Embedded Domain Specific Languages by Syntax Overloading

WG2.11, Halmstad, 25th June 2012

ecb10@st-andrews.ac.uk

University of St Andrews

Edwin Brady

Introduction

This talk is about IDRIS, a programming language with *dependent types*.

- `cabal update; cabal install idris`
- `http://idris-lang.org/`
- `http://idris-lang.org/documentation/`

In particular, we will see *by example* how to use IDRIS to build *Embedded Domain Specific Languages*. Code is at:

`http://idris-lang.org/documentation/edsl-talk`

The IDRIS Programming Language

IDRIS is a general purpose pure functional programming language, with support for theorem proving. Features include:

- Full *dependent types*, dependent pattern matching
- Dependent *records*
- *Type classes* (Haskell style)
  - Numeric overloading, Monads, `do`-notation, idiom brackets, . . .
- *Tactic* based theorem proving
- High level constructs: `where`, `case`, `with`, monad comprehensions, syntax overloading
- *Totality* checking, cumulative universes
- Interfaces for *systems* programming (e.g. C libraries)

Dependent Types in IDRIS

IDRIS syntax is influenced by Haskell. Some data types:

```
data Nat = O | S Nat

infixr 5 ::  -- Define an infix operator
data Vect : Set -> Nat -> Set where  -- List with size
     Nil  : Vect a 0
     (::) : a -> Vect a k -> Vect a (1 + k)
```

The type of a function over vectors describes invariants of the input/output lengths, e.g..

```
append : Vect a n -> Vect a m -> Vect a (n + m)
append Nil       ys = ys
append (x :: xs) ys = x :: append xs ys
```

Dependent Types in IDRIS

We can use Haskell style type classes to constrain polymorphic functions, e.g., pairwise addition a vectors of numbers:

```
total
vAdd : Num a => Vect a n -> Vect a n -> Vect a n
vAdd Nil       Nil       = Nil
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys
```

(Aside: The `total` keyword means that it is an error if the totality checker cannot determine that the function is total)

Demonstration

Conclusions

IDRIS is a language specifically designed for EDSL implementation:

- do-notation, idiom brackets
- Syntax overloading

We have seen how to use dependent types to build EDSLs with expressive type systems

- EDSL captures *state* properties of systems programs
- Syntax overloading for ease of application development
  - Provide an *abstraction* to hide the details of the type system

Future Work

What domains might benefit from this approach?

- Security protocols?
- Type-safe, secure web programming?

Software engineering concerns

- Compositionality and scalability
- Modularity and reuse
- Theorem proving is often necessary: domain-specific decision procedures?
- Rich type systems are highly suited to *interactive* development — what tool support can we provide?

Compiler efficiency

- Currently compiles via C — investigate e.g. LLVM. . .