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

January 13, 2003

Simon Marlow and Simon Peyton Jones
Microsoft Research, Cambridge

Abstract

The abstract

1 Introduction

*** Language has arity-ful functions by design ****

There are two basic evaluation models for higher-order languages: *push/enter* and *eval/apply*. They differ in the way they deal with the fundamental primitive, namely function application. In the eval/apply model, the caller first *evaluates* the function, and then *applies* it to its arguments. In the push/enter model, the caller *pushes* the arguments onto the stack, and *enters* the function, which applies itself to whatever arguments it finds on the stack.

Both models make sense for both call-by-value languages, such as ML and call-by-need (or lazy) languages, such as Haskell. The push/enter model appears to be particularly suitable for higher-order languages in which *currying* is encouraged. We will explain why in Section 2.

Strangely, however, no one knows which model is more efficient. The trouble is that the evaluation model has a pervasive effect on the implementation, so it is hard to implement both. In practice, implementors simply make a choice based on their gut feel.

The Glasgow Haskell Compiler (GHC) is the world's leading compiler for Haskell, a language that certainly encourages currying, and has always used the push/enter model. Recently, however, we have modified its back end and run-time system to use the eval/apply model. (We explain why in Section **??**.) This change gives us a unique opportunity to make a detailed, quantitative comparison of the two models, in the credible setting of state-of-the-art compiler and a large benchmark suite.

More specifically, our contributions are these:

- empty

There are lots of nitty-gritty details in this paper, for which we make no apology. The choice between push/enter and eval/apply is not determined by one or two big issues. Instead, each model has a list of small advantages and disadvantages, and it is absolutely unclear how these effects balance out against each other. For that reason, rather than give simple bottom-line figures, we try to quantify the major effects, so that they will be useful for other settings in which the constant factors differ.

2 Efficient currying

We focus exclusively on languages for which *currying* is important. 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 they *encourage* it.

2.1 Implementing currying

Suppose we have the following definition:

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

and the compiler is faced with the following call:

```
g 3 4
```

The call is to `g`, 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 (tail) 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.

Now consider this variant:

```
h :: Int -> [a] -> a
h x = if x==0 then \ys -> head ys
                else \ys -> tail (head ys)
```

The function h is still "known", but it only takes one argument. Now how should we compile the call (h 3 4)? Here the eval/apply and push/enter models differ. Under the eval/apply model we first make a direct call to h, passing the argument 3 in a register, expecting it to return a function closure; then we tail-call that function closure, passing argument 4. Under the push/enter model we first push the "pending" argument 4 onto the stack, and then make a tail call to h passing argument 3 in a register as before.

So push/enter and eval/apply differ principally in how they handle the call to the "unknown" function that h returns. This apparently-small difference has pervasive effects. In order to make them more explicit we must give a more precise description of what the two evaluation models are.

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 exactly the STG language introduced in [3], but we have adjusted some of the details for this paper.

The idea is that each syntactic construct 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 umber, d. We have more to say about unboxed values in Section 3.3.
- A function application performs a tail call. The application is in A-normal form — that is, each argument is an atom (literal or variable) — so there is no argument preparation to perform first. As we discussed earlier, it is crucially important to have an efficient way to pass multiple arguments to a curried function, so the language syntax has multi-argument application, rather than the binary application that is presented to the Haskell programmer. However, there is no guarantee that the function's arity matches the supplied number of arguments; indeed, 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.

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)
```

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 language does not provide a syntactic form of expression for constructor applications, or for anonymous lambdas; instead, they must be explicitly allocated using `let`. We can see an example of this in the `Cons` branch of `map`, which builds the result `r` with a `let` before returning it. Since the result has to be allocated anyway (unless it is an unboxed value), this simplification simply reflects reality.

3.1 Heap objects

There are 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 *PAP*. If it is appplied to more than n arguments, it consumes the first n arguments and (presumably) computes a function that consumes the remaining arguments.

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

Variables	x, y, f			
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
Expressions	e	$::=$	a	Atom
		$\mid$	$f\, a_1 \ldots a_n$	Application ($n \geq 1$)
		$\mid$	$\oplus\, a_1 \ldots a_n$	Saturated primitive operation ($n \geq 1$)
		$\mid$	`let` $x = obj$ `in` e	
		$\mid$	`case` e `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

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 [1].

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 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 we connect 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 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 [4]. Its significance for this paper is that the stack contains 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.

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

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 [2], 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 (sligtly 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.

SLPJ: Mutter mutter: unboxed literal alternatives?

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 [1]. 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 (CALL). 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:

- 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.

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 $H[v]$ 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)
$f\ a_1 \ldots a_n;\ \ s;\ \ H[f \mapsto FUN(x_1 \ldots x_n \rightarrow e)]$	$\Rightarrow\ \ e[a_1/x_1 \ldots a_n/x_n];\ \ s;\ \ H$ if f is a known function	(CALL)
$\oplus\ a_1 \ldots a_n;\ \ s;\ \ H$	$\Rightarrow\ \ a;\ \ s;\ \ H$ where a is the result of applying the primitive operation $\oplus$ to arguments $a_1 \ldots a_n$	(PRIMOP)

Rules for push/enter

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

Rules for eval/apply

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

Figure 2. The evaluation rules

- 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 CALL, 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.

Another possiblity 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.

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.

5 Implementing the models

In this section we briefly describe how the two evaluation models are mapped onto a real machine by the Glasgow Haskell Compiler.

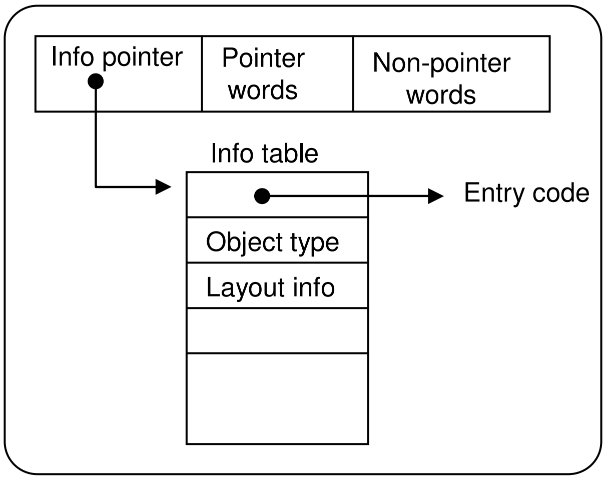

Figure 3. A heap object

5.1 Heap objects

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. All the kinds of object (FUN, CON, THUNK,BLACKHOLE) have a size that is statically fixed by their info table.

5.2 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:

[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.

- An update continuation *Upd* x • 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 5.4). 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` • `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, becuase 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 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.

5.3 Call continuations

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 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 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 with a single one.

An important complication is that we need different call continuations when some of the arguments are unboxed. Why? Because the call-continuation return address must (like any return address) have layout information to guide the garbage collector. So we need not just N continuations, but (in principle) 3^N[2]. Hence the importance of measurements, to identify the common cases.

5.4 Pending arguments

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 grabs 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 here.

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, called `Su`, to point to the topmost update frame or `case` frame. Then the function can see if there are enough arguments by taking the difference between the stack pointer and `Su`. (The function knows knows not only how many arguments it is expecting, but how many words they occupy.) This is the so-called *argument satisfaction check*.

Whenever we push an update frame, we save `Su` in the frame, and make `Su` point to the top of the new update frame. However, for almost all `case` frames, it is entirely unnecessary to save `Su`. 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 functions, 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.

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 eval/apply story. It would not arise in a pure lambda-calulus 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 scrutinse a function.

6 A qualitative comparison

The choice of evaluation model has a pervasive effect on the code generator and run-time system, not all of them obvious. In this section we describe all the effects that we have come across.

[2]The 3 comes from the three basic cases we deal with: pointer, 32-bit non-pointer and 64-bit non-pointer.

6.1 Stack frames and pending arguments

The root cause of many of these effects is the presence of "pending arguments" on the stack in the push/enter model.

Now 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. This stack discipline gives rise to at least two difficulties: (a) accurate garbage collection is harder (Section 6.2), and (b) it is harder to generate code in a portable assembly language such as `C--` (Section 6.3).

These pending arguments give rise to two significant complications, which we discuss next.

6.2 Accurate garbage collection

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 **??**. Each `case` expression pushes a return address 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[3] a static descriptor, allocated by the compiler, that decribes the layout of the stack frame. The descriptor identifies 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, as described in Section 6.1. 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. First, identifying which are pointers and which are non-pointers; as the example in Section 6.1 showed, there may be a mixture. Second, 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. *SLPJ: Lies, all lies.* 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.

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 addess that haralds the next stack frame. We were wrong about that: the problem of distinguishing pointers from return addresses is much trickier than it looks. In earlier versions of GHC we maintained the following invariant: *code (pointed to by a return address) is always a lower address than any of the heap.* A comparison against a threshold value then suffices to distinguish a return address from a heap pointer. However, as GHC has developed the code/heap invariant has become increasingly complicated and expensive to maintain:

- Some heap objects are allocated statically. They are often read-only, so it is tempting to allocate them in the text segment. But that means they get mixed up with code, breaking the invariant. They must be allocated in a different linker segment, and that segment must be allocated after the code.

  The system breaks down altogether if dynamic linking is required. Now we cannot ensure that all static objects are allocated at a higher address than all code.

- GHC builds its dynamic heap by dynamically requesting blocks of address space from the operating system. With older operating systems we could rely on the operating system to allocate such dynamic requests at higher addresses than all code, but this does not reliably hold with some recent variants of Linux *SLPJ: Simon, is this right?*.

  Again, dynamic linking makes this much worse. There is absolutely no way to avoid the interleaving of dynamically-linked code and dynamically-allocated heap objects.

So the latest version of GHC keeps a map of where the code chunks are, in a tree structure, and searches this tree to determine whether a putative pointer points into code space or heap space. This data structure is tiresome to maintain, and expensive to consult. *SLPJ: Not right! We still need it.*

6.3 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 signficant disadvantage of the eval/apply 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 some 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

[3] This access can be done using a hash table, or by putting the descriptor immediately before the code, or in some other way.

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 6.2). 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 incorporate pending arguments in* `C--`*'s design*. 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.

6.4 Keeping the `Su` pointer

6.5 Unsafe coerce

Use `forall a.a` in `Typeable`, and for `HValue`.

7 Measurements

7.1 Call patterns

Classify call calls to *unknown* functions with at least one argument by argument pattern (e.g. ppp, ppnpp, etc). (We don't care about known functions because we generate inline code for their calls.)

These results are independent of push/enter vs eval/apply.

7.2 Number of arguments

Classify calls with at least one argument into:

- (Independent of Push/enter vs eval/apply.) Known function with:
  - Exactly the right number of arguments
  - Too few arguments
  - Too many arguments
- (Eval/apply only.) Unknown function, which after evaluation turns out to be a PAP, has:
  - Exactly the right number of arguments
  - Too few arguments (so we must copy the PAP).
  - Too many arguments (push excess args)
- (Eval/apply only.) Unknown function, which after evaluation turns out to be a FUN, has:
  - Exactly the right number of arguments.
  - Too few arguments (build a PAP).
  - Too many arguments (push excess args).

Perhaps do this classification statically (for code size) and dynamically (for execution time)? Or is that overkill?

Number of multi-chunk calls?

7.3 Function evaluation

How often does a call to an unknown function find an already-evaluated function? If the proportion is high, we might want to do semi-tagging in the `stg_ap` functions. If we find an evaluated function, with enough args, we don't need to save the args on the stack, so it's all very like a known call. How often does this happen?

7.4 PAP construction

We expect more PAPs to be constructed by eval/apply. How many?

I don't think there should be any other differences in allocation. Are there?

PAP sizes should be smaller with eval/apply, because there is no tagging. Quantify? (Likely to be very small effect, because only unboxed values are tagged.)

7.5 Info tables

Effects on info table sizes and code size. Maybe something about object layout encoding (code-size and execution time saving from having special cases).

7.6 Memory reads and writes generally

8 Notes

Lazily filled-in stack tags. (Partial in case of over-saturated call.)

No arg-satis code.

Seq/catch frames no longer need Su.

Seq frames still present, but just for vectored returns.

Hack for vectored return (0 after), and DLL closures (0 before) no longer needed.

No update chain. Lazy black-holing, and update frame squeezing.

9 References

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

[2] 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.

[3] 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.

[4] 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.