DSLs in Haskell

Ryan Newton

2015.07.15

INDIANA UNIVERSITY


12-18
JULY, 2015
Abstract Submission Deadline
April 1, 2015
{SYMBIOTIC LIFESTYLE}
UNIVERSIDADE DE LISBOA
PORTUGAL
iss-symbiosis.org
EIGHTH
CONGRESS
OF THE
INTERNATIONAL
SYMBIOSIS
SOCIETY
Organizing Committee:
Silvana Munzi
Cristina Cruz
Faculdade de Ciências da
Universidade de Lisboa
Rusty Rodriguez
Adaptive Symbiotic Technologies
Irene Newton
Indiana University, Bloomington
Scientific Committee:
Silvana Munzi
Cristina Cruz
Rusty Rodriguez
2015congress@internationalsymbiosissociety.org
Ciências
ULisboa
CE3C
International
Symbiosis
Society

Haskell features for DSL Construction

Front-end

- Template Haskell: typed and untyped splices
- Rebindable syntax
- Type-safe observable sharing
- Alternate "Prelude"s
- Type classes + overloaded literals

Middle-end

Compiler construction technologies

- Scrap-your-boilerplate (SYB)
- Syntactic library
- GADT ASTs for type preservation
- Nanopass tooling
- Finally-tagless abstract syntax

Back-end

- Quasiquoters: foreign syntax blocks
- Type-safe backends (e.g. LLVM)
- Finally-tagless for mixed shallow/deep embedded exec.

You have cabal & GHC 7.8.4, right?

- Note, lots of competing “easy” Haskell installers:
  - Haskell Platform
  - Halcyon
  - Stackage.org (“stack”)
  - Kronos Haskell

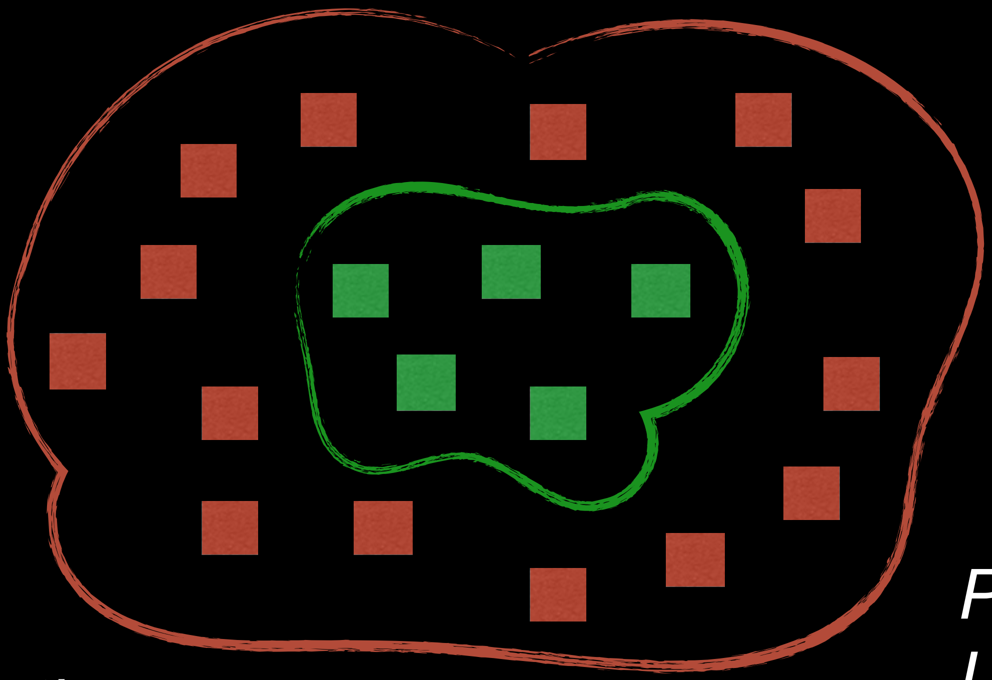
- Now please grab this repo. Either URL:
  - `git@github.com:iu-parfunc/haskell_dsl_tour.git`
  - https://github.com/iu-parfunc/haskell_dsl_tour.git

Themes & concepts… let’s talk about

Bundling


SAMSUNG
My Verizon M...
Mobile Hotspot
Verizon Tones
VZ Protect
VZ Navigator
Accessories
NFL Mobile
Games

Scientific Journals

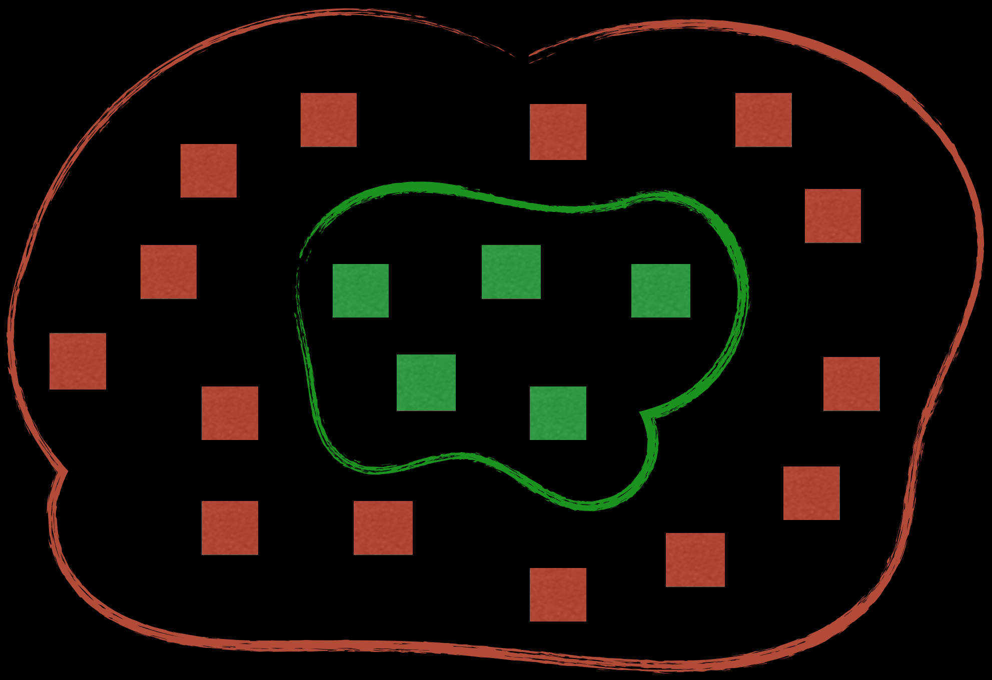


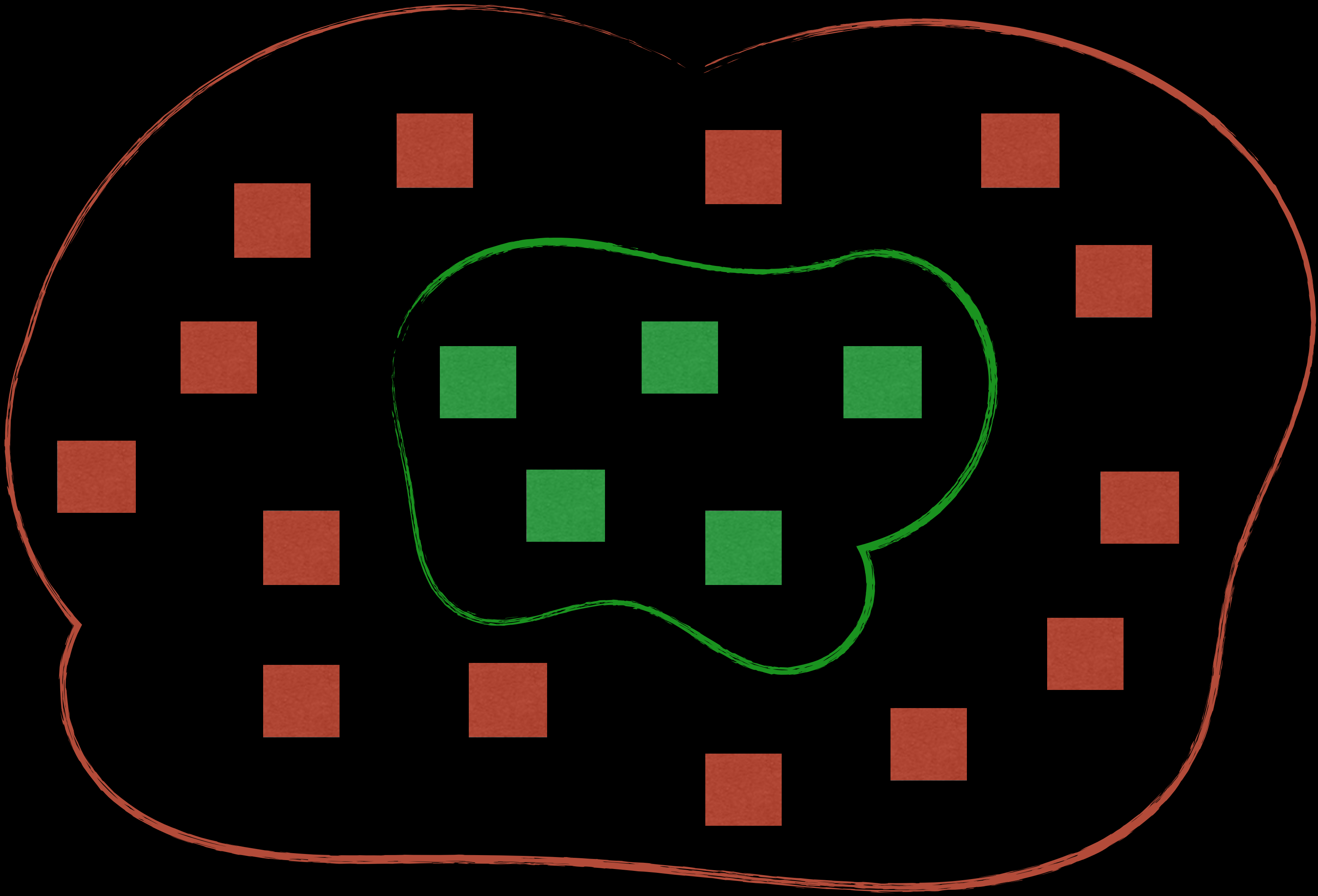
Operating system

Programming Language

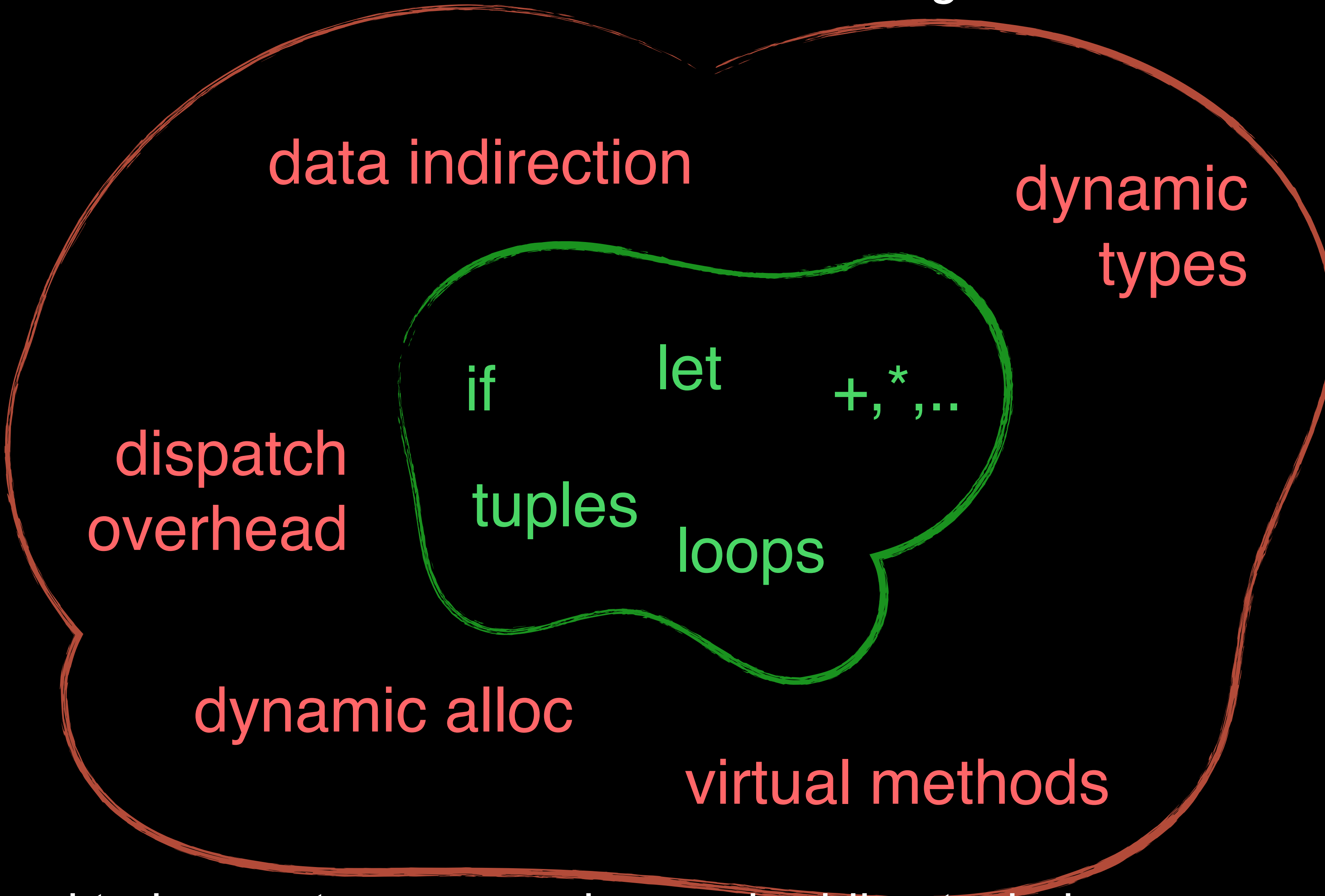
Answer: DSLs

Answer: unikernels





Abstraction without regret



Good today: metaprogramming, embedding techniques

Still immature: fine grained capability tracking, phase polymorphism

Second Theme: Type safety

- Front-end embeddings that use GADTs to retain types.
- Middle end: GADT ASTs that propagate types through compilation.
- Backend:
  - Syntax-safe quasi-quote splices
  - Type safe LLVM bindings

Main examples drawn from:

- Accelerate

- (Mini / nano) Feldspar

Haskell features for DSL Construction

Front-end

- Template Haskell: typed and untyped splices
- Rebindable syntax
- Type-safe observable sharing
- Alternate “Prelude”s
- Type classes + overloaded literals

Middle-end

Compiler construction technologies

- Scrap-your-boilerplate (SYB)
- Syntactic library
- GADT ASTs for type preservation
- Nanopass tooling
- Finally-tagless abstract syntax

Back-end

- Quasiquoters: typed syntax blocks
- Type-safe backends (e.g. LLVM)
- Finally-tagless for mixed shallow/deep embedded exec.

Haskell features for DSL Construction

Front-end

- Template Haskell: typed and untyped splices
- Rebindable syntax
- Type-safe observable sharing
- Alternate "Prelude"s
- Type classes + overloaded literals

Middle-end

Compiler construction technologies

- Scrap-your-boilerplate (SYB)
- Syntactic library
- GADT ASTs for type preservation
- Nanopass tooling
- Finally-tagless abstract syntax

Back-end

- Quasiquoters: typed syntax blocks
- Type-safe backends (e.g. LLVM)
- Finally-tagless for mixed shallow/deep embedded exec.

- Type classes (simple overloading)
- Rebindable syntax
  - Alternate preludes
- Unsafe sharing observation on which safe can be built (McDonell, ICFP'13)
- Template Haskell

When is overloading not enough?

- Needs a better story for:
  - data type definitions
  - pattern matching

Example: a concurrent data structure DSL

- Small set of operations:
  - define data types
    - sums + products
    - mutable locations + mutable arrays
  - bind recursive functions
  - basic operations
    - readIORef, writeIORef, casIORef, fetchAndAdd

Template Haskell for DSLs

- Paper: "Optimising Embedded DSLs using Template Haskell"
  - makes things easier when Haskell is the target lang for the DSL
  - (Yes, like LISP.)

- But I think the more compelling case is handling declarations.

Final caveat

- Front end stuff should *not* over constrain a DSL's fate
  - Build multiple front ends:
    - different languages
    - different embedding technologies
  - Build your core tech into an *engine* (VM) which has a clean API.
- Examples:
  - ArBB, Copperhead (2), Accelerate (in progress)

Haskell features for DSL Construction

Front-end

- Template Haskell: typed and untyped splices
- Rebindable syntax
- Type-safe observable sharing
- Alternate "Prelude"s
- Type classes + overloaded literals

Middle-end

Compiler construction technologies

- Scrap-your-boilerplate (SYB)
- Syntactic library
- GADT ASTs for type preservation
- Nanopass tooling
- Finally-tagless abstract syntax

Back-end

- Quasiquoters: typed syntax blocks
- Type-safe backends (e.g. LLVM)
- Finally-tagless for mixed shallow/deep embedded exec.

Compiler construction is folklore

- ask 3 major authors/maintainers, get 3 stories
- Basics:
  - algebraic sum types
  - OOP hierarchies (Exp superclass, If subclass)
  - expression problem
- Plus: generic traversal (SYB), binder representation …

Cool compiler tricks in Haskell

- Finally tag-less
- Sum type “thinning” with class constraints and phantom types
- Open unions / expression problem (Syntactic)
- SYB to walk trees, fv in 3 lines, not O(N)

Finally tagless

- Parameterizes over syntax representation

- Remains agnostic to deep/shallow embedding
  - form an AST, if desired, OR
  - just desugar into Haskell code
    (no explicit codegen step)

github.com/hakaru-dev/hakaru/

```haskell
-- TODO: incorporate HNat
class (Order repr 'HInt , Num        (repr 'HInt ),
       Order repr 'HReal, Floating   (repr 'HReal),
       Order repr 'HProb, Fractional (repr 'HProb))
    => Base (repr :: Hakaru * -> *) where
  unit       :: repr 'HUnit
  pair       :: repr a -> repr b -> repr ('HPair a b)
  unpair     :: repr ('HPair a b) -> (repr a -> repr b -> repr c) -> repr c
  inl        :: repr a -> repr ('HEither a b)
  inr        :: repr b -> repr ('HEither a b)
  uneither   :: repr ('HEither a b) ->
                (repr a -> repr c) -> (repr b -> repr c) -> repr c
  true       :: repr 'HBool
  false      :: repr 'HBool
  if_        :: repr 'HBool -> repr c -> repr c -> repr c

  unsafeProb :: repr 'HReal -> repr 'HProb
  fromProb   :: repr 'HProb -> repr 'HReal
  fromInt    :: repr 'HInt  -> repr 'HReal
```

Where’s the grand synthesis?

- My belief:

- People have deployed so much cleverness, that if you try all of the techniques at once, your brain explodes.

- But that doesn’t mean that we won’t eventually figure it out.

Still not there yet, even solo

- A good nanopass story
  - One example tool: our p523 compiler toolchain
  - Given grammar0 + delta1..deltaN,
  - Generates ASTs and common functions

Syntactic

- See nanofeldspar example

Prototype nanopass tool

- NO sophisticated types
- Codegen tool that generates dumb types
- SExp lang defs:

```
(l01-parse-scheme
   (%remove Expr)
   (%add
    (Expr
     (quote Datum)
     (let    ([UVar Expr] *) Body)
     (letrec ([UVar Expr] *) Body)
     (lambda (UVar *) Body)
     (if Expr Expr Expr)
     (begin Expr * Expr)
     (set! UVar Expr)
     (ValPrim Expr *)
     (EffectPrim Expr *)
```

```
(l02-convert-complex-datum
 (%remove (Expr quote))
 (%add (Expr (quote Immediate)))
 )

(l03-uncover-assigned
 (%remove Body)
 (%add (Body (assigned (UVar *) Expr)))
 )

(l04-purify-letrec
 (%remove (Expr letrec lambda))
 (%add
  (Expr (letrec ([UVar Lamb] *) Body))
  (Lamb (lambda (UVar *) Body))
```

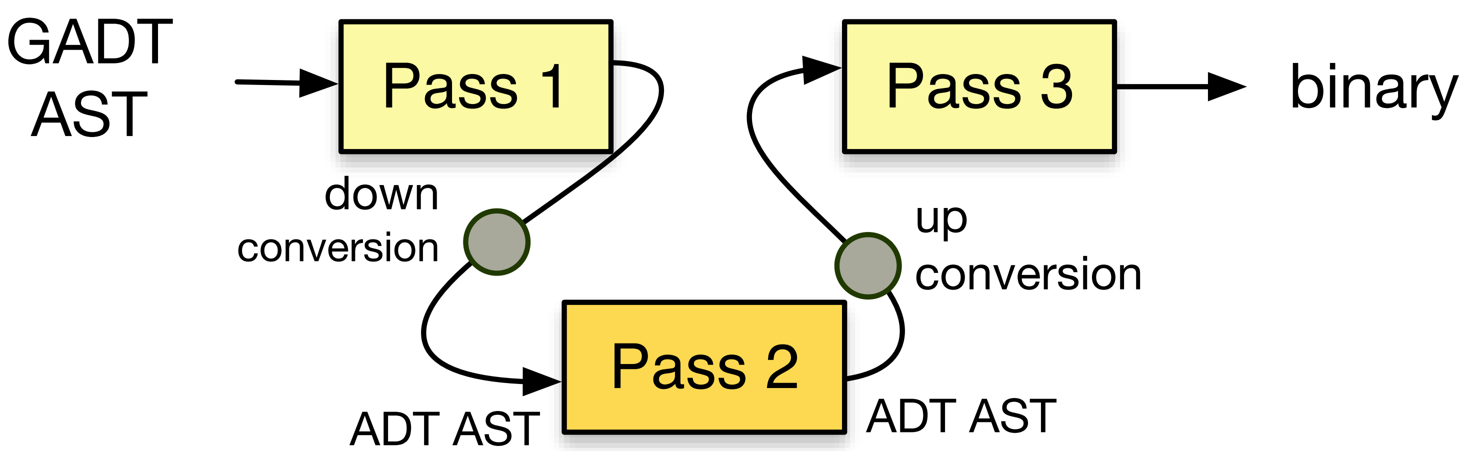
SYB techniques

- Haskell SYB libraries
  - type directed
- Poor-man's:
  - (gtraverse *tree fn combine*)
  - (*fn exp fallthru*)   handle it, or…
  - (*fallthru exp*)
- Even the latter gets the asymptotic benefits
  - nanopass codegen can create gtraverse easily

Optimizations on GADT ASTs

- See mini-accelerate exercises

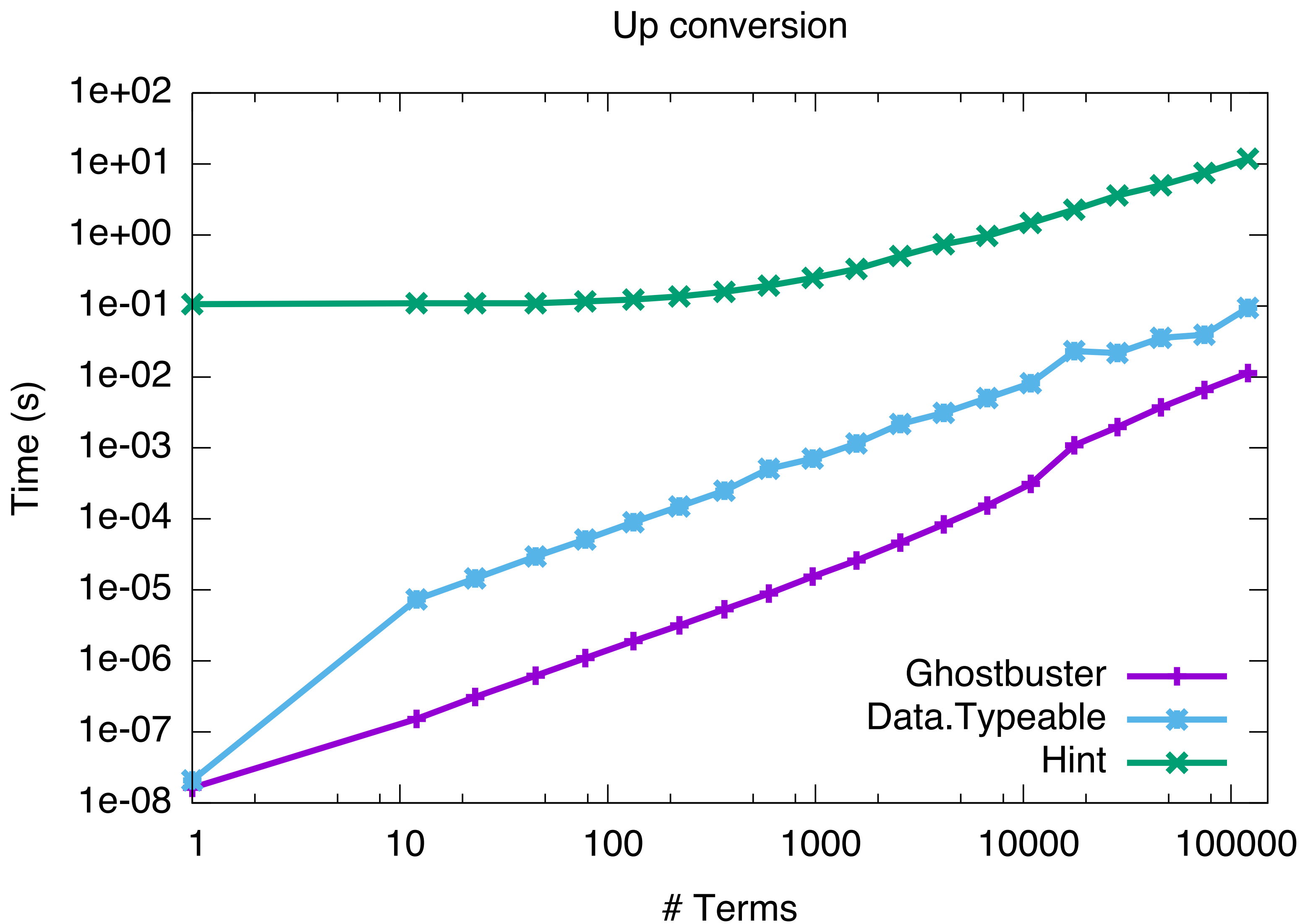
Multi-representation + conversions



Code walkthrough

- See ./middle_end/multi-level_AST

INDIANA UNIVERSITY



Haskell features for DSL Construction

Front-end

* Template Haskell: typed and untyped splices
* Rebindable syntax
* Type-safe observable sharing
* Alternate “Prelude”s
* Type classes + overloaded literals

Middle-end

Compiler construction technologies

* Scrap-your-boilerplate (SYB)
* Syntactic library
* GADT ASTs for type preservation
* Nanopass tooling
* Finally-tagless abstract syntax

Back-end

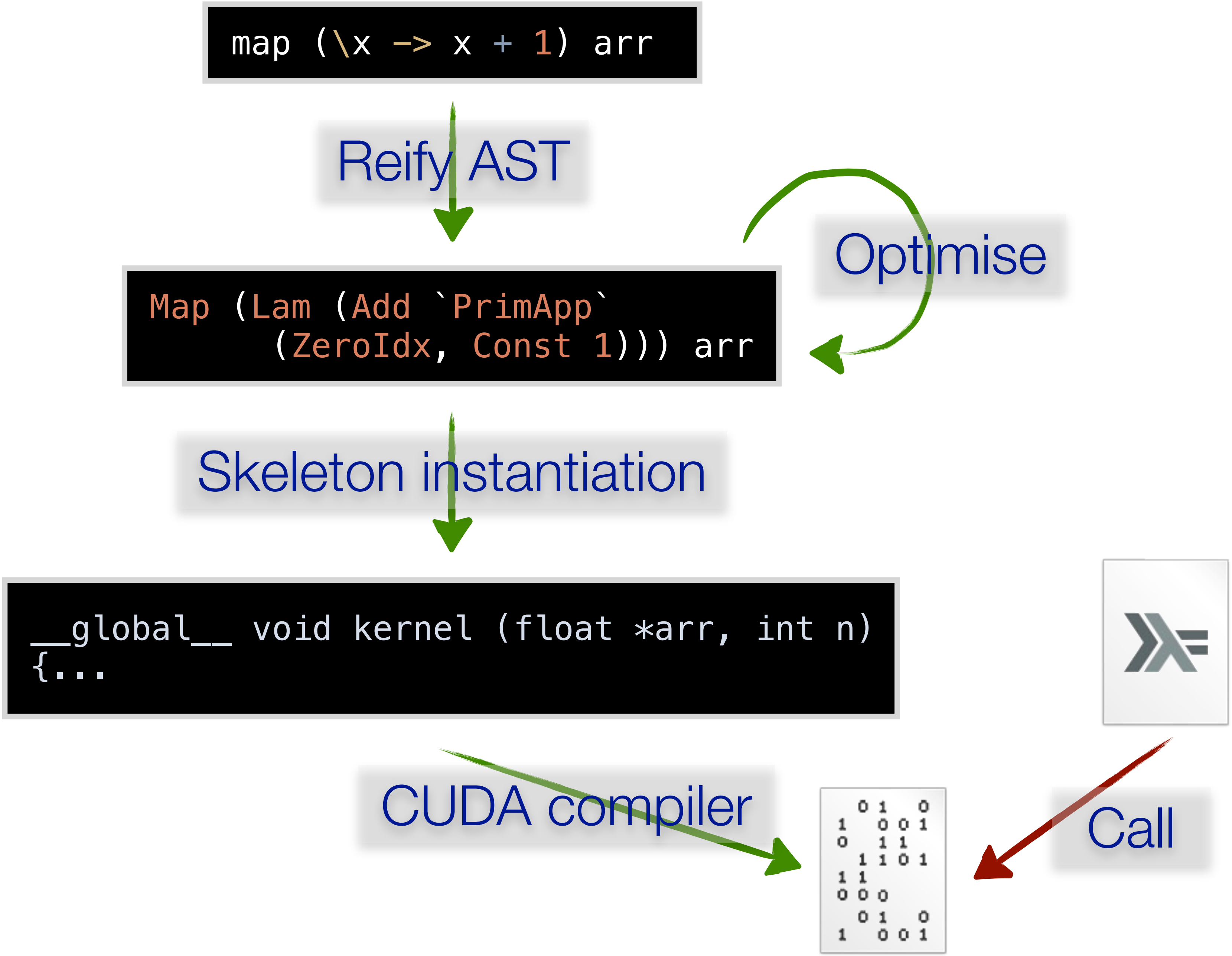
* Quasiquoters: foreign syntax blocks
* Type-safe backends (e.g. LLVM)
* Finally-tagless for mixed shallow/deep embedded exec.

Back-end

INDIANA UNIVERSITY

Quasi-quotation for C generation

- language-c-quote package
  - Actually C, CUDA, OpenCL support


map (\x -> x + 1) arr
Reify AST
Map (Lam (Add `PrimApp`
(ZeroIdx, Const 1))) arr
Optimise
Skeleton instantiation
__global__ void kernel (float *arr, int n)
{...
CUDA compiler
Call
0 1 0
1 0 0 1
0 1 1
1 1 0 1
1 1
0 0 0
0 1 0
1 0 0 1

```
mkMap dev aenv fun arr = return $
  CUTranslSkel "map" [cunit|

  $esc:("#include <accelerate_cuda.h>")
  extern "C" __global__ void
  map ($params:argIn, $params:argOut) {
    const int shapeSize = size(shOut);
    const int gridSize  = $exp:(gridSize dev);
    int ix;

    for ( ix =  $exp:(threadIdx dev)
        ; ix <  shapeSize
        ; ix += gridSize ) {
          $items:(dce x       .=. get ix)
          $items:(setOut "ix" .=. f x)
        }
  } |]
  where ...
```

Type-safe LLVM bindings

- Haskell'15:

Type-safe Runtime Code Generation: Accelerate to LLVM

Trevor L. McDonell[1] Manuel M. T. Chakravarty[2] Vinod Grover[3] Ryan R. Newton[1]

[1]Indiana University Bloomington [2]University of New South Wales [3]NVIDIA Corporation

- Preserves "Exp Int" all the way to LLVM IR

Type-preserving LLVM bkend

```
data Instruction a where
  Add :: NumType a        -- reified dictionary
      → Operand a
      → Operand a
      → Instruction a
```

```
data family Operands :: *
data instance Operands () = OP_Unit
data instance Operands Int = OP_Int (Operand Int)
data instance Operands Int8 = OP_Int8 (Operand Int8)
  ...
data instance Operands (a,b)
  = OP_Pair (Operands a) (Operands b)
```

Closing note: not just codegen, runtime too