which we make no apology — indeed, we believe that articulating these details is one of our main contributions.

2 Background: efficient currying

The choice between push/enter and eval/apply is only important if the language encourages *currying*. In a higher-order language one can write a multi-argument function in two ways:

```
f :: (Int,Int) -> Int
f (x,y) = x*y

g :: Int -> Int -> Int
g x y = x*y
```

Here, `f` is un-curried. It takes a single argument that is a pair, unpacks the pair, and multiplies its components. On the other hand, `g` is curried. Notionally at least, `g` takes one argument, and returns a function that takes a second argument, and multiplies the two. The type of `g` should be read right-associatively, thus:

```
g :: Int -> (Int -> Int)
```

Currying appeals to our sense of beauty, because multi-argument functions come "for free"; one does not need data structures to support them.

In any higher-order language one can *write* curried functions, simply by writing a function that returns a function, but languages differ in the degree to which their syntax *encourages* it. For the purposes of this paper, we assume that currying is to be regarded as the native way to define multi-argument functions, and that we wish to make multi-argument curried functions as fast as possible.

We said that "notionally at least `g` takes one argument", but suppose that, given the above definition of `g`, the compiler is faced with the call `(g 3 4)`. The call is to a *known function* — one whose definition the compiler can "see". It would be ridiculous to follow the currying story literally. To do that, we would call `g` passing one argument, `3`, get a function closure in return, and then call that function, again passing one argument, `4`. No, in this situation, any decent compiler must load the arguments `3` and `4` into registers, or on the stack, and call the code for `g` directly, *and that is true whether the basic evaluation model is push/enter or eval/apply.* In the rest of this paper we will take it for granted that saturated calls to "known" functions are compiled using an efficient argument-passing convention. The push/enter and eval/apply models differ only in how they handle calls to "unknown" functions.

3 Language

To make our discussion concrete we use a small, non-strict intermediate language similar to that used inside the Glasgow Haskell Compiler. Its syntax is given in Figure 1. In essence it is the STG language [11], but we have adjusted some of the details for this paper.

Although the push/enter vs eval/apply question applies equally to strict and non-strict languages, we treat a non-strict one here because it is the slightly more complicated case.

The idea is that each syntactic construct in Figure 1 has a direct operational reading. We give these operational intuitions here, and we will make them precise in Section 4:

- A literal is an *unboxed* 32-bit integer, i, or 64-bit double-precision floating-point number, d. We have more to say about unboxed values in Section 3.3.
- A call, $f^k\ a_1 \ldots a_n$, applies the function f to the arguments $a_1 \ldots a_n$. The application is in A-normal form [5] — that is, each argument is an atom (literal or variable) — so there is no argument preparation to perform first. The superscript k describes the statically-known information about the function's arity. It takes two forms:
  - f^n, where n is an integer, indicates that the compiler statically knows the arity of f, usually because there is a lexically-enclosing binding for f that binds it to a FUN object with arity n.
  - $f^\bullet$ indicates that the compiler has no static information about f's arity. It would be safe to annotate every application with $\bullet$.

  There is no guarantee that the function's arity (whether statically known or not) matches the number of arguments supplied at the call site; indeed, in a lazy language the function might turn out to be an unevaluated thunk.
- A `let` expression (and only a `let`) allocates an object in the heap. We discuss the forms of heap object in Section 3.1. In this paper we will only discuss simple, non-recursive `let` expressions. GHC supports a mutually-recursive `letrec` as well, of course, but recursive bindings do not affect the issues discussed this paper, so we omit them to save clutter. The top-level definitions of a program are recursive, however.
- A `case` evaluates a sub-expression, called the *scrutinee*, and optionally performs case analysis on its value. More concretely, `case` saves any live variables that are needed in the case alternatives, pushes a return address, and then evaluates the scrutinee. At the return address, it performs case analysis on the returned value. All `case` expressions are exhaustive: either there is an *ANY* alternative as a catch-all, or the patterns cover all the possibilities in the data type.

3.1 Heap objects

The language does not provide a syntactic form of expression for constructor applications, or for anonymous lambdas; instead, they must be explicitly allocated using `let`. In general, `let` performs heap allocation, and the right hand side of a `let` is a *heap object*. There are exactly five kinds of heap objects:

$FUN(x_1 \ldots x_n \rightarrow e)$ is a function closure, with arguments x_i and body e (which may have free variables). The function is curried — that is, it may be applied to fewer than n, or more than n, arguments — but it still has an *arity* of n. If it is applied to fewer than n arguments, no useful work can be done, and we build a partial application object (*PAP*). If it is applied to more than n arguments, it consumes the first n arguments and (guaranteed by the type system) computes a function that consumes the remaining arguments.

$PAP(f\ a_1 \ldots a_n)$ represents a partial application of function f to arguments a_i. Here, f is guaranteed to be FUN object, and the arity of that FUN is guaranteed to be strictly greater than n.

$CON(C\ a_1 \ldots a_n)$ is a data value, the saturated application of constructor C to arguments a_i.

$THUNK\ e$ represents a thunk, or suspension. When its value is needed, e is evaluated, and the thunk overwritten with (an indirection to) the value of e.

BLACKHOLE is used only during evaluation of a thunk, never in a source program. While a thunk is being evaluated, it is replaced by *BLACKHOLE* to avoid space leaks and to catch certain forms of divergence [7].

Of these, FUN, *PAP* and *CON* objects are *heap values*, and cannot be evaluated any further.

A top-level definition creates a statically-allocated object, at a fixed address, whereas a `let` allocates a heap object dynamically.

3.2 Case expressions

The language offers conventional algebraic data type declarations, such as

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
data Bool   = False | True
data List a = Nil | Cons a (List a)
```

Variables	x, y, f, g			
Constructors	C			Defined in data type declarations
Literals	lit	$::=$	$i \mid d$	Unboxed integer or double
Atoms	a, v	$::=$	$lit \mid x$	Function arguments are atomic
Function arity	k	$::=$	$\bullet$	Unknown arity
		$\mid$	n	Known arity $n \geq 1$
Expressions	e	$::=$	a	Atom
		$\mid$	$f^k\, a_1 \ldots a_n$	Function call $(n \geq 1)$
		$\mid$	$\oplus\, a_1 \ldots a_n$	Saturated primitive operation $(n \geq 1)$
		$\mid$	$\texttt{let}\ x = obj\ \texttt{in}\ e$	
		$\mid$	$\texttt{case}\ e\ \texttt{of}\ x\ \{alt_1; \ldots; alt_n\}$	$(n \geq 1)$
Alternatives	alt	$::=$	$C\, x_1 \ldots x_n \ \rightarrow\ e$	$(n \geq 0)$
		$\mid$	$ANY \ \rightarrow\ e$	Default alternative
Heap objects	obj	$::=$	$FUN(x_1 \ldots x_n \rightarrow e)$	Function (arity $= n \geq 1$)
		$\mid$	$PAP(f\, a_1 \ldots a_n)$	Partial application (f is always a FUN with $arity(f) > n \geq 1$)
		$\mid$	$CON(C\, a_1 \ldots a_n)$	Saturated constructor $(n \geq 0)$
		$\mid$	$THUNK\ e$	Thunk
		$\mid$	$BLACKHOLE$	[only during evaluation]
Programs	$prog$	$::=$	$f_1 = obj_1; \ldots; f_n = obj_n$	

Figure 1. Syntax

Values of type `Tree` are built with the constructors `Leaf` and `Branch`, and can be discriminated and taken apart with a `case` expression. The boolean type `Bool` is just a regular algebraic data type, so that a conditional is implemented by a `case` expression. Constructors are always saturated; unsaturated constructors can always be saturated by eta expansion.

To give the idea, here is the Haskell definition of the `map` function:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

and here is its rendition into our intermediate language:

```
nil = CON Nil

map = FUN (f xs ->
        case xs of
          Nil -> nil
          Cons y ys -> let h = THUNK (f y)
                           t = THUNK (map f ys)
                           r = CON (Cons h t)
                       in r
      )
```

The top-level definition of `nil` is automatically generated by GHC, so that there is a value to hand for `map` to return in the `Nil` case alternative. A similar top-level definition is generated for each nullary constructor.

The scrutinee of a `case` expression is an *expression* rather than than an *atom*. This is important, because it lets us write, for example, `case (null xs) of ...`, rather than

```
let y = THUNK (null xs) in case y of ...
```

There is no need to construct a thunk!

The main reason that `case` connects evaluation with case analysis is that it allows us to make the *vectored return* optimisation. Consider the `case` expression in the `map` example above. Since there are just two branches, the `case` can push (a pointer to) a jump table, or vector, of two return addresses, one for each branch. When returning the value, the returning code can choose the appropriate return address, eliminating the test. Whether or not vectored returns are worth the bother is an interesting question, but the idea applies equally to both push/enter and eval/apply, so we do not consider it further here.

An unusual feature of `case` is the extra binder x in its syntax.

$$e ::= \texttt{case}\ e\ \texttt{of}\ x\ \{alt_1; \ldots; alt_n\}$$

The idea is that x is bound to the value of the scrutinee. If the scrutinee was always atomic nothing would be gained but, as we have seen, it is not. Here is a slightly contrived example:

```
case (reverse xs) of y {
     Nil       -> Nil ;
     Cons x xs -> append y y
}
```

This expression evaluates `(reverse xs)`; if the result is `Nil`, it returns `Nil`, otherwise it returns the reversed list appended to itself. Since the returned value of `(reverse xs)` is present in the implementation, it makes sense to have a name for it! (Still, where the binder is irrelevant in our examples, we sometimes omit it.)

The most common application is to model call-by-value, by using `case` instead of `let`. For example, here is how we might compile the call `(f (reverse xs))` if we knew that `f` was strict:

```
case (reverse xs) of y { ANY -> f y }
```

The `case` expression is also use to implement Haskell's `seq` operator. In Haskell, the expression ``(a `seq` b)`` evaluates `a`, discards the result, and then evaluates `b`. The `seq` function is used purely for its effect (evaluation), almost invariably to control space usage. In our system, `seq` is implemented like this:

```
seq = FUN (a b -> case a of { ANY -> b })
```

It is important to realise that the first argument to `seq` can have any type at all: in particular, it may be a function or a data value.

3.3 Unboxed values

Another slightly unusual feature of our language is the use of *unboxed values*. Most values are represented by a pointer to a heap object, including all data structures, function closures, and thunks. Our intermediate language also supports a handful of primitive, unboxed data types, of which we consider only `Int#` and `Double#` here. An `Int#` is a 32-bit integer, in the native machine representation; it is not a pointer. Similarly, a `Double#` is a 64-bit double-precision floating-point value in IEEE representation. These unboxed values can be passed as a arguments to a function, returned as results, stored in data structures, and so on. For example, here is how the (boxed) type `Int` is defined, as an ordinary algebraic data type:

```
data Int = I# Int#
```

That is, an `Int` value is a heap-allocated data structure, built with the `I#` constructor, containing an `Int#`.

Having explicit unboxed values allows us to make boxing and unboxing operations explicit in our intermediate language. For example, here is how `Int` addition is defined:

```
plusInt :: Int -> Int -> Int
plusInt a b
  = case a of
      I# x -> case b of
                I# y -> case x +# y of
                          z -> I# z
```

The first `case` expression evaluates the argument `a` (in case it is a thunk) and takes it apart; the second `case` does the same to `b`; the `case x +# y of ...` adds the two unboxed values using the primitive addition operator `+#`, while the final use of `I#` boxes the result back into an `Int`.

This approach to unboxed values is described in detail elsewhere [12]. Its significance for this paper is that both heap objects and the stack may contain a mix of pointer and non-pointer values.

4 The two evaluation models

It is now time to become precise about what we mean by a "push/enter" or "eval/apply" model. We do so by giving an operational semantics that exposes the key differences between these models, while still hiding some representation details that only confuse the picture. Douence and Fradet give a completely different, combinator-based, formalism that allows them to contrast push/enter with eval/apply [2], but the one we give here maps more directly to operational intuitions.

Figure 2 gives the operational semantics for both evaluation models, using a small-step transition relation of the form

$$e_1;\ s_1;\ H_1 \ \Rightarrow\ e_2;\ s_2;\ H_2$$

The components of the program state are:

The code e, is the expression under evaluation, in the syntax of Figure 1.

The stack s, is a stack of continuations that says what to do when the current expression is evaluated.

The heap H, is a finite mapping from variables (which we treat as synonymous with heap addresses) to heap objects. The latter have the syntax given in Figure 1.

The stack continuations, k, take the following forms:

k	::=	$\texttt{case} \bullet \texttt{of}\ x\,\{alt_1;\ldots;alt_n\}$	
	|	$Upd\ t\ \bullet$	Update thunk t with returned value
	|	$(\bullet\ a_1 \ldots a_n)$	Apply the returned function to $a_1 \ldots a_n$ [eval/apply only]
	|	$Arg\ a$	Pending argument [push/enter only]

The meaning of these continuations should become clear as we discuss the evaluation rules. The rules themselves are fairly dense, so the following subsections explain them in some detail. After that, we sketch how the operational semantics is mapped onto a real machine by the Glasgow Haskell Compiler.

4.1 Rules common to both models

The first block of evaluation rules in Figure 2 are common to both push/enter and eval/apply.

The first rule, LET, says what happens when the expression to be evaluated is a `let` form. Following Launchbury [8], we simply allocate the right-hand side obj in the heap, using a fresh name x', extend the heap thus $H[x' \mapsto obj]$. The use of a fresh name corresponds to allocating an unused address in the heap. Lastly, we substitute x' for x in e, the body of the `let`, before continuing. In a real implementation this substitution would be managed by keeping a pointer to the new object in a register, or accessing it by offset from the allocation pointer, but we do not need to model those details here.

The next group of four rules deal with `case` expressions. Rule CASE, starts the evaluation of a `case` expression by pushing a `case` continuation on the stack, and evaluating the scrutinee, e. When evaluation is complete, a value v (either a literal or a pointer to a heap value) is returned to the `case` continuation by RET.

If v is (a pointer to) a constructor, rule CASECON applies; it resumes the appropriate branch of the `case`, binding the constructor arguments to x_i and the constructor application itself to x. If the returned value does not match any other `case` alternative, the *ANY* alternative is used (rule CASEANY). These two rules precede CASE because they overlap it, and we use the convention that the first applicable rule takes precedence. To reduce clutter, we use the (slightly unusual) convention that no binding is ever removed from the heap. For example, in rule CASECON the heap H on the right-hand side of the rule still has a binding for v.

The next two rules deal with thunks. If the expression to be evaluated is a thunk, we push an update continuation (or *update frame*), $Upd\ t\ \bullet$, which points to the thunk to be updated (rule THUNK). While the thunk t is being evaluated we update the heap so that t points to a *BLACKHOLE*. No left-hand sides match *BLACKHOLE* so evaluation will "get stuck" if we try to evaluate a thunk during its own evaluation. This simple trick has been known for a long time, and is also crucially important to avoid space leaks [7]. When evaluation is complete, we overwrite the thunk with the value (rule UPDATE).

The last two rules deal with *saturated* applications of *known* functions, either primitive operations (PRIMOP) or user-defined ones (KNOWNCALL). Both are very simple and can be compiled efficiently, with fast parameter-passing mechanisms. Notice that the call to f is a *tail* call. No continuation is pushed; instead control is simply transferred to f's body.

The big remaining question is how function application is handled when the function is unknown, or is applied to too many or too few arguments. And that is the key point at which the two evaluation models differ, of course.

4.2 The push/enter model

The rules in the second block of Figure 2 are the ones specific to the push/enter model. First consider rule PUSH, which deals with function applications. It simply pushes the arguments onto the stack, as *pending arguments*, using the *Arg* continuation, and enters the function. The next three rules deal with what "entering the function" means:

Rules common to push/enter and eval/apply

$\texttt{let}\ x = obj\ \texttt{in}\ e;\ s;\ H$	$\Rightarrow\ e[x'/x];\ s;\ H[x' \mapsto obj]$ x' fresh	(LET)
$\texttt{case}\ v\ \texttt{of}\ x\ \{\ldots; C\ x_1 \ldots x_n \rightarrow e; \ldots\};\ s;$ $H[v \mapsto CON(C\ a_1 \ldots a_n)]$	$\Rightarrow\ e[v/x, a_1/x_1 \ldots a_n/x_n];\ s;\ H$	(CASECON)
$\texttt{case}\ v\ \texttt{of}\ x\ \{\ldots; ANY \rightarrow e\};\ s;\ H$	$\Rightarrow\ e[v/x];\ s;\ H$ if v is a literal or $H[v]$ is a value, and does not match any other case alternative	(CASEANY)
$\texttt{case}\ e\ \texttt{of}\ x\ \{\ldots\};\ s;\ H$	$\Rightarrow\ e;\ \texttt{case} \bullet \texttt{of}\ x\ \{\ldots\} : s;\ H$	(CASE)
$v;\ \texttt{case} \bullet \texttt{of}\ x\ \{\ldots\} : s;\ H$	$\Rightarrow\ \texttt{case}\ v\ \texttt{of}\ x\ \{\ldots\};\ s;\ H$ if v is a literal or $H[v]$ is a value	(RET)
$x;\ s;\ H[x \mapsto THUNK\ e]$	$\Rightarrow\ e;\ Upd\ x \bullet : s;\ H[x \mapsto BLACKHOLE]$	(THUNK)
$y;\ Upd\ x \bullet : s;\ H$	$\Rightarrow\ y;\ s;\ H[x \mapsto H[y]]$ if $H[y]$ is a value	(UPDATE)
$f^n\ a_1 \ldots a_n;\ s;\ H[f \mapsto FUN(x_1 \ldots x_n \rightarrow e)]$	$\Rightarrow\ e[a_1/x_1 \ldots a_n/x_n];\ s;\ H$	(KNOWNCALL)
$\oplus\ a_1 \ldots a_n;\ s;\ H$	$\Rightarrow\ a;\ s;\ H$ where a is the result of applying the primitive operation $\oplus$ to arguments $a_1 \ldots a_n$	(PRIMOP)

Rules for push/enter

$f^k\ a_1 \ldots a_m;\ s;\ H$	$\Rightarrow\ f;\ Arg\ a_1 \ldots Arg\ a_m : s;\ H$	(PUSH)
$f;\ Arg\ a_1 \ldots Arg\ a_n : s;\ H[f \mapsto FUN(x_1 \ldots x_n \rightarrow e)]$	$\Rightarrow\ e[a_1/x_1 \ldots a_n/x_n];\ s;\ H$	(FENTER)
$f;\ Arg\ a_1 \ldots Arg\ a_m : s;\ H[f \mapsto FUN(x_1 \ldots x_n \rightarrow e)]$	$\Rightarrow\ p;\ s;\ H[p \mapsto PAP(f\ a_1 \ldots a_m)]$ if $m \geq 1$; the top element of s is not of the form $Arg\ y$; p fresh	(PAP1)
$f;\ Arg\ a_{n+1} : s;\ H[f \mapsto PAP(g\ a_1 \ldots a_n)]$	$\Rightarrow\ g;\ Arg\ a_1 \ldots Arg\ a_n : Arg\ a_{n+1} : s;\ H$	(PENTER)

Rules for eval/apply

$f^\bullet\ a_1 \ldots a_n;\ s;\ H[f \mapsto FUN(x_1 \ldots x_n \rightarrow e)]$	$\Rightarrow\ e[a_1/x_1 \ldots a_n/x_n];\ s;\ H$	(EXACT)
$f^k\ a_1 \ldots a_m;\ s;\ H[f \mapsto FUN(x_1 \ldots x_n \rightarrow e)]$	$\Rightarrow\ e[a_1/x_1 \ldots a_n/x_n];\ (\bullet\ a_{n+1} \ldots a_m) : s;\ H$ if $m > n$	(CALLK)
	$\Rightarrow\ p;\ s;\ H[p \mapsto PAP(f\ a_1 \ldots a_m)]$ if $m < n$, p fresh	(PAP2)
$f^\bullet\ a_1 \ldots a_m;\ s;\ H[f \mapsto THUNK\ e]$	$\Rightarrow\ f;\ (\bullet\ a_1 \ldots a_m) : s;\ H$	(TCALL)
$f^k\ a_{n+1} \ldots a_m;\ s;\ H[f \mapsto PAP(g\ a_1 \ldots a_n)]$	$\Rightarrow\ g^\bullet\ a_1 \ldots a_n\ a_{n+1} \ldots a_m;\ s;\ H$	(PCALL)
$f;\ (\bullet\ a_1 \ldots a_n) : s;\ H$	$\Rightarrow\ f^\bullet\ a_1 \ldots a_n;\ s;\ H$ $H[f]$ is a FUN or PAP	(RETFUN)

Figure 2. The evaluation rules

- First, the function f might turn out to be a FUN object of arity n, and there might be n or more arguments on the stack. In that case (rule FENTER), we can proceed to evaluate the body the function, binding the actual arguments to the formal parameters as usual. Any excess pending arguments are left on the stack, to be consumed by the function that e (presumably) evaluates to.
- What if there aren't enough pending arguments on the stack? This could happen either because a function-valued thunk pushed an update frame, or because a `case` expression evaluated a function (see Section 3.2). In either case, we must construct a value to return to the "caller" and that value is a partial application, or *PAP*, as rule PAP1 shows.
- What if f is a *PAP* and not a FUN? In that case, we simply unpack the *PAP*'s arguments onto the stack, and enter the function (rule PENTER).

The three cases above do not exhaust the possible forms of f. It might also be a $THUNK$, but we have already dealt with that case (rule THUNK). It might be a CON, in which case there cannot be any pending arguments on the stack, and rules UPDATE or RET apply.

4.3 The eval/apply model

The last block of Figure 2 shows how the eval/apply model deals with function application. The first three rules all deal with the case of a FUN applied to some arguments:

- If there are exactly the right number of arguments, we behave exactly like rule KNOWNCALL, by tail-calling the function. Rule EXACT is still necessary — and indeed has a direct counterpart in the implementation — because the function might not be statically known.
- If there are too many arguments, rule CALLK pushes a *call continuation* on the stack, which captures the excess arguments. This is the essence of eval/apply. Given an application `(f x y)` where `f` takes one argument, first call `(f x)`, and then apply the resulting function to `y`.
- If there are too few arguments, we build a *PAP* (rule PAP2), which becomes the value of the expression.

These rules work by dynamically inspecting the arity of the function closure in the heap, which works fine for both known and unknown calls, but we can do better for known calls. Rule KNOWNCALL has already dealt with the saturated known case, and it is probably not worth the bother of treating under- and over-saturated known calls specially because they are very uncommon (see Section 8).

Another possibility is that the function in an application is a $THUNK$ (rule TCALL). This case is very like the over-applied function of rule CALLK; we push a call continuation and enter the thunk. (This in turn will push an update frame via rule THUNK.)

Finally, the function in an application might be a partial application of another function f' (rule PCALL). In that case we unpack the *PAP* and apply f' to its new arguments. Since f' is sure to be a FUN, this will take us back to one of the cases in rules EXACT, CALLK or PAP2.

That concludes the rules for function application. We need one last rule, RETFUN, which returns a function value (*PAP* or FUN) to a call continuation, in the obvious way. This rule re-activates a call continuation, exactly as rule RET re-activates a `case` continuation.

4.4 Summary

The crucial difference between push/enter and eval/apply is this. When a function of statically-unknown arity is applied, two pieces of information come together at run-time: the arity of the function and the number of arguments in the call. The two models differ in whether they place responsibility for arity-matching with the function itself, or with the caller:

Push/enter: the *function*, which statically knows its own arity, examines the stack to figure out how many arguments it has been passed, and where they are. The nearest analogy is C's "varargs" calling convention.

Eval/apply: the *caller*, which statically knows what the arguments are, examines the function closure, finds its arity, and makes an exact call to the function.

This distinction leads to pervasive implementation choices and constraints, a claim we substantiate in Sections 5 and 6.

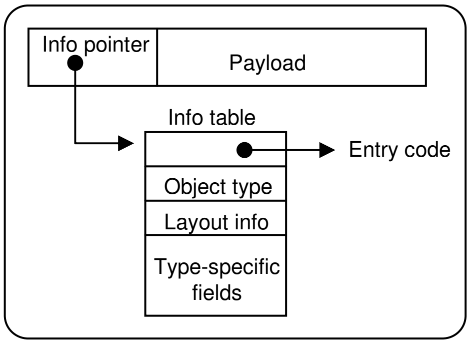

Figure 3. A heap object

4.5 Heap objects

To provide the context for our subsequent discussion, we now sketch briefly how GHC maps the operational semantics onto a real machine. Figure 3 shows the layout of a heap object. In GHC, the first word of every object is called the object's *info pointer*, and points to an immutable, statically-allocated *info table*. The remainder of the object is called the *payload*, and may consist of a mixture of pointers and non-pointers.

The info table contains:

- Executable code for the object. For example, a FUN object has code for the function body.
- An object-type field, which distinguishes the various kinds of objects (FUN, *PAP*, CON etc) from each other.
- Layout information for garbage collection purposes, which describes the size and layout of the payload. By "layout" we mean which fields contain pointers and which contain non-pointers, information that is essential for accurate garbage collection.
- Type-specific information, which varies depending on the object type. For example, a FUN object contains its arity; a CON object contains its constructor tag, a small integer that distinguishes the different constructors of a data type; and so on.

In the case of a PAP, the size of the object is not fixed by its info table; instead, its size is stored in the object itself. The layout of its fields (e.g. which are pointers) is described by the (initial segment of) an argument-descriptor field in the info table of the FUN object which is always the first field of a PAP. The other kinds of heap object all have a size that is statically fixed by their info table.

A very common operation is to jump to the entry code for the object, so GHC uses a slightly-optimised version of the representation in Figure 3. GHC places the info table at the addresses *immediately before* the entry code, and reverses the order of its fields, so that the info pointer *is* the entry-code pointer, and all the other fields of the info table can be accessed by negative offsets from this pointer. This is a somewhat delicate hack, because it involves juxtaposing code and data, but (sadly) it does improve performance significantly (on the order of 5%). Again, however, is not germane to this paper and we ignore it from now on.

4.6 The evaluation stack

In GHC, the evaluation stack s, in Section 4, is represented by a contiguous block of memory[1]. The abstract stack of Section 4 is a stack of continuations, k. These continuations are each represented concretely by a *stack frame*. The stack frames for the two continuations common to both push/enter and eval/apply are these:

- An update continuation $Upd\ x\ \bullet$ is represented by a small stack frame, consisting of a return address and a pointer to the thunk to be updated, x. In the push/enter model, an update frame must contain a second word, which points to the next update frame down in the stack (see Section 5). Having a return address in the update frame means that a value can simply return to the topmost return address, without having to test whether the top frame is an update continuation, a `case` continuation.

  The return address for every update frame can be identical, though; it points to a hand-written code fragment, part of the runtime system, that performs the update, pops the update frame, and returns to the next frame.

- A `case` continuation `case` $\bullet$ `of` $x\,\{alts\}$ is represented by a return address, together with the free variables of the alternatives *alts*, which must be saved on the stack across the evaluation of the scrutinee. For example, consider this function:

  ```
  f :: (Int,Int) -> (Bool,Int) -> Int
  f x y = case h1 x of
             (_,b) -> case h2 y of
                          w -> w+b
  ```

  Across the call to `(h1 x)`, we must save `y` on the stack, because it is used later, but we need not save `x`; then across the call to `(h2 y)` we must save `b`, but we need not save `y`.

  Unlike an update frame, the return address for each `case` expression is different: it points to code for the case alternatives of that particular `case` expression.

In both cases, the frame can be thought of as a stack-allocated function closure: the return address is the info pointer, and it "knows" the layout of the rest of the frame — that is, where the pointers, non-pointers and (in the case of `case` continuations) dead slots are. In our implementation, the stack grows downward, so the return address is at the lowest address, and a stack frame looks exactly like Figure 3. A return address has an info table that the garbage collector uses to navigate over the frame.

In the next sections we describe how the other two continuations are implemented: the *Arg* continuation for push/enter (Section 5) and the $(\bullet\ a_1 \ldots a_n)$ continuation for eval/apply (Section 6).

5 Implementing push/enter

The push/enter model uses the stack to store pending arguments, represented by continuations of form *Arg a*. Unlike the other continuations, these have no return address. When a function with arity n is entered, it begins work by grabbing the top n arguments from the stack (rule FENTER), not by returning to them! This is precisely the difference alluded to in Section 4.4: the function is in control.

How does the function know how many arguments are on the stack? It needs to know this so that it can perform rule FENTER or PAP1 respectively. In GHC the answer is this: we dedicate a register[2], called `Su`, to point to the topmost update frame or `case` frame, rather like the frame pointer in a conventional compiler. Then the function can see if there are enough arguments by taking the difference between the stack pointer and `Su`. (The function knows not only how many arguments it is expecting, but how many words they occupy.) This is the so-called *argument satisfaction check*.

Every function is compiled with two entry points. The *fast entry point* is used for known calls; it expects its arguments in registers (plus some on the stack if there are too many to fit in registers). The *slow entry point* expects all its arguments on the stack, and begins by performing the argument-satisfaction check. If the argument-satisfaction check fails, the slow entry point builds a PAP and returns to the return address pointed to by `Su`; if it succeeds, the slow entry point loads the arguments from the stack into registers and jumps (or falls through, in fact) to the fast entry point.

5.1 Reducing the number of `Su` pushes

In conventional compilers, the frame pointer is really only needed to support debugging, and some compilers provide a flag to omit it, thereby freeing up a register. We cannot get rid of `Su` altogether, but when pushing a new frame it is often unnecessary to save `Su` and make it point to the new frame. Consider:

```
case x of { (a,b) -> .... }
```

We know for sure that `x` will evaluate to a pair, not to a function! There is no need to make `Su` point to the `case` frame during evaluation of `x`. *The only time we need to do so is when the scrutinee cannot statically be determined to be a non-function type.* The classic example is the polymorphic `seq` function:

```
seq :: a -> b -> b
seq a b = case a of { ANY -> b }
```

In some calls to `seq`, `a` will evaluate to a function, while in others it will not. In the former case we must ensure that `Su` points to the `case` frame, so that rule PAP1 applies.

In principle, the same is true about update frames, but in practice there are several reasons that we want to walk the chain of update frames (see Section 7) so GHC always saves `Su` in every update frame.

To avoid that some `case` frames have a saved `Su` and some do not, we instead *never* save `Su` in a `case` frame. Instead, in the (rare) situation of a non-data-typed `case`, we push *two* continuations, a regular `case` continuation, and, on top of it, a `seq` *frame* containing `Su`. A `seq` frame is like an update frame with no update: it serves only to restore `Su` before returning to the `case` frame underneath.

This whole `seq`-frame business is a tiresome complication of the push/enter story. It would not arise in a pure lambda-calculus language that did not permit functions to be evaluated except by applying them. Haskell used to have this property, but it lost it when the `seq` function was introduced, and the intermediate language of this paper institutionalises this change by allowing a `case` to scrutinise a function.

[1] In fact, GHC supports lightweight concurrency, so there are many threads. Each has its own stack, of limited size. The compiler generates explicit stack-overflow tests, and grows the stack when necessary. None of this is relevant to the discussion of this paper, so we do not discuss concurrency or stack overflow any further.

[2] or a memory location on register-starved architectures

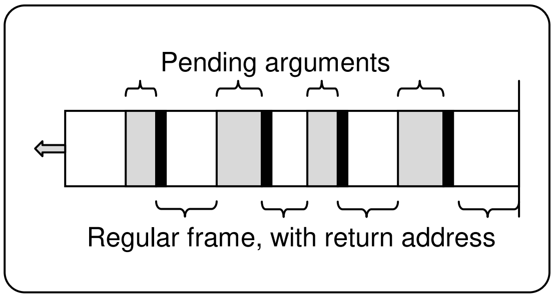

Figure 4. Stack layout for push/enter

5.2 Accurate stack walking

The most painful aspect of the push/enter model is the problem of pending arguments. Consider these functions:

```
g :: Int -> Int -> Int# -> Double# -> Int
g x = ....

f :: Int -> Int
f x = g x x 3 4.5
```

Under the push/enter model, we push the pending arguments `x`, `3`, and `4.5` onto the stack before making the tail call `g x`. The function `g` might compute for a very long time before returning a function that consumes the pending arguments. During this period, the pending arguments simply sit on the stack waiting to be consumed.

An accurate garbage collector must be able to identify every pointer in the stack. The push/enter model leads to stack layout that looks like Figure 4. Each `case` expression pushes a return address (shown black) on the stack before entering the scrutinee. That return address "knows about" a stack frame of variables saved on the stack (or perhaps in callee-saves registers) that are live after the call. The garbage collector can use the return address to access the info table for the return address, just as it does for a heap-allocated closure. The info table describes the layout of the stack frame, including exactly where in the frame the (live) pointers are stored, so that the garbage collector can follow them; it also gives the size of the frame, so that the garbage collector knows where to start looking for the next frame.

The stack frames are the easy (and well-understood) part. However, between each stack frame are zero or more pending arguments (shown grey). The difficulty is that *there is no description of their number or layout* in the stack data structure. The function that pushed them "knows" what they are, and the function that consumes them knows too — but an arbitrarily long period may elapse between push and consumption, and during that time the garbage collector must somehow deal with them. There are two sub-problems:

- Identifying which are pointers and which are non-pointers; as the example above showed, there may be a mixture.
- Distinguishing the last pending argument from the next return address on the stack, which heralds a new stack frame.

One alternative is to have a separate stack for pending arguments, which solves the second of these sub-problems, but not the first. Or, the separate stack could be for pending non-pointer arguments only, which solves the first sub-problem, but not the second. However, a separate stack carries heavy costs of its own, to allocate it, maintain a pointer to the stack top, and check for overflow. We do not consider this alternative further.

A second alternative, and the one that GHC uses, is as follows. We solve the problem of distinguishing pending pointers from non-pointers by *tagging* the non-pointers. The tag is a word, pushed on top of the non-pointer argument, that indicates how many words long the non-pointer argument is (usually 1 or 2). A tag can always be distinguished from a pointer argument, because pointer arguments never point to very low addresses.

Incidentally, a non-alternative is to use a conservative garbage collector. Firstly, to plug space leaks we would then have to use extra memory writes to stub off dead pointers, something the frame layout maps deal with automatically; this turns out to be very important in practice. Second, there are other reasons that GHC's runtime system has to walk the stack accurately: to black-hole thunks under evaluation, and to raise exceptions. Third, stacks may have to move in order to grow; GHC's lightweight concurrency precludes simply allocating a gigantic stack for each thread.

5.2.1 Distinguishing pointers from return addresses

We chose not to tag pointers because pointers are much more common (so the tag pushes would be correspondingly more burdensome), and because it seemed relatively easy to distinguish them from the return address that heralds the next stack frame. We were wrong about that: the problem of distinguishing pointers from return addresses is much trickier than it looks.

First some background: GHC allocates some heap objects statically, compiled directly into the binary. We can therefore divide this problem into two:

- Distinguishing pointers to dynamic heap objects from all other pointers. This is something the garbage collector needs to do anyway, because it needs to know whether to copy the object referenced by a given pointer or not. We could do this by examining the info table of the object, but it's more efficient if the test can be done without dereferencing the pointer and polluting the cache, especially if it turns out that we aren't otherwise going to touch the object that it points to (static objects are assumed to be in the old generation in GHC's generational collector, so they rarely get touched).

  So, we need a test that can determine whether a given pointer points to dynamic heap or not. The details aren't important, because we need this test both for eval/apply and push/enter.

- Distinguishing pointers to static objects from return addresses. Having established that we have a pointer to static memory, we have to be able to tell whether it is a return address or a pointer to a static heap object.

  In GHC we did this by keeping static objects in a separate linker segment from the code: static objects are data, whereas return addresses are in the text segment. Determining the border between text and data can usually be done, although it is non-portable and usually needs to be implemented in a different way for each new platform the compiler is ported to. Furthermore, this breaks down when dynamic linking is added to the mix, because there may be many text and data segments scattered throughout the address space. One alternative, which we used on Win32 systems with DLLs, was to place a zero word before every static closure and use this to discriminate, making use of the fact that a return address is never preceded by a zero word. The problem with this is that it means dereferencing the pointer, which is something we were trying to avoid for efficiency reasons.

The problem of distinguishing pointers from return addresses would completely go away if we pushed `Su` into every frame, so that the `Su` chain linked all frames, not just update frames — but that imposes a non-trivial run-time cost by increasing memory traffic.

5.2.2 *Distinguishing pointers from non-pointers*

Even tagging non-pointers is tiresome. When calling the fast entry point of a function, we can pass some arguments in registers, but when there are too many we pass them on the stack. It would make sense for the stack layout of these overflow parameters to be the same as the latter part of the stack layout expected by the slow entry point (which takes all its arguments on the stack). The latter has tagged slots for non-pointers, so the former had better do so too. But we do not want to take the instructions to explicitly tag the slots when making a fast call, so we allocate space for the tags but do not fill the tags in. (In a call to a known function when too many arguments are supplied, we must generate code to tag the "extra" arguments but not the "known" ones.)

So the invariant at the fast entry point is that there is space for the tags of the non-pointer arguments passed on the stack, but these slots are non necessarily initialised. The fast entry point typically starts with a heap-overflow check; if it fails, it must remember to fill in the tags, so that the top frame of the stack is self-describing.

The exact details are unimportant here. The point is that, while tagging non-pointers in the stack is feasible and reasonably efficient, it imposes a significant complexity burden on both code generator and the the run-time system.

5.3 Generating `C--`

Some compilers generate native code directly, but a very popular alternative route is to generate code in C, or a portable assembly language such as `C--`, leaving to another compiler the tasks of instruction selection, register allocation, instruction scheduling, and so on. A significant disadvantage of the push/enter model is that it makes this attractive route much harder, or at least much less efficient.

The problem, again, is the pending arguments. Suppose that we want to generate C. We plainly cannot push the pending arguments onto the C stack, because C controls its own stack layout. There is just no way to have C stack frames separated by chunks of pending arguments.

The only way out of this is to maintain a separate stack for pending arguments. In fact, GHC uses C as a code generator, and it keeps *everything* on the separately-maintained stack: pending arguments, saved variables, return addresses, and so on. Indeed, GHC does not use the C stack at all, so we only have to maintain a single stack.

Unfortunately, we thereby give up much of the benefit of the portable assembly language. If we do not use the C stack, we cannot use C's parameter-passing mechanisms. Instead, we pass arguments either in global variables that are explicitly allocated in registers (using a `gcc` directive) or on the explicit stack. We have to perform our own liveness analysis to figure out what variables are live across a call, and generate code to save them to to the explicit stack. In short, we only use C to compile basic blocks, managing the entire call/return interface manually.

There are other reasons why we could not use C's stack, however. There is no easy way to check for stack overflow, or to move stacks around (both important in our concurrent Haskell system). C may save live variables across a call, but does not generate stack descriptors for the garbage collector (Section 5.2). Portable exception handing is tricky. And so on.

`C--`, on the other hand, is a portable assembly language designed specifically to act as a back end for high-level-language compilers. It provides explicit and very general support for tail calls, garbage collection, exception handling, and concurrency, and so addresses many of C's deficiencies. Yet, *we have found no general or clean way to extend `C--`'s design to incorporate pending arguments*. So, like C, `C--` provides no way to push an arbitrary number of words on the stack that should persist beyond the end of the current call.

The bottom line is this. The pending arguments required by the push/enter model are incompatible with any portable assembly language known to us, except by using that language in a way that vitiates many of its advantages. We count this as a serious strike against the push/enter model.

6 Implementing eval/apply

Next, we turn our attention to the implementation details for eval/apply. The eval/apply model uses *call continuations*, of form $(\bullet\ a_1 \ldots a_n)$, which are represented by a stack frame consisting of a return address, together with the arguments $a_1 \ldots a_n$. This return address is entered when a function has evaluated to a value (FUN or PAP), and returns. This is the moment when the complicated rules (EXACT, CALLK, PAP2, and so on) are needed, and that involves quite a lot of code. So we do not generate a fresh batch of code for each call site; instead, we pre-generate a range of call-continuation return addresses, for 1, 2, 3, ... N arguments.

What if we need to push a call continuation for more than N arguments? Then we push a succession of call continuations, each for as many arguments as possible, given the range of pre-generated return addresses. In effect, this reverts to something more like the argument-at-a-time function application process, except that we deal with the arguments N at a time. We can measure how often this happens, and arrange to pre-generate enough call continuations to cover 99% of the cases. The remainder are handled by pushing multiple call continuations.

An important complication is that we need different call continuations when some of the arguments are unboxed. Why? Because: (a) the calling convention for the function that the continuation will call may depend on the types of its arguments (e.g. a floating-point argument might be passed in a floating-point register); and (b) the call-continuation return address must (like any return address) have layout information to guide the garbage collector. So cannot get away with just N continuations, but (in principle) we need 3^N. The "3" comes from the three basic cases we deal with: pointer, 32-bit non-pointer and 64-bit non-pointer. There might well be more if, for example, a 32-bit float was passed in a different register than a 32-bit integer. Hence the importance of measurements, to identify the common cases.

6.1 Generic application in more detail

To be more concrete, we will imagine that we compile Haskell into `C--` [13] (we will introduce any unusual features of `C--` as we go along). Here is the code that the call `(f 3 x)`, where `f` is an unknown function, might generate:

```
jump stgApplyNP( f, 3, x )
```

This transfers control — the "`jump`" indicates a tail call — to a pre-generated piece of run-time system code, `stgApplyNP`, where the "`NP`" suffix means "one 32-bit non-pointer, and one pointer". The first parameter is the address of the closure for `f`. It's just as if the original Haskell call had been `(stgApplyNP f 3 x)`, where `stgApplyNP` is a known function, so we make a fast call to it.

The run-time system provides a whole bunch of `stgApplyNP` functions, for various argument combinations. Indeed, we generate them by feeding the desired argument combinations to a generator program.

Figure 5 shows (approximately) is the code we generate for `stgApplyNP`. In this code we assume that `TYPE(f)` is a macro that

```
stgApplyNP( f, a, b ) {
/* Apply f to arguments a and b */

   switch TYPE(f) {
      case THUNK:
         fun_code = CODE(f) ;
         f = fun_code( f );
             /* a,b saved across this call */
        jump stgApplyNP( f, a, b )

      case FUN:
        switch ARITY(f) {
          case 1: /* Too many args */
                  fun_code = CODE(f) ;
                    f = fun_code( f, a );
                       /* b saved across this call */
                  jump stgApplyP( f, b );

          case 2: /* Exactly right! */
                   fun_code = CODE(f) ;
                    jump fun_code( f, a, b );

          other: /* Too few args */
                 ...check for enough heap
                    space to allocate PAP...
                 r = ...build PAP for (f a b)...
                 return( r )
         }

      case PAP:
         switch PAP_ARITY(f) {
           case 1: /* Too many args */
                  f = applyPapN( f, a ) ;
                  jump stgApplyP( f, b );

           case 2: /* Just right */
                  jump applyPapNP( f, a, b )

           other: /* Too many args */
                  ...check for enough heap...
                  r = ...build PAP for (f a b)...
                  return( r )
}}
```

Figure 5. The generic apply function `StgApplyNP`

gets the type field from the info table of heap object `f`, `ARITY(f)` gets the arity from the info table of a `FUN` object, and so on. `CODE(f)` gets the fast entry point of the function, which takes the function arguments in registers (plus stack if necessary).

First, the function might be a `THUNK`; in that case, we evaluate it (by calling its entry point, passing the thunk itself as an argument), before looping around to `stgApplyNP` again.

Next, consider the `FUN` case, which begins by switching on the arity of the function:

- `case 2`: if it takes exactly two arguments, we just jump to the function's code, passing the arguments `a` and `b`. We also pass a pointer to `f`, the function closure itself, because the free variables of the function are stored therein.

  Note that if we end up taking this route, then the function arguments might not even hit the stack: `a` and `b` can be passed in registers to `stgApplyNP`, and passed again in registers when performing the final call. This is an improvement over push/enter, where arguments to unknown function calls are always stored on the stack.

- `case 1`: if the function takes fewer arguments than the number required by `f` — in this case there is just one such branch — we must save the excess arguments, make the call, and then apply the resulting function to the remaining arguments. The code for an N-ary `stgApply` must have a case for each $i < N$. So we get a quadratic number of cases, but since it's all generated mechanically, and the smaller arities cover almost all cases, this is not much of a problem in practice.

- `other`: otherwise the function is applied to too few arguments, so we should build a partial application in the heap.

The third case is that `f` might be a partial application. The three cases are similar to those for a `FUN`, but they make use of an auxiliary family of functions `applyPapX` etc which apply a saturated PAP. This apply operation is not entirely straightforward, because PAP contains a statically-unknown number of arguments. One solution is to copy the argument block from the PAP, followed by the argument(s) to `applyPapX` to a temporary chunk of memory, and call a separate entry point for the function that expects its arguments in a contiguous chunk of memory. The advantage of this approach is that it requires no knowledge of the calling convention. Another solution (currently used by GHC) is to exploit knowledge of the calling convention to make a generic call; in GHC's case we just copy the arguments onto the stack. *SLPJ: I'm unsure whether to say more here, or less!*

There are several opportunities for optimisation. First, we can have specialised `FUN` types for functions of small arity (1, 2, 3, say); that way we could combine the node-type and arity tests. Second, a top level function has no (non-constant) free variables, so there is no need to pass its function closure as its first argument. We would need another `FUN` node type to distinguish this case.

6.2 Too many arguments

What do we do with an unknown call for which there is no pre-generated `stgApplyX` function? Answer, we just split it into two (or more) chunks. For example, suppose we only had `stgApplyX` functions for a single argument. Then our call `(f 3 x)` would compile to:

```
f1 = stgApplyN( f, 3 );
jump stgApplyP( f1, x );
```

Of course, `x` must be saved on the stack across the call to `stgApplyN`.

6.3 Seq frames and vectored returns

In Section 5.1 we discussed the tiresome need for `seq` frames in push/enter. Can we get rid of them under eval/apply? Yes, we could, but we don't, for an entirely separate reason. GHC implements an optimisation called *vectored returns*, whereby a `case` expression pushes not just one return address, but a (pointer to a) vector, or jump table, of return addresses, one for each constructor of the data type, at least when there are only a few constructors in the type (c.f. Section 3.2). This trick eliminates the test-and-branch that is otherwise needed at the return point.

But now consider a truly polymorphic `case`:

```
seq :: a -> b -> b
seq a b = case a of { ANY -> b }
```

The return address pushed by this `case` must be able to cope with either a vectored or non-vectored return, because we do not know the type of `seq`'s argument. So in fact we arrange that the vector of return addresses runs *backwards* from the address we push on the stack, and starts one word before that address. Now we can make a "polymorphic" return address, that has both a vector and regular return code.

This is such a gruesome trick that we do not make the code generator capable of producing such a return address. Instead, we have a few hand-built ones in the run-time system. On the few occasions that the code generator encounters a polymorphic case (i.e. one in which the scrutinised type is unknown) it pushes a regular return address and then a `seq` frame, the latter using the hand-built polymorphic return address from the run-time system.

The reason we retain vectored returns despite this complication is that they buy a few percent in performance. For the purposes of this paper, though, the bottom line is that `seq`-frames are not eliminated by the move to eval/apply.

7 A qualitative comparison

Having described the two implementations, we now summarise the main differences.

In favour of eval/apply:

- Much easier to map to a portable assembly language, such as `C--` or C.
- No need to distinguish return addresses from heap pointers. This is a big win.
- No tagging for non-pointers; this reduces complexity and makes stack frames and PAPs a little smaller.
- No need for the `Su` pointer, perhaps saving a register; and update frames become one word smaller, because there is no need to save `Su`.
- Because the arity-matching burden is on the caller, not the caller, run-time system support functions, callable from Haskell, become more convenient to write.
- When calling an *unknown* function with the right number of arguments, the arguments can be passed in registers rather than on the stack. Push/enter pretty much mandates passing argunents to unknown functions in memory, on the stack.

In favour of push/enter:

- Appears to be a natural fit with currying.
- Eliminates some `PAP` allocations compared to eval/apply.
- The payload of a `PAP` object can be self-describing because the arguments are tagged. In contrast, an eval/apply `PAP` object relies on its `FUN` to describe the layout of the payload; this results in some extra complication in the garbage collector, and an extra global invariant: a `PAP` must contain a `FUN`, it cannot contain another `PAP`.

Plain differences:

- Push/enter requires a slow entry point for each function, incorporating the argument-satisfaction check. Eval/apply does not need this, but (in some renditions) may require an entry point in which the arguments are in a contiguous memory block.
- The `Su` pointer makes it easy to walk the chain of update frames. That is useful for two reasons. First, at garbage collection time we want to black-hole any thunks that are under evaluation (see [7]). Second, a useful optimisation is to squeeze out chains of adjacent update frames, which we also do at garbage-collection time. Under eval/apply, however, one can still find the update frames by a single stack walk; but it may take a little longer because the stack-walk must examine other frames on the stack in order to hop over them. Notice, though, that there is nothing to stop us adding an `Su` register, pointing to the topmost update frame, to the eval/apply model, if that turned out to be faster for the reasons just described. We have not tried this.

Program	Unknown calls (%)	Known calls (%) $<$	$=$	$>$
anna	25.7	0.6	74.3	0.0
cacheprof	25.3	0.2	74.7	0.0
compress	1.6	0.0	98.4	0.0
fem	5.3	0.0	94.7	0.0
fulsom	25.1	0.2	74.9	0.0
hidden	13.8	0.0	86.2	0.1
infer	18.8	0.1	81.2	0.0
scs	17.1	0.0	82.9	0.2
circsim	14.5	0.0	85.5	0.0
fibheaps	13.6	0.0	86.4	0.7
typecheck	27.5	0.5	72.5	0.0
simple	49.2	0.0	50.8	0.0
Min	0.0	0.0	21.2	0.0
Max	78.8	4.3	100.0	1.6
Average	20.7	0.2	79.3	0.1

Figure 6. Anatomy of calls

From this list we conclude two things. First, it is essentially impossible to come to a rational conclusion about performance based on these differences. The only way is to build both both models and measure the difference. Second, the eval/apply model seems to have decisive advantages in terms of complexity. Yes, the `stgApplyX` generator is a new component, but it is well isolated, and not too large (it amounts to some 580 lines of Haskell including comments). The big wins are that complexity elsewhere is reduced, and it is easier to map the code to a portable assembly language.

The bottom line is this: if eval/apply is no more expensive than push/enter, it is definitely to be preferred.

8 Measurements

Our measurements are made on the Glasgow Haskell Compiler version 5.04 (approximately; it does not correspond exactly to any released version). We made measurements across the entire `nofib` benchmark suite of around 50 programs; we present detailed figures for a representative set of a dozen larger benchmarks, but the tables also give minimum, maximum and mean figures *across the whole suite*.

8.1 The anatomy of calls

First of all, we present data on the dynamic frequency of the different categories of function call. All these figures are independent of evaluation model; they are simply facts about programs in our benchmark suite, as compiled by GHC.

Figure 6 show the relative dynamic frequency of:

- Calls to an unknown (lambda-bound or case-bound) function.
- Calls to a known (let-bound) function with (a) too few arguments, (b) exactly the right number of arguments, and (c) too many arguments.

Note that "known" simply means that a let(rec) binding for the function is statically visible at the call site; the function may be bound at top level, or may be nested. GHC propagates arity information across module boundaries, which greatly increases the number of known calls. Also notice that every over-saturated application of a known function gives rise to a subsequent call to the unknown function returned as its result; these unknown calls are

Program	Eval/apply change (%) Tot alloc Words	PAPs Number	Words
anna	+1.7%	+934.7%	+761.2%
cacheprof	-0.0%	+9516.1%	+8326.7%
circsim	+0.0%	+15.4%	+9.1%
compress	-0.0%	+41.8%	+26.2%
fem	+0.0%	+1807.1%	+1622.6%
fibheaps	+0.9%	+12.7%	+12.7%
fulsom	+0.1%	+31576.0%	+35546.8%
hidden	+0.0%	+34.2%	+33.2%
infer	+0.2%	+184.6%	+164.3%
scs	+0.0%	+3454.5%	+3051.9%
simple	+0.0%	+2720.0%	+2543.8%
typecheck	+1.2%	+2840000.0%	+3036915.5%
Min	-2.7%	-12.0%	-11.5%
Max	+2.9%	+2840000.0%	+3036915.5%
Geometric Mean	+0.1%	+173.8%	+158.2%

Figure 8. Change in allocation

	Eval/apply change (%)			
Program	Code size	Instrs	Mem rd	Mem wr
anna	-5.13%	+2.01%	+2.55%	-3.27%
cacheprof	-4.02%	+10.79%	+10.33%	+0.35%
circsim	+0.28%	+0.29%	+1.09%	-9.48%
compress	+2.23%	+1.80%	+3.16%	+3.69%
fem	-0.89%	-5.59%	-3.25%	-7.76%
fibheaps	+1.05%	+3.36%	+4.54%	-3.14%
fulsom	-2.15%	-2.50%	-2.36%	-7.95%
hidden	-2.41%	+3.32%	+4.00%	-6.14%
infer	-1.63%	+2.49%	+2.40%	-0.98%
scs	-2.37%	+0.65%	+1.41%	-2.44%
simple	-1.85%	+3.51%	+2.54%	-4.74%
typecheck	+4.64%	+6.84%	+6.68%	-4.77%
Min	-5.19%	-10.17%	-8.06%	-13.65%
Max	+7.67%	+11.65%	+20.81%	+21.40%
Geometric Mean	+1.80%	+0.00%	+1.08%	-4.85%

Figure 9. Space and time

included in the "unknown calls" column. For example, each execution of the call `(id f x)` would count as one call to a known function (`id`) with too many arguments, and one call to the unknown function returned by `id`.

These numbers lead to two immediate conclusions. First, known calls are common, and sometimes dominate, but unknown calls can be the majority in some programs. Unknown calls must be handled efficiently. Second, known calls are almost always saturated; the efficiency of handling under- or over-saturated known calls is not important, and they can be treated like unknown calls.

Figure 7 classifies the unknown calls of Figure 6, by their argument patterns. This data is helpful in deciding how many different versions of `stgApply` to generate. We don't care about known functions because we generate inline code for their calls. The column headings use one character per argument to indicate the pattern with the key: `p` = pointer, `v` = void. `pp`, for example, means a call with two pointer arguments. A "void" argument is an argument of size zero; such arguments are used for the "state token" used for implementing the `IO` monad. The general conclusion is clear: a double-handful of 9 argument patterns is enough to cope with 99.9% of all situations.

8.2 Allocation

Heap allocation is largely unaffected by the change from push/enter to eval/apply. However, there are some small effects to consider. First, eval/apply will allocate a PAP when returning a function applied to too few arguments, when push/enter may get away without heap allocation because the function can find its missing arguments on the stack. Second, the PAPs in eval/apply may be slightly smaller because there is no need to tag their non-pointer components. Figure 8 quantifies these effects, showing the percentage change observed when we move from push/enter to eval/apply. The amusingly large percentages arise in programs that allocate only a handful of PAPs under push/enter, but significant numbers under eval/apply. However, even in these cases, total allocation hardly budges.

8.3 The bottom line

What really matters in the end is time and space. Figure 9 shows the percentage change we measured in moving from push/enter to eval/apply. Somewhat to our surprise, there is really not much to choose between the two models. Static code size increases very slightly; instruction count really doesn't change much (up a bit here, down a bit there). There are a few more memory reads, but significantly fewer memory writes.

9 Related work

Two of the most popular and influential abstract machines for lazy languages, the G-machine [6] and the Three Instruction Machine (TIM) [3], both use push/enter. As a result, many compilers for lazy languages, including GHC and `hbc`, use push/enter.

However Faxén's OCP compiler for the lazy language Plain uses eval/apply [4]. Rather than have generic `stgApplyXX` application procedures, OCP creates specialised function entry points. For each function `f` of arity n, and for each $i < n, j <= n - i$, OCP makes an entry point `f_`ij that expects to find i arguments in a PAP object, and j extra arguments passed in registers. That looks like an awful lot of entry points, but a global flow analysis allows OCP to prune many entry points that cannot be used. The possiblity of such specialisation is an additional benefit of eval/apply (see [1] for an extreme version). Eager Haskell, an unusual implementation of Haskell based on eager evaluation, also uses eval/apply [10].

Caml, a call-by-value language, uses push/enter for the interpreter [9], but eval/apply for the compiler, largely for the reasons outlined in Section 7.

10 Conclusions

Our main conclusion is easy to state: for a high-performance, compiled implementation use eval/apply! There is not much to choose between the two models on performance grounds, and eval/apply makes it noticeably easier to manage the complexity of a compiler and runtime system for a higher order language, as Section 7 explained. Many of the complexities of push/enter are caused by efficiency hacks, though. For an interpreter, where performance is not such an issue, these hacks are not important, and push/enter may well be more elegant.

Acknowledgments

Many thanks to Robert Ennals, Karl-Filip Faxén, Xavier Leroy, Jan-Willem Maessen, Greg Morrisett, and Norman Ramsey for giving the paper a careful read.

Program	Argument pattern (% of all unknown calls)									
	`v`	`p`	`pv`	`pp`	`ppv`	`ppp`	`pppv`	`pppp`	`ppppp`	`OTHER`
anna	0.0	29.6	0.0	69.3	0.0	1.1	0.0	0.0	0.0	0.0
cacheprof	0.0	91.6	0.0	8.1	0.0	0.3	0.0	0.0	0.0	0.0
compress	0.4	73.9	0.0	12.9	0.0	12.7	0.0	0.0	0.0	0.0
fem	0.0	91.3	0.0	8.1	0.0	0.6	0.0	0.0	0.0	0.0
fulsom	0.0	17.5	0.0	82.5	0.0	0.0	0.0	0.0	0.0	0.0
hidden	0.2	48.7	0.0	14.3	0.0	36.8	0.0	0.0	0.0	0.0
infer	0.0	51.8	0.0	48.1	0.0	0.1	0.0	0.0	0.0	0.0
scs	1.4	19.6	0.0	79.0	0.0	0.0	0.0	0.0	0.0	0.0
circsim	0.0	70.2	0.0	8.6	0.0	21.2	0.0	0.0	0.0	0.0
fibheaps	0.0	43.2	13.7	43.1	0.0	0.0	0.0	0.0	0.0	0.0
typecheck	0.0	89.5	0.0	10.5	0.0	0.0	0.0	0.0	0.0	0.0
simple	0.0	20.1	0.0	79.9	0.0	0.0	0.0	0.0	0.0	0.0
Min	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Max	58.6	100.0	13.7	100.0	15.5	98.9	6.2	11.3	0.3	0.1
Average	5.2	54.4	0.3	34.4	0.3	5.2	0.1	0.1	0.0	0.0

Figure 7. Argument patterns

11 References

[1] U. Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, Sweden, April 1999.

[2] R. Douence and P. Fradet. A systematic study of functional language implementations. *ACM Transactions on Programming Languages and Systems*, 20(2):344–387, March 1998.

[3] J. Fairbairn and S. Wray. TIM - a simple lazy abstract machine to execute supercombinators. In G. Kahn, editor, *Proc IFIP conference on Functional Programming Languages and Computer Architecture, Portland*, pages 34–45. Springer Verlag LNCS 274, Sept. 1987.

[4] K.-F. Faxén. *Analysing, Transforming and Compiling Lazy Functional Programs*. PhD thesis, Department of Teleinformatics, Royal Institute of Technology, June 1997.

[5] C. Flanagan, A. Sabry, B. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM Conference on Programming Languages Design and Implementation (PLDI'93)*, pages 237–247. ACM, June 1993.

[6] T. Johnsson. Efficient compilation of lazy evaluation. In *Proc SIGPLAN Symposium on Compiler Construction, Montreal*. ACM, June 1984.

[7] R. Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–80, Jan 1992.

[8] J. Launchbury. A natural semantics for lazy evaluation. In *20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM, Jan. 1993.

[9] X. Leroy. The Zinc experiment: an economical implementation of the ML language. Tr 117, inria-rocquencourt, INRIA, Feb. 1990.

[10] J.-W. Maessen. *Hybrid Eager and Lazy Evaluation for Efficient Compilation of Haskell*. PhD thesis, Massachusetts Institute of Technology, June 2002.

[11] S. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, Apr. 1992.

[12] S. Peyton Jones and J. Launchbury. Unboxed values as first class citizens. In R. Hughes, editor, *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666, Boston, 1991. Springer Verlag.

[13] S. Peyton Jones, N. Ramsey, and F. Reig. `C--`: a portable assembly language that supports garbage collection. In G. Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 1–28, Berlin, Sept. 1999. Springer.