High-Performance Client-Side Web Applications through Haskell EDSLs

Anton Ekblad
Chalmers University of Technology, Sweden
antonek@chalmers.se

Abstract

We present *Aplite*, a domain-specific language embedded in Haskell for implementing performance-critical functions in client-side web applications. In Aplite, we apply partial evaluation, multi-stage programming and techniques adapted from machine code-targeting, high-performance EDSLs to the domain of web applications. We use Aplite to implement, among other benchmarks, procedural animation using Perlin noise (Perlin 2002), symmetrical encryption and K-means clustering, showing Aplite to be consistently faster than equivalent hand-written JavaScript – up to an order of magnitude for some benchmarks. We also demonstrate how Aplite's multi-staged nature can be used to automatically tune programs to the environment in which they are running, as well as to inputs representative of the programs' intended workload.

High-performance computation in the web browser is an attractive goal for many reasons: interactive simulations and games, cryptographic applications and reducing web companies' electricity bills by outsourcing expensive computations to users' web browsers. Similarly, functional programming in the browser is attractive due to its promises of simpler, shorter, safer programs. In this paper, we propose a way to combine the two.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (Functional) Programming; D.3.2 [*Language Classifications*]: Specialized application languages

Keywords domain-specific languages, web applications, multi-stage programming

1. Introduction

As the rise of the web as an application platform continues unabated, we expect our browsers to be able to do more and more. Applications that were unthinkable as web applications just a few years ago – word processing, photo editing and even relatively complex computer games – are now everyday occurrences in the browser. This is hardly surprising: the speed and ease with which one can develop and deploy a web application, not having to deal with a combinatorial explosion of possible client systems and configurations, is most appealing. Equally so is the low barrier for users to try out a new application. Another compelling argument for developing for the web is the software-as-a-service business model, where software is leased on a monthly basis rather than sold. This enables updates and bug fixes to be deployed to users as they become available, rather than having to wait for the next release cycle.

JavaScript: Language of the Web Unfortunately, JavaScript, the de facto language of the web, has its share of shortcomings. While developers don't have to contend with a multitude of operating systems and system configurations when writing JavaScript, they instead must contend with a slightly smaller multitude of web browsers and JavaScript engines. Although a program written for one modern JavaScript engine will generally have the same semantics when moved to another one – in contrast to native applications – the same can not be said for performance. A program that runs at a passable speed on Google's V8 JavaScript engine may run significantly slower on Mozilla's SpiderMonkey, and vice versa. During our experiments, we observed some programs to run as much as an order of magnitude faster on V8 than on SpiderMonkey, while modifying the program to perform well on SpiderMonkey instead caused it to run significantly slower on V8.

Even if we manage to produce code that performs similarly across different browsers, we might not get the speed that we want from our JavaScript code. JavaScript can perform on par with lower-level languages under the right circumstances. The *ASM.js* low-level subset of JavaScript, for instance, can perform well enough to be used to implement demanding 3D games (Zakai 2013). However, ASM.js is intended as a compilation target, and is thus not well suited to being written by humans. Disciplined use of plain JavaScript can give some of the benefits of ASM.js, but JavaScript contains several constructs that are highly problematic from an optimization point of view: higher order functions, the *eval* function, reflection and reliance on weak, dynamic typing can all have adverse effects on JavaScript performance, to name a few. A single performance misstep may lead to sub-par performance in *all* browsers. Optimization-unfriendly code, in addition to being slow in and of itself, may in some cases cause deoptimization of *all surrounding code* by invalidating assumptions needed for efficient compilation of a given function (Antonov 2015). The untyped nature of JavaScript also means that the programmer needs to manually keep track of the types of all values their programs operate on, adding manual type checks and additional testing to compensate for a lack of static guarantees. Being forced to write fast, clever, low-level code only exacerbates this problem.

Considering that we generally do not trust developers to produce type-safe code through sheer discipline – hence the use of type checkers – it would seem unwise to trust those same developers to be disciplined enough to produce code which is guaranteed to avoid performance pitfalls. Doubly so when the performance pitfalls differ depending on the user's choice of web browser.

Functional Languages to the Rescue? As functional programmers, can we use functional programming languages to solve the

Haskell'16, September 22-23, 2016, Nara, Japan
ACM. 978-1-4503-4434-0/16/09...
http://dx.doi.org/10.1145/2976002.2976015

aforementioned problems? Most major functional languages can now be compiled to JavaScript, including some designed specifically for the web. Haskell through GHCJS (Nazarov et al. 2015), Haste (Ekblad 2015a) or UHC (Dijkstra et al. 2013), O'Caml through Js_of_ocaml (Vouillon and Balat 2014) are all instances of the former, while Elm (Czaplicki 2012) and PureScript (Freeman 2015) are instances of the latter.

However, programs written in many of these languages can be quite slow when executed in the browser, especially for computationally heavy tasks. Not only can the JavaScript engine's garbage collector be a poor match for functional programs, but the JavaScript-targeting compilers also generally lack the hundreds of person-years of effort that went into their native cousins, and many of the tricks used to speed up execution of functional languages on stock hardware do not apply to JavaScript. Additionally, JavaScript code compiled from functional languages often make heavy use of constructs, higher order functions in particular, that are hard for JavaScript engines to fully optimize (Ekblad 2015a).

EDSLs to the Rescue! We don't want to force developers to manually enforce type-correctness and disciplined, performance-oriented programming, as is needed when using JavaScript. However, as poor performance is one of the major problems we would like to solve, we can also not accept trading convenience and safety for even *lower* performance, as with functional languages compiled to JavaScript.

Instead, we look to *EDSLs* – embedded, domain-specific languages – for a solution. The idea of using domain specific languages to achieve high performance is not new: in order to make a language excel in a specific domain, its syntax and types are restricted so that only problems in the given domain are expressible. Without having to support the generality of a general-purpose programming language, the compiler is free to focus on optimizing for the given domain, being able to make assumptions about programs that would otherwise be invalid. Meanwhile, less performance-sensitive code may be written in the slower, more expressive host language, giving us either performance or convenience on a case by case basis.

A type-safe, domain-specific language for computations which are highly optimizable can elegantly solve the aforementioned problems with resorting to hand-written JavaScript. Furthermore, embedding the language in a higher-level host language yields additional benefits. EDSL code may take advantage of the host language for advanced meta-programming capabilities (Axelsson et al. 2010). This embedding also elegantly enables run-time specialization of EDSL code: being compiled on-line by the host language program itself, EDSL code may be specialized to user input during run-time, possibly yielding more efficient code; a technique commonly known as partial evaluation (Futamura 1999). The host program may take advantage of its knowledge of the browser it is currently executing in to tune the code generator to produce the most efficient code possible for the current environment.

It should be noted that EDSLs for the browser is not an all-or-nothing proposition. Low-level, performance-focused EDSLs are likely to yield the best results performance-wise, while using Haskell without any EDSLs at all may give programmers more expressive power at the cost of performance. However, one might also consider taking the middle road: a higher-level, web-focused EDSL might yield a smaller performance benefit than a lower-level one, but allows the programmer to still work at a comparatively higher level of abstraction. One might also consider mixing EDSLs of different layers of abstraction, giving programmers more freedom to choose the tool for the task at hand. This approach is discussed further in section 6.

Our Contribution This paper makes the following contributions:

- In sections 2 and 4, we describe the *Aplite* multi-backend EDSL, targeting a low-level subset of JavaScript based on *ASM.js*, compiled and executed on demand in the user's browser. Aplite constitutes a special case of multi-stage programming, allowing specializing code to the browser as well as to user input. With these properties, Aplite is designed to mitigate the performance problems of computationally heavy web applications.
- In section 3, we demonstrate how the multi-staged, multi-backend nature of Aplite can be used to automatically select at run-time the most efficient backend for any given program in the current browser environment, and to seamlessly integrate Aplite code into Haskell programs, using type families to separate pure Aplite kernels from impure ones allowing both to be imported using the same interface.
- In section 5, we implement several programs, including procedural animations using Perlin noise, symmetric encryption and K-means clustering, using Aplite, and show consistent performance improvements over equivalent, hand-written JavaScript programs, up to an order of magnitude for some benchmarks.

2. Aplite: A High-Performance JavaScript EDSL

The Aplite language is implemented as a strongly typed DSL embedded in Haskell. More specifically, it is intended for use with web-targeting Haskell dialects such as GHCJS, Haste or UHC. It has two main design goals: to allow numeric computations to be expressed in a way that is efficiently executable across JavaScript implementations and to easily integrate with high-level Haskell code. It effectively creates a hybrid programming environment, where control flow and event handling may be expressed in high-level Haskell, while performance-critical computations may be outsourced to highly efficient Aplite kernels. The language is deeply embedded and intended to serve as a high-performance core on which to base higher-level functional abstractions. Thus it takes on a role similar to that of the deep embedding in Axelsson's and Svenningsson's work on combining deep and shallow embedded DSLs (Svenningsson and Axelsson 2012). This makes the language easily extensible: rich features may be added on top, compiling down to a relatively small and simple core. The code generator and user interface of the language may thus be extended and improved independently of each other.

Due to the first design goal – restricting ourselves to efficient code only – the language is quite simple. It allows programmers to express bitwise operations over integers, arithmetic operations over floating point and integer values, and access to mutable arrays, references, conditional statements and loops in an *Aplite* monad, which is reified using the technique presented by Svenningsson and Svensson (Svenningsson and Svensson 2013).

The list of supported constructs is heavily guided by what constructs are supported by the low-level, highly efficient *ASM.js* subset of JavaScript, which is covered in more detail in section 4.1. Lambda abstractions and general recursion are not supported by Aplite's syntax. Instead we rely on Haskell functions to break up Aplite programs in manageable units, inlining all applications. The pros and cons of this approach are explored further in section 6.

Figure 2 shows a simple example of an Aplite program, squaring the contents of an array.

Unlike many EDSLs, Aplite programs are not compiled beforehand, nor are they complete applications on their own. Aplite programs have no way of communicating with the outside world, being limited to a strictly computational role. Instead, Aplite programs are *imported* into the host program using the *aplite* bridging function. This function takes as its input an Aplite function over n arguments $a_1 \rightarrow ... \rightarrow b$, and returns a native Haskell function over n arguments $a'_1 \rightarrow ... \rightarrow b'$, where the Haskell types $a'_{1..n}$ and b' are

```
-- Expression language
type CExp a
instance Num a ⇒ Num (CExp a)
instance Bits a ⇒ Bits (CExp a)

-- Comparisons
(#&&) :: CExp Bool → CExp Bool → CExp Bool
(#==) :: JSType a ⇒ CExp a → CExp a → CExp Bool
...

-- Numerics
sqrt_ :: JSType a ⇒ CExp a → CExp a
...

-- Command language
type Aplite a
instance Monad Aplite

-- References
type Ref a
initRef   :: CExp a → Aplite (Ref a)
getRef    :: Ref a → Aplite (CExp a)
setRef    :: Ref a → CExp a → Aplite ()
modifyRef :: Ref a → (CExp a → CExp a) → Aplite ()

-- Arrays
type Arr i e
newArr    :: Ix i
          ⇒ CExp i → Aplite (Arr i e)
getArr    :: Ix i
          ⇒ Arr i e → CExp i → Aplite (CExp e)
setArr    :: Ix i
          ⇒ Arr i e → CExp i → CExp e → Aplite ()
modifyArr :: Ix i
          ⇒ Arr i e → CExp i → (CExp e → CExp e)
          → Aplite ()

-- Control flow
data Border i = Incl i | Excl i
for :: Integral i
    ⇒ (i, Int, Border i)
    → (CExp i → Aplite ())
    → Aplite ()
while :: Aplite Bool → Aplite () → Aplite ()
iff :: CExp Bool → Aplite () → Aplite ()
    → Aplite ()
ifE :: CExp Bool → Aplite (CExp a) → Aplite (CExp a)
    → Aplite (CExp a)
```

Figure 1: Constructs of the Aplite language

```
square :: CExp Int → Arr Int Double → Aplite ()
square len arr =
  for(0, 1, Excl len) $ λi →
    modifyArr arr i (**2)
```

Figure 2: An Aplite example: squaring an array of numbers

```
squareArray :: Int → IOUArray Int Double → IO ()
squareArray = aplite square

main :: IO ()
main = do
  arr ← newListArray (0, 9) [1..10]
  squareArray 10 arr
  getElems arr >>= print
```

Figure 3: Calling Aplite from Haskell

isomorphic to the Aplite types $a_{1..n}$ and b respectively. Figure 3 shows how functions written using Aplite can be called from the host program. A mutable array containing the numbers 1 to 10 is created, passed to the *square* function from figure 2 which updates the array to contain the squares of its original contents, converted to a list and printed.

Note how the type of the imported *squareArray* function differs from that of *square*: the *Aplite* monad is replaced by *IO*, the *Word32* is no longer locked up in Aplite's *CExp* expression type, and the array has become a standard unboxed, mutable Haskell array. On import, the *aplite* function automatically marshals arguments and return values between Haskell and Aplite, provided that the type of the Aplite function is isomorphic to that of the import. Attempting to import an Aplite function under an incompatible type signature yields a Haskell type error at compile-time. Compilation of any well-typed Aplite program is guaranteed to succeed.

This automatic conversion of arguments and return values not only makes it easy to use Aplite in a higher-level Haskell control program, but also ensures that values can not flow unchecked, by closure or otherwise, between Haskell and Aplite code. This property lets us import certain Aplite functions as though they were pure. While any Aplite function may be imported into the *IO* monad, functions which are guaranteed to contain no observable effects can safely be imported as pure Haskell functions as well, as in the example given by figure 4. A function is considered safe to import purely if it does not accept any mutable arrays as arguments. Mutable references are disallowed in imports in general, and thus do not affect the purity of an import. Returning a mutable array is perfectly safe however, as the ban on mutable arguments ensures that any such array must have been created within the function. This means that even if the function were to have mutated the array, nobody else would have had a reference to the array to observe the mutation taking place. Attempting to import a potentially effectful function as a pure function yields a type error. The type-level implementation of these restrictions is further discussed in section 3.

Aplite takes advantage of Haskell's non-strict semantics to provide on-demand compilation and loading of functions. Up until the point where evaluation of *squareArray* is actually forced, it is merely a chunk of syntax tree, sitting around awaiting eventual compilation. This means that not only can we defer the cost of compilation until we know that compiling a function would be productive, but functions which end up not being used can be pruned during the compiler's normal dead code elimination pass, lowering the amount of data that needs to be sent to client browsers before execution can begin.

Figure 1 gives an overview of the constructs of the Aplite language. The *JSType* type class contains all permissible base types of the language: signed and unsigned integers, floating point numbers, and booleans. The *Border* construct describes a border value for a for-loop: a border value of *Incl n* indicates that a for-loop with loop variable i will execute while $i <= n$, while *Excl n* indicates that the loop will execute while $i < n$.

```
fib :: Int → Double
fib = aplite $ λx → do
  r1 ← initRef 0
  r2 ← initRef 1
  for (1, 1, Excl x) $ λi → do
    x1 ← getRef r1
    x2 ← getRef r2
    setRef r1 x2
    setRef r2 (x1 + x2)
  getRef r1
```

Figure 4: Calculating the nth Fibonacci number with Aplite

```
type family ApliteSig a where
  ApliteSig (a → b) = ApliteArg a → ApliteSig b
  ApliteSig a       = ApliteResult a

type family ApliteArg a where
  ApliteArg Double         = CExp Double
  ApliteArg Int32          = CExp Int32
  ...
  ApliteArg (IOUArray i e) = Arr i e

type family ApliteResult a where
  ApliteResult (IO (IOUArray i e)) = Aplite (Arr i e)
  ApliteResult (IO ())             = Aplite ()
  ApliteResult (IO a)              = Aplite (CExp a)
```

Figure 5: Deriving Aplite types from Haskell equivalents

3. Interfacing with Haskell

An Aplite function is imported into its host programs by passing it to the *aplite* function and assigning the result an appropriate type signature, giving the type at which we would like to use the function in our Haskell host program. A corresponding Aplite-level type signature is derived from this Haskell-level type signature in order to check that it is actually possible to import the function at the type that we want. Deriving the Aplite-level type from the Haskell-level type instead of the other way around ensures that we do not have to explicitly type Aplite programs before passing them to *aplite* – the appropriate type will be inferred from the Haskell-level type, ambiguity avoided.

3.1 Deriving Aplite Types

The process of deriving an Aplite-level type from its Haskell-level equivalent is relatively straightforward. We recurse through arguments of the Haskell-level type, changing the types of the arguments into their Aplite equivalents. Figure 5 shows the procedure implemented using closed type families (Eisenberg et al. 2014). The *ApliteArg* type family contains conversions for all supported argument types, while the *ApliteResult* family covers all supported return values. The *CExp* type is the type of Aplite expressions.

However, this only covers imports in the *IO* monad. In order to support pure imports of "safe" functions as well, we must extend this scheme to cover the notion of pure and impure functions, and somehow guard against impure functions being imported purely – recall that a function is considered impure if it accepts a mutable array as an argument.

We do this by introducing another type family – *Purity* – which looks at the return type of any function type to determine if it is

```
data Pure
data Impure

type family Purity a where
  Purity (a → b) = Purity b
  Purity (IO a)  = Impure
  Purity a       = Pure

type family ApliteSig a where
  ApliteSig (a → b) = ApliteArg a (Purity b)
                        → ApliteSig b
  ApliteSig a       = ApliteResult a

type family ApliteArg a p where
  ApliteArg Double         p      = CExp Double
  ApliteArg Int32          p      = CExp Int32
  ...
  ApliteArg (IOUArray i e) Impure = Arr i e

type family ApliteResult a where
  ApliteResult (IO (IOUArray i e)) = Aplite (Arr i e)
  ApliteResult (IO ())             = Aplite ()
  ApliteResult (IO a)              = Aplite (CExp a)
  ApliteResult (IOUArray i e)      = Aplite (Arr i e)
  ApliteResult a                   = Aplite (CExp a)
```

Figure 6: Aplite type derivation extended with purity

of the form *IO* a, in which case the function is *Impure*, or some other type a, which means that the function is *Pure*. We then extend the *ApliteArg* family with an extra parameter p, giving the purity of the function the argument belongs to. By simply omitting the type instance for the case where the argument is a mutable array and p is anything but *Impure*, we ensure that type checking of such functions fail.

This slightly tricky part done, all that is left is to extend the *ApliteResult* family of permissible return types with the non-IO cases, and we're done. Figure 6 updates the code from figure 5, extending it to support pure imports as well as impure.

3.2 From Function to Concrete Program

While deriving the Aplite type that corresponds to an import signature is all well and good, we also need a corresponding value-level transformation. Figure 7 shows the implementation of this transformation.

In order to be able to import Aplite functions, we must first reduce them from lambda expressions to concrete programs. Using the *square* function from figure 2 as an example, we must apply it to some *CExp Length* and some *Arr Index Double* to obtain a fully saturated value of type *Aplite ()*, which we can then compile into our intermediate representation.

We do this by recursing over the arguments of the function. For each argument, we generate a unique name n and store it in a list of arguments. We then construct an Aplite expression representing a variable v with the name n, using the *varExp* helper function, and apply the function being exported to v, resulting in a function with one fewer argument. After the function has been fully applied in this way we bind the resulting, now saturated, Aplite program with a function *return_*, which produces an actual, function-terminating return statement in the resulting code (as opposed to monadic return, which does not cause termination), and return the final program along with its list of arguments.

```
class Export f where
  type Result f
  export :: Id → [(Type, Id)] → f
         → ([(Type, Id)], Aplite ())

instance Export (JSType a, Export b) ⇒
         Export (CExp a → b) where
  type Result (CExp a → b) = Result b
  export n args f = export (succ n) args' f'
    where
      argType = jsType (undefined :: CExp a)
      argName = "arg" ++ show n
      args'   = (argType, argName) : args
      f'      = f (varExp argName)

instance ReturnValue a ⇒ Export (Aplite a) where
  export _ args f = (reverse args, f >>= return_)
```

Figure 7: Exporting Aplite functions

In figure 2, the previously mentioned *return_* function resides in a *ReturnValue* type class, as it needs to be overloaded for Aplite's *CExp* expression type as well as for arrays. The type of each argument in the intermediate representation, needed by the code generator, is calculated using the *jsType* function which resides in the type class by the same name. For brevity's sake, we will not give the full implementation of either type class here; this informal descriptions diven in this paragraph will have to suffice.

3.3 Compilation and Loading

After being thusly transformed, the finished Aplite program and its argument list are passed to the code generator, further discussed in section 4, where it is compiled into a string containing a JavaScript function representing the program. While a string is great for inspecting the generated code, it would normally not do us much good when it comes to actually loading and executing it. In our case, however, we will use the *Haste.Foreign* (Ekblad 2015b) foreign function interface to load the generated code.

Haste.Foreign is a lightweight foreign function interface designed specifically for JavaScript-targeting Haskell dialects. Using JavaScript's native *eval* function as its backend, it allows importing raw strings of JavaScript code into Haskell as full-fledged functions. Much like our *aplite* function imports an Aplite function under whatever – matching – type signature we give it, Haste.Foreign provides a *ffi* function which imports a string of JavaScript under a given type signature. Given that JavaScript is a dynamically typed language, there is no requirement that the JavaScript string is somehow well-typed with respect to the type at which it is imported. The only restriction placed on imported functions is that their arguments and return value are marshallable between Haskell and JavaScript, and that functions are imported in the IO monad, reflecting the fact that JavaScript code may perform arbitrary side-effects.

While the general safety of importing dynamic strings of code at arbitrary types can certainly be questioned, it fits our purposes quite nicely. There is, however, one bit of impedance mismatch: Haste.Foreign's restriction to imports in the IO monad does not square well with our wish to import pure Aplite functions when provably safe. This means that we can not simply import our compiled Aplite functions using the type at which we intend to call it. The solution is to derive yet another type for our Aplite program: its *FFI type*, which we will use to import the generated JavaScript. This type is quite simple; if the Haskell type of an Aplite function is $a_1 \to ... \to$ *IO* b, then its FFI type remains $a_1 \to ... \to$ *IO* b. If its FFI type is $a_1 \to ... \to b$ for any other b, then its import type becomes $a_1 \to ... \to$ *IO* b.

```
type family FFISig a where
  FFISig (a → b) = a → FFISig b
  FFISig (IO a)  = IO a
  FFISig a       = IO a

class EscapeIO a p where
  type Escaped a p
  escapeIO :: p → a → Escaped a p

instance EscapeIO b p ⇒ EscapeIO (a → b) p where
  type Escaped (a → b) p = a → Escaped b p
  escapeIO p f = λx → escapeIO p (f x)

instance EscapeIO (IO a) Pure where
  type Escaped (IO a) Pure = a
  escapeIO _ = unsafePerformIO

instance EscapeIO (IO a) Impure where
  type Escaped (IO a) Impure = IO a
  escapeIO _ = id
```

Figure 8: FFI types and escaping from the IO monad

After importing our function through Haste.Foreign at its FFI type, we must now somehow strip away the final *IO*, if the user requested the import to be pure. We do this by recursing through the arguments of the function, leaving them untouched, until we reach the *IO* computation at the bottom of the type. There, we apply *unsafePerformIO* to the computation, allowing the function to escape the *IO* monad. This use of *unsafePerformIO* is completely safe, as the *ApliteSig* type family ensures that attempting to import an Aplite function with observable effects purely results in a type error.

Figure 8 shows the derivation of an Aplite function's FFI type, and the release of a pure imported function from the IO monad. Note the use of an additional parameter p in the *Escaped* type class to indicate the purity of the function being examined.

3.4 All Together Now

Putting together the pieces of the puzzle, we end up with several restrictions on both the Aplite and Haskell-level types of the functions we wish to import:

- The function's *Aplite type* must be an instance of *Export* so that we may gather up its arguments and convert it into a concrete Aplite program.
- The function's *FFI type* must be an instance of Haste.Foreign's *FFI* type class, to ensure that programs are actually importable.
- The function's *FFI type* must be an instance of the *EscapeIO* type class, allowing us to let pure imports out of the *IO* monad.
- The function's *Haskell type* must be equivalent to the result of escaping its *FFI type* from the *IO* monad.

With these restrictions now gathered in one place, all that's left is to glue together the pieces described throughout this section. Figure 9 shows the complete implementation of the *aplite* function, broken down into type-annotated steps for readability. Note the use of the *JSString* type, reprenting a JavaScript-native string, to avoid costly conversion between Haskell's *String* type and the JavaScript's more efficient native representation that the *eval* function underlying

```
type ApliteExport a =
  ( Export (ApliteSig a)
  , Haste.Foreign.FFI (FFISig a)
  , EscapeIO (FFISig a) (Purity a)
  , a ~ Escaped (FFISig a) (Purity a)
  )

aplite :: ∀a. ApliteExport a ⇒ ApliteSig a → a
aplite prog = escapeIO (undefined :: Purity a) prog4
  where
    prog2 :: ([(Type, Id)], Aplite ())
    prog2 = export 0 [] prog

    prog3 :: JSString
    prog3 = CodeGen.generate prog2

    prog4 :: FFISig a
    prog4 = Haste.Foreign.ffi prog3
```

Figure 9: The *aplite* function revealed

Haste.Foreign demands as its input. Note also that *ApliteExport* here is a constraint, and as such requires the *ConstraintKinds* Haskell extension.

4. Code Generation

Aplite draws its inspiration from the Feldspar (Axelsson et al. 2010) high-performance DSP and array processing EDSL. In fact, it is based on *imperative-edsl* (Axelsson 2015), a lightweight Feldspar offshoot indended to provide an efficient low-level base language on top of which higher-level functional abstractions may be built in a resource-aware manner.

While imperative-edsl compiles down to a symbolic representation of the C language, this is not ideal for our use case. C supports some functionality that can not be implemented efficiently in JavaScript – most notably unstructured jumps – and is in general a larger language than what we need from an intermediate representation. For Aplite, we have modified imperative-edsl to instead use a typed, specialized subset of JavaScript, where each sub-expression is annotated with its type and where only efficient language constructs are representable. We deem a language construct to be efficiently representable if it is translatable into a JavaScript construct which can in turn be translated into some efficient machine code construct by the JavaScript engine. In short, this includes:

- Local, mutable variables
- Logic, arithmetic and bitwise operators
- If-, for- and while-statements
- Array reads and writes

The intermediate language also has a very limited type system: only signed and unsigned integers, double precision floating point numbers, and arrays over said types are allowed. In general, the intermediate language corresponds closely to an abstract representation of ASM.js.

As one of the main motivations for this work is the inability of JavaScript code to perform consistently across JavaScript engines, Aplite supports multiple backends and aims to enable developers to choose the backend that best suits their particular situation. The backend used to compile a particular function may be configured by replacing the *aplite* function seen in figures 3 and 4 by the *apliteWith* function, which accepts an additional backend configuration parameter as input. Since this parameter is configurable at run-time – as this is when Aplite programs are compiled – the developer may also take user input and the browser environment in which their application is currently executing into account when choosing how to optimize their performance-critical functions. Section 4.3 describes how selection of the optimal backend for any given Aplite program can be automated.

Additionally, having multiple backends can be greatly useful in ascertaining the correctness of the backends, as each backend serves as an oracle for each other backend. Testing the backends against each other provides a cheap way to create a large number of tests with relatively little overhead. While such a testing regimen is not enough to conclude that both backends are completely free from errors, it provides a relatively solid defense against regressions in either backend.

We have implemented two code generators targeting different flavors of JavaScript: plain, but low-level, JavaScript, and ASM.js.

4.1 ASM.js

ASM.js is a subset of the JavaScript language which has a one-to-one mapping to machine code, chosen to act as an efficient compilation target for JavaScript-targeting compilers (Herman et al. 2014). When a supported web browser encounters such JavaScript, it more or less bypasses its normal interpretation pipeline and instead compiles the code straight down to machine code. ASM.js is cleverly designed to be backwards-compatible: any valid ASM.js program is also a valid program in plain JavaScript. Thus, any ASM.js program will work in any modern browser, with the only drawback that unsupported browsers will run the code significantly more slowly.

Unlike normal JavaScript, ASM.js is typed, using standard operators to act as type annotations: a $+$ prefix indicates that a value is a double precision floating point number, a $|0$ suffix – bitwise *or* by zero – indicates a signed integer, and a $>>>$ 0 suffix – logical right shift by zero – indicates an unsigned integer. These annotations must be sprinkled liberally throughout the program: JavaScript's only concept of numbers is its double precision floating point *Number* type, and the type annotations are chosen to force their result to obey the semantics of the type they represent. Performing bitwise operations on a JavaScript number forces it to behave as a 32-bit signed integer, except for logical right shift which forces unsigned 32-bit integer behavior instead. Hence their selection as type annotations. Each subexpression must thus carry the appropriate annotation to avoid reverting back to standard behavior for JavaScript numbers.

ASM.js programs have severe restrictions on the functionality they can use. Only value types – numbers – may be used, so that memory may be allocated exclusively on the stack, avoiding the need for garbage collection. Any more complex, or persistent, data must be stored in an explicit heap represented as a JavaScript *typed array* – an efficient, unboxed representation of a raw string of bytes – the size and location of which are given when an ASM.js module is compiled. The heap is wrapped in one or more views, which lets programs access its contents as elements – integral and floating point numbers of different sizes – rather than as a string of bytes. Indices into these views must be bit-shifted right by two or three, depending on the element size of the view being used. Bit-shifting an index into a 32-bit integer view of the heap to the right by two divides it by four, giving the byte offset at which the indexed value resides. ASM.js uses this information to retrieve the address at which the given element can be found – the index before bit-shifting is the byte index of the element – which is needed when addressing memory at the machine level to which ASM.js compiles.

ASM.js code may only call functions that are either available in a designated standard library object – essentially a tiny subset of libc – or specified at compile-time in a special FFI object. In

```
var squareHeapModule = (function(stdlib, ffi, heap) {
  "use asm";

  var imul = stdlib.imul;
  var intHeap = new stdlib.Int32Array(heap);

  function squareHeap(len) {
    len = len|0;
    var i = 0;
    var tmp = 0;
    for(i = 0; (i|0) < (len|0); i = (i + 1)|0) {
      tmp = intHeap[(i << 2) >> 2];
      intHeap[(i << 2) >> 2] = imul(tmp, tmp);
    }
  }

  return ({squareHeap: squareHeap});
})(Math, null, new ByteArray(0x1000));
```

Figure 10: Squaring n elements on the heap with ASM.js

order to be recognized by the browser and accordingly optimized, ASM.js code must reside in a *module* – a concept which in JavaScript generally refers to a function which when executed exports an object containing one or more library functions – which has a very particular layout. References to external functions through the FFI or standard library objects must be copied into local variables, to ensure that they are not modified from the outside during execution. Any functions containing ASM.js code must be declared – locally, again to avoid outside modification – and returned as part of an object containing the interface to the module. Figure 10 gives a complete example of an ASM.js module.

These restrictions make communication between ASM.js and other JavaScript code cumbersome. This is not necessarily a drawback for its intended use case however, as it is mainly intended as a compilation target for whole applications, rather than small snippets of performance-critical code in a larger JavaScript application.

The intermediate language to which Aplite is compiled before the code generation step was essentially designed around the needs of an ASM.js code generator, hence its insistence on type annotations at every subexpression. The ASM.js backend is thus essentially a pretty-printer for the intermediate format. It also produces the module structure described above and some scaffolding essential to setting up the ASM.js environment. Recall that all ASM.js programs must have a single heap, specified at compile-time. Since we cannot substitute this heap for something else at run-time, any array data passed into an Aplite function must be copied into the function's heap. When the function returns, that same data must be copied back *out* of the heap and into its original array again, as any mutation of input arrays must be observable from the calling program.

These array-related caveats make the ASM.js backend relatively unsuited for functions which work on large arrays: even should the ASM.js code generator be the fastest one for the task, the cost of copying millions of elements into and out of arrays quickly becomes prohibitive.

4.2 Plain JavaScript

The plain JavaScript backend is quite similar to the ASM.js backend, but dispenses with the elaborate module declarations and much of the line noise of the type annotations. Some annotations are still necessary – bitwise *or*ing or right-shifting by zero after arithmetic to avoid overflow for instance – but quite a few can be done away with. Most crucially, the JavaScript backend does not use an explicit

```
var squareHeapModule = (function() {
  function squareHeap(len, arr) {
    var i, tmp;
    for(i = 0; i < len; ++i) {
      tmp = arr[i];
      arr[i] = Math.imul(tmp, tmp);
    }
  }
  return ({squareHeap: squareHeap});
})();
```

Figure 11: Squaring n elements using "plain" JavaScript

heap. While it does use the same typed arrays that underpin the ASM.js heap, it uses an unbounded number of separate array objects instead of squeezing all the data into one big array. This means that we no longer need to copy data into and out of the heap: data can be shared directly between Aplite programs and unboxed Haskell arrays. Figure 11 gives an equivalent example, in the "plain" JavaScript flavor, to the ASM.js module in figure 10.

This gives the plain JavaScript backend a distinct advantage when working with potentially large arrays. This format is also preferable for web browsers which do not support ASM.js: while supported web browsers ignore the excessive bitwise operations-as-type-annotations of ASM.js at run-time, simply using them to guide compilation down to code which implements the same functionality in hardware, unsupported browsers may in the worst case be completely unable to optimize the operations away, ending up performing a lot of unnecessary work. The output of the plain JavaScript backend, containing less casts, annotations and strange hoops for interpreters to jump through in general, may thus be more palatable to some browsers. Even where ASM.js is supported, the plain JavaScript backend may be preferable. In our experiments, several benchmarks fared better with the plain JavaScript backend than with the ASM.js one in both browsers, despite both being ASM.js-aware.

4.3 Automatic Backend Selection

Embedded DSLs make experimentation and exploratory programming significantly easier than when writing the target code by hand: small changes to the source code yield large changes in the target program, and switching backends altogether may yield dramatic effects on performance. However, figuring out the appropriate backend for every situation is a non-trivial task in its own right: web browsers are frequently updated, and, as discussed in section 5, the optimal backend also depends heavily on the program under compilation.

Thanks to Aplite's multi-staged nature, we can relieve the programmer of this burden by providing a means to automatically select the appropriate backend for the program under consideration and the present execution environment. To this end, we add a new function *apliteSpec* which, in addition to a function $f :: a$, which contains an indirection to a compiled Aplite function, returns a specialization handle h *:: SpecHandle* a for f. A specialization handle contains the abstract syntax tree of an Aplite function, allowing it to be recompiled at need, and a reference to the compiled Aplite function to which f is an indirection.

The handle h may then be passed to a function *specialize* along with a benchmarking function $b :: a \rightarrow$ *IO Time*, which may perform any required benchmarking setup and measure the execution time of its argument in whichever manner the developer finds appropriate. Then, *specialize* recompiles the syntax tree contained in h with both backends, applies b to each resulting function, and replaces the underlying function of f with whichever implementation turned

```
type Time = Double

data SpecHandle a = SpecHandle
  { funcAst  :: ApliteFunc
  , funcCode :: IORef Dynamic
  }

apliteSpec :: ∀a. ApliteExport a
           ⇒ ApliteSig a → (a, SpecHandle a)
apliteSpec = unsafePerformIO $ do
  r ← newIORef (apliteWith ct prog :: a)
  return ( unsafePerformIO $ readIORef r
         , SpecHandle (compileToAST prog) r)

specialize :: ∀a. ApliteExport a
           ⇒ SpecHandle a
           → (a → IO Time)
           → IO ()
specialize SpecHandle{..} bench = do
    tf ← bench f
    tg ← bench g
    writeIORef funcCode $ if tf ≤ tg then f else g
  where
    f = apliteFromAST defaultTuning funcAst :: a
    g = apliteFromAST asmjsTuning funcAst :: a
```

Figure 12: The *apliteSpec* and *specialize* functions

```
-- Time the execution of a computation
time :: IO a → IO Time

square :: Word32 → IOUArray Index Double → IO ()
(square, specSquare) = aplite $ λlen arr →
  for(0, 1, Excl len) $ λi →
    modifyArr arr i (**2)

main :: IO ()
main = do
  displayLoadScreen
  specialize specSquare $ λsquare → do
    arr ← newListArray (0, 9) [1..10]
    square 10 arr
    time $ square 10 arr
  startApplication
```

Figure 13: Using automatic backend selection

out to be the fastest one. Figure 12 gives an implementation of this scheme, and figure 13 shows an example of how it may be used by the developer. In the example, we create an array to use as test data, apply the function we want to benchmark once, to ensure that the initial overhead of lazy compilation doesn't affect the timings, and then time a second application of the function to the test data. The *specialize* function then uses the obtained timing information to determine which backend is faster.

This method generalizes nicely to any number of backends and code tunings. By letting the user provide a list of backends to consider in addition to the current arguments of *specialize*, the user could also be allowed to guide specialization by ruling out parts of a possibly large number of backends and tunings from the start.

Automatic backend selection is implemented here as an effectful operation rather than as a pure function from an unspecialized Aplite program to a specialized one, to give the programmer more control. Depending on the number and nature of the functions to specialize and their inputs, specialization may take some time to perform. The programmer may thus want to be able to control when, and if, this specialization happens – for instance to display a loading screen while specialization is underway. Having specialization as a pure function would mean that the newly specialized functions would either have to be created on the top level, denying the programmer this control, or be threaded throughout the entire application.

5. Performance Evaluation

To evaluate the performance of the Aplite language, we bring out a range of benchmarks, selected to cover Aplite's performance across a wide range of tasks: prime factorization, procedural animation using Perlin noise, K-means clustering, matrix multiplication, symmetric encryption and CRC32 checksums. While some of our experiments did show Aplite-generated JavaScript outperforming native, GHC-compiled Haskell code, the focus of Aplite is to augment the performance of client-side JavaScript and browser-targeting Haskell code. For this reason, we compare the performance of Aplite to that of hand-written JavaScript and the web-targeting Haste implementation of Haskell, rather than server-targeting native Haskell.

Methodology Each program was implemented using Aplite and compared to an equivalent hand-written JavaScript version of the same program.[1] For illustrative purposes, the Aplite implementation of the *matrix* benchmark is shown in figure 14, and its hand-written JavaScript counterpart is shown in figure 15.

The benchmark programs are briefly described below.

- *crc32* - calculating the CRC32 checksum of 100 MB of data. The data is passed to the Aplite function as a mutable array, which stresses the ASM.js backend disproportionately due to the copying issues described in section 4. This benchmark uses the highly optimized standard implementation of CRC32 from the SheetJS open source library (SheetJS team 2014) for its JavaScript version.
- *K-means* - K-means clustring of 100,000 random points into five clusters in two dimensions. The points and initial cluster centre points are uniformly distributed across the plane. The K-means benchmark is specialized to the number of clusters using the partial evaluation we get for free by compiling our language at run-time. Simlarly to *crc32*, points and clusters are passed to the Aplite function in mutable arrays, exercising the ASM.js backend quite significantly.
- *perlin* - procedurally generated, infinite animation of two-dimensional clouds using Perlin noise. The benchmark measures the time taken to generate 10 200x200 pixel frames of the animation, calculated from the average frame rate of the animation.
- *factors* - a naïve algorithm for finding the prime factors of a number: factors are calculated by iterating through all odd numbers up to the square root of the target number. The benchmark measures the time the algorithm takes to figure out that 5,467,154,436,746,477 is a prime. This benchmark only uses arrays to store found factors and spends most of its time performing the actual factorization, making it an indicator of performance over simple, arithmetic functions.
- *matrix* - multiplication of two 600x600 element matrices. The hand-written JavaScript version of this benchmark uses the

[1] All benchmarks can be downloaded from https://github.com/valderman/aplite-benchmarks.

```
matMult :: Word32 → Mat → Mat → Mat → IO ()
matMult = apliteWith TUNING $ λm m1 m2 out → do
  for (0, 1, Excl m) $ λi → do
    for (0, 1, Excl m) $ λj → do
      sumRef ← initRef 0
      for (0, 1, Excl m) $ λk → do
        x ← getArr m1 (i*m+k)
        y ← getArr m2 (k*m+j)
        modifyRef sumRef (+(x*y))
      getRef sumRef >>= setArr out (i*m+j)
```

Figure 14: The *matrix* benchmark in Aplite

```
function mult(m, m1, m2, out) {
    for(var i = 0; i < m; ++i) {
        for(var j = 0; j < m; ++j) {
            var sum = 0;
            for(var k = 0; k < m; ++k) {
                sum = m1[i*m+k]*m2[k*m+j];
            }
            out[i*m+j] = sum;
        }
    }
}
```

Figure 15: The *matrix* benchmark in JavaScript

same typed, unboxed arrays as Aplite produces. Modifying the benchmark to instead use the slightly more idiomatic standard JavaScript arrays increases its execution time by about 10 percent. As the matrices are passed to Aplite as arrays, totalling almost a million elements, this benchmark triggers the array copying in the ASM.js backend much like the *crc32* benchmark.

- *xtea* - 4 MB of data is encrypted using the XTEA block cipher in CBC mode. While XTEA has known weaknesses and is thus not suitable for production use, it is simple to implement and verify, and it contains many of the same structures used in stronger ciphers.

The Aplite programs and the non-Aplite Haskell programs were compiled using version 0.5.4.2 of the Haste compiler, with the `--opt-all` flag, enabling all optimizations. All programs were run 10 times using Chrome 50.0 and Firefox 45.1 under Debian GNU/Linux, on a Core i5 6300U machine with 16 GB of RAM, and the median run time for each program calculated. Before each timed run, the test data was loaded into memory and the Aplite programs compiled and executed once, to ensure that test data generation and compilation times did not interfere with the test results. Some Haskell programs are problematic to implement in a way that admits a fair comparison with hand-written JavaScript and the Aplite backends, and were thus not measured implemented in Haste Haskell – an instance of this being the *factors* benchmark which relies heavily on an efficient implementation of *fmod*, which is currently not provided by Haste. In light of the remaining benchmarks and the fact that Haskell programs are almost invariably slower than their hand-written JavaScript counterparts when compiled to JavaScript (Ekblad 2015a), we feel confident in judging Aplite's performance relative to Haste Haskell even so.

The results for Chrome are shown in figure 16, and the results for Firefox in figure 17. Execution times are given in milliseconds

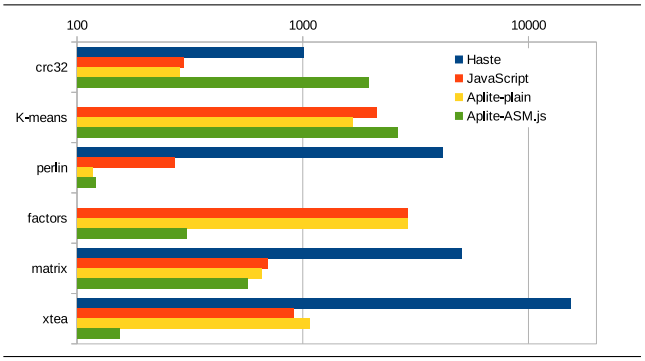

Figure 16: Chrome execution times in milliseconds

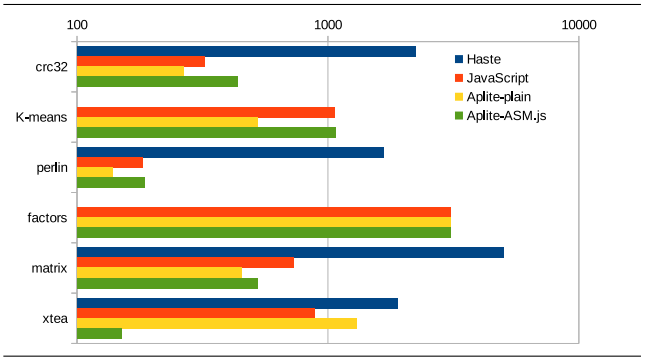

Figure 17: Firefox execution times in milliseconds

on a logarithmic scale. The JavaScript bars indicate programs hand-written in JavaScript, while "Aplite-plain" and "Aplite-ASM.js" refer to Aplite's plain JavaScript and ASM.js backends respectively.

We can see that the additional array copying incurred by the ASM.js backend is taking its toll in the *crc32*. Modifying the *crc32* benchmark to avoid copying, by manually writing the input array into the ASM.js heap at compile-time, its execution time drops to slightly above 300 milliseconds – almost on par with the hand-written JavaScript or plain JavaScript backend – confirming our suspicion that the extra array copying is the culprit. The *K-means* and *matrix* benchmarks, which also deal with relatively large arrays, do not see much improvement from being altered similarly however, both being more computationally heavy than *crc32* and dealing with far smaller arrays. Regardless, performance of the ASM.js backend is clearly tilted in favor of computation-intensive workloads rather than memory-intensive ones.

The *factors* benchmark on Firefox provides one of the more disappointing results: both backends are just about equally performant as the hand-written JavaScript version. As this function is comparatively short and simple however, it stands to reason that browsers would be able to optimize both versions equally. This does not explain the results for the same benchmark on Chrome however, where the ASM.js backend produces a tenfold performance improvement. This goes to show that JavaScript performance can be quite unpredictable, even under the best of circumstances. Meanwhile, the *xtea* benchmark delivers a speedup of nearly an order of magnitude on both browsers. Curiously, this is the only benchmark for which Firefox – the originator of ASM.js – produces better results with the ASM.js backend than with the plain JavaScript one. On Chromium, this picture is more nuanced, with the ASM.js backend giving the best results for half of the benchmarked programs.

Summarizing the results, we see Aplite outperforming hand-written JavaScript in most benchmarks, sometimes by a large margin, and beating non-Aplite Haskell in every benchmark. While its performance relative to hand-written JavaScript varies by benchmark and browser, for each benchmark and browser there is always at least one Aplite backend that performs at least as well as the hand-written JavaScript implementation.

The increased run-time performance comes at a slight cost: Aplite programs need to be compiled before they can be executed, and as a multi-stage language, this compilation happens at run-time, in the user's browser. This cost is not excessive, however: all benchmarks presented in this section have compilation times from 20 to 180 milliseconds, depending on the benchmark, browser and backend. The ASM.js backend is generally faster than its plain JavaScript counterpart, and Firefox generally compiles Aplite programs faster than Chromium. Compilation times were measured as the difference between the first run of an Aplite function and a subsequent run over identical input, as the compilation process is mostly lazy. This means that the compilation times given here are slightly longer than the time actually spent in the Aplite code generator, but give a truer picture of what overhead to expect using Aplite.

6. Discussion and Related Work

Multi-Stage Programming A combined Haskell-Aplite program is essentially a special case of multi-stage programming, where the stages are restricted to Haskell compile-time, Haskell run-time (or Aplite compile-time), and Aplite run-time. Taha et al explored a more general system for multi-stage programming with their *MetaML* language (Taha and Sheard 2000). Although more expressive in its generality and offering intriguing possibilities for specialization on multiple levels, their approach is somewhat less suitable for our purposes. Being embedded in Haskell, Aplite allows programmers to use the full expressive power of stock Haskell, including a vast collection of community-provided libraries. More crucially, Aplite also derives much of its speed from its *lack* of generality. Giving Aplite programs the ability to specialize and compile an arbitrary number of Aplite stages would add optimization-unfriendly dynamism to all but the innermost stage. It is possible that clever use of multi-stage programming would enable performance gains that would offset the penalty thus incurred, but this still remains to be investigated. Rompf and Odersky present another interesting approach to multi-stage programming with *Lightweight Modular Staging* (Rompf and Odersky 2010). Like Aplite, but unlike MetaML, LMS is based on embedded DSLs as opposed to being built into the host language. Rather than a complete EDSL, LMS provides a framework for building multi-stage DSLs – an approach worth investigating if one were to extend Aplite with further stages.

If one were to investigate a more general multi-stage programming system for the web further, the *Sunroof* EDSL by Bracker and Gill (Bracker and Gill 2014) demands further consideration. Like Aplite, Sunroof is a language embedded in Haskell which compiles down to JavaScript code. Unlike Aplite, Sunroof targets the full generality of the JavaScript language instead of a highly performant subset, going so far as to confine the Haskell parts of the system to the server. This approach does not lend itself well to performance-critical computations. Network latencies are both staggeringly high compared to the run-time of even the most expensive of the benchmarks in section 5 and highly unpredictable, and supporting the full JavaScript language places higher demands on programmer discipline to avoid straying into optimization-unfriendly territory. A Sunroof implementation where the Haskell implementation could execute on the client would likely provide at least a modest performance improvement over plain Haskell, however. Such a system might play an important part in a more expressive multi-stage programming environment for web applications, adding an intermediate Sunroof stage between the Haskell and Aplite stages.

While not properly another stage in that it would not allow another layer of specialization, such a system may be even further generalized through a multi-tier programming system such as *Haste.App* by Ekblad and Claessen (Ekblad and Claessen 2014). Haste.App allows a Haskell program to be automatically sliced into a server and a client part, giving a seamless and type-safe abstraction over the client-server communication that web applications must otherwise contend with. Integrating the aforementioned pieces would yield a combined web development environment encompassing server-side Haskell, client-side Haskell, Sunroof and Aplite.

The *Feldspar* EDSL by Axelsson et al (Axelsson et al. 2010), the ideas of which Aplite builds upon, might be used to extend such an environment even further: similar to how Aplite allows efficient computation on the client, integrating Feldspar via Haste.App would give the server added computational capabilities for the cases where even Aplite is not fast enough, or where the results of the computation are for some reason only relevant to the server.

While still not as general as MetaML, such an environment would allow programmers considerable flexibility in choosing the right tool for each part of their application, within the same language framework. One may discuss at which point adding more stages and layers to such a system becomes a curse rather than a blessing: going deeper may offer more power, but may also cause considerable confusion (Nolan 2010). Nonetheless, we feel that this is definitely a venue worth exploring.

High-Performance EDSLs Aplite is not the only EDSL to make loading of compiled code easy. Both Feldspar and the *meta-repa* language by Ankner and Svenningsson (Ankner and Svenningsson 2013) use Template Haskell to automate the loading of their respective programs. Feldspar compiles down to highly performant C code which must then be compiled separately with some C compiler and manually loaded, making it on the surface relatively cumbersome to deal with. Using Template Haskell however, the entire process – from Feldspar to C to machine code being loaded into memory – may be automated (Persson 2014). Originally developed in order to facilitate testing, enabling Feldspar programs to be easily executed and examined within Haskell, the approach could also be used to speed up native Haskell programs using Feldspar, similar to what Aplite does for web-targeting Haskell programs. As this process happens entirely during compile-time, it does not allow Feldspar programs to be specialized over their execution environment or user input, however. Meta-repa takes another approach. Instead of compiling down to C, it uses Template Haskell to compile down to highly specialized and efficient Haskell code instead. Unlike Feldspar, this avoids the dependency on an external C compiler, allowing the whole compilation process to take place within the Haskell compiler. As with Feldspar, this process happens entirely at compile-time and programs can thus not be specialized to run-time factors. Although many EDSLs opt to execute their entire compilation pipeline at compile-time, tapping into C or Haskell compilers that may not be available in their target environment, projects like *Harpy*, a machine code generation DSL by Grabmüller and Kleeblatt, demonstrate that generating and loading native code at run-time is most definitely a viable option (Grabmüller and Kleeblatt 2007).

Taking high-performance EDSLs even further, several such languages target GPUs rather than conventional CPUs (Svensson et al. 2010; McDonell et al. 2015; Mainland and Morrisett 2010). One such language, *Nikola* by Mainland and Morrisett, is quite similar to Aplite: both languages provide on-demand, run-time compilation (and, consequently, the ability to specialize over input and environment) of n-ary functions and seamless integration between host and embedded code. In addition, Nikola supports the verbatim inclusion of code written in its target language; something

which Aplite does not. Nikola functions are always pure, due to the strict separation between the CPU executing the host program and the GPU executing the Nikola kernels. In contrast, Aplite kernels may perform some side-effects visible to the host program and vice versa, giving the impetus for Aplite's use of type families to determine which functions can be imported purely and which can not. Aplite also improves upon Nikola by introducing multiple backends and allowing automatic specialization of kernels to the each kernel's input and execution environment.

Efficient JavaScript Compilation There are also more all-or-nothing approaches to the issue of producing efficient JavaScript. The ASM.js JavaScript subset was first conceived as a compilation target for general purpose languages, more specifically Mozilla's *Emscripten* compiler (Zakai 2011). Through its LLVM frontend, Emscripten is able to compile a wide range of languages down to efficient JavaScript. There are, however, reasons why one would rather use a language which directly targets JavaScript in conjunction with a performance-oriented EDSL. For one, web browsers ship with extensive run-time functionality, which programs compiled with Emscripten need to re-implement on their own. This may partially negate some of its efficiency gains, but more crucially it involves shipping larger code to users. While the size of a native binary stored on a hard drive is usually not an issue, the prospect of sending tens of megabytes of JavaScript to every user before their application can even begin to load certainly is. Additionally, not all browsers support ASM.js and, as we have seen, not all programs see performance benefits from using it.

There is an initiative – WebAssembly – to replace ASM.js with a cross-browser compatible, binary format with the same semantics (Eich 2015). While this may somewhat mitigate the large code sizes that affect ASM.js programs, it is still not a perfect fit for high-level languages, much for the same reasons that ASM.js is not. Work is ongoing on improving this standard however, so it is not inconceivable that it would one day make a compelling alternative to compiling high-level languages directly to JavaScript. However, even should that day come, high-performance EDSLs still have their place. High-level languages for native platforms are usually outperformed by highly optimized domain-specific (or just lower-level) languages.

6.1 Limitations

Mitigating Copying Costs As we showed in section 5, using Aplite brings major performance benefits over plain Haskell code and as well as over hand-written JavaScript. This does, however, assume that the correct backend for the task at hand is chosen. The benchmarks show clearly that using the wrong backend may impact performance severely, part of which is caused by the extra copying of arrays incurred by the ASM.js backend.

This performance penalty might be mitigated in a number of ways. The lowest-hanging fruit in this area would be introducing immutable arrays. In the benchmark where extraneous copying hurt the most – *crc32* – the input array is never mutated. Recall that half the cost of array copying is incurred when a function returns, when data is copied out from the heap into an array in order to make mutation visible to the outside world. For an immutable array we would not incur this cost at all, as it can obviously not have been mutated, which means that its representations in the original array and in the ASM.js heap must be identical, and so there is no need to copy the array's contents back out again.

Too Many Variables When loading JavaScript code dynamically as described in section 3, certain web browsers – including both Chrome and Firefox – refuse to optimize functions over a certain size, as measured by the number of local variables. None of our benchmarks have hit this limitation, but as this limit is relatively low – between 300 and 600 variables – it is quite conceivable that large Aplite functions would hit upon this limit. As Aplite relies heavily on metaprogramming, inlining functions rather than calling them, this may present a problem. As the local variable limit is counted per-function, automatically breaking Aplite programs into appropriately sized chunks, placing each in a separate function, would solve this problem neatly although possibly at the expense of a slight performance hit.

High-Level Abstractions Apart from the powerful metaprogramming cabalities afforded by the use of Haskell as its host language, Aplite as presented is sporting a relative lack of high-level abstractions for a functional language. Efficiently implementing high-level abstractions over arrays and numbers is not impossible; the rich set of abstractions present in the Feldspar language is ample proof of this. As described in section 2, Aplite is designed to act as a fast, low-level core for higher-level abstractions. As such, extending the language's user interface with more convenient constructs is relatively easy. This is partially employed, albeit in a more ad-hoc manner, in the benchmarks, where Haskell-level metaprogramming is used to implement, among other things, virtual data structures and unrolled bounded recursion over lists.

7. Conclusions and Future Work

Conclusions We have presented the Aplite embedded domain-specific language for high-performance, client-side web applications. Its multi-staged programming model allows programmers to seamlessly mix high-level control flow code written in Haskell with highly efficient, low-level computational kernels written in Aplite. Aplite builds on previous work on high-performance EDSLs, adapting the concepts to the web domain where unpredictable performance of idiomatic JavaScript makes a strong case for compiling some higher-level metalanguage down to a predictably efficient JavaScript subset. Its capability to provide predictable performance is further enhanced by the use of multiple backends, with automatic run-time backend selection based on the measured performance of Aplite programs, allowing the same source code to be compiled in the most suitable manner possible for any given task. The use of Haskell as a metalanguage enables the practical use of partial evaluation and multi-stage programming, giving developers fine-grained tools to optimize their computational kernels by specializing them to user input, and experimenting with different optimizations at their leisure.

Depending on the task, Aplite can bring performance benefits of an order of magnitude over both non-Aplite Haskell code and hand-written JavaScript. While not all of its backends perform equally well for all programs and browsers, for each investigated program and browser there is at least one Aplite backend which performs at least as well as hand-written, idiomatic JavaScript, and built-in support to automatically select the proper backend for each situation. We claim that performance-oriented EDSLs are eminently applicable to the domain of web applications. In implementing Aplite, we have believe we present strong evidence that this is indeed the case.

Future Work While we have shown that Aplite fulfills its promise of enabling high-performance functional web applications, we are just beginning to explore the design space. Certain programs in the space targeted by Aplite may benefit from execution on a GPU. Given that Aplite's intermediate representation is already severely restricted, exploring the possibilities of a GPU backend may be a good way forward. Similarly, the capability to run programs in parallel on different CPU cores is another possibility worth exploring. Such backends – or perhaps even a dedicated GPU stage – may make useful building blocks in the more encompassing multi-stage web programming environment we envisioned in section 6. Another interesting possibility would be a *hybrid backend*, in which

parts of the code is compiled into ASM.js and parts into plain JavaScript. This offers the possibility of taking advantage of ASM.js for computation-intensive parts of the program, while mitigating or avoiding the copying penalty for memory-intensive parts.

Presently, communication between Haskell and Aplite is one-way: Haskell calls, and Aplite responds. As ASM.js has some limited FFI capabilities, it would be possible to give Aplite programs the capability to call back into Haskell, provided that the called functions have types that are compatible with the rather limited set of types approved for the ASM.js FFI. This would add interesting prototyping capabilities: Aplite programs could be written in plain Haskell at first, and gradually converted into pure Aplite as the design of the application solidifies or the performance demands grow. This also ties into the issue of higher-level abstractions discussed in section 6.1. While our language presently shows eminent performance, implementing higher-level abstractions remains a high-priority future work item.

Acknowledgments

Many thanks to Koen Claessen, Emil Axelsson, Michał Pałka and Atze van der Ploeg for their valuable feedback, discussion and encouragement.

This work was funded by the Swedish Foundation for Strategic Research, under grant RAWFP.

References

J. Ankner and J. D. Svenningsson. An EDSL approach to high performance Haskell programming. In *ACM SIGPLAN Notices*, volume 48, pages 1–12. ACM, 2013.

P. Antonov. Optimization killers. https://github.com/petkaantonov/bluebird/wiki/Optimization-killers, 2015.

E. Axelsson. The imperative-edsl package. http://hackage.haskell.org/package/imperative-edsl, 2015.

E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169–178. IEEE, 2010.

J. Bracker and A. Gill. Sunroof: A monadic DSL for generating JavaScript. In M. Flatt and H.-F. Guo, editors, *Practical Aspects of Declarative Languages*, volume 8324 of *Lecture Notes in Computer Science*, pages 65–80. Springer International Publishing, 2014. ISBN 978-3-319-04131-5. doi: 10.1007/978-3-319-04132-2_5. URL http://dx.doi.org/10.1007/978-3-319-04132-2_5.

E. Czaplicki. Elm: Concurrent FRP for functional GUIs. *Senior thesis, Harvard University*, 2012.

A. Dijkstra, J. Stutterheim, A. Vermeulen, and S. Swierstra. Building JavaScript applications with Haskell. In R. Hinze, editor, *Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 37–52. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41581-4. doi: 10.1007/978-3-642-41582-1_3. URL http://dx.doi.org/10.1007/978-3-642-41582-1_3.

B. Eich. From ASM.js to WebAssembly. https://brendaneich.com/2015/06/from-asm-js-to-webassembly/, 2015.

R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 671–683, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535856. URL http://doi.acm.org/10.1145/2535838.2535856.

A. Ekblad. *A Distributed Haskell for the Modern Web*. Licentiate thesis, Chalmers Institute of Technology, 2015a.

A. Ekblad. Foreign exchange at low, low rates. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*, IFL '15, pages 2:1–2:13, New York, NY, USA, 2015b. ACM. ISBN 978-1-4503-4273-5. doi: 10.1145/2897336.2897338. URL http://doi.acm.org/10.1145/2897336.2897338.

A. Ekblad and K. Claessen. A seamless, client-centric programming model for type safe web applications. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 79–89, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3041-1. doi: 10.1145/2633357.2633367. URL http://doi.acm.org/10.1145/2633357.2633367.

P. Freeman. PureScript. http://www.purescript.org/, 2015.

Y. Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4): 381–391, 1999.

M. Grabmüller and D. Kleeblatt. Harpy: Run-time code generation in Haskell. In *Haskell Workshop 2007*. ACM Press, September 2007.

D. Herman, L. Wagner, and A. Zakai. The ASM.js draft specification. http://asmjs.org/spec/latest/, 2014.

G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. *ACM Sigplan Notices*, 45(11):67–78, 2010.

T. L. McDonell, M. M. T. Chakravarty, V. Grover, and R. R. Newton. Type-safe runtime code generation: Accelerate to LLVM. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 201–212, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3808-0. doi: 10.1145/2804302.2804313. URL http://doi.acm.org/10.1145/2804302.2804313.

V. Nazarov, H. Mackenzie, and L. Stegeman. GHCJS Haskell to JavaScript compiler. https://github.com/ghcjs/ghcjs, 2015.

C. Nolan. Inception. http://www.imdb.com/title/tt1375666/, 2010.

K. Perlin. Improving noise. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 681–682. ACM, 2002.

A. Persson. Towards a functional programming language for baseband signal processing. 2014.

T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *ACM SIGPLAN Notices*, volume 46, pages 127–136. ACM, 2010.

SheetJS team. Sheetjs. http://sheetjs.com/, 2014.

J. Svenningsson and E. Axelsson. Combining deep and shallow embedding for EDSL. In *Trends in Functional Programming*, pages 21–36. Springer, 2012.

J. D. Svenningsson and B. J. Svensson. Simple and compositional reification of monadic embedded languages. In *ACM SIGPLAN Notices*, volume 48, pages 299–304. ACM, 2013.

J. Svensson, K. Claessen, and M. Sheeran. GPGPU kernel implementation and refinement using Obsidian. *Procedia Computer Science*, 1(1):2065–2074, 2010.

W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical computer science*, 248(1):211–242, 2000.

J. Vouillon and V. Balat. From bytecode to JavaScript: the js_of_ocaml compiler. *Software: Practice and Experience*, 44(8):951–972, 2014.

A. Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.

A. Zakai. Big web app? Compile it! http://kripken.github.io/mloc_emscripten_talk/, 2013.