Making a Fast Curry: Push/Enter vs. Eval/Apply for Higher-order Languages

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 intuition 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, when the function is unknown: the *push/enter* model or the *eval/apply* model (Peyton Jones, 1992). To illustrate the difference, consider the higher-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
```

Here `k` is an *unknown function*, passed as an argument; global flow analysis aside, the compiler does not know what function `k` is bound to. What code should the compiler generate to execute the call `k x y` in the body of `zipWith`? It can't blithely pass two arguments to `k`, 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.

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.

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 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 affects many other design choices in subtle but pervasive ways. We identify and discuss these effects in Sections 5 and 6, and contrast them in Section 7. There are lots of nitty-gritty details here, for which we make no apology — they were far from obvious to us, and articulating these details is one of our main contributions.
  In terms of its impact on compiler and run-time system complexity, eval/apply seems decisively superior, principally because push/enter requires a stack like no other: stack-walking is more difficult, and compiling to an intermediate language like C or `C--` is awkward or impossible.
- We give the first detailed quantitative measurements (Section 8) that contrast the two approaches, based on a credible, optimising compiler (the Glasgow Haskell Compiler, GHC). We give both bottom-line results such as wall-clock time, total instruction count and allocation, and also some more insightful numbers such as breakdowns of call patterns.
  Our experiments show that the execution costs of push/enter and eval/apply are very similar, despite their pervasive differences. What you gain on the swings you lose on the roundabouts.

Our conclusion is simple, and contradicts the abstract-machine heritage of the lazy functional-language community: eval/apply is a clear win. We have now adopted eval/apply for GHC.

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.

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.

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. Our measurements show that on average around 20% of calls are to unknown functions, and on average 40% of those calls (8% of all calls) have more than one argument (Section 8), although these figures can vary significantly from program to program.

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
		$\mid$	$f^k\ a_1 \dots a_n$	Function call ($n \geq 1$)
		$\mid$	$\oplus\ a_1 \dots a_n$	Saturated primitive operation ($n \geq 1$)
		$\mid$	$\texttt{let}\ x = obj\ \texttt{in}\ e$	
		$\mid$	$\texttt{case}\ e\ \texttt{of}\ \{alt_1; \dots; alt_n\}$	($n \geq 1$)
Alternatives	alt	$::=$	$C\ x_1 \dots x_n\ \to\ e$	($n \geq 0$)
		$\mid$	$x \to e$	Default alternative
Heap objects	obj	$::=$	$FUN(x_1 \dots x_n \to e)$	Function (arity $= n \geq 1$)
		$\mid$	$PAP(f\ a_1 \dots a_n)$	Partial application (f is always a FUN with $arity(f) > n \geq 1$)
		$\mid$	$CON(C\ a_1 \dots a_n)$	Saturated constructor ($n \geq 0$)
		$\mid$	$THUNK\ e$	Thunk
		$\mid$	$BLACKHOLE$	[only during evaluation]
Programs	$prog$	$::=$	$f_1 = obj_1; \dots; f_n = obj_n$	

Fig. 1. Syntax

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 (Peyton Jones, 1992), 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, and because our quantitative data is for Haskell.

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, $f^k\ a_1 \dots a_n$, applies the function f to the arguments $a_1 \dots a_n$. The application is in A-normal form (Flanagan *et al.*, 1993) — 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.

- 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 a default alternative as a catch-all, or the patterns cover all the possibilities in the data type. We often omit the curly braces in our informal examples, using layout instead.

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 other than the x_i). 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.

$PAP(f\; a_1 \ldots a_n)$ represents a partial application of function f to arguments $a_1 \ldots a_n$. 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 $a_1 \ldots a_n$.

$THUNK\; e$ 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 (Jones, 1992).

Of these, FUN, PAP and CON objects are *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)
```

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 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!

3.3 Unboxed values

Another slightly unusual feature of our language is the use of *unboxed values* (Peyton Jones & Launchbury, 1991). Supporting unboxed values is vital for performance, but it has significant consequences for the implementation: both heap objects and the stack may contain a mix of pointer and non-pointer 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 ->
    I# (x +# y)
    }}
```

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`.

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 (Douence & Fradet, 1998), although their treatment only considers single-argument functions whereas we are interested in how to perform multiple application without building intermediate function closures. Furthermore, the semantics we present 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. We use the notation "`:`" to means cons in the context of a stack.

Rules common to push/enter and eval/apply

$$\texttt{let}\ x = obj\ \texttt{in}\ e;\ \ s;\ \ H \quad\Rightarrow\quad e[x'/x];\ \ s;\ \ H[x' \mapsto obj] \qquad \text{(LET)}$$
x' fresh

$$\texttt{case}\ v\ \texttt{of}\ \{\ldots; C\ x_1 \ldots x_n \rightarrow e; \ldots\};\ \ s;\ \ H[v \mapsto CON(C\ a_1 \ldots a_n)] \quad\Rightarrow\quad e[a_1/x_1 \ldots a_n/x_n];\ \ s;\ \ H \qquad \text{(CASECON)}$$

$$\texttt{case}\ v\ \texttt{of}\ \{\ldots; x \rightarrow e\};\ \ s;\ \ H \quad\Rightarrow\quad e[v/x];\ \ s;\ \ H \qquad \text{(CASEANY)}$$
if v is a literal or $H[v]$ is a value, and does not match any other case alternative

$$\texttt{case}\ e\ \texttt{of}\ \{\ldots\};\ \ s;\ \ H \quad\Rightarrow\quad e;\ \ \texttt{case}\ \bullet\ \texttt{of}\ \{\ldots\} : s;\ \ H \qquad \text{(CASE)}$$

$$v;\ \ \texttt{case}\ \bullet\ \texttt{of}\ \{\ldots\} : s;\ \ H \quad\Rightarrow\quad \texttt{case}\ v\ \texttt{of}\ \{\ldots\};\ \ s;\ \ H \qquad \text{(RET)}$$
if v is a literal or $H[v]$ is a value

$$x;\ \ s;\ \ H[x \mapsto THUNK\ e] \quad\Rightarrow\quad e;\ \ Upd\ x\ \bullet : s;\ \ H[x \mapsto BLACKHOLE] \qquad \text{(THUNK)}$$

$$y;\ \ Upd\ x\ \bullet : s;\ \ H \quad\Rightarrow\quad y;\ \ s;\ \ H[x \mapsto H[y]] \qquad \text{(UPDATE)}$$
if $H[y]$ is a value

$$f^n\ a_1 \ldots a_n;\ \ s;\ \ H[f \mapsto FUN(x_1 \ldots x_n \rightarrow e)] \quad\Rightarrow\quad e[a_1/x_1 \ldots a_n/x_n];\ \ s;\ \ H \qquad \text{(KNOWNCALL)}$$

$$\oplus\ a_1 \ldots a_n;\ \ s;\ \ H \quad\Rightarrow\quad a;\ \ s;\ \ H \qquad \text{(PRIMOP)}$$
where a is the result of applying the primitive operation $\oplus$ to arguments $a_1 \ldots a_n$

Rules for push/enter

$$f^k\ a_1 \ldots a_m;\ \ s;\ \ H \quad\Rightarrow\quad f;\ \ Arg\ a_1 : \ldots : Arg\ a_m : s;\ \ H \qquad \text{(PUSH)}$$

$$f;\ \ Arg\ a_1 : \ldots : Arg\ a_n : s;\ \ H[f \mapsto FUN(x_1 \ldots x_n \rightarrow e)] \quad\Rightarrow\quad e[a_1/x_1 \ldots a_n/x_n];\ \ s;\ \ H \qquad \text{(FENTER)}$$

$$f;\ \ Arg\ a_1 : \ldots : Arg\ a_m : s;\ \ H[f \mapsto FUN(x_1 \ldots x_n \rightarrow e)] \quad\Rightarrow\quad p;\ \ s;\ \ H[p \mapsto PAP(f\ a_1 \ldots a_m)] \qquad \text{(PAP1)}$$
if $m \geq 1$; the top element of s is not of the form $Arg\ y$; p fresh

$$f;\ \ Arg\ a_{n+1} : s;\ \ H[f \mapsto PAP(g\ a_1 \ldots a_n)] \quad\Rightarrow\quad g;\ \ Arg\ a_1 : \ldots : Arg\ a_n : Arg\ a_{n+1} : s;\ \ H \qquad \text{(PENTER)}$$

Rules for eval/apply

$$f^{\bullet}\ a_1 \ldots a_n;\ \ s;\ \ H[f \mapsto FUN(x_1 \ldots x_n \rightarrow e)] \quad\Rightarrow\quad e[a_1/x_1 \ldots a_n/x_n];\ \ s;\ \ H \qquad \text{(EXACT)}$$

$$f^k\ a_1 \ldots a_m;\ \ s;\ \ H[f \mapsto FUN(x_1 \ldots x_n \rightarrow e)]$$
$$\Rightarrow\quad e[a_1/x_1 \ldots a_n/x_n];\ \ (\bullet\ a_{n+1} \ldots a_m) : s;\ \ H \qquad \text{(CALLK)}$$
if $m > n$
$$\Rightarrow\quad p;\ \ s;\ \ H[p \mapsto PAP(f\ a_1 \ldots a_m)] \qquad \text{(PAP2)}$$
if $m < n$, p fresh

$$f^{\bullet}\ a_1 \ldots a_m;\ \ s;\ \ H[f \mapsto THUNK\ e] \quad\Rightarrow\quad f;\ \ (\bullet\ a_1 \ldots a_m) : s;\ \ H \qquad \text{(TCALL)}$$

$$f^k\ a_{n+1} \ldots a_m;\ \ s;\ \ H[f \mapsto PAP(g\ a_1 \ldots a_n)] \quad\Rightarrow\quad g^{\bullet}\ a_1 \ldots a_n\ a_{n+1} \ldots a_m;\ \ s;\ \ H \qquad \text{(PCALL)}$$

$$f;\ \ (\bullet\ a_1 \ldots a_n) : s;\ \ H \quad\Rightarrow\quad f^{\bullet}\ a_1 \ldots a_n;\ \ s;\ \ H \qquad \text{(RETFUN)}$$
$H[f]$ is a FUN or PAP

Fig. 2. The evaluation rules

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. To reduce clutter, we use the 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 stack continuations, κ, take the following forms:

κ	$::=$	$\texttt{case} \bullet \texttt{of} \{alt_1; \ldots; alt_n\}$	
	$\mid$	$Upd\ t\ \bullet$	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. 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 (Launchbury, 1993), 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. If the returned value does not match any other `case` alternative, the default 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.

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 (Jones, 1992). 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:

- 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 of 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

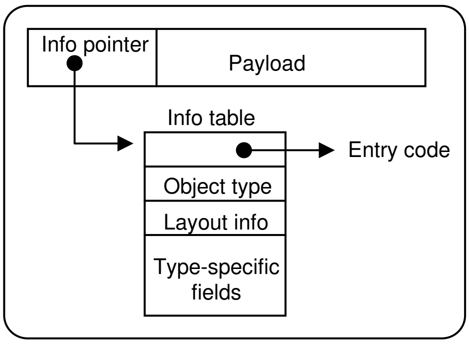

Fig. 3. A heap object

— 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 g (rule PCALL). In that case we unpack the PAP and apply g to its new arguments. Since g 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 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. For example, the object $CON(C\ a_1 \ldots a_n)$ would be represented by an object whose info pointer represented the constructor C and whose payload is the arguments $a_1 \ldots a_n$.

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.

4.5 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, κ. 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 $\bullet$ 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 or 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` $\{alts\}$ 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 the Introduction: 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` ("u" for "update"), 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 { x -> 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.

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

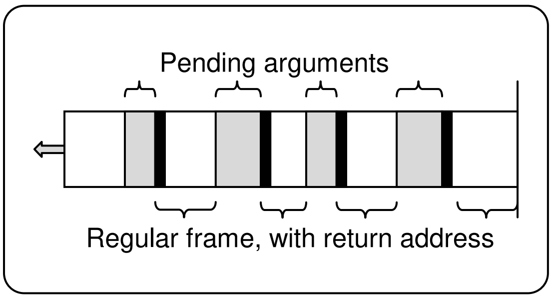

Fig. 4. Stack layout for push/enter

5.2 Accurate stack walking

The most painful aspect of the push/enter model is the problem of representing *Arg* continuations, which hold 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. Update and `case` continuations, whose representation was discussed in Section 4.5, are represented by "regular" stack frames, consisting of a return address (shown black) on top of a block of data (shown white) *whose exact layout is "known" to the return address.* The garbage collector can use the return address to access the info table for the return address (Section 4.5 again), 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.

These regular stack frames are the easy (and well-understood) part. However, between each regular stack frame are zero or more *Arg* continuations, or 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.

Another 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.

Failing these alternatives, the obvious approach is to add a tag word to each Arg continuation. The tag word distinguishes pointer-carrying from non-pointer-carrying Arg continuations, specifies the size of latter kind, and can be distinguished from the return address that heralds the next regular stack frame. Easy enough, but inefficient. In the following two sections we describe two optimisations that GHC uses to reduce the tagging cost.

5.2.1 Omitting tags on pointers

Our first optimisation is to not to tag pointer arguments at all. This is attractive because pointer arguments dominate (see Section 8). Furthermore it looks relatively easy to distinguish a pointer from the return address that heralds the next stack frame, whereas non-pointer arguments, which can hold any bit-pattern whatsoever, cannot be distinguished in general. We were wrong to think it was easy, though: the problem of distinguishing pointers from return addresses is much trickier than it looks, as we now discuss.

GHC allocates some heap objects statically, compiling them directly into the binary. So a pointer on the stack may belong to one of three classes:

$\mathcal{R}$: pointers to return addresses
$\mathcal{D}$: pointers to dynamic heap objects
$\mathcal{S}$: pointers to a static objects

When traversing the stack, we want to identify pointers in $\mathcal{R}$. We do this in two stages:

Step 1: identify pointers belonging to $\mathcal{D}$. Stack-walking aside, the garbage collector needs to distinguish $\mathcal{D}$ from $\mathcal{S}$ anyway, because it needs to know whether to move a live object or not.

If we knew that a pointer was in $\mathcal{D}$ or $\mathcal{S}$, then we could distinguish the two 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).

GHC therefore implements the static/dynamic test without dereferencing the pointer, using an address-based test – we know exactly where the dynamic heap is – and since that test distinguishes $\mathcal{D}$ from all other pointers, we can re-use it to distinguish $\mathcal{D}$ from $\mathcal{R}$ or $\mathcal{S}$ when stack walking.

Step 2: distinguish $\mathcal{R}$ from $\mathcal{S}$. In earlier versions of 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 could be solved in another way: by saving `Su` in a known place every regular frame. Then the stack-walker could rely on an `Su` chain linking every regular frame, so it would always know where the next regular frame began. However, building a chain of all frames would impose a non-trivial run-time cost by increasing memory traffic.

5.2.2 Lazy tagging

Tagging non-pointer pending arguments carries only a modest run-time cost, because (in Haskell at least) it is rare to call a function that returns a function that consumes non-pointer arguments. GHC therefore tags non-pointer Arg continuations straightforwardly, with a tag word pushed on top of the non-pointer argument, containing the length in words of the non-pointer argument (usually 1 or 2). A tag can always be distinguished from a pointer argument, because pointer arguments never point to very low addresses.

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 — fast calls to functions taking non-pointer arguments are not at all rare — 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 99.9% of the cases (Section 8). 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--` (Peyton Jones *et al.*, 1999) (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 `stgApply` functions, for various argument combinations. Indeed, we generate them by feeding the desired argument combinations to a generator program.

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`.

6.2 The run-time stgApply functions

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

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

Fig. 5. The generic apply function `StgApplyNP`

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.

6.3 Variations on the theme

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. At the time of writing, GHC does not implement either of these optimisations.

In the `FUN` case of Figure 5 we used a macro `CODE(f)` which extracted the fast entry point of a function closure `f`. Since this is a very common operation, we use this fast-entry code as the info pointer of the closure (see Figure 3), so that we can get the fast-entry code with a single memory reference. There is a down-side to this choice, however: functions are no longer self-evaluating. Under push/enter, we had the convention that jumping to the code for a closure would always evaluate the closure or, in the case of a function, apply the function to the available arguments on the stack and return the result. Hence to evaluate a closure to head normal form we could just enter the closure's code with no arguments on the stack (`Su`==`Sp`).

If we use the fast-entry code of a function as its closure's info pointer, we cannot evaluate an arbitrary closure to head normal form simply by entering it. Instead, we must first check the type of the closure: if it is a function we can return the result immediately, otherwise we enter the closure. Fortunately evaluating an arbitrary closure is rare; most of the time the code generator knows the type of the closure being entered, and can generate the right kind of eval sequence. The classic function that does require the polymorphic eval code is `seq`, which evaluates its first argument without knowing its type.

An alternative approach would be to give every function closure an info pointer that returns immediately, and have a separate entry point in a function's info table (accessed by `CODE(f)`) for calling the function. This would make polymorphic eval code simpler, but would would result in larger info tables and an extra indirection when calling an unknown function.

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 (Section 5.2.1).

- 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 callee, 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 arguments 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 (Jones, 1992). Second, a useful optimisation is to collapse sequences of adjacent update frames into a single frame, by choosing one of the objects to be updated and making all the others be indirections to it. 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.

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 88 programs, and our tables will give minimum, maximum and mean figures *across the whole suite.* However, for reasons of presentation we couldn't include detailed results for all 88 programs in the tables, so we have left out some of the programs with less interesting results (but the aggregate results were still calculated using the whole suite). Outlying results, many of which are discussed in the text, are highlighted in a grey box.

The `nofib` benchmark suite contains programs ranging from micro-benchmarks (`tak`, `rfib`) to larger programs solving "real" problems: `cacheprof` is a program for automatically translating assembly code to insert instructions for dynamic cache profiling, `compress` is an implementation of LZW compression, `prolog` is a Prolog interpreter, and `hidden` is a program for hidden-line removal in 3D rendering. We make no apology for including the micro-benchmarks: in practice even the larger programs often have small inner loops, and the micro-benchmarks are useful for illustrating the boundary cases.

Where appropriate, we will attempt to explain any unusual or extreme results. We investigated individual programs using the following tools:

- GHC has a lightweight profiling system called "ticky-ticky" profiling, which counts the occurrence of certain events during a program run. The events include global counts such as the number of allocations of various kinds and the number of updates, but also per-function counts of the number of calls and allocations within each function. The latter are particularly useful for identifying inner loops for further investigation.
- Cachegrind (part of Valgrind (Seward, n.d.)), the tool we use for counting instructions and memory references, can also give these counts at the granularity of a labelled code block. We found it particularly helpful to compare these results between the push/enter and eval/apply versions of a program, to quickly identify sections of code that were performing a different number of operations - most blocks remained the same or close between the two models. Of course instruction counts and memory references are only a rough indicator of real performance, though.

.

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.

Program	Lines	Calls	Uneval (%)	Unknown (%) <	Unknown (%) =	Unknown (%) >	Known (%) <	Known (%) =	Known (%) >
anna	9561	4047084	0.8	0.0	25.5	0.0	0.6	73.8	0.0
atom	188	10237920	0.0	0.0	5.2	0.0	0.0	94.8	0.0
boyer2	723	295984	0.0	0.0	0.0	0.0	0.0	100.0	0.0
boyer	1014	1387158	0.5	0.0	10.5	0.0	0.5	88.9	0.0
bspt	2141	273402	0.0	0.0	1.9	0.0	0.0	98.1	0.0
cacheprof	2151	19597901	0.3	0.0	25.2	0.0	0.2	74.5	0.0
cichelli	244	5790007	0.0	0.0	19.3	0.0	0.0	80.7	0.0
circsim	668	30421443	0.0	0.0	14.5	0.0	0.0	85.5	0.0
clausify	179	2186312	0.0	0.0	1.7	0.0	0.0	98.3	0.0
comp_lab_zift	884	8581682	0.0	0.0	20.9	0.0	0.0	79.1	0.0
compress2	199	1721537	0.0	0.0	2.6	0.0	0.0	97.4	0.0
compress	736	7816380	0.0	0.0	1.6	0.0	0.0	98.4	0.0
cse	464	24878	1.4	0.4	7.4	0.0	0.2	91.7	0.3
exp3_8	93	8079893	0.0	0.0	0.0	0.0	0.0	100.0	0.0
expert	525	10755	0.3	0.2	19.6	0.0	0.1	80.1	0.1
fem	1286	1440406	0.0	0.0	5.4	0.0	0.0	94.6	0.0
fibheaps	296	1548796	5.1	5.8	8.3	0.0	0.0	85.3	0.6
fluid	2401	392664	2.4	0.0	48.3	0.0	0.1	50.4	1.2
fulsom	1397	4333456	0.4	0.0	25.0	0.0	0.2	74.8	0.0
gamteb	701	2239319	0.0	0.0	7.1	0.0	0.1	91.2	1.6
genfft	502	1587626	0.0	0.0	7.2	0.0	0.0	92.8	0.0
gg	812	397004	0.0	0.0	23.5	0.0	0.1	76.4	0.1
grep	356	102	4.9	4.9	27.5	0.0	3.9	62.7	1.0
hidden	521	36030177	0.1	0.0	13.8	0.0	0.0	86.1	0.1
hpg	2067	2470202	5.3	3.0	21.1	1.1	2.0	72.7	0.1
infer	594	1823681	0.1	0.0	18.8	0.0	0.1	81.1	0.0
integer	68	125287103	0.0	0.0	49.3	0.0	0.0	50.7	0.0
knights	887	233593	0.0	0.1	40.1	0.0	0.0	59.8	0.0
lcss	60	4155607	0.0	0.0	49.0	0.0	0.0	51.0	0.0
life	53	8395883	0.0	0.0	8.2	0.0	0.0	91.8	0.0
lift	2033	21173	18.7	0.5	31.5	0.3	3.6	63.9	0.1
listcopy	527	6372584	0.0	0.0	1.8	0.0	0.0	98.2	0.0
maillist	175	629501	9.5	8.3	35.3	0.0	1.9	53.9	0.5
mandel	498	8396707	0.3	0.0	62.9	0.0	0.0	37.1	0.0
mkhprog	803	59097	0.4	0.3	2.9	0.0	0.0	96.8	0.0
nucleic2	3391	870440	0.0	0.0	10.1	0.0	0.0	89.9	0.0
para	1781	30122407	0.0	0.0	45.0	0.0	0.0	55.0	0.0
paraffins	91	1254290	0.0	0.0	49.5	0.0	0.0	50.5	0.0
parser	3139	865802	0.6	0.0	37.4	0.2	0.0	62.4	0.0
parstof	1280	184871	0.2	0.0	52.9	0.0	0.0	47.1	0.0
pic	527	168978	1.3	0.0	14.8	0.0	0.0	85.0	0.1
pretty	265	1562	3.6	3.6	9.2	0.0	0.1	87.1	0.1
prolog	641	64723	2.1	0.1	24.9	0.1	2.0	73.0	0.0
puzzle	170	7936980	0.0	0.0	31.8	0.0	0.0	68.2	0.0
reptile	1522	359506	0.2	0.1	4.6	0.0	0.0	95.2	0.1
rsa	74	369801	0.0	0.0	0.0	0.0	0.0	100.0	0.0
scc	100	629	1.1	1.0	49.1	0.0	0.2	49.8	0.0
sched	555	856125	0.0	0.0	0.0	0.0	0.0	100.0	0.0
scs	585	28431366	0.5	0.0	17.3	0.0	0.0	82.5	0.2
simple	1129	14398577	0.0	0.0	49.2	0.0	0.0	50.8	0.0
sorting	162	40322	0.0	0.0	22.3	0.0	0.0	77.7	0.0
symalg	1146	80079	0.1	0.0	1.2	0.0	0.0	98.7	0.1
tak	16	2494307	0.0	0.0	0.0	0.0	0.0	100.0	0.0
treejoin	121	3604474	0.0	0.0	10.5	0.0	0.0	89.5	0.0
typecheck	658	18043268	0.5	0.0	27.3	0.0	0.5	72.2	0.0
veritas	11124	21133	1.9	0.4	6.9	0.0	0.1	92.4	0.2
wang	357	1325827	0.0	0.0	4.8	0.0	0.0	95.2	0.0
x2n1	35	1289082	0.0	0.0	78.8	0.0	0.0	21.2	0.0
Min			0.0	0.0	0.0	0.0	0.0	21.2	0.0
Max			18.7	8.3	78.8	1.1	3.9	100.0	1.6
Average			1.0	0.4	20.3	0.0	0.2	79.0	0.1

Fig. 6. Anatomy of calls

Figure 6 shows the relative dynamic frequency of:

- Calls to an unknown (lambda-bound or case-bound) function which turned out to be unevaluated (as a percentage of the total calls),
- Calls to unknown functions with (a) too few arguments, (b) exactly the right number of arguments, and (c) too many arguments (each as a percentage of the total calls),
- Calls to a known (let-bound) function with (a) too few arguments, (b) exactly the right number of arguments, and (c) too many arguments (again, each as a percentage of the total calls).

The last six columns of the table together cover all calls, and add up to 100%. 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 or unknown function gives rise to a subsequent call to the unknown function returned as its result; these unknown calls are included in one of the "unknown calls" columns. 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 three immediate conclusions. First, known calls are common, and often dominate, but unknown calls can be the majority in some programs (e.g. `x2n1`, `mandel`). 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 (c.f. Section 4.3). Third, even unknown calls are almost always to an evaluated function with the correct number of arguments, so it is worthwhile optimising this case. For example, we can pass the arguments to the generic apply function in registers, in the hope that it can just pass them directly to the function[3].

Another thing to note from these results is the wide variety of behaviours; even amongst the larger programs there is significant variation in the proportion of unknown calls made: `bspt` with 2141 lines makes only 1.9% unknown calls, but `fluid` with 2401 lines makes 48.3% unknown calls. One might perhaps guess that larger programs would exhibit "average" behaviour, but this is not reliably the case; execution is often dominated by a handful of inner loops.

There are few remarkable results in Table 6. An extreme out-lier is `x2n1`, which has the highest proportion of unknown function calls (78.8%). The `x2n1` program is micro-benchmark characterised by lots of floating point operations. The inner loop contains this function:

```
f :: Int -> Complex Double
f n = mkPolar 1 ((2*pi)/fromIntegral n) ^ n
```

[3] Our current implementation does perform this optimisation, however. One reason is that on x86, our primary implementation architecture, there are no registers available for parameter passing anyway.

The function `mkPolar` ends up fully inlined and reduced to a constructor application and a couple of primitive floating point operations, similarly the division and `fromIntegral` are reduced to primitives.

The exponentiation operator, `(^)`, unfortunately remains overloaded. It is defined in the `Prelude`, and makes calls to overloaded functions on each iteration; and by definition each call to an overloaded function will be unknown at the call site. Overloading is a plentiful source of unknown function calls. Other optimisations could improve the quality of the code here, but that is an orthogonal issue as far as this paper is concerned.

8.2 Argument patterns

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 in the implementation of the `IO` and `ST` monads. The state token is always passed as the last argument, which is why we need `ppv` but not `pvp` and `vpp`, for example.

The general conclusion is clear: a double-handful of 9 argument patterns is enough to cope with 99.99% of all situations.

Some programs have unusual results:

- `fibheaps` has an unusually large number of `pv` calls. A simple inspection of the program shows that it contains a lot of code in the `ST` monad, which accounts for the high use of the `pv` pattern.
- `maillist` does a lot of file manipulation work in the `IO` monad. This accounts for its use of the `ppv` pattern.
- `boyer2`, `grep` and `sched` appeared to perform a high proportion of `v` calls, but in fact these programs performed a very low number of unknown calls in total (26 for `grep`, and 12 for `boyer2` and `sched`). It just so happened that the small amount of `IO` monad code at the top level of the program accounted for many of those unknown calls.

Several of the programs appear to have a preference for one or two of the argument patterns. For example, `wang` performs almost exclusively `pp` calls. Investigating the program reveals why: these calls all come from a local copy of the `foldr` function (this benchmark is automatically generated code):

```
f_foldr::(t1 -> t2 -> t2) -> t2 -> [t1] -> t2;
f_foldr a_op a_r []=a_r;
f_foldr a_op a_r (a_a:a_x)=a_op a_a (f_foldr a_op a_r a_x);
```

Calls to the unknown function `a_op` in the body of `f_foldr` are `pp` calls, and by

	Argument pattern (% of all unknown calls)									
Program	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
atom	0.0	4.8	0.0	0.6	0.0	94.6	0.0	0.0	0.0	0.0
boyer2	58.3	41.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
boyer	0.0	92.8	0.0	7.2	0.0	0.0	0.0	0.0	0.0	0.0
bspt	0.5	70.2	0.0	6.2	0.0	11.8	0.0	11.3	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
cichelli	0.0	10.4	0.0	89.6	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
clausify	0.0	0.4	0.0	99.6	0.0	0.0	0.0	0.0	0.0	0.0
comp_lab_zift	0.0	3.4	0.0	96.6	0.0	0.0	0.0	0.0	0.0	0.0
compress2	1.1	98.8	0.0	0.0	0.0	0.0	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
cse	0.4	59.2	0.0	9.9	0.0	30.6	0.0	0.0	0.0	0.0
exp3_8	5.6	59.3	0.0	34.5	0.6	0.0	0.0	0.0	0.0	0.0
expert	2.5	62.6	0.1	32.4	0.4	2.0	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
fibheaps	0.0	43.2	13.7	43.1	0.0	0.0	0.0	0.0	0.0	0.0
fluid	0.0	61.9	0.0	34.2	0.0	3.9	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
gamteb	0.1	96.4	0.0	3.5	0.0	0.0	0.0	0.0	0.0	0.0
genfft	0.0	1.6	0.0	98.4	0.0	0.0	0.0	0.0	0.0	0.0
gg	0.0	53.7	0.0	46.3	0.0	0.0	0.0	0.0	0.0	0.0
grep	58.6	34.5	3.4	0.0	3.4	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
hpg	24.1	55.2	0.0	10.4	2.7	1.4	6.2	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
integer	0.0	50.0	0.0	50.0	0.0	0.0	0.0	0.0	0.0	0.0
knights	0.0	4.1	0.0	95.9	0.0	0.0	0.0	0.0	0.0	0.0
lcss	0.0	1.1	0.0	0.0	0.0	98.9	0.0	0.0	0.0	0.0
life	0.0	3.3	0.0	0.2	0.0	96.5	0.0	0.0	0.0	0.0
lift	0.1	57.9	0.0	40.4	0.0	1.6	0.0	0.0	0.0	0.0
listcopy	0.2	28.5	0.0	71.3	0.0	0.0	0.0	0.0	0.0	0.0
maillist	31.8	35.1	1.1	1.1	15.5	14.4	1.1	0.0	0.0	0.0
mandel	0.0	13.4	0.0	86.6	0.0	0.0	0.0	0.0	0.0	0.0
mkhprog	12.2	48.7	2.1	31.5	3.7	1.9	0.0	0.0	0.0	0.0
nucleic2	0.0	56.0	0.0	44.0	0.0	0.0	0.0	0.0	0.0	0.0
para	0.0	22.5	0.0	77.4	0.0	0.0	0.0	0.0	0.0	0.0
paraffins	0.0	99.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1
parser	0.0	11.5	0.0	88.5	0.0	0.0	0.0	0.0	0.0	0.0
parstof	0.0	94.3	0.0	5.7	0.0	0.0	0.0	0.0	0.0	0.0
pic	8.7	75.8	0.0	15.4	0.0	0.0	0.1	0.0	0.0	0.0
pretty	3.6	2.6	0.0	93.9	0.0	0.0	0.0	0.0	0.0	0.0
prolog	0.4	78.2	0.0	21.3	0.0	0.0	0.0	0.0	0.0	0.0
puzzle	0.0	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
reptile	0.2	72.1	0.0	26.9	0.0	0.9	0.0	0.0	0.0	0.0
rsa	29.1	69.6	1.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0
scc	2.9	12.8	0.0	84.3	0.0	0.0	0.0	0.0	0.0	0.0
sched	58.3	41.7	0.0	0.0	0.0	0.0	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
simple	0.0	20.1	0.0	79.9	0.0	0.0	0.0	0.0	0.0	0.0
sorting	0.3	66.5	0.0	33.3	0.0	0.0	0.0	0.0	0.0	0.0
symalg	8.3	55.0	0.0	36.8	0.0	0.0	0.0	0.0	0.0	0.0
tak	1.9	61.8	0.0	35.7	0.6	0.0	0.0	0.0	0.0	0.0
treejoin	0.2	99.8	0.0	0.0	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
veritas	1.2	47.3	0.0	50.2	0.0	1.1	0.0	0.0	0.3	0.0
wang	0.0	0.0	0.0	100.0	0.0	0.0	0.0	0.0	0.0	0.0
x2n1	0.0	17.6	0.0	82.4	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

Fig. 7. Argument patterns

looking back at Figure 6 we can see that they were all in fact calls to functions of two arguments.

The other programs which have a high proportion of one particular argument pattern are similar: there is often a single unknown call in the inner loop of the program.

8.3 The bottom line

What really matters in the end is time and space. Figure 8 shows the percentage change we measured in moving from push/enter to eval/apply. The runtime figures are wall-clock times, averaged over 5 runs, discounting any programs that ran for less than 0.5 seconds on our 1GHz Pentium III (around half of the suite). The machine was otherwise unloaded at the time of the test.

Somewhat to our surprise, there is only a small difference between the two models, with eval/apply edging out push/enter by around 2-3% of runtime on average. We discuss the runtime differences in more detail in the rest of this section.

The table also gives differences in code size, heap allocations, instructions executed and memory read/write references. Code size differences are due to two main factors:

- Increased size of the runtime due to the addition of the `stgApply` functions.
- Reduction of the size of individual compiled modules, due to the removal of the per-function slow entry code.

In small programs, the increased size of the runtime outweighs the per-module reduction, and we see a small overall increase in code size. On larger programs (e.g. `anna`, `veritas`) the per-module reduction starts to win out, and we see a reduction in code size.

Heap allocation is largely unaffected by the change from push/enter to eval/apply, as can be seen in the "Alloc" column of Figure 8. The small change in allocation is the difference between two factors pulling in opposite directions. Firstly, eval/apply will allocate a PAP when returning a function applied to too few arguments, whereas push/enter may get away without heap allocation because the function can find its missing arguments on the stack. Hence eval/apply will allocate more PAPs. Secondly, however, the PAPs in eval/apply may be slightly smaller than those for push/enter, because there is no need to tag their non-pointer components (Section 4.4).

Instructions and memory references were measured using the Cachegrind tool, which is a part of the Valgrind dynamic program analysis toolset. Cachegrind has the ability to produce per-function instruction and memory reference counts, which we used to try to narrow down and explain differences in real-time performance. When examining programs in this way, we found that differences between the push/enter and eval/apply versions of programs fell into the following categories:

- **Updates.** Under eval/apply, update frames are one word smaller than under push/enter (2 words instead of 3), because there is no `Su` register to save in

Program	Code size	Alloc	Instrs	Memory reads	Memory writes	Run-time
	Eval/apply change (Δ%)					
anna	-5.1	+1.7	+2.0	+2.6	-3.3	-0.8
atom	-0.8	+0.0	-7.4	-5.2	-12.0	-5.5
boyer	+4.1	+0.5	-2.5	-1.3	-10.5	-
boyer2	+4.0	+0.0	-2.7	-0.1	-3.2	-
bspt	-0.8	-0.0	-6.1	-3.6	-8.3	-
cacheprof	-4.0	+0.4	+10.8	+10.3	+0.3	+4.1
cichelli	+2.9	-0.0	-1.8	+0.4	-6.1	-2.4
circsim	+0.3	-0.0	+0.3	+1.1	-9.5	-4.7
clausify	+6.1	-0.0	-1.5	+2.0	+0.8	-
comp_lab_zift	+3.3	-0.0	-1.3	+0.2	-9.6	-7.5
compress	+2.2	-0.0	+1.8	+3.2	+3.7	+1.8
compress2	+3.0	-0.0	-0.7	-0.4	-0.3	-1.9
cse	+5.0	-0.0	-5.9	-4.3	-8.8	-
exp3_8	+1.5	+0.0	-2.5	+1.6	-9.8	-23.2
expert	+1.3	-2.4	+0.6	+1.7	-6.3	-
fem	-0.9	+0.0	-5.6	-3.3	-7.8	-
fibheaps	+1.1	+0.9	+3.4	+4.5	-3.1	-
fluid	-2.8	+0.1	+5.8	+5.9	-4.9	-
fulsom	-2.2	+0.1	-2.5	-2.4	-8.0	-3.7
gamteb	-0.8	+0.1	-0.5	+0.8	-0.8	+2.2
genfft	+5.8	-0.0	-8.1	-6.3	-11.0	-
gg	-2.6	+0.1	-0.8	+0.5	-5.1	-
grep	+1.8	+0.1	-0.0	+0.0	-0.5	-
hidden	-2.4	+0.0	+3.3	+4.0	-6.1	+2.1
hpg	-2.8	+0.2	+4.0	+5.4	-4.6	-5.6
infer	-1.6	+0.2	+2.5	+2.4	-1.0	+0.0
integer	+2.0	+0.0	+3.8	+3.5	-13.7	-0.8
knights	+2.7	-0.2	+7.5	+6.1	-0.5	-
lcss	+1.4	-0.0	+1.8	+0.4	-6.9	-0.4
life	+6.5	+0.0	-5.0	-3.5	-9.3	-5.4
lift	+2.5	-0.1	-2.2	-1.2	-8.9	-
listcopy	+5.6	+0.0	-10.2	-8.1	-11.3	-2.4
maillist	+3.4	+0.0	+3.8	+3.4	-3.6	-
mandel	-0.6	+0.0	+4.4	+3.4	-5.2	+3.3
mkhprog	+2.5	+0.0	-6.8	-4.4	-8.4	-
nucleic2	+0.6	-0.0	-4.3	-3.3	-7.1	-8.6
para	+2.2	-0.0	+6.2	+5.6	-8.8	+6.9
paraffins	+0.7	-0.0	+1.0	+0.8	-1.9	+1.1
parser	-1.1	-0.0	+1.8	+2.6	-4.3	-
parstof	-0.3	+0.0	+11.6	+9.4	-3.6	-
pic	-0.8	-0.0	+0.2	+0.7	-1.1	-
pretty	+0.8	-0.0	-1.3	-0.8	-5.9	-
prolog	+1.5	+2.8	+5.2	+5.4	-1.2	-
puzzle	+5.3	-0.0	+1.2	+1.1	-5.9	-12.1
reptile	-1.9	-0.0	-7.7	-4.9	-8.8	-
rsa	+0.5	-0.0	+1.3	+1.6	+1.5	+0.0
scc	+7.7	+0.1	+1.2	+1.2	-1.0	-
sched	+6.2	-0.0	-1.2	+0.2	-7.6	-
scs	-2.4	+0.0	+0.7	+1.4	-2.4	-3.6
simple	-1.8	+0.0	+3.5	+2.5	-4.7	+1.5
sorting	+3.9	-0.0	+1.8	+2.6	-5.3	-
symalg	-1.9	+0.0	+0.0	+0.1	+0.2	-3.1
tak	+1.9	+0.4	+9.1	+20.8	+21.4	-
treejoin	+3.6	+0.0	-2.7	-1.6	-5.7	-0.9
typecheck	+4.6	+1.2	+6.8	+6.7	-4.8	+3.1
veritas	-5.2	-0.7	-4.6	-3.5	-8.0	-
wang	+0.8	-0.0	-1.9	-1.6	-4.7	-3.0
x2n1	-0.4	-0.0	+5.5	+3.9	-5.9	-20.5
Min	-5.2	-2.4	-10.2	-8.1	-13.7	-23.2
Max	+7.7	+2.8	+11.6	+20.8	+21.4	+6.9
Geometric Mean	+1.8	+0.1	+0.0	+1.1	-4.9	-2.8

Fig. 8. Space and time

the frame. In the x86 implementation on which these results were taken, the `Su` register was actually stored in a memory location due to the lack of real machine registers. The absence of `Su` in eval/apply therefore shows up as a reduction in memory reads/writes in Figure 8.

- **Walking the chain of update frames.** The benefit due to the reduction in the size of update frames is balanced to some extent by the extra work that has to be done when traversing the chain of update frames on the stack, as described in Section 7.
- **Unknown call overhead.** The difference in unknown call behaviour shows up as a high instruction count in the `stgApply` routines for eval/apply, compared to instructions spread out amongst the slow entry points of the functions being called in push/enter. Comparing the totals gives instruction counts that are roughly the same, with eval/apply perhaps taking a few more instructions in unknown call overhead. However, we expect there to be a performance benefit due to the extra locality in the eval/apply version.
  We expect an architecture with more argument registers available to benefit more from eval/apply, because many unknown calls would take place with arguments in registers compared to the stack under push/enter.
- **Calling conventions.** In general, when calling a function, our code generator dedicates one register to pass the address of the function closure. This is unnecessary in the case of a top-level function, which has no free variables, so in principle it would be possible to re-use the function-pointer register as an argument register for calls to top-level functions.
  For push/enter we implemented this optimisation, so that top-level functions have a different calling convention than non-top-level ones. However, under eval/apply we found that a consistent calling convention for both top-level and non-top-level functions avoided a lot of complexity in the `stgApply` functions. Moreover, if the calling convention for all functions is the same, we can adopt that same calling convention for the `stgApply` functions, and hence optimise the common case where the function is evaluated and has the correct arity, and `stgApply` is just transferring control directly to the function. In practical terms, this means we can pass arguments in registers to `stgApply`, and `stgApply` can simply jump to the entry code for the function.
  On the x86 architecture, in fact there are no registers available for parameter passing, although we do have a machine register for passing the address of the function closure. This means that with eval/apply, *no* arguments to top-level functions are passed in registers, compared to one argument with push/enter. We do not have measurements that isolate the effect of this difference taken by itself, but we believe it to have little real effect on runtimes. In some of the programs we investigated we saw some small savings in push/enter due to the argument register, and we also saw cases where it made the code worse (because the register had to be immediately saved on the stack on entry to the function, perhaps requiring an extra stack check).
- **Entry convention.** In Section 6.3 we discussed the fact that we chose to make the info pointer of a function closure into its fast-entry code, at the

cost of extra tests in polymorphic eval code. This decision affects the `para` benchmark, as we discuss below.

We can now offer some explanation for some of the programs with outlying results in Figure 8, in terms of the factors outlined above:

- `exp3_8` was 23.2% faster. This is largely due to the fact that `exp3_8` spends most of its time doing updates. In this case, the extra memory reads when traversing the update frame chain balance out the memory reads saved by the smaller size of update frames, but the difference in writes is much greater.
- `x2n1` was 20.5% faster. These savings again appear to be mostly due to improvements in the update code: the heaviest-hit basic block in `x2n1` is the update routine. There are lots of small changes in instruction counts for basic blocks across this program however, so this is probably not the whole story. This program performs a lot of unknown calls, and so we see a lot of activity in the `stgApply` routines in the eval/apply version, compared with activity spread across the slow entry points for various functions in the push/enter case, as we would expect. We also noticed some differences due to calling conventions.
- `puzzle` was 12.1% faster. Again, updates dominate the runtime, and eval/apply consequently gains a bonus.
- `para` was 6.9% slower with eval/apply. There are some savings in the update code as usual, but there are some losses in the unknown call code. There are some further losses due to having to check closure types for an eval in the runtime (see entry conventions above). Interestingly, while investigating this program we discovered one place in the eval/apply version of the runtime which was checking the closure type for an eval unnecessarily; fixing that reduced the difference in instructions for this test from +6.2% to +3.2%, but did not have any effect on runtime[4].

9 Related work

Two of the most popular and influential abstract machines for lazy languages, the G-machine (Johnsson, 1984) and the Three Instruction Machine (TIM) (Fairbairn & Wray, 1987), 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 (Faxén, 1997). 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 possibility of such specialisation is an additional benefit of eval/apply (Boquist (Boquist, 1999) describes an extreme version). Eager

[4] We did not re-run the entire testsuite with this change.

Haskell, an unusual implementation of Haskell based on eager evaluation, also uses eval/apply (Maessen, 2002).

Caml, a call-by-value language, uses push/enter for the interpreter (Leroy, 1990), 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 of a higher order language, 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. We are confident of this result for a non-strict language, and we believe that the benefit is likely to be more pronounced for a strict one.

Many of the complexities of push/enter are caused by efficiency hacks, however. For an interpreter, where performance is not such an issue, these hacks are not important, and push/enter may well be a more elegant solution.

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

References

Boquist, Urban. 1999 (April). *Code optimisation techniques for lazy functional languages.* Ph.D. thesis, Chalmers University of Technology, Sweden.

Douence, Rémi, & Fradet, Pascal. (1998). A systematic study of functional language implementations. *Acm transactions on programming languages and systems*, **20**(2), 344–387.

Fairbairn, Jon, & Wray, Stuart. (1987). TIM - a simple lazy abstract machine to execute supercombinators. *Pages 34–45 of:* Kahn, G (ed), *Proc ifip conference on functional programming languages and computer architecture, portland.* Springer Verlag LNCS 274.

Faxén, Karl-Filip. 1997 (June). *Analysing, transforming and compiling lazy functional programs.* Ph.D. thesis, Department of Teleinformatics, Royal Institute of Technology.

Flanagan, C, Sabry, A, Duba, B, & Felleisen, M. (1993). The essence of compiling with continuations. *Pages 237–247 of: ACM Conference on Programming Languages Design and Implementation (PLDI'93).* ACM.

Johnsson, Thomas. (1984). Efficient compilation of lazy evaluation. *Proc sigplan symposium on compiler construction, montreal.* ACM.

Jones, R. (1992). Tail recursion without space leaks. *Journal of functional programming*, **2**(1), 73–80.

Launchbury, J. (1993). A natural semantics for lazy evaluation. *Pages 144–154 of: 20th ACM Symposium on Principles of Programming Languages (POPL'93).* ACM.

Leroy, X. 1990 (Feb.). *The Zinc experiment: an economical implementation of the ML language.* Tr 117, inria-rocquencourt. INRIA.

Maessen, Jan-Willem. 2002 (June). *Hybrid eager and lazy evaluation for efficient compilation of haskell.* Ph.D. thesis, Massachusetts Institute of Technology.

Peyton Jones, Simon, Ramsey, Norman, & Reig, Fermin. (1999). `C--`: a portable assembly language that supports garbage collection. *Pages 1–28 of:* Nadathur, G (ed), *International conference on principles and practice of declarative programming*. Lecture Notes in Computer Science, no. 1702. Berlin: Springer.

Peyton Jones, SL. (1992). Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of functional programming*, **2**(2), 127–202.

Peyton Jones, SL, & Launchbury, J. (1991). Unboxed values as first class citizens. *Pages 636–666 of:* Hughes, RJM (ed), *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*. Lecture Notes in Computer Science, vol. 523. Boston: Springer Verlag.

Seward, Julian. *Valgrind*. `http://valgrind.kde.org`.