Generalised algebraic data types for type-safe language embeddings

Christoph Herrmann
Univ. St Andrews, Scotland

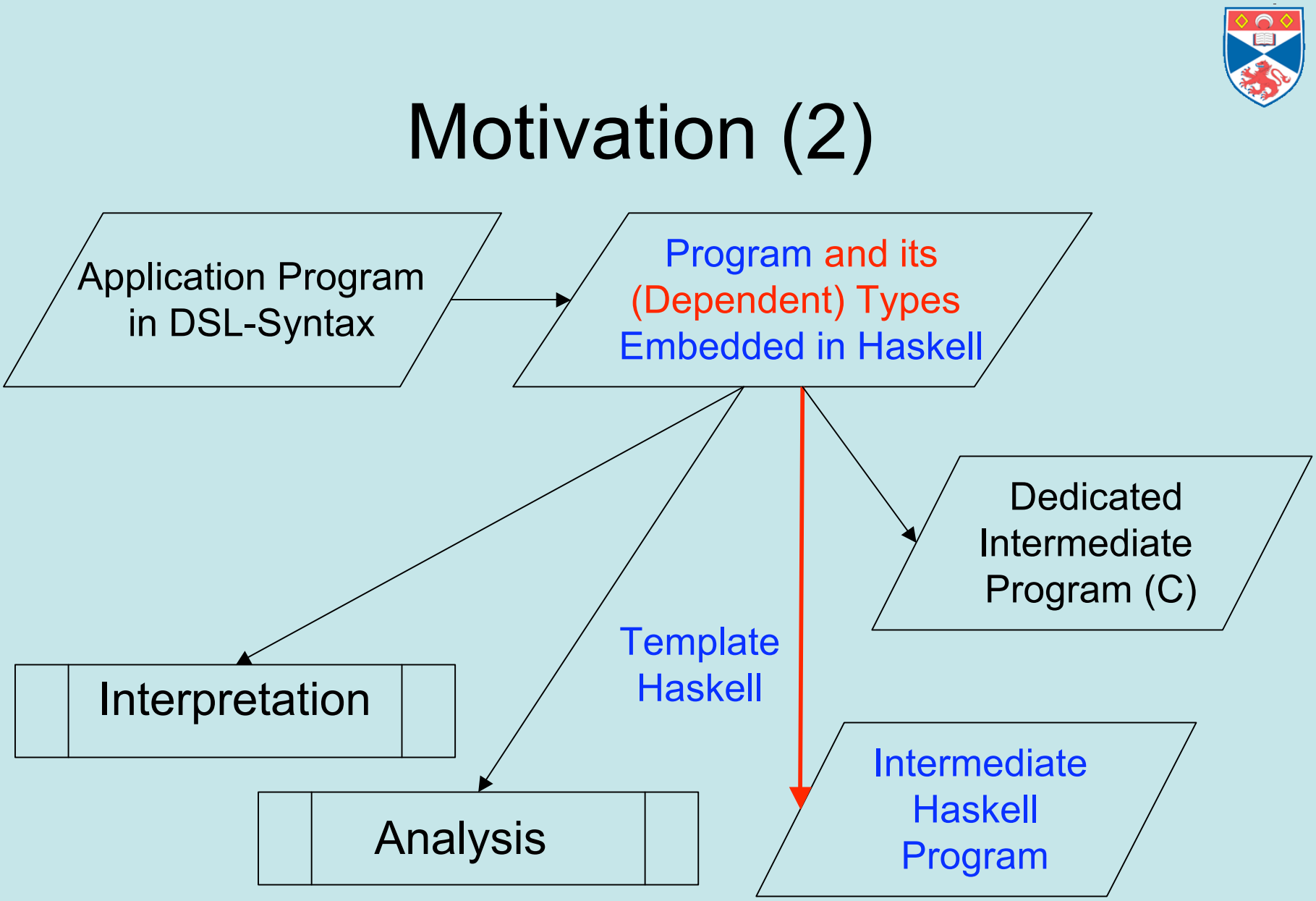
IFIP WG-2.11 Meeting
Dragør, 20 Aug 2007

Overview

- Motivation
- Language embedding in Haskell
  - Adding type-safety with GADTs
  - Staging with Template Haskell
- Special examples
  - Type-safe stack operations
  - Type classes for arithmetic constraints
- Related work
- Conclusions

Motivation (1)

- Language embedding
  - Rapid prototyping
- Enforcing types by the host language
  - Little implementation effort
  - Proof comes for free
- Generalised algebraic data types (GADTs)
  - Extend syntax with types
  - Permit simple dependent types
  - Available in Haskell



Motivation (2)

Application Program in DSL-Syntax

Program and its (Dependent) Types Embedded in Haskell

Dedicated Intermediate Program (C)

Template Haskell

Interpretation

Analysis

Intermediate Haskell Program

Language Embedding in Haskell (1)

- Represent grammar of embedded language by an algebraic data type
- Define all syntax-directed actions on the data constructors (often inductively)
  - Interpretation
    - Run-time values
    - Abstract interpretation
  - Compilation
    - Staged with Template Haskell, target language: Haskell
    - Other targets, especially C or bytecode representations

Language Embedding in Haskell (2)

```
data Expr =
   C Int                -- integer constant, e.g., C 5
 | T                    -- Boolean constant true
 | Add Expr Expr        -- Expr + Expr
 | Eq  Expr Expr        -- Expr = Expr
 | If  Expr Expr Expr -- if Expr then Expr else Expr

interpret (Add x y)  = interpret x + interpret y
interpret (If x y z) = if interpret x then interpret y
                                      else interpret z

compile (C x)     = emit (“push “ ++ show x)
compile (Add x y) =    compile x ++ “;”
                    ++ compile y ++ “;”
                    ++ emit “add”
```

Type-safety with GADTs (1)

- Problem with above representation
  - Previous representation would permit `Eq T (C 1)`
  - Potential consequences
    - Crash or erroneous results in interpretation or execution of compiled code
    - Optimization can propagate error, makes it hard to find

      e.g., `Eq T (C 1)` → `C 1`
- Need to discriminate expressions

```
EqInt :: Expr Int -> Expr Int -> Expr Bool
```

Type-safety with GADTs (2)

- Normal algebraic data types
  - E.g., `data Expr a = … | C 1 | F | Expr a + Expr a`
  - You cannot restrict `a` individually for particular constructors
- Generalised algebraic data types, example:

```
data Expr :: * -> * where
    C   :: Int -> Expr Int
    T   :: Expr Bool
    Add :: Expr Int -> Expr Int -> Expr Int
    Eq  :: Expr a -> Expr a -> Expr Bool
    If  :: Expr Bool  -> Expr a -> Expr a -> Expr a
```

Staging with Template Haskell

- (Verified) staged interpretation = (Verified) Compilation
- Change interpreter from dealing with values to compose code that computes these values
- TH-examples:

```
c (Eq x y)   = [| $(c x) == $(c y) |]
c (If x y z) = [| if $(c x) then $(c y)
                  else $(c z) |]
```

- No tags in the generated code
- Types in the source transfer to target program, in contrast to string emission:

```
c (Eq x y) = c x ++ ";" ++ c y ++ ";" ++ emit "EQ"
```

Type-safe stack operations (1)

Goal: enforce that top stack elements have the appropriate types for the operation, relying on the Haskell type checker

```
data (:$)  -- the type of the empty stack
data p:<s  -- top element has type p, rest stack type s

data Stack :: * -> * where
  Empty :: Stack (:$)
  Push  :: a -> Stack xs -> Stack (a<: xs)
  Pop   :: Stack (a :< xs) -> Stack xs
  UOp   :: (a->b) -> Stack (a :< xs)
                  -> Stack (b :< xs)
  BOp   :: (a->b->c) -> Stack (a :< (b :< xs))
                     -> Stack       (c :< xs)
```

Type-safe stack operations (2)

```
*TB> let x = Push (1::Int) (Push (2::Float) Empty)
*TB> :t x
x :: Stack (<: Int (<: Float (:$)))

*TB> let y = BOp (+) x
<interactive>:1:16:
    Couldn't match `Int' against `Float'
      Expected type: Stack ((:<) Int ((:<) Int xs))
      Inferred type: Stack ((:<) Int ((:<) Float (:$)))

*TB> let z = (Push (6::Int) x)
*TB> :t z
z :: Stack (<: Int (<: Int (<: Float (:$))))
*TB> :t BOp (+) z
BOp (+) z :: Stack (:< Int (<: Float (:$)))
```

Type classes for constraints (1)

- Haskell's type class system provides Horn clause rules to specify type class instances
- Type classes can be used to exclude instances of algebraic data types for which membership in the type class cannot be derived
- Without type classes these rules have to be encoded in the program:
  - Application program parts have to carry an explicit proof (using GADT instances) that their function is type correct
  - Additional auxiliary arguments of functions
- In ghc-6.6.1 type classes cannot be used to help the type inference find the right instance, they can only kill instances
- Planned for ghc-6.8: rules specified for type classes to be added to the type inference system

Type classes for constraints (2)

```
data Zero = Zero
data Nat x => Succ x = Succ x

class Nat a where -- with instance declarations for Zero and Succ

class Less a b where
instance Less Zero (Succ x) where
instance (Less x y) => Less (Succ x) (Succ y) where

-- GADT:
…
Read :: Less index size =>              -- enforce that index<size
        Mem a size -> Adr index ->      -- then addressing permitted
        Stack xs -> Stack (a :< xs)     -- and value read put on stack
```

Related work (Systems)

- Epigram: McBride & McKinna
  - Type-level functions
- Concoqtion: Taha et al.
  - Based on MetaOCaml and the Coq proof assistant

Related work (Papers)

- Gibbons, Wang and Oliveira: “Generic and Indexed Programming”, *Trends in Functional Programming*, 2007.
- Perry, Mackey, Reis, Ligatti, August and Walker: “Fault-tolerant Typed Assembly Language”, *PLDI* 2007.
- McKinna and Wright: “A type-correct, stack-safe, provably correct expression compiler in EPIGRAM”, to appear in *Jornal of Functional Programming*.
- Brady, Hammond: “A verified staged interpreter is a verified compiler”, *Proc. 5th Int. Conf. Generative Programming and Component Engineering*, ACM Press, 2006.
- Brady, Hammond and McKinna: “Embedding a Language with Certified Size Constraints in a Dependently Typed Metalanguage”, draft, accessible on `http://www.cs.standrews.ac.uk/~eb/drafts/resourcelang.pdf`

Conclusions

- Reuse of host language types for the embedded language
- GADT constructors encode state transitions
  - works for stack machines like for finite automata
- Type connection between generator and generated program
  - known in an extreme form in MetaOCaml
  - here in relaxed form: still permits object program types depend on compile-time values
- Supports simple form of dependent types
  - Example: arithmetic constraints