Dependent Types For DSLs

Edwin Brady

eb@cs.st-andrews.ac.uk

University of St Andrews
DSL4EE, Gothenburg, 16th June 2011
Introduction

This talk is about a technique for Domain Specific Language implementation. It will cover:

1. An overview of functional programming with dependent types, using the language **IDRIS**.

2. *Embedded Domain Specific Language (EDSL)* implementation.
   - A type safe interpreter
   - *Verified* resource management using DSLs
     ♦ e.g. for networks, security, concurrency, . . .

3. *For discussion*: what other domains fit this approach?
Idris

**Idris** is an experimental purely functional language with dependent types ([http://idris-lang.org/](http://idris-lang.org/)).

- Compiled, via C, with some optimisations.
- Loosely based on Haskell, similarities with Agda, Epigram.
- Available from Hackage:
  - `cabal install idris`
  - Requires Boehm GC, `port install boehmgc`
- Tutorial notes online:
  - [http://idris-lang.org/tutorial](http://idris-lang.org/tutorial)
Idris

**Idris** is an experimental purely functional language with dependent types ([http://idris-lang.org/](http://idris-lang.org/)).

- Compiled, via C, with some optimisations.
- Loosely based on Haskell, similarities with Agda, Epigram.
- Available from Hackage:
  - `cabal install idris`
  - Requires Boehm GC, `port install boehmgc`
- Tutorial notes online:
  - [http://idris-lang.org/tutorial](http://idris-lang.org/tutorial)
- “Research quality software”
Some Idris Features

Idris has several features to help support EDSL implementation...

- Full-Spectrum Dependent Types
- Compile-time evaluation
- Efficient executable code, via C
- Unification (type/argument inference)
- Plugin decision procedures
- Overloadable do-notation, idiom brackets
- Simple foreign function interface

...and I try to be responsive to feature requests!
Dependent Types in Idris

Dependent types allow types to be parameterised by *values*, giving a more precise description of data. Some data types in Idris:

```idris
data Nat = O | S Nat;

infixr 5 :: ; -- Define an infix operator

data Vect : Set -> Nat -> Set where -- List with size
  VNil : Vect a O

  | (::) : a -> Vect a k -> Vect a (S k);
```

We say that *Vect* is *parameterised* by the element type and *indexed* by its length.
Functions

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

e.g. the type of \texttt{vAdd} expresses that the output length is the same as the input length:

\begin{verbatim}
vAdd : Vect Int n -> Vect Int n -> Vect Int n;
vAdd VNil VNil = VNil;
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys;
\end{verbatim}

The type checker works out the type of \texttt{n} implicitly, from the type of \texttt{Vect}.
Input and Output

I/O in Idris works in a similar way to Haskell. e.g. \texttt{readVec} reads user input and adds to an accumulator:

\texttt{readVec : Vect Int n -> IO ( p ** Vect Int p );}
\texttt{readVec xs = do \{ putStrLn "Number: ";}
\texttt{\hspace{1cm} val <- getInt;}
\texttt{\hspace{1cm} if \ val == -1 \ then \ return \ <| \ _ , \ xs \ | >}
\texttt{\hspace{1cm} \else (readVec (val :: xs));}
\texttt{\}};

The program returns a \textit{dependent pair}, which pairs a \textit{value} with a \textit{predicate} on that value.
Libraries

Libraries can be imported via `include "lib.idr"`. All programs automatically import `prelude.idr` which includes, among other things:

- **Primitive types** `Int`, `String`, `Float` and `Char`, plus `Nat`, `Bool`
- Tuples, dependent pairs.
- `Fin`, the finite sets.
- `List`, `Vect` and related functions.
- `Maybe` and `Either`
- The `IO` monad, and foreign function interface.
A Type Safe Interpreter

A common introductory example to dependent types is the type safe interpreter. The pattern is:

- Define a data type which represents the language and its typing rules.
- Write an interpreter function which evaluates this data type directly.

[demo: interp.idr]

[code available at http://idris-lang.org/examples/dsl4ee.tgz]
A Type Safe Interpreter

Notice that when we run the interpreter on functions without arguments, we get a translation into Idris:

```
Idris> interp Empty test
\ x : Int . \ x0 : Int . x + x0
Idris> interp Empty double
\ x : Int . x+x
```
A Type Safe Interpreter

We have *partially evaluated* these programs. If we can do this reliably, and have reasonable control over, e.g., inlining, then we have a recipe for *efficient* verified EDSL implementation:

1. Design an EDSL which guarantees the resource constraints, represented as a dependent type
2. Implement the interpreter for that EDSL
3. Specialise the interpreter for concrete EDSL programs, using a partial evaluator
Resource Usage Verification

We have applied the type safe interpreter approach to a family of domain specific languages with resource usage properties, in their type:

- File handling
- Memory usage
- Concurrency (locks)
- Network protocol state

I will outline a generic framework for the construction of resource aware DSLs
Resource Aware DSLs

Our aim is to define a language for tracking resource usage \textit{statically}. It will take the following form, a data type parameterised over a start and end state:

\begin{verbatim}
data RLang : Set -> ResState -> ResState -> Set where ...
\end{verbatim}

An interpreter, given an environment of resources, runs a program which updates the environment:

\begin{verbatim}
rinterp : \{s,s\':ResState\} -> ResEnv s -> RLang a s s' -> IO (a & s);
\end{verbatim}
Our concern is whether a resource is *valid* at a given time. We define resource types, and include a *time slice* in the state:

```haskell
data ResTy = RTy Set;
ResState n = (Nat & Vect ResTy n);

rty : ResTy -> Set;
```

We parameterise *resources* over the time they are valid, and their location in a resource list:

```haskell
data Resource : Nat -> Fin n -> ResTy -> Set where
    Res : {i:Fin n} -> rty a -> Resource t i a;
```
Resource environments

An environment contains concrete resource values (compare to the well-typed interpreter earlier)

data ResEnv : Vect ResTy n -> Set where
    Empty   : ResEnv VNil
    | Extend  : rty r -> ResEnv xs -> ResEnv (r :: xs);
Resource IO monad

We can now define a \textit{resource state} monad, parameterised over the current state.

\begin{verbatim}
data ResIO : Set -> ResState n -> ResState n -> Set where
    ResIOP : (ResEnv (snd s) -> IO (a & ResEnv (snd s'))) -> ResIO a s s';

BIND : ResIO t s s' -> (t -> ResIO u s' s'') -> ResIO u s s'';
RETURN : a -> ResIO a s s;
\end{verbatim}

Operations in this monad give a \textit{DSL} for managing resources in general.
Resource IO operations

For example, as in Haskell’s State monad we may need to GET and PUT state:

GET : (i:Fin n) ->
    ResIO (Resource (fst s) i (vlookup i (snd s))) s s;

PUT : {i:Fin n} ->
    Resource (fst s) i (RTy a) -> rty b ->
    ResIO () s (Later s i b);

GET gives a value valid in the current time slice. PUT updates the time slice, using Later, which increments the time slice portion of the state.
Resource IO operations

We can **USE** a value stored in a resource, provided the resource is valid in the current time slice:

```
USE : {i:Fin n} ->
    (rty a -> IOr b) -> Resource (fst s) i a -> ResIO b s s;
```

While the types of **GET**, **PUT** and **USE** may look complex (to ensure that resources are used only when valid) using them in a realistic example is more straightforward.

[**demo:** safe-file.idr]
Conclusions

We have seen how I
DRIS can be used to implement type-safe languages, with I
DRIS’s type system enforcing the type safety of the object language.

■ Resource safety in particular is an important problem

This is not unique to I
DRIS!

■ Techniques equally applicable to Agda, Coq, Guru, Trellys, Haskell (with GADTs)…
Lots of interesting (resource related) problems fit into the EDSL framework:

■ Concurrency, time/space usage, security, power consumption, AI/planning . . .

These are all problems in Computer Science (because that’s what I know!)

■ Where else might resource aware DSLs and dependent types in general fit?
Related Work

- “Parameterised Notions of Computation”
  — Robert Atkey,
  In MSFP 2006

- “The Power of Pi”
  — N. Oury and W. Swierstra,
  In ICFP 2008

- “Security Typed Programming Within Dependently Typed Programming”
  — J. Morgenstern and D. Licata,
  In ICFP 2010
Further Reading

■ “Scrapping your Inefficient Engine: using Partial Evaluation to Improve Domain-Specific Language Implementation”
  — E. Brady and K. Hammond,
  In ICFP 2010.

■ “Domain Specific Languages (DSLs) for Network Protocols”
  — S. Bhatti, E. Brady, K. Hammond and J. McKinna,
  In Next Generation Network Architecture 2009.

■ “IDRIS — Systems Programming meets Full Dependent Types”
  — E. Brady, In PLPV 2011.


■ http://idris-lang.org/tutorial/