Embrace, Defend, Extend: A Methodology for Embedding Preexisting DSLs

Case in Point, Stream^{Hs}: StreamIt in Haskell

Abhishek Kulkarni Ryan R. Newton
Indiana University
{adkulkar, rrnewton}@indiana.edu

Abstract

Domain-specific languages offer programming abstractions that enable higher efficiency, productivity and portability specific to a given application domain. Domain-specific languages such as StreamIt have valuable auto-parallelizing code-generators, but they require learning a new language and tool-chain and may not integrate easily with a larger application. One solution is to transform such standalone DSLs into embedded languages within a general-purpose host language. This prospect comes with its own challenges, namely the compile-time and runtime integration of the two languages. In this paper, we address these challenges, presenting our solutions in the context of a prototype embedding of StreamIt in Haskell. By demonstrating this methodology, we hope to encourage more reuse of DSL technology, and fewer short-lived reimplementations of existing techniques.

Categories and Subject Descriptors D.3.2 [*Concurrent, Distributed, and Parallel Languages*]

General Terms Languages, Performance

Keywords Streaming, Parallel, Haskell

1. Introduction

Parallel domain-specific languages (DSLs) come in two major flavors: standalone, or embedded in a host language. Because significant value exists in the [parallel] code-generators associated with DSLs, we argue it is best that they be accessible from multiple languages. Indeed, in a minority of DSLs, this is already the case; the ideal scenario is a well defined target language and API for emitting it, which can then support multiple front-ends. LLVM and ArBB (Intel Array Building Blocks [**?**]) are two examples in this category.

For the majority of DSLs, however—standalone or single-language embedded—there remains the question of how to reap their benefits from a wider context. Of course, one approach is to reimplement everything in every language context. This is already happening with GPU DSLs [**? ? ? ? ? ?**] and other array-parallel libraries [**? ?**]. Yet this approach diffuses effort. Perhaps a better scenario is to focus community effort around best-in-class implementations for each relevant computational model: for example, the *StreamIt* language for the case of sychronous data-flow.

In this paper, we describe a methodology for retrofitting a DSL like StreamIt to become an embedded DSL (EDSL) in a high-level host-language (Haskell). The benefits of such an embedding extend beyond merely writing the same programs in a different language. EDSLs are necessarily *metaprograms*. By virtue of embedding in a more widely used general purpose language, EDSLs offer the power of full set of libraries and tools in the meta-language[1]. In this paper we examine examples of increased abstraction and safety enabled by this metaprogramming capability.

We advocate a three-layered approach for embedding a standalone DSL:

• **Embrace**: First, directly expose a set of language constructs isomorphic to the target DSL, making it possible to conveniently emit syntax and bootstrap the rest of the system. Aim to cover the entire language in this phase. Further, in some cases care must be taken to ensure that the host runtime interoperates well with the target runtime (e.g. avoid oversubscription, see Section **??**).

• **Defend**: Parallel DSLs, while excelling at one thing, often have other warts—inadequacies in the type system, lack of bounds checking, bugs. To make a DSL safe to use as a part and parcel of a type-safe, memory-safe language, the end user should be defended against these risks. If desired, an additional layer of static verification using advanced type representations may be added at this phase.

• **Extend:** In the meta-language, it may be possible to build *new* abstractions that make it even easier to program using the DSL. In the context of Haskell this may include using existing interfaces (type classes) such as `Monad`, `Applicative`, or `Arrow`, and otherwise adapting the DSL to the idioms of the host language (Section **??**).

The end-result is a package such as *Stream^{Hs}*: our prototype merger of Haskell and StreamIt. In the next section we provide background on the StreamIt language, and then explain what each of the three steps above mean in the context of embedding StreamIt in Haskell (Sections **??**, **??**, and **??**). We then evaluate the mechanisms we use to wed Haskell and StreamIt compile-time and runtime (Sections **??** and **??**).

2. StreamIt Background, Very Briefly

StreamIt is a simple C-like language that is both *restricted*—missing allocation, function pointers, unions, and many other features—and *extended* with a special notation for defining Stream processing elements called *filters* and wiring together graphs of

[1] We view this as the most salient advantage of EDSLs; in the opinion of the authors, saving the effort of building a parser and front-end is a minor economy.

FPCDSL '13, September 22, 2013, Boston, MA, USA.

http://dx.doi.org/10.1145/2505351.2505355

these filters [**?** **?**]. Figure **??** lists the StreamIt language supported by our current Stream^{Hs} prototype.

A canonical example of StreamIt code is the finite impulse response (FIR) filter below:

```
float→float filter FIR (int N, float[N] weights) {
  work push 1 pop 1 peek N
  {
     float result = 0;
     for (int i = 0; i < N; i++) {
       result += weights[i] * peek(i);
     }
     pop();
     push(result);
  }
}
```

Each filter has only a single input and single output stream which are accessed with `push`, `pop`, and `peek`. Further, StreamIt relies on on explicitly declared static data rates[2], as seen in the `push 1 pop 1 peek N` line. The above example can be implemented with a sliding window, as it pops and pushes only a single element each time it "fires", but it peeks at the last `n` elements.

Performance Before proceeding, a note on why StreamIt is of interest in the first place: due to its restricted kernel language and fully exposed communication patterns and data-rates, StreamIt supports efficient static scheduling and is performance-portable across a range of architectures. From 2002 to 2012, a number of different StreamIt backends were built, targeting multicores, GPUs, Cell processors, tiled processors, and clusters. This level of effort justifies an attempt to reuse StreamIt rather than build a new implementation just to change the front-end interface.

3. *Embrace*: StreamIt in Haskell

In Stream^{Hs}, StreamIt code can be directly transliterated on a line by line basis to Haskell. Our FIR filter becomes:

```
fir :: Var Int → Var (Array Float)
    → Filter Float Float ()
fir n weights =
  work Rate{pushRate=1, popRate=1, peekRate=ref n} $ do
    result ← float' 0
    for (0, ref n) $ λi → do
      let i' = ref i
      result += ref (weights ! i') * peek i'
    pop
    push (ref result)
```

This direct embedding does not change the level of abstraction. Yet because the implementation handles invocation of the StreamIt compiler and resulting binary, it becomes easier to use StreamIt from within a Haskell program. Further, the above program can call arbitrary Haskell programs internally, in the process of constructing the stream graph. This effectively makes Haskell a powerful macro language.

3.1 Type Safety

Above, the `fir` function returns a value of type `Filter` parameterized by three type arguments `Float`, `Float`, and `()`. What does this mean? "`Filter inty outy res`" is a monadic type that asserts the enclosing filter pops elements of type `inty`, pushes elements of type `outy`, and returns a value (in the Haskell meta-language) of type `res`. In the above example, this ensures that the `push` statement can *only* push values of type `Float`. Further, the types "`Var Int`" and "`Exp Int`" are used to represent variables and expressions in the target language. Because of the extra `Int` parameters, it is possible to enforce that the StreamIt programs will be well-typed. Thus Stream^{Hs} subsumes the StreamIt compiler's typechecking, guaranteeing that type errors will be encountered at Haskell compile-time, rather than deferring them to StreamIt compile-time.

Because type safety is about *defending* the programmer from errors, it could be addressed in the next section. However, nothing is lost in terms of expressiveness due to type-safe embedding, so there's little reason *not* to do it from the get-go.

[2] Some StreamIt backends also have limited support for unknown rates, and for "teleporting messages" that are sent out of band. We omit discussion of those features for simplicity.

Types	τ	::=	`Int` | `Float` | `Array` τ | `Void`	
	x, y	$\in$	variables	
	a	$\in$	arithmetic expressions	
	b	$\in$	boolean expressions	
	ℓ	$\in$	labels	
	a	::=	$n \in \mathbb{N} \mid r \in \mathbb{R} \mid a_0 + a_1 \mid a_0 - a_1$ $\mid a_0 \times a_1 \mid a_0/a_1 \mid a_0 \geq a_1$ $\mid a_0 \leq a_1 \mid a_0 \neq a_1$	
	b	::=	`True` | `False` | $b_0 \wedge b_1 \mid b_0 \vee b_1 \mid \neg b_0$	
Expressions	e	::=	$a \mid b \mid e[e] \mid \cdots$	
Rates	r	::=	(e, e, e)	push/pop/peek rates
Commands	c	::=	$x : \tau$	declaration
		|	`if` b `then` c_0 `else` c_1	conditionals
		|	`for` (c_0, b, c_1); `do` c_2	loops
		|	`while` b; `do` c_0	
		|	`println` e	output
		|	$x = e$	assignment
		|	$x[e_1]...[e_n] = e$	assignment
		|	$c_0; c_1$	sequence
Filters	F	::=	c	commands
		|	$x = f$	assignment
		|	`init` c	initialization
		;	`work` r c	work
Filter Expr	f	::=	$e \mid$ `peek` $e \mid$ `pop` | `push` e	
Composite Filters	P	::=	c	commands
		|	$\texttt{add}_n\ \ell\ e_1 \ldots e_n$	add filter
		|	`add FileWriter`$\langle\tau\rangle$ *path*	
		|	`add FileReader`$\langle\tau\rangle$ *path*	
		|	`split` $n \mid$ `join` n	splitjoins
TopLvlDecls	M	::=	`def` $\ell(x_0 : \tau_0 \ldots x_n : \tau_n)\,\{F\}$	
		|	`def` $\ell(x_0 : \tau_0 \ldots x_n : \tau_n)\,\{P\}$	

Figure 1. Stream^{Hs} syntax, listed in psuedocode that serves as a standin either for the original StreamIt and or our embedding.

3.2 First Layer, Implementation

This layer of our prototype is based on existing techniques in language embedding. Importing `Language.StreamIt.Raw` exposes all of the basic primitives and types. Higher-order Abstract Syntax (HOAS [**?**]) is used to model StreamIt variable bindings using appropriately typed Haskell variable bindings.

One interesting part of the implementation is its flexible approach to when and how the StreamIt DSL compiler is invoked, and the resulting binary is stored and loaded. This is described in Sec-

tion **??**. Further, we defer discussion of data-movement between Haskell and StreamIt to Section **??**.

4. *Defend*: Array Bounds, Bugs, and Incomplete Metaprogramming

StreamIt is *almost* a deterministic-by-construction language. If it were, every StreamIt program would always evaluate to the same series of values, and it would be possible to call StreamIt code from Haskell *without* the `IO` monad, meaning that StreamIt could be used anywhere, even within pure functions[3]. Further, even *if* StreamIt remains locked in the `IO` monad, to retain the guarantee that Haskell is a safe language in spite of using our library we must still ensure that StreamIt cannot crash the program. (In some cases we run StreamIt in a separate process, so this is impossible.)

Bounds The main reason that a StreamIt program might go wrong is because of an out of bounds array or stream access returning an unbound result or crashing StreamIt. It is straightforward, however, to define another module on top of our first layer that wraps the original StreamIt `peek` and array primitives with bounds checks. Such a module can be declared `SafeHaskell`—a guarantee that no backdoors, such as calling arbitrary native code, are used, and thus "types cannot lie". This restricted variant of the language is useful in executing untrusted code, for example on a server [**?**].

Of course, runtime bounds checks result in additional overhead. In Section **??**, we present a more declarative way of writing filters that has the side-benefit of eliminating bounds checks.

StreamIt Metaprogramming Many parallel DSLs employ meta-programming (multi-stage evaluation). This includes both standalone DSLs [**? ? ?**] and embedded ones [**? ? ? ? ?**]. Metaprogramming offers the potential for *abstraction without performance penalty*. This is not specific to DSLs; perhaps the most widespread example is C++ template meta-programming. But for DSLs, abstraction capabilities (higher order functions, objects, etc) often cause particular problems for optimization framework's, such as StreamIt's, static scheduling, or are infeasible on the target platform—especially embedded chips and accelerators.

Therefore it is no surprise that StreamIt itself includes a metaprogramming facility. While the language does not include *explicit* metaprogramming in the form of quotation and anti-quotation syntax, it does include compile-time partial evaluation that inlines procedures and evaluates all code outside of the stream filter's bodies, leaving only an explicit graph topology. For example, the below code uses compile-time computation to build a pipeline with a variable number of repeated copies of the `FIR` filter.

```
int→int pipeline Rep(int n)
{
  if (n > 0) {
    add FIR(..);
    add Rep(n-1);
  }
}
```

So what is the problem? Alas, StreamIt is a research language, and while ostensibly the entire language is supported at compile-time, in reality with a small amount of poking around a programmer can uncover things that do not work right at compile-time or work differently in the Java reference implementation (`--library`) and the C backend(s)[4]. For example, `println` statements at compile-time are completely lost in the C backends (using the last official release, StreamIt 2.1.1.), and arrays that are mutated at compile-time seem to expose some bugs.

Nevertheless, virtually all of the most interesting DSLs today are research prototypes. Thus we view it as a legitimate goal of a reuse effort such as this to paper over bugs (or fix them, of course!) and expose a sound subset to users of our library. Further, even if there were no bugs or limitations, a tension would still remain, because anything that is computed at StreamIt compile-time (e.g. array sizes), may not therefore be easily available at Haskell compile-time.

Option 1: Replace StreamIt's meta-eval with Haskell In this scenario, Haskell simply never leaves any work for the StreamIt compile-time evaluator. That is, while our base layer exposes full StreamIt (including meta-programming), we could also inline and elaborate everything ourselves, presenting StreamIt with a fully elaborated stream graph.

Unfortunately, it is a common pattern to generate many repeated filters when the stream graph bounds are statically known, for example in a recursive algorithm like Mergesort[5], or in the EEG application from our previous work [**?**]. We do not want to force the StreamIt compiler to deal with this duplicated code[6].

Option 2 (Winner): Limited StreamIt meta-eval The option we end up choosing is a compromise. We expose stream constructors—like `Rep` above—as Haskell functions, but rather than fully inlining them, we inline only where doing so would expose *new static information* that fixes array bounds or peek-rates. For example, consider the following Haskell version of the repeated-FIR pipeline.

```
rep :: Int → StreamIt Int Int ()
rep 0 = return ()
rep n = pipeline $
          do add FIR (...)
             add1 rep (n-1)

main :: StreamIt Void Void ()
main = pipeline $
 do add intSource
    add1 rep 10
    add intPrinter
```

Note that a separate monad, `StreamIt` is used to describe graph-wiring code, as opposed to individual filter code (*i.e.* the `Filter` monad). `add` adds a node (`Filter`) or a sub-graph (`StreamIt`) to the existing stream graph. Stream constructors parameterized by arguments are added with `add1`, `add2`, etc. that take a variable number of "`Var a`" arguments. These retain flexibility for the EDSL implementation to observe the function call and decide how to generate code for it. A technique called *Observable Sharing* [**? ?**] can be used to determine when two function calls are the same. (In contrast, raw Haskell function calls are "invisible" to the EDSL, which receives only the resulting value.)

Assuming the arguments to FIR (which are omitted for simplicity above) never change, then there is no static information to be gained by unrolling the pipeline. Rather, we leave that for StreamIt to do. The criteria for inlining is simple: starting at the "leaf" filters, check whether the body contains array sizes or peek rates that are *not* resolved to constants; if so, inline that constructor and repeat.

---

[3] The merits of this are debatable, however, because it does in fact do actual IO, including writing a temporary file to disk to store the StreamIt output.

[4] One example of fishing for bugs can be found in this file: `https://github.com/iu-parfunc/haskell-streamit/blob/master/Examples/example_BUGS.str`. In fact, this file also includes one bug in the type-checker, which makes it fortunate that in Stream^{Hs}, Haskell's typechecker renders StreamIt's redundant.

[5] See StreamIt's included benchmarks, as well as the Haskell version at: `https://github.com/iu-parfunc/haskell-streamit/blob/master/Examples/Mergesort/MergeSort.hs`.

[6] The 2.1.1 release of StreamIt fully duplicates code for all replicas of a stream filter, resulting in large binaries. But we do not want Stream^{Hs} to remove the option to fix this.

Moreover, our restricted formulation leaves *only* the inlining of constructors for StreamIt meta-programming system. That is, if one imports `Language.StreamIt.Safe` in Haskell, it is not possible to construct arbitrary imperative code for StreamIt to evaluate at compile-time, which, again is where we have found a concentration of bugs. Instead, we use StreamIt's native meta-programming only for code-deduplication and only under certain circumstances.

5. *Extend*: Functional Filters and Combinators

Finally, once we have a safe way to express StreamIt computations in Haskell, we can ask whether we can take advantage of metaprogramming to improve the programming interface further: either in performance or productivity.

A first sore point is that StreamIt exposes only a low-level imperative language within stream filters. Thus our StreamIt EDSL, while embedded in Haskell, is not an idiomatic Haskell library. For example, programmers cannot use structured control-flow constructs such as `fold` and `map`, rather they express everything using `for` loops, as in the FIR-filter example.

In fact, Haskell programmers already have a set of idioms to deal with streams; they use stream-processing all the time in the form of Haskell's ubiquitous lazy lists! The FIR filter example, written as a recursive function over infinite lazy lists would be naturally expressed in Haskell as follows:

```
firLs n weights = loop ls
  where
  loop ls =
    let window = take n ls in
    sum (zipWith (*) weights window)
      : loop (tail ls)
```

where `ls` is the implicit input stream to the filter `firLs`. While the FIR example is simple, and either representation is perfectly clear, in more complex examples we begin to see the advantages of the above list combinators:

- *Windows* of streams are themselves first-class values, as returned by `take`. With more complex indexing relationships it can become difficult to see what a particular `peek` is supposed to be accessing.
- Separate data-parallel combinators are modular and recognizable: `sum`, `zipWith`, etc. The user does not need to mix the guts of these operations inside a `for` loop.
- Less chance of mistakes due to the ordering of `pop`s and `peek`s. The danger is because `pop` advances the cursor used by `peek`, which makes evaluation order very important (consider the expression `pop() + peek(0)` in the original StreamIt).

5.1 Solution: Functional Filters

Fortunately, we are able to add a new array type and a set of array and stream combinators that let us write a StreamIt filter in the same style as the list-processing example above.

```
fir n weights = funFilter loop
  where
  loop strm loopK → do
    window ← take n strm
    sum (zipWith (*) weights window)
      <:> loopK (tail strm)
```

While the basic style is the same, there are a few differences from the list-processing version.

- The program is monadic. This is necessary to correctly type `push`/`pop`/`peek` operations that are implicitly generated by some of the combinators.
- The `loop` function does not recur directly through itself. Rather, it takes a stand-in for itself, `loopK`. This formulation of recursion is sometimes called the *poor man's Y-combinator*.
- The list cons operator (`:`) is replaced by stream-cons, `<:>`. Likewise, functions like `zipWith`, `take`, `tail` refer to alternate versions defined in `Language.StreamIt.Safe`.

To the best of our knowledge, this use of poor-man's Y to represent loops in the target-language of a multi-stage programming system is a novel feature of this system. In the next section, we will explain how the combination of metaprogramming and array fusion enables the above example to generate code that is actually *more efficient* than the original, imperative FIR filter.

5.2 Stream and Array Representation

During metaprogram evaluation, abstract handles on streams have the following simple representation:

```
data Stream a = Stream [Exp a] Int
```

This contains a cursor (zero-based offset) as well as a list of elements stream-cons'd on the front, with the head of the list being the most recently added. This representation defines a stream in a *differential* manner, relative to a second stream. That is, "`Stream [a,b,c] 5`", is equivalent to "`append [a,b,c] (drop 5 s2)`". Notice that we need not record *which* stream the cursor indexes into (`s2`); this is because in StreamIt filters there is only a *single* input stream, so our stream handles are always relative to that.

Given this representation, some stream functions, like `tail`, are extremely straightforward:

```
tail :: Stream a → Stream a
tail (Stream [] cursor) = Stream [] (cursor+1)
tail (Stream ls cursor) = Stream (Prelude.tail ls) cursor
```

But what about `take`? The `take` function returns a *window* of stream elements, represented as an array. Arrays can take many forms in Haskell libraries, and there has been much work into designing array representations to support *fusion* of operations over arrays [**?** **?**]. Perhaps the most widely known is the widely used *Stream Fusion* approach [**?**], which is used by the standard `Data.Vector` library.

In understanding exotic array representations, it is instructive to view arrays as *functions* from a domain of indices to values: `Int` $\rightarrow \alpha$. Given such a mental model, the fact that arrays are sometimes manifest in memory in contiguous locations is just a form of memoization. Likewise, it is possible to represent a manifest, in-memory array by its accessor function. We call arrays represented by these functions *pull arrays*. With a pull-array representation, it is clear that writing a `map` function need not require creating a temporary array in memory, only composing functions. Eliminating temporaries by combining stages of processing is what we mean by *fusion* (or to be precise, fusion and deforestation).

In Stream^{Hs}, we use a bundled combination of two *different* representations for arrays: push and pull arrays. Each operator that returns such an `SArray`, must formulate both representations. These concepts are suitable adapted to include `Exp` and `Filter` types, as seen below:

```
-- StreamIt (meta) arrays:
data SArray inT ouT a =
  SArray
  { pushrep :: PushArray inT ouT a
  , pullrep :: PullArray a
  , arrlen  :: Exp Int
  }
data PushArray inT ouT a =
     PushArray (Rcvr a → Filter inT ouT ())
type PullArray a = Bool → Exp Int → Exp a
```

```
type Rcvr a = Exp Int → Exp a → Filter inT ouT ()
```

Push arrays were introduced by Svensson et al. [**?**]. Their central idea is to change who is responsible for iterating over the domain of the array from the *consumer* (pull) to the producer (push). That is, given an `Int` $\rightarrow \alpha$ pull representation, ultimately a consumer must loop through all locations in the array, evaluating the function at each. The push array, by contrast, takes a callback for writing out individual elements (`Rcvr`). The push array is responsible for iterating over the array's domain and pushing all elements to the consumer.

Of course, none of this indirection—function calls back and forth between consumers and producers—happens at runtime. Moreover, bundling two different representations does not have a runtime impact. Rather, the `SArray` dual-representation is there so that clients can choose the best representation for the operation they want to perform at code generation time. Push arrays are usually preferable, but sometimes pull arrays are required. For example, `zipWith` cannot use the `pushrep` for *both* of its inputs; that would generate two loops. Rather, it allows one to push and allows that pusher to pull the other input.

Bounds checks, when and where Normally, evaluating a pull array function (`Exp Int` → `Exp a`) entails emitting a bounds check. Even if the size of the array is statically known, it cannot be known whether the client (supplying the `Exp Int`) will attempt an out-of-bounds access.

You may notice above that our definition of pull arrays has an extra parameter: `Bool`. This flag controls whether invoking the pull function triggers a bounds check, and thus it is *not* exposed to the end user. To see how this is used, consider a simple program fragment, `sum arr`, which is implemented as `fold (+) 0 arr`. This reduction will ultimately generate a scalar `for`-loop in the generated StreamIt code. However, because `fold` uses the loop bounds to create its iteration space, it can *guarantee* that it will never make an out of bounds access, and thus it may employ the unchecked variant of the pull function.

The push-array representation also avoids bounds checks on reductions. Here, the `fold` function provides a receiver-callback to the push-array representing `arr`. The receiver simply adds values it observes to an accumulator. Because the receiver does not care about the index argument, it ignores it, consumes the values only, and does not require a bounds check.

In some cases, push-arrays have a decided advantage. Consider `sum (take 10 $ hd <:>strm)`. In this program fragment we are adding a new element to the front of a stream, and then taking the first ten elements: nine of them will be genuine `peek`s into the stream, and the first will be `hd`. The pull representation of the array will need to include a conditional, and thus will look something like this:

```
λ ix → if (ix == 0)
       then return hd
       else peek (ix - 1)
```

Whereas in the push-array case, the two different regions of the array simply become sequentially composed calls to the value-receiver, eliminating the conditional as well as bounds checks:

```
do rcvr 0
   for (1,9) (λi → peek i >>= rcvr)
```

After the calls to `rcvr` above are inlined, they will be replaced with direct updates to the reduction variables. Further, push-arrays are especially advantageous in an environment of static array sizes, because downstream consumers of push-arrays can know exactly how which indices will be supplied to the receiver without runtime checks. In fact, the current library of `SArray` operations (1) never generates bounds checks unless raw peeking or array reference operators are used, and (2) never generates a temporary array, always fully fusing operations.

Limitations The current `SArray` type is not stored in an `Exp` and is distinct from the `Exp (Array t)` type that is used for simple, direct use of StreamIt arrays (though conversion is possible). This means that `SArrays` cannot be used from contexts where `Exps` are expected, for example, they cannot currently be returned from a dynamic conditional expression. This facility could be added, but it's not clear that it would be a good idea. StreamIt does not support pointers, so arrays may only be manipulated by value (copied). Returning an array from a conditional would necessitate copying the entire array.

6. Embedding Practicalities

Thusfar we have said little about the compilation workflow and about the runtime. These form the topic of this Section.

6.1 Trouble-free Staged Compilation

One of the major problems facing new DSLs is that they introduce new build workflows and compilers that are often themselves non-trivial to install. *Runtime* compilation of such DSLs not only can result in high runtime overhead, but it also means that the end-user must have a working instance of the DSL compiler (*e.g.* StreamIt) in order to run. If possible, it would be preferable to instead ship statically-linked binaries without this dependence on the runtime environment.

The way to accomplish that is to compile StreamIt code not during Haskell runtime, but during Haskell compile-time. This is possible when the host language has its own metaprogramming features (*i.e.* Template Haskell [**?**]). The downside is that compiling the embedded language along with the host means that one must statically choose which StreamIt backend and settings to use. It is impossible to know, at Haskell compile-time, anything about the future runtime machine(s), such as their number of cores.

When compiling the host and target language together, there are many choices for how to deploy the resulting files. We tried different staged compilation approaches to strike a balance between build complexity and performance. We evaluate the performance of compilation, loading and execution for each of our configurations. The staged compilation techniques Stream^{Hs} supports are:

- **Embedded Binary:** The StreamIt compiler `strc` is invoked by the Haskell compiler at compile-time using Template Haskell and the generated StreamIt executable is read and stored as a compressed bytestring *inside* the Haskell executable[7]. At runtime, the embedded binary is extracted and executed using the `exec` system call. The extra penalty of writing back the embedded binary cannot be avoided without using a modified loader to execute the in-memory ELF image. However, this approach provides seamless, single-command compilation driven by the Haskell compiler `ghc`. This technique also scales to other DSLs which can only produce standalone binaries, and cannot easily be modified to produce linkable libraries.

- **Shared Library:** In case of StreamIt, we were able to patch it to produce a shared library that can be linked or dynamically loaded (`dlopen`) at runtime. This method avoids storing large sections of static data in the binary and it speeds up both the compilation and loading time by avoiding extra copies of the StreamIt binary. Further, it opens up the possibility of running the StreamIt program in thread(s) within the same process as the host.

In both cases above, the compile function is lifted and executed at compile-time returning a result to the Haskell compiler. Unfor-

[7] The GPU EDSL, Nikola [**?**], also employed this approach.

No. of Filters	Size (KB)	StreamIt (s)	StreamIt RAM (s)	Stream^{Hs} embed(s)[8]	Stream^{Hs} embed RAM(s)	Stream^{Hs} dynload(s)
2	12	1.56	1.49	6.18	6.06	2.95
22	36	2.13	2.03	12.07	12.1	3.84
102	132	4.24	3.97	18.43	16.22	7.10
202	252	6.74	6.32	–	–	11.24
502	620	15.22	13.86	–	–	24.34
1002	1228	30.6	28.64	–	–	49.73

Table 1. Compile-time of different linking/embedding methods. RAM refers to using a Ram-disk for compilation outputs rather than a traditional file system.

tunately, there is currently no way in this context (template expansion) to programmatically *add* a new library, namely the StreamIt-generated `.a` or `.so`, to the GHC process's list of libraries to link. Thus producing and then linking a shared library requires modifications to the user's build system. Specifically, the libraries produced during template expansion must be added to the final link command.

In the following type-signature, `Q THS.Exp` is a Template Haskell type for the quoted Haskell expression that results from StreamIt compilation.

```
compile :: (Elt a, Elt b, Typeable a, Typeable b) ⇒
           Target → StreamIt a b () → Q THS.Exp
run     :: Target → Embedding → IO ()

main     = run StreamIt $(compile StreamIt fft2)
```

Compilation speed To assess the compilation speed of the above alternatives, we used a synthetic StreamIt program consisting of a source filter, a series of identity filters and a sink filter. We vary the number of identity filters to control the size of the resulting executable (Column 2 of Table **??**). The compiler was also run with a memory-backed filesystem (aka RAM disk) to mitigate the cost of extra copies involved. As shown in Table **??**, the compression and embedding of the StreamIt binary is expensive, however a memory-backed device marginally improves the compilation speed. We can also see that a significant amount of time is due to ordinary Haskell compile time, plus overhead from template Haskell and running the EDSL's code-generator to produce StreamIt code. Currently the EDSL code emission is unoptimized (*e.g.* uses `String`).

Load/Execute speed The runtime overhead of executable embedding can be significant for larger binary sizes. We evaluated the performance for a hello-world binary with increasing embedded binary sizes. The results are shown in Figure **??**. In each case, the program was run with 0 iterations and the numbers are relative to that of running the generated StreamIt program directly with 0 iterations. By default, we use gzip compression before embedding the binary. We also measured the load time by packing the generated executable with UPX [**?**] at maximum compression level. From Figure **??**, we see that a memory-backed filesystem considerable mitigates the copy costs and incurs lower runtime overheads. Owing to the large compression ratio of almost 30% however, the performance gain due to reduced writes trumps the unpacking overhead for UPX packed StreamIt binaries. As a result, the combination of using UPX and writing temporaries to RAM disk (`/dev/shm`) provides significantly better performance.

[8] GHC failed to link programs with a large number of filters due to the higher resulting executable size. This is most likely due to an inefficient in-memory representation of the embedded binary.

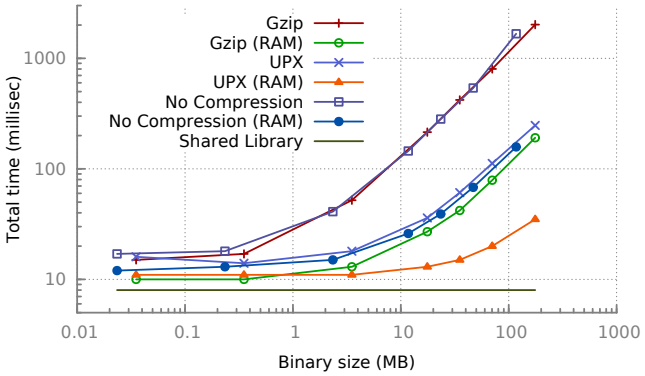

Figure 2. Load and execute overhead for a *hello-world* binary of different sizes. Embedded as a Haskell byte array with (1) Gzip compression, or (2) as a UPX packed executable, or with (3) no compression at all.

6.2 Communication between the Host and DSL-generated program

Some DSLs are designed with an *open world assumption*, *e.g.* they generate code meant to be incorporated into a larger host program. The Pochoir stencil compiler [**?**], CnC coordination language [**?**], Obsidian GPU language [**?**], and Kanor [**?**] are all examples of this approach. Unfortunately, many DSLs have a closed world assumption—they produce standalone binaries corresponding to DSL programs[9]. StreamIt falls into this category, having no foreign function interface and no facility for reading data anywhere but from files.

Nevertheless, even in this scenario, embedding techniques can still produce a usable EDSL. In particular, Stream^{Hs} takes care of creating named pipes to communicate between Haskell and StreamIt. (We could perhaps do even better with a FUSE file system, but our implementation of this is incomplete.) In spite of it's statically-scheduled nature, the OS will still force StreamIt to block on IO. And because StreamIt uses global barriers, when using the SMP backend, all threads will become blocked, none will busy-wait.

7. Evaluation

7.1 Composition Scenarios

Part of the closed-world assumption is that StreamIt traditionally achieves throughput (and granularity *agnosticism*[10]) based on a static-scheduling methodology that assumes exclusive access to a set of cores during each scheduler "epoch" (aka iteration). Programs with this assumption do not support dynamic load balancing (e.g. work stealing) and can be difficult to compose within a larger application. For example, if one core is unexpectedly busy, it's epoch-slice will not be finished on time when the other threads reach the global-barrier at the end of the epoch.

In fact, StreamIt exhibits a pathological version of this problem because (1) it pins all worker threads to specific cores and (2) it executes a global barrier that busy-waits. For example, on a four core desktop processor (3.10Ghz Westmere running RHEL6), the

[9] Embedded DSLs suffer from an analogous problem wherein they are only callable from one host language, with no ability to create a compiled artifact or separate generated code. Accelerate [**?**] and other GPU languages often fall into this category.

[10] That is, you may write arbitrarily fine-grained StreamIt programs without suffering scheduler overhead.

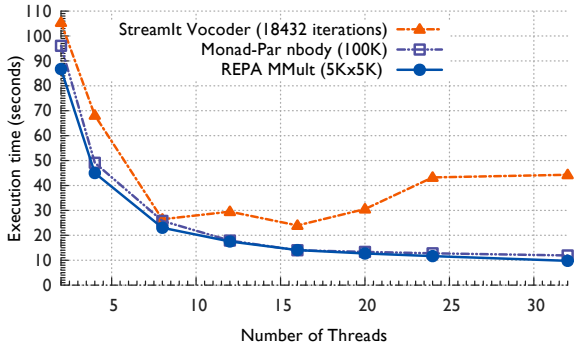

Figure 3. Baseline performance of a sequential Haskell program with synchronous calls to (1) the StreamIt Vocoder benchmark, (2) the Monad-Par nbody benchmark and (3) REPA matrix multiplication benchmark. Thread settings > 16 represent hyperthreading. Results are from the streamit-3.0 development branch, with a 16-core Intel E5-2670, and `strc -O2 --smp`.

StreamIt Vocoder benchmark takes 68.5 seconds realtime when partitioned onto four worker threads. If another process runs and keeps a *single core busy*, dropping to $0.75X$ the processing power, then the time increases to 194.3 seconds, or $2.83X$ slower. In fact, even *overpartitioning*, and running 8 threads on the same machine (with no interlopers) is just as bad (195.6 seconds), in spite of the fact that the threads are spread evenly: two per core. Note that part of the problem in this example is that the barriers happen fairly frequently, at $100Hz$.

Given these limitations, the only runtime composition strategy we can be confident will work effectively is:

• **Sequential Haskell program, synchronous call to StreamIt:** In this scenario the host program blocks, yielding the entire machine to the generated StreamIt code, without interference.

However, a surprising result of our experiments is that the runner up for least-bad composition is:

• **Parallel Haskell program, gang-scheduled components:** Some Haskell libraries such as REPA [**?**], utilize the same assumption as StreamIt—that they may launch P balanced tasks for P processors and expect them to run simultaneously. (But REPA does not busy-wait.)

In general, it is still better to run these two whole-machine tasks sequentially. However, as shown in Figure **??**, the overhead of running both together is less than a 25% slowdown as long as hyperthreading is not used. Further, at two threads the simultaneous version is actually faster!

Processors	Vocoder (perturbed)	Vocoder (baseline)	nbody-seq (perturbed)
32 (P)	48.632	44.301	78.134
31 (P-1)	44.212	44.049	58.919
24 (P-8)	44.242	43.236	51.406

Table 2. Sequential Haskell program (nbody-seq) running simultaneously with the StreamIt Vocoder benchmark. When all 32 processors are used, a 10% slowdown in Vocoder is seen as nbody-seq keeps one core busy.

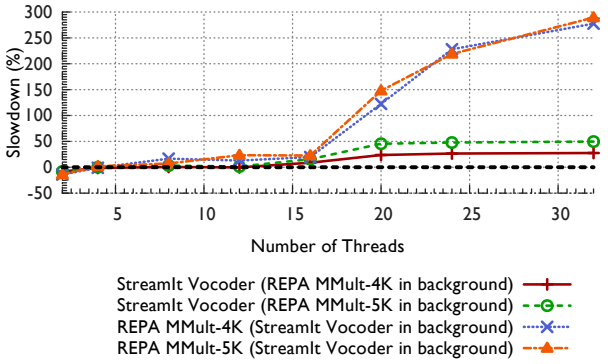

Figure 4. Running the StreamIt Vocoder benchmark in parallel with gang-scheduled, parallel matrix multiplication (REPA). For up to 16 cores, both benchmarks execute in isolation resulting in a negligible slowdown.

7.2 Future Work: Better Composition

In the future we plan to replace the global barrier used in StreamIt. This is a small, isolated part of the code that is easy to change. After that we will be able to experiment with the following two scenarios, which currently perform very badly:

• **Sequential Haskell program, asynchronous call to StreamIt:** This scenario is problematic, with the Haskell program disrupting the scheduled whole-machine StreamIt epoch, and even after solving the busy-waiting problem, there will be no fruitful computation to fill the gaps left by a disrupted StreamIt epoch.

One solution is to force the StreamIt compiler to target $P-1$ processors, which maintains full performance. A second solution is to have StreamIt target many more than P processors, to enable a form of load balancing (over-partitioning). This should reduce dead-time relative to the P processor version. From Table **??**, we see that this scenario causes a negligible slowdown, $1.09X$ at 32 cores as opposed to $2.83X$ at 4 cores, due to fewer number of iterations per core, and conseqeuntly fewer global barriers.

• **Parallel Haskell program, abundant parallelism:** In this scenario, even if StreamIt epoch slices are displaced in time, all processors should stay productive, utilizing the thread-migration and spark-stealing [**?**] load-balancing capabilities of the Haskell runtime, such as in Monad-Par [**?**]. Hence Haskell should be able to fill any gaps left by out-of-sync StreamIt epochs. In fact, we can

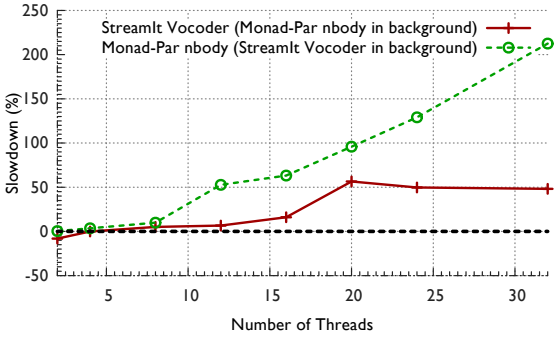

Figure 5. Performance degradation when running the StreamIt Vocoder benchmark in parallel with a nbody benchmark with abundant, dynamic parallelism implemented with Monad-Par.

confirm this from Figure **??** where the relative slowdown of the Monad-Par benchmark is significantly lesser than gang-scheduled REPA benchmark (ref. Figure **??**).

8. Conclusions

In this paper we present a case study embedding a language designed without such a scenario in mind, overcoming a number of obstacles in the process. We believe this can generally be done for parallel DSLs. Further, given that the most effective parallel DSLs are for particularly *narrow* domains (synchronous streaming, stencils, DSP transforms, image filters, etc), we expect a practical parallel programming environment in the future will require unfettered access to a number of *different* DSLs. To achieve this, one approach is to engineer several DSLs on top of a single compiler and runtime (*e.g.* Delite [**?**]). This paper proposes a second approach: retrofitting separately designed DSLs to fit into a host environment that will allow composition.