Abstract
We address the increasingly varied capabilities of specialized computing platforms by introducing an extensible family of functionally-limited mini-languages, implemented as embedded domain specific languages (EDSLs) in Haskell, that may be composed to harness the computational features offered by different hardware platforms. We present a novel modular representation of the EDSL that enables zero-overhead evaluation when running on the GHC runtime, deep term introspection for optimization purposes as a prelude to serialization for consumption by a lower-level compiler, and rich interplay between these two views that re-casts interpreters as optimizers.

Categories and Subject Descriptors D.2.7 [Programming Techniques]: Parallel Programming; D.2.11 [Software Architectures]: Domain-specific architectures; D.3.3 [Language Constructs and Features]: Frameworks

Keywords EDSL, Haskell, GPU

1. Introduction
The long-awaited era of mandatory parallelism has arrived, but it doesn’t look exactly as most of us anticipated. While we have been warned that single-threaded performance would not keep improving forever, we took solace in the future availability of many of those really quite good processor cores we had become so familiar with. While we have gotten a few more cores, there has been a coincident rebirth of interest in specialized hardware blocks. Though graphics processing units (GPUs) hinted at this, mobile computing has ignited an explosion in the variety of widely available specialized hardware, all in the name of conserving battery life. The variegated parallel computing landscape is not one defined by simple repetition of powerful processor cores, but is instead characterized by a panoply of computational power expressed in all manner of specialized silicon.

A selection of the muses behind this work are shown in Figure 1. We must work on robots smaller than a shoe box, powered by single-board computers with multiple CPU cores and integrated GPUs. We must work on human-sized robots carrying multiple server-grade CPUs, tens of gigabytes of RAM, and a discrete GPU. We must work on the 8-bit microcontrollers with a few kilobytes of RAM that intermediate so many of the interactions between software and hardware. Targeting these varied platforms means that we may be interested in programs that only make use of a specific set of arithmetic operations. Or those that do not allocate memory. Or those that make use of fast local memory shared among groups of threads executing in parallel. Or those that make use of hardware-accelerated image access operations that exploit specialized memory arrangements for sampling from 1-, 2-, or 3-dimensional dense arrays.

Taming the complexity of available hardware resources is a task well-suited to programming languages. However, we do not seek to identify a single high-level language which may be compiled to a variety of hardware, as this often serves to hide from the programmer the key limitations and capabilities that characterize individual hardware platforms. Thus, rather setting out to design a programming language, we have developed an approach to developing a family of related languages that expose the key features of specialized hardware while still benefiting from membership in a cohesive language ecosystem.

When considering significantly varied hardware platforms, software APIs – surely an extralingual consideration – become hard-
ware capabilities, and the engineer is well advised to hew closely
to the fast-path or end up with a program whose value is negated by
poor performance. Thus our aim is to compose feature-poor, low-
level object languages from sets of available language features that
expose capabilities such as arithmetic operations, memory access,
or even complex algorithms. We may then write software libraries
whose types describe the required features, allowing them to be
embedded in programs whose types will, in turn, reflect the union
of required features of all components.

While we will describe the use of the design described here in
Section 3, the implementation’s scope distracts from the core tech-
nique. To this end, we will first describe how we build a family of
EDSLs, then move on to several examples. We begin with a stand-
dard presentation of a “final tagless” encoding of an object lan-
guage in Haskell using type classes. We then introduce an “indexed
initial” encoding to capture the modularity of the tagless encod-
ing with an explicit representation of object language syntax. We
finally work our way to the “partially tagless” encoding in which
we combine the syntactic convenience and zero-overhead evaluation
properties of a tagless encoding with the perspicuousness of a
tagged initial encoding wherever such a representation is useful
during code generation.

2. A Simple Language

We begin with a simple language expressed in a finally tagless
[3] style. Such a shallow embedding [2], wherein we define the
semantic operations of the language, rather than explicit reification
of syntax, lets us hit the ground running for the important reason
that evaluation of the programs written in the embedded language is
equivalent to evaluation of Haskell programs: there is no interpreter
overhead. The constructions below use several recent features of
the GHC compiler; the source of this paper is a valid Haskell file
that may be compiled with at least GHC-7.8.4.

Our language will, as usual, have literal integer values and
addition. It will also support a notion of lambda abstraction in the
style of Parametric Higher-order Abstract Syntax (PHOAS)
[5]. This will enable us to let the metatanglange do the book-
keeping associated with the tracking of binders, and keep us honest
about parametricity, but still give us sufficient traction to inspect
the structure of terms in the object language.

```haskell
class ArithF e where
  lit :: Int → e Int
  add :: e Int → e Int → e Int

class AbsF e where
  lam :: (e a → e b) → e (a → b)
  app :: e (a → b) → e a → e b
```

Finally, we draw attention to the potential for supporting some-
what opaque operations that may be offered by specialized hard-
ware. The running example is a fancy operation that we have been
assured is very efficiently implemented on our target hardware.

```haskell
class FancyF e where
  fancyOp :: e Int → e Int
```

We refer to each of these classes as a sub-language to reflect
that they are most likely to be used in combination rather than indi-
vidually. A nice feature of the tagless style when used with Haskell
is that it immediately gives a lightweight form of modularity to lan-
guage construction. Each sub-language may be defined in its own
Haskell module along with derived functionality. Only later must
the desired features be brought into scope at once, and combined in
a composite class constraint.

The Haskell backend – in which our object language is simply
Haskell itself – is here implemented by the Identity functor.

```haskell
instance ArithF Identity where
  lit = pure
  add = liftA2 (+)
```

```haskell
instance AbsF Identity where
  lam f = Identity (runIdentity o f o Identity)
  app = (\$)
```

We will also implement software emulation for the fancy oper-
ation, but remember that this may be a complex definition.

```haskell
instance FancyF Identity where
  fancyOp = fmap (+42)
```

We will define our first object language as the union of two of
these sub-languages, each providing specific features.

```haskell
type MyLang e = (ArithF e, FancyF e)
```

We can write a test program that adds two literal values.

```haskell
testSum :: MyLang e ⇒ e Int

testSum = add (lit 2) (lit 3)
```

An evaluator for the native Haskell backend is the runIdentity
function that unwraps a value from the Identity data constructor.
Evaluating “runIdentity testSum” results in 5.

Of more interest is a backend that performs code generation to be
fed into another compiler.

```haskell
newtype Code a = Code {getCode :: Int → String}
```

The Code newtype is a function from lambda nesting level to
serialized code, while the type constructor has a phantom type
reflecting the type of the object language term to be serialized. We
will serialize our terms to a pseudo-language that looks a bit like
a functional language with lambda abstraction mixed in with infix
arithmetic.

```haskell
instance ArithF Code where
  lit = Code o const o show
  add (Code x) (Code y) =
    Code $ λ n → \("+" + n + " + " + y n + ")"
```

```haskell
instance AbsF Code where
  lam f = Code $ λ n →
    let x = "x_" ++ show n
    Code body = f (Code $ const x)
    subst = body (n + 1)
  in concat ["\(\"+\", x, \" \rightarrow \", subst, \")\"]
  app (Code f) (Code x) =
    Code $ λ n →
    concat [("\(" + f n, \" , x n, \")\")]
```

```haskell
instance FancyF Code where
  fancyOp (Code x) =
    Code $ λ n →
    "(hardwareOperation " + x n + ")"
```

With these instances in hand, we can verify that our Code type
produces syntax we could plausibly pass to another compiler. Evalu-
ating “getCode testSum 0” results in "(2 + 3)". Our little
language also lets us return functions, serialized in a simplistic
haskell-like syntax.

```haskell
testLam :: (AbsF e, ArithF e) ⇒ e (Int → Int)
  testLam = λ x → add x x
```
Evaluating “getLam testLam 1” renders such a program as $(\lambda x_1 \to (x_1 + x_1))$.

This construction has several desirable properties. It re-uses the metalanguage’s (Haskell’s) type system for type safety. It re-uses the metalanguage’s facility for lambda abstraction to keep track of binders. It permits modular construction of a compound language by separating the definition of each language feature as a distinct type class. Finally, the “backends,” or interpreters are defined separately, and, again, modularly. This lets one grow track of binders. It permits modular construction of a compound the metalanguage’s (Haskell’s) type system for type safety. It re-

2.1 Indexed Initial Encoding

A familiar complaint about tagless encodings is that they make context-sensitive operations somewhat difficult. Though this perceived weakness is being chipped away by continuing work on tagless encodings [11], initial encodings remain popular for the ease of inspection of the syntax trees. In such an encoding, the terms of the object language are explicitly reified as values in the metalanguage. These values may be inspected and manipulated as a step distinct from interpretation. Such accessibility is typically used to implement optimizations of the object language terms using metalanguage facilities.

In our case, we would at the very least like to perform arithmetic simplification where possible. One might hope that a downstream compiler receiving a serialized program containing the expression “$2 + 3$” would simplify it before generating machine instructions, but such reasonable expectations are often disappointed in embedded systems development. Instead, we should do what we can to perform general program optimizations before passing responsibility to a downstream compiler.

The modularity of the tagless approach was listed as a benefit when first introduced. This separation of concerns need not be squandered when moving to a parallel initial encoding that does reify the syntax of the object language. The approach taken here is to use a data family to define an open sum type for the Representation of our object language terms.

data family Repr :: k \to [k] \to (* \to *) \to * \to *

This family is indexed by a type indicating a particular sub-language, a set of sub-languages that may appear in sub-terms, a type constructor for binder references, and the type of the object language term.

Terms in the language will package up a Repr with a value-level tag indicating the sub-language that provides the necessary semantics. This is accomplished by pairing the representation of an object language term with a singleton [7] type.

The indexed initial GADT (generalized algebraic data type) encoding (denoted by an IG suffix) of our first object language will consist of the ArithIG and FancyIG features.

data LangIG = ArithIG | FancyIG

The singletons machinery for each language variant we discuss will look virtually identical, so it is only shown for this first implementation. This consists of an indexed data family and a type class that provides the unique value that inhabits each singleton type.

data family LangSing :: k \to *
data instance LangSing (a :: LangIG) where
SArithIG :: LangSing ArithIG
SFancyIG :: LangSing FancyIG
class ISing (a :: k) where sing :: LangSing a
data TermME :: [ LangME ] → ( * → * ) → * → * where
  TermME :: ( lang ∈ langs, EvalME lang )
  ⇒ LangSing lang
  ⇒ Repr lang langs e a
  ⇒ TermME langs e a

termME :: ( lang ∈ langs, Ising lang, EvalME lang )
  ⇒ Repr lang langs e a → TermME langs e a

termME = TermME sing

The data instances for the sub-languages are unchanged from the
first version, modulo a change in the suffixes on identifiers. However, now we define our evaluator’s behavior for each sub-
language independently.

instance EvalME ArithME where
  evalME _ ( LitME x ) = x
  evalME k ( AddFI x y ) = k x + k y

instance EvalME FancyME where
  evalME k ( FancyOpME x ) = k x + 42

This implementation may once again be spread across multiple
Haskell modules, allowing for the expansion of the language with-
out having to revisit existing definitions. The desired flexibility is
provided by defining the evaluation function for each type through
type class instances rather than a single function that dispatches on
data type constructors.

type MyLangME = [ ArithME, FancyME ]

runEvalFI :: ( ∀e. TermME MyLangME e a ) → a
runEvalFI t = go t
  where go :: TermME MyLangME Identity a → a
        go ( TermME _ x ) = evalME go x

testSumME :: TermME MyLangME e Int
testSumME = termME ( AddME ( termME ( LitME 2 ) )
                 ( termME ( LitME 3 ) ) )

Again, the expected result of “runEvalME testSumME” pro-
ducing 5 is achieved.

2.3 Finally Initial

An aspect of the above construction whose clumsiness is some-
what hidden by the simplicity of the object language is that our
EvalME class effectively recapitulates our tagless encoding: the
tagless encoding defines semantics for each language feature, and
these semantics are repeated for the subsequently introduced initial
terms. We can rectify that by connecting the two encodings (denoted with an FI suffix).

data LangFI = ArithFI | FancyFI

type family Finally ( l :: k ) ( e :: * → * ) :: Constraint

class EvalFI ( lang :: LangFI ) where
evalFI :: Finally lang e
  ⇒ ( ∀a. TermFI langs e a → a )
  ⇒ Repr lang langs e r → e r

data TermFI :: [ LangFI ] → ( * → * ) → * → * where
  TermFI :: ( lang ∈ langs, EvalFI lang, Finally lang e )
  ⇒ LangSing lang
  ⇒ Repr lang langs e a
  ⇒ TermFI langs e a

The definitions of termFI, and the data instances stay true to the
form established in the first iteration, but the modular evaluator
is somewhat changed. The Finally type family is used to connect
each sub-language syntax tag with its corresponding tagless seman-
tics.

  type instance Finally ArithFI e = ArithF e

instance EvalFI ArithFI where
  evalFI _ ( LitFI x ) = lit x
  evalFI k ( AddFI x y ) = add ( lit ( k x ) ) ( lit ( k y ) )

  type instance Finally FancyFI e =
    ( ArithF e, FancyF e )

instance EvalFI FancyFI where
  evalFI k ( FancyOpFI x ) = fancyOp ( lit ( k x ) )

The evaluator is similar to the modular evaluator iteration, but
the implementations of the pieces of the evaluator tie back into the
semantics provided by the tagless encoding. This connection is
provided by a closed type family, AllFinal, that builds a composite
constraint by applying a constraint constructor function to each of a
list of types.

  type MyLangFI = [ ArithFI, FancyFI ]

runEvalFI :: ( ∀e. AllFinal MyLangFI e → a )
  ⇒ TermFI MyLangFI e a
runEvalFI t = go t
  where go :: TermFI MyLangFI Identity b → b
        go ( TermFI _ x ) = runIdentity ( evalFI go x )

testSumFI :: ( ArithFI ∈ langs, ArithFI e )
  ⇒ TermFI langs e Int

testSumFI = termFI ( AddFI ( termFI ( LitFI 2 ) )
                   ( termFI ( LitFI 3 ) ) )

We can verify that evaluating “runEvalFI testSumFI :: Int” produces 5.

2.4 Partially Final

The careful support of multiple backends has thus far been lightly
motivated as providing an opportunity for software emulation of
hardware features. This is a genuine benefit, as embedded software
development is often conducted when the target hardware is unsta-
ble or not yet available in sufficient quantity to support the needs of
software development. However, intertwining of the software emu-
lolation and code generation lets us exploit the former in the act of
producing the latter. To demonstrate this, we will now consider an
object language that makes use of all three sub-languages we have
described thus far (the partially tagless final variant, denoted with a
PF suffix).

We will close the circle here, and use the original type classes
providing the finally tagless encoding to index an initially encoded
representation rather than a new data type promoted to a kind.
Thus, instead of using a kind, that we would name LangPF if we
were continuing the established pattern, we will use the kind of our
semantics-providing type classes, namely, ( * → * ) → Constraint.

This time, the continuation passed to the evaluator will not
return a bare value, but an object language term.

class EvalPF ( lang :: ( * → * ) → Constraint ) where
evalPF :: ( lang ∈ langs, lang e )
  ⇒ ( ∀a. TermPF langs e a → e a )
  ⇒ Repr lang langs r → e r

The utility of this distinction is made clear by specializing the
type of evalPF to a TermPF whose type of references is a
TermPF. This is the interface through which we will implement
partial evaluation of the terms of our object language. The default
The definition lets us pass the evaluator through terms that we do not know how to reduce.

```haskell
class PEval (lang :: (* → *)) → Constraint where
pevalPF :: (lang ∈ langs, lang (TermPF langs e)) ⇒ (∀ a. TermPF langs (TermPF langs e) a
                          → TermPF langs e a)
               → Repr lang langs (TermPF langs e) r
               → TermPF langs e r

default pevalPF
:: (lang ∈ langs, EvalPF lang, lang (TermPF langs e))
⇒ (∀ a. TermPF langs (TermPF langs e) a
               → TermPF langs e a)
               → Repr lang langs (TermPF langs e) r
               → TermPF langs e r
pevalPF = evalPF
```

The definition of `TermPF` follows the pattern of earlier definitions. However the familiar `Repr` data instances are now paired with instances of the tagless, semantic type classes for members of the `Repr` family.

```haskell
data TermPF :: [(∗ → ∗) → Constraint] → (∗ → ∗) → ∗ → ∗ where
TermPF :: (lang ∈ langs, EvalPF lang, PEval lang, lang e)
⇒ LangSing lang
⇒ Repr lang langs e a
⇒ TermPF langs e a

instance (ArithF ∈ langs, ArithF e)
⇒ ArithF (TermPF langs e) where
lit = termPF ∘ LitPF
add x y = termPF (AddPF x y)

instance (FancyF ∈ langs, ArithF ∈ langs, FancyF e)
⇒ FancyF (TermPF langs e) where
fancyOp = termPF ∘ FancyOpPF
```

The modular partial evaluators for the sub-languages will inspect the results of the evaluation of their sub-terms to determine if they can be simplified. In concrete terms, if the operands of an addition operation are both literal values, then we can simplify the addition in place.

```haskell
pattern AsLit x ← TermPF SArithPF (LitPF x)

litPF :: (ArithF e, ArithF ∈ langs)
⇒ Int → TermPF langs e Int
litPF = termPF ∘ LitPF

instance EvalPF ArithF where
evalPF _ (LitPF x) = lit x
evalPF k (AddPF x y) = add (k x) (k y)

instance PEval ArithF where
pevalPF _ (LitPF x) = lit x

pevalPF k (AddPF x y) =
case (k x, k y) of
  (AsLit x’, AsLit y’) → litPF $ x’ + y’
  (x’, y’) → add x’ y’

instance EvalPF FancyF where
evalPF k (FancyOpPF f x) = fancyOp (k x)
```

The definition of the `PEval` instance for `ArithPF` was done in a simple style to demonstrate the value of the connection between encodings we have maintained. It simply uses the Haskell `+` operator to add two literal `Int` values. This is effective, but belies the power available us. In order to properly consider partial evaluation, we will need terms that we can not necessarily evaluate down to numbers.

```haskell
data instance Repr AbsF langs e a where
LitPF :: Int → Repr ArithF langs e Int
AddPF :: TermPF langs e Int
⇒ TermPF langs e Int
⇒ Repr ArithF langs e Int

instance (ArithF ∈ langs, ArithF e)
⇒ ArithF (TermPF langs e) where
lit = termPF ∘ LitPF
add x y = termPF (AddPF x y)

instance (FancyF ∈ langs, ArithF ∈ langs, FancyF e)
⇒ FancyF (TermPF langs e) where
fancyOp = termPF ∘ FancyOpPF
```

Earlier, we used Haskell’s `+` operator to evaluate addition between two statically known `Ints`. Here, we will substitute a statically known `Int` into the body of a function using Haskell’s own function application.

```haskell
pattern AsLam x ← TermPF SAbsPF (LamPF x)

instance PEval AbsF where
pevalPF k (LamPF f) = lam $ k o f o termPF o VarPF
                       ∘ (∀ x. VarPF x) = x
pevalPF k (AppPF f x) = app (k f) (k x)
```

Running `pevalPF` doesn’t leave us with a Haskell value, but instead with another `TermPF` that may be simpler than the one we started with. The interface for partially evaluating any safe term in our language is provided by `partialEval`. The `All` closed type family provides a constraint that each language feature is supported by the representation.

```haskell
type MyLangPF = [ArithF, FancyF, AbsF]
partialEval :: All MyLangPF e
⇒ (∀ f. All MyLangPF f
⇒ TermPF MyLangPF f a)
⇒ TermPF MyLangPF e a
partialEval t = go t
where go :: TermPF langs (TermPF langs e) a
⇒ TermPF langs e a
  go (TermPF _ x) = pevalPF go x
```

The evaluator is very similar to the previous iteration, with the only difference being that the `Identity` data constructor is stripped off only at the very end of evaluation rather than at each step.
runEvalPF :: (∀ e. All MyLangPF e
⇒ TermPF MyLangPF e a) → a
runEvalPF t = runIdentity (go t)
where go :: TermPF langs Identity b → Identity b
     go (TermPF _ x) = evalIPF go x

We could define yet another testSum-like term using this fourth iteration of our language construction, but at this point we have come around full circle and can instead work with a value constructed with the semantic, tagless embedding. For example, evaluating “runEvalPF testSum :: Int” produces 5.

The real win with this final iteration of our construction is our ability to work with terms that cannot be immediately normalized to numeric values. We will explore this capability by working with function values, whose serialization we may render by converting the tagged encoding to the Code type through the tagless encoding.

i2f :: TermPF langs e a → e a
i2f (TermPF _ x) = evalIPF i2f x

Here is a slight variation of our running summation test program,

testProg :: (AbsF e, ArithF e) ⇒ e (Int → Int)
testProg = lam $ λx → add x (add (lit 2) (lit 3))

Serializing this program with “getCode testProg 1” gets us \((λx_1 → (x_1 + (2 + 3)))\). But we can also consider serializing the same program after partial evaluation.

peCode :: Code (Int → Int)
peCode = i2f (partialEval testProg)

Serializing this value with “getCode peCode 1” results in \((λx_1 → (x_1 + 5))\).

We will use another program to exercise our ability to β-reduce applications where inlining the argument makes sense.

testApp :: (AbsF e, ArithF e) ⇒ e Int
testApp = app (lam $ λx → add x x) (lit 21)

The original syntax of the test application is \(((λx_1 → (x_1 + x_1)) 21)\). But we can run this program through our partial evaluator,

peApp :: Code Int
peApp = i2f (partialEval testApp)

which leaves us with 42.

At long last, we can demonstrate using the software implementation of a hardware-accelerated operation when optimizing the code generation of a program targeting a specialized hardware platform.

instance PEval FancyF where
pevalPF k (FancyOpPF x) = case k x of
    AsLit x' → litPF o runIdentity $ fancyOp (lit x')
    x' → fancyOp x'

In contrast to the PEval instance for ArithPF where we used Haskell’s + operator to implement addition, here we use the tagless encoding’s fancyOp definition to simplify a syntactic term whose head is FancyOpPF.

testFancy :: (AbsF e, ArithF e, FancyF e)
     ⇒ e (Int → Int)
testFancy = lam $ λx →
          add x (fancyOp (add (lit 5) (lit 10)))

The raw syntax for this test program is \(λx_1 → (x_1 + (\text{hardwareOperation} (5 + 10)))\). Partially evaluating the program,

peFancy :: Code (Int → Int)
peFancy = i2f (partialEval testFancy)

gets us to \(λx_1 → (x_1 + 57)\). This mechanism means that software emulation of hardware capabilities may be integrated directly into an optimization pipeline that feeds into a specialized compiler for a target hardware platform.

3. Scaling Up: HOCL

The technique presented thus far is the backbone of the Haskell Open Compute Language (HOCL) project. HOCL is a loose assembly of sub-languages – collections of tightly related features – that may be brought together in a piecemeal fashion to express some bit of functionality. These components comprise a toolkit with which one may construct programs targeting a computational substrate capable of dealing with the union of sub-languages used to express the program.

Our aim in this work is to be able to work with component libraries that can be re-used across hardware platforms from 8-bit microcontrollers, to multi-core CPUs, to distributed systems. But in order to investigate the effectiveness of the approach, we first apply our tools to a more familiar domain: graphics processing unit (GPU) programming.

We will specifically consider the generation of OpenCL (Open Computing Language) [1] code from HOCL fragments. OpenCL is a cross-vender, cross-platform parallel programming standard. While its initial use has been as an alternative to NVIDIA’s CUDA and Microsoft’s DirectCompute on high-end GPUs, its open nature is leading to increasing availability on embedded platforms.

OpenCL is a C-like language that adds first-class support for small vector types, and specialized image types. The former map well onto problems dealing with two- or three-dimensional geometry or color information, while the latter free GPU designers to use non-linear addressing schemes, hierarchical caching, and fixed-function interpolation hardware for accessing memory devoted to image storage. An interesting aspect of OpenCL and GPU-focused languages is an implicit iteration over a programmer-supplied number of work items. Each work item is viewed as an independent entity, while in fact the compiler is able to use this independent threads model of computation to better utilize available hardware, potentially coupling distinct threads quite tightly to improve utilization.

A useful approximate model from a high-level point of view is that the loop over all work items has been fully unrolled, allowing each work item to run in parallel. Looking back up from a low-level point of view, we can see that discouraging inter-thread communication is a natural way to gain freedom in how a loop may best be unrolled across hardware resources.

HOCL is capable of generating OpenCL programs to be executed by an OpenCL driver, or running the same programs in native Haskell. All HOCL programs begin as Haskell values built with the tagless encoding. When the OpenCL backend is used, these terms are run through the partial evaluator, then serialized to an intermediate representation consisting of imperative statements. These statements are passed through a suite of optimizations such as common subexpression elimination, let-floating, and dead-code elimination before being serialized as OpenCL syntax.

The Haskell backend offers a programming model similar to OpenCL wherein the programmer specifies how many work items to kick off. HOCL’s Haskell backend allocates ranges of work items to each available hardware thread until all work items have been completed. One caveat of OpenCL running on the CPU is that

Compositional EDSL construction.
Compositional EDSL construction.

it does not support specialized image operations. Arrays are supported on every target, but only the GPU can work with image types, where an OpenCL implementation may leverage specialized hardware for caching and indexing into images. The Haskell backend supports images, but in a manner that is equivalent or slightly slower than arrays, so only arrays are used in the following tests when running natively in Haskell.

All performance is measured with GHC-7.8.4 on a 1.3 GHz Intel Core i5-4250U processor with two physical cores with an integrated Intel HD Graphics 5000 GPU. The processor is equipped with Intel’s Hyper-Threading technology, so presents itself to the operating system as four cores. Each example will be paired with a short analysis to highlight its contribution to the overall picture.

3.1 Pan

The first demonstration is an implementation of Conal Elliot’s Pan [8] language for functional image specification. Pan’s key design feature is the recognition that functions from 2D points to colors are an excellent representation of scalable images. The representation has the additional benefit that functions are very nice to work with within a functional programming language, thus enabling the definition of small building blocks from which more complicated images may be assembled. This demonstrates the expressive power of using a functional EDSL to produce imperative code.

We begin restating the Pan design in HOCL terms by defining a few key type synonyms that lay bare the representation choice.

```
type Point e = e (V2 Float)
type Color = V4 Float
type ImageC e = Point e → e Color
type Transform e = Point e → Point e
```

A point is a pair of `Floats` in the language `e`. A color image is a function from a point in `e` to four `Floats`. A transformation is represented quite directly as a function on points. Such modest beginnings provide the building blocks with which complex, visually interesting figures may be constructed.

We can warm up with a recreation of one of Elliott’s early examples: concentric blue and yellow rings, Figure 2. The image is formed by linearly interpolating between a solid blue image and a solid yellow image with a radial cosine function. A few of the supporting definitions are shown before the definition of `ybRings` itself.

```
blueI :: Hoel e ⇒ ImageC e
blueI = const blue
lerpI :: Hoel e
```

A Haskell expression based on this function is evaluated to generate the following OpenCL program.

```
kernel void myKernel(write_only image2d_t xA) {
  const int2 tmp0 = (int2)(get_global_id(0),
    get_global_id(1));
  const int2 tmp1 = (int2)(get_global_size(0),
    get_global_size(1));
  const float tmp2 = (cos(
    fast_length(
      (float2)(10.0f / (float)(tmp1.x)) * 
    convert_float2(tmp0 -
      (tmp1 / (int2)(2, 2)))) * 
  3.1415927f) + 1.0f) * 0.5f;
  write_imagef(xA, tmp0,
    ((float4)(tmp2 * 
      (float4)(1.0f, 1.0f, 0.0f, 1.0f)) + 
    ((float4)(1.0f - tmp2) * 
      (float4)(0.0f, 0.0f, 1.0f, 0.0f));
  return;
}
```

We see that the OpenCL state accessor functions, like that which obtains a thread’s global ID, are called only once with any given argument, and that their return values are re-used throughout the composite program. Similarly, the expensive `cos` application is evaluated only once. This exploitation of sharing, recovered during low-level optimization of the HOCL program, is crucial for enabling compositionality at the Haskell level of specification.

One of the visually richer Pan examples is a figure extending the previous one: concentric yellow and blue rings that have a radial ripple applied before being swirled about their origin, Figure 3(a). This program is expressible in HOCL without much fussing over the original Pan syntax.

```
fig22 :: Hoel e ⇒ e (V4 Float)
fig22 = (swirl 8 o rippleRad 5 0.3 $ crop mask ybRings)
where mask = uscale 5 5 udisk
```

Pan is an exciting example as it lets us investigate the performance of HOCL not just as we vary the number of processor cores, but as we change over from GHC running a Haskell program to the OpenCL runtime executing an OpenCL kernel on the CPU, to the OpenCL runtime executing a kernel on the GPU. This grants us an opportunity to see not just relative scaling with hardware resources, but the relationship of absolute performance across backends. As suggested by Section 2, a HOCL backend is selected by invoking an evaluator-like function with a HOCL program that is fully parametric with respect to its concrete representation. A function like `runIdentity` is used to evaluate a HOCL program as a Haskell function, while code generation is performed similarly to the `peFancy` example above wherein we partially evaluate a program, serialize its representation to concrete OpenCL syntax, then pass that program representation to an OpenCL compiler provided by a hardware driver.

Moving from right to left in Figure 3(b), we see how the Haskell backend for HOCL scales across CPU cores by leveraging the GHC runtime system (RTS). In this case, we evaluate a Haskell function to compute the value of each pixel, and instruct the RTS how many
Haskell specification of an algorithm. The way in which each pixel of performance characteristics across compilers given a single program to parallelize extremely well – before a less than linear scaling as we move from hardware processor cores to CPU threads as provided by Hyper-Threading.

At this point in the analysis, it is important to note that the design above is using four single-precision floating point numbers to represent a single pixel. One area in which GHC is not particularly adept is leveraging the “single instruction, multiple data” (SIMD) instructions offered by the CPU. The OpenCL compiler, on the other hand, is more than ready to exploit the availability of such features. The more than double performance improvement we see when moving from GHC (running with its +RTS -N flag that frees it to use all available processor cores) to OpenCL on the CPU is right in line with what we would hope for from greater SIMD utilization.

Finally, we let OpenCL target the GPU. While the definition of \texttt{fig22} remains unchanged, this time we feed it an OpenCL image rather than an array, and define that image to use a single byte for each color channel. This 4x reduction in memory bandwidth adds to the GPU's advantage in SIMD width to offer a significant performance boost. The ability to take advantage of the reduced-precision floating point operations available on the GPU is a great advantage of writing program fragments that are isolated from so many implementation details.

3.2 Conway's Game of Life

The performance analysis of the Pan example showed a continuum of performance characteristics across compilers given a single Haskell specification of an algorithm. The way in which each pixel is computed in an independent manner is easily parallelized, while the final calculation that produces a color for each rasterized pixel is actually fairly complicated arithmetically. We will now turn our attention to a test case that highlights the strength of the OpenCL compiler as compared to GHC, but, more importantly, has a slightly different memory access pattern that highlights the benefit of explicitly surfacing hardware features in the object language. In this case, we will see the effect of GPU image handling support directly compared to a traditional linearly-arranged array.

John Conway’s solitaire variant, known simply as Life for its resemblance to the ebb and flow of a colony of living organisms[9], is a popular demonstration of parallelism. Its time evolution is defined entirely by a starting configuration consisting of a binary state encoded in each cell of a rectangular grid, and the repeated application of three rules,

1. Survivals. Every counter with two or three neighboring counters survives for the next generation.
2. Deaths. Each counter with four or more neighbors dies (is removed) from overpopulation. Every counter with one neighbor or none dies from isolation.
3. Births. Each empty cell adjacent to exactly three neighbors – no more, no fewer – is a birth cell. A counter is placed on it at the next move.[9]

We translate this into program code that works with numeric values represented by a single byte, where a zero indicates an empty cell. The logic to update an individual cell, or counter, is implemented by convolving the $3 \times 3$ neighborhood centered at the cell with a stencil that is 1 everywhere, except in the center where it is 16. To compute the new state of the cell, we check if the result of the convolution indicates that the cell was alive and had precisely two neighbors; if so, it survives. If the cell had three neighbors, the convolution would produce a number whose least significant four bits would equal the decimal value 3, whether the central cell was initially dead or alive. A low-level kernel that reads from the previous state and writes to the output state, each represented as a two-dimensional image, is shown here.

\begin{verbatim}
conway :: Hocl e
⇒ e (ImgM V2 Word8)
⇒ e (Img V2 Word8)
⇒ e (HIO ()

conway dst = writeImage dst globalID
  app (lam step) go where
  step x = cond (x ==? 16) 1 0)

We describe this kernel as low-level as it relies on OpenCL’s division of labor across discrete work items. That implied iteration is exposed to the programmer as the primary execution API of OpenCL, \texttt{enqueueNDRangeKernel}, which takes among its arguments the number of work items to execute. That the kernel is in fact the body of some outer control structure is visible in the arguments to \texttt{globalID} which identifies the current work item. Here, the 2D work item identification mechanism is used to index the state of the game board.

The stencil used in the convolution is known at code generation-time as a Haskell value, and is passed to the \texttt{Array} sub-language to be placed in OpenCL’s constant memory pool. A library function, \texttt{stencilImg}, is used to generate the necessary iteration over the stencil using the \texttt{Iteration} sub-language. The \texttt{cond} function
produces a conditional expression using the \textit{Bool} sub-language, while the bitwise union and intersection operators, \((\mid\mid)\) and \((\&\&).\) are provided by the \textit{Bitwise} sub-language. Finishing things off is a call to \textit{writeImage}, provided by the \textit{ImageM} sub-language for working with mutable images.

This Haskell program is evaluated, producing an initial encoding used for arithmetic simplification, then serialized through the tagless encoding into a series of imperative statements. These statements are run through a handful of optimization passes that make use of the fact that the statement generator makes extremely limited use of mutation. Finally, the optimized program is serialized and sent to the OpenCL driver for compilation to an active hardware platform (i.e. a CPU or GPU, depending on how OpenCL is initialized). The OpenCL kernel generated for the above Haskell definition is shown below with only a few edits to limit line length,

\begin{verbatim}
const sampler_t sampler_FALSE_EDGE_NEAREST =
  CLK_NORMALIZED_COORDS_FALSE |
  CLK_ADDRESS_CLAMP_TO_EDGE |
  CLK_FILTER_NEAREST;

constant uint c11[9] =
  {1u, 1u, 1u, 1u, 16u, 1u, 1u, 1u, 1u};

kernel void myKernel(write_only image2d_t xA,
  read_only image2d_t xB) {
  int x1 = 1;
  uint x2 = 0u;
  const int2 tmp0 = (int2)(get_global_id(0),
    get_global_id(1));
  for(int i0 = -1; i0 < 2; i0 = i0 + 1) {
    uint x8 = x2;
    for(int i7 = -1; i7 < 2; i7 = i7 + 1) {
      const uint tmp1 = c11[x1 + i7];
      const uint tmp2 =
        read_imageui(xB,
          sampler_FALSE_EDGE_NEAREST,
          tmp0 + (int2)(i7, i0)).x;
      x8 = x8 + (tmp1 * tmp2);
    }
    x1 = x1 + 3;
    x2 = x8;
  }
  write_imageui(xA, tmp0,
    (x2 == 18u) ? 1u :
      ((3u == (x2 & 15u)) ? 1u : 0u));
  return;
}
\end{verbatim}

The performance of this program, shown in Figure 4, exposes the inefficient handling of byte-sized data by GHC and the vector library. In this case, while running with the Haskell backend may have portability benefits, or improve the available profiling tools, the OpenCL backend represents a significant performance improvement. When we move from the CPU to the GPU, we can highlight the performance delta between arrays and images. The GPU, running the exact same array-based kernel as the CPU, is nearly four times faster, but it can run twice as fast again if the game state is carried by images rather than arrays.

### 3.3 Distance Functions

As a natural tangent to the Pan examples, we finally consider 3D functional rendering using distance function rendering techniques wonderfully explicated by Ólafur Quílez [13]. In this model, a geometric solid is represented as a function that computes the distance between any 3D point and the surface of the solid. If it is a signed distance function, then points within the interior of the object may be distinguished from those outside. With these representations in hand, one may render a scene by casting rays from a camera’s point-of-view until the rays intersect the surface of a solid. As with Pan, the choice of representation makes composition and transformation delightfully elegant.

Quílez provides a collection of geometric primitives specified as OpenGL shading language (GLSL) programs, which may be ported to HOCL without much change. The largest change is gaining polymorphism in the type used to represent the field underlying the vector space we are working in. This again lets us operate with varying precision depending on our accuracy and performance needs. This example stresses computation capabilities when evaluating distance functions, as well as memory access when considering shadowing and final image composition. This example demonstrates that the fine-grained specification of a dialect allows HOCL to play the very specific role of a GLSL stand-in without compromising computational generality.

We begin by defining what a distance function is,

\[
\text{type } DFun \ e a = e (V3 \ a) \rightarrow e a
\]

The signed distance from any point to the surface of a sphere centered at the origin may be computed by comparing the distance between the point and the origin with the sphere’s radius.

\[
sphere :: Floaty \ e a \Rightarrow e a \rightarrow DFun \ e a
\]

An infinitely long cylinder of a given radius, parallel to the Y axis is defined by comparing the difference between the cylinder axis’s X-Z location and the projection of the point onto the X-Z plane with the cylinder’s radius.

\[
cylinder :: Floaty \ e a \Rightarrow e (V3 \ a) \rightarrow DFun \ e a
\]

Such primitives may then be unioned, subtracted, and even blended to achieve smooth, organic results. Figure 3.3 shows a scene built from a prism with rounded corners sunk and blended into a large rectangular pedestal. The vertical prism then has a cylinder subtracted from it, and the entire scene is rendered with a moving light source and soft shadows.

This example runs at 40-50Hz at 640 × 480 on the aforementioned Intel HD 5000. While this scene is not particularly complex, this style of rendering is recently gaining in popularity thanks to...
the greater availability of GPUs that offer the dense parallelism necessary. That the GLSL programs translate so nicely to HOCL is evidence that the core pieces needed for this software-style rendering are provided despite HOCL not being specifically designed as a language for graphics.

The choice to focus on GPU programming raises the question of how the approach presented here compares to existing Haskell EDSLs for GPU programming. The Obsidian [6, 14], Nikola [12], and Accelerate [4] libraries all provide a mechanism for Haskell programmers to take advantage of GPU hardware. The differences between HOCL and these libraries are both incidentally and significant. Incidentally, these earlier libraries all primarily target NVIDIA’s CUDA language. This is not a viable target for the embedded systems we considered when starting in on this work. More significantly, each of these packages is aimed at providing a more or less low-level imperative code generation. A satisfying story for type-safe optimization is presented by Kameyama [11] by introducing a small bit of ceremony in how function parameters are referred to. This change permits type-safe scope extrusion, wherein, for example, some definitions are lifted out of loops. A let-floating optimization present in HOCL is comparable, but its implementation is not guaranteed correct by virtue of type checking.

Acknowledgments
Thanks to Gershom Bazerman, Jared Nicholson, and Gabor Greif for helpful feedback on this presentation.

References


