Embedded Domain Specific Language Implementation using Dependent Types

Edwin Brady

eb@cs.st-andrews.ac.uk

University of St Andrews

GPCE/SLE, Eindhoven, 10/10/10

Introduction

This tutorial is in two parts. 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
   - Network protocols as EDSLs
   - Code generation via specialisation
   - Performance data

Idris

IDRIS is an experimental purely functional language with dependent types (`http://idris-lang.org/`).

- Compiled, via C, with some optimisations.
- Loosely based on Haskell, similarities with Agda, Epigram.
- Available from Hackage:
  - `cabal install idris`
- Tutorial notes online:
  - `http://idris-lang.org/tutorial`

Idris

IDRIS is an experimental purely functional language with dependent types (`http://idris-lang.org/`).

- Compiled, via C, with some optimisations.
- Loosely based on Haskell, similarities with Agda, Epigram.
- Available from Hackage:
  - `cabal install idris`
- Tutorial notes online:
  - `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:

```
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 `vAdd` expresses that the output length is the same as the input length:

```
vAdd : Vect Int n -> Vect Int n -> Vect Int n;
vAdd VNil VNil = VNil;
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys;
```

The type checker works out the type of `n` implicitly, from the type of `Vect`.

Input and Output

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

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

The program returns a *dependent pair*, which pairs a *value* with a *predicate* on that value.

The `with` Rule

The `with` rule allows dependent pattern matching on intermediate values:

```
vfilter : (a -> Bool) -> Vect a n -> (p ** Vect a p);
vfilter f VNil = <| _, VNil |>;
vfilter f (x :: xs) with (f x, vfilter xs f) {
   | (True, <| _, xs' |>) = <| _, x :: xs' |>;
   | (False, <| _, xs' |>) = <| _, xs' |>;
}
```

The underscore _ means either match anything (on the left of a clause) or infer a value (on the right).

Libraries

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

- Primitive types `Int`, `String` 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`]

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

As an example, I will outline the construction of a DSL for a simple network transport protocol.

Example — Network Protocols

Protocol correctness can be verified by *model-checking* a finite-state machine. However:

- There may be a large number of states and transitions.
- The model is needed *in addition to* the implementation.

Model-checking is therefore not *self-contained*. It can verify a protocol, but not its *implementation*.

Example — Network Protocols

In our approach we construct a self-contained domain-specific framework in a dependently-typed language.

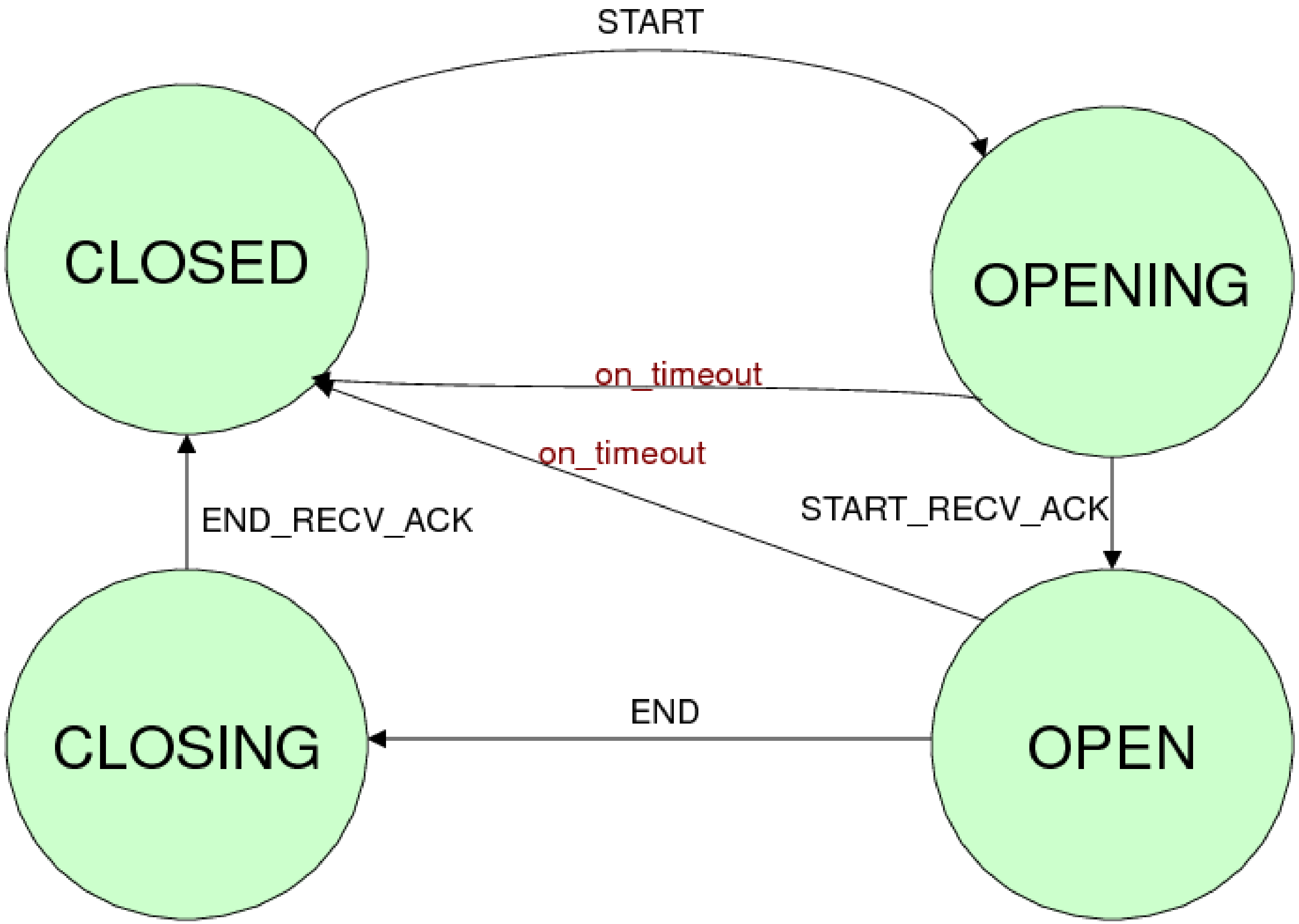
- We can express correctness properties *in the implementation itself*.
- We can express the *precise form of data* and *ensure it is validated*.
- We aim for *Correctness By Construction*.

ARQ

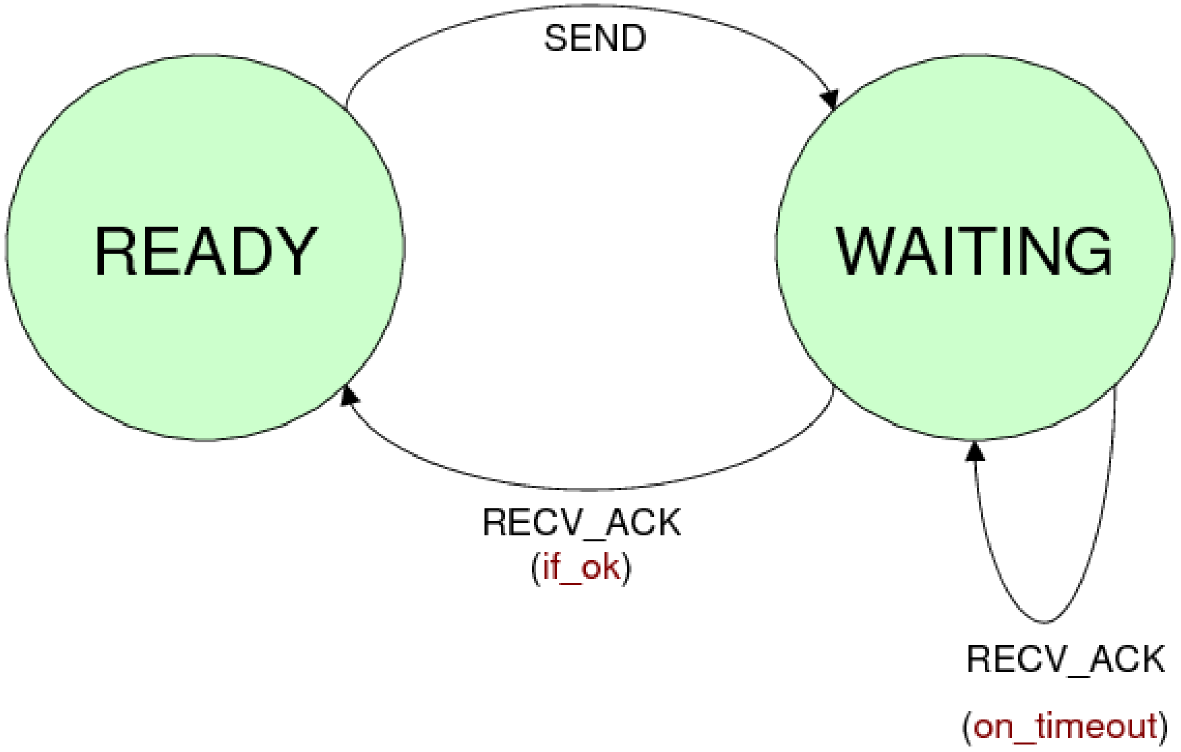
Our simple transport protocol:

- Automatic Repeat Request (ARQ)
- Separate *sender* and *receiver*
- State
    - *Session* state (status of connection)
    - *Transmission* state (status of transmitted data)

Session State



Transmission State



Session Management

- `START` — initiate a session
- `START_RECV_ACK`
  — wait for the receiver to be ready
- `END` — close a session
- `END_RECV_ACK`
  — wait for the receiver to close

Session Management

- `START` — initiate a session
- `START_RECV_ACK`
  — wait for the receiver to be ready
- `END` — close a session
- `END_RECV_ACK`
  — wait for the receiver to close

When are these operations valid? What is their effect on the state? How do we apply them correctly?

Session Management

We would like to express contraints on these operations, describing when they are valid, e.g.:

Command	Precondition	Postcondition
`START`	`CLOSED`	`OPENING`
`START_RECV_ACK`	`OPENING`	`OPEN` (if `ACK` received)
		`OPENING` (if nothing received)
`END`	`OPEN`	`CLOSING`
`END_RECV_ACK`	`CLOSING`	`CLOSED` (if `ACK` received)
		`CLOSED` (if nothing received)

Sessions, Dependently Typed

How do we express our session state machine?

- Make each transition an operation in a DSL.
- Define the *abstract syntax* of the DSL language as a dependent type.
- Implement an *interpreter* for the abstract syntax.
- *Specialise* the interpreter for the ARQ implementation.

This is the recipe we followed for the well typed interpreter . . .

Session State, Formally

`State` carries the session state, i.e. states in the Finite State Machine, plus additional data:

```
data State = CLOSED
           | OPEN TState -- transmission state
           | CLOSING
           | OPENING
```

`TState` carries the transmission state. An open connection is either waiting for an `ACK` or ready to send the next packet.

```
data TState = Waiting Seq      -- seq. no.
            | Ready Seq        -- seq. no.
```

Network Protocol EDSL

```
data ARQ : State -> State -> Set -> Set where
   START :  ARQ CLOSED OPENING ()
 | START_RECV_ACK
    : (if_ok       : ARQ (OPEN (Ready First)) st’ t) ->
      (on_timeout : ARQ OPENING st’ t) ->
            ARQ OPENING st’ t
 | END   :  ARQ (OPEN (Ready n)) CLOSING ()
 | END_RETRY
         : ARQ CLOSING CLOSING ()
 | END_RECV_ACK
    : (if_ok:       ARQ CLOSED st’ t) ->
      (on_timeout: ARQ CLOSING st’ t) ->
            ARQ CLOSING st’ t
   ...
```

Network Protocol EDSL

```
data ARQ : State -> State -> Set -> Set where
   ...
 | WITHIN :  Time -> (action     : ARQ st st' t) ->
                     (on_timeout : ARQ st st' t) ->
                      ARQ st st' t
 | IF     :  Bool -> (if_true  : ARQ st st' t) ->
                     (if_false : ARQ st st' t) ->
                      ARQ st st' t
 | RETURN :  t -> ARQ st st t
 | BIND   :  ARQ st st' t ->
            (k : t -> ARQ st' st'' t') ->
             ARQ st st'' t';
```

Network Protocol EDSL Interpreter

The interpreter for `ARQ` is parameterised over the actual network data, and keeps track of time to check for timeouts.

```
params (s:Socket, host:String, port:Int) {
   interpBy : Time -> (prog:ARQ st st’ t) [static] ->
              IO (Maybe t);
   ...
   interpBy t END
        = checkTime t (sendPacket s host port (CTL S_BYE));
   ...
}

checkTime : Time -> IO t -> IO (Maybe t);
```

Sending Packets

An example program, which opens a connection, sends a batch of packets, then closes it, within `i` microseconds:

```
sendNumber : Time -> Nat -> ARQ CLOSED CLOSED ();
sendNumber i tot
    = WITHIN i
        (do { open_connection 500000;
              session 500000 0 tot First;
              close_connection 500000;
        (TRACE "Timed out");
```

The types ensure that the protocol is followed; any protocol violation is a *type error*.

Sending Packets

The following function sends `tot` packets, with no payload, with timeout `i` per packet.

```
session : Time -> Nat -> Nat -> (sq:Seq) ->
          ARQ (OPEN (Ready sq)) CLOSING ();
session i n tot sq =
      IF (n == tot)
          END
         (do { TRACE ("Sending " ++ showNat n);
               send sq i;
               session i (S n) tot (Next sq); });
```

Sending Packets (Specialised)

Partial evaluation of the `ARQ` interpreter with this program yields:

```
sessionS : Socket -> String -> Int -> Time ->
           Time -> Nat -> Nat -> Seq -> IO (Maybe ());
sessionS s h p t i n tot sq = do {
  checkTime t (if (n == tot)
     then checkTime t (sendPacket s h p (CTL S_BYE))
     else do { putStr ("Sending " ++ showNat n);
               checkTime t (sendS s h p t sq i);
               checkTime t
                  (sessionS s h p t (S n) i tot (Next sq))); };
```

Results

We have implemented a number of examples using the DSL approach, and compared the performance of the interpreted and specialised versions with equivalent programs in C and Java.

- File handling
  - Copying a file
  - Processing file contents (e.g. reading, sorting, writing)
- Functional language implementation
  - Well-typed interpreter extended with lists

Results

Run time, in seconds of user time, for a variety of DSL programs:

Program	Spec	Gen	Java	C
`fact1`	0.017	8.598	0.081	0.007
`fact2`	1.650	877.2	1.937	0.653
`sumlist`	3.181	1148.0	4.413	0.346
`copy`	0.589	1.974	1.770	0.564
`copy_dynamic`	0.507	1.763	1.673	0.512
`copy_store`	1.705	7.650	3.324	1.159
`sort_file`	5.205	7.510	2.610	1.728
`ARQ`	0.751	0.990	—	—

Results

Run time, in seconds of user time, for a variety of DSL programs:

Program	Spec	Gen	Java	C
`fact1`	0.017	8.598	0.081	0.007
`fact2`	1.650	877.2	1.937	0.653
`sumlist`	3.181	1148.0	4.413	0.346
`copy`	0.589	1.974	1.770	0.564
`copy_dynamic`	0.507	1.763	1.673	0.512
`copy_store`	1.705	7.650	3.324	1.159
`sort_file`	5.205	7.510	2.610	1.728
`ARQ`	0.751	0.990	—	—

Results

Run time, in seconds of user time, for a variety of DSL programs:

Program	Spec	Gen	Java	C
`fact1`	0.017	8.598	0.081	0.007
`fact2`	1.650	877.2	1.937	0.653
`sumlist`	3.181	1148.0	4.413	0.346
`copy`	0.589	1.974	1.770	0.564
`copy_dynamic`	0.507	1.763	1.673	0.512
`copy_store`	1.705	7.650	3.324	1.159
`sort_file`	5.205	7.510	2.610	1.728
`ARQ`	0.751	0.990	—	—

Conclusions

IDRIS's type system occupies a "sweet spot" where partial evaluation is particularly effective.

- Tagless interpreters
- Existing evaluator; only minor changes required
- Comparable performance to *hand written* C/Java . . .
    - . . . but *verified* resource usage, via EDSLs

This is not unique to IDRIS!

- Techniques equally applicable to Agda, Coq, Guru, Trellys, Haskell (with GADTs). . .

Conclusions

Lots of interesting (resource related) problems fit into the EDSL framework:

- Concurrency (managing locks)
- Time/space usage
  - Important for *hard real-time systems*
- Power consumption
- AI/Planning (valid plan guaranteed to reach a goal)
- Security (managing access to resources)
- . . .

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, draft 2010.
- `http://www.cs.st-andrews.ac.uk/~eb/hacking/ARQdsl.html`
  — ARQ DSL implementation
- `http://idris-lang.org/tutorial/`