Implementing Domain Specific Languages using Dependent Types and Partial Evaluation

Edwin Brady

eb@cs.st-andrews.ac.uk

University of St Andrews

EE-PigWeek, January 7th 2010

Introduction

This talk is about *applications* of dependently typed programming. It will cover:

- Briefly, an overview of functional programming with dependent types, using the language *Idris*.
- *Domain Specific Language (DSL)* implementation.
  - A type safe interpreter
  - Code generation via specialisation
  - Network protocols as DSLs
  - Performance data

Idris

Idris is an experimental purely functional language with dependent types (`http://www.cs.st-and.ac.uk/~eb/Idris`).

- Compiled, via C, with reasonable performance (more on this later).
- Loosely based on Haskell, similarities with Agda, Epigram.
- Some features:
  - Primitive types (`Int`, `String`, `Char`, ...)
  - Interaction with the outside world via a C FFI.
  - Integration with a theorem prover, Ivor.

Why Idris?

Why Idris rather than Agda, Coq, Epigram, . . . ?

- Useful to have freedom to experiment with high level language features.
- I want to see what we can achieve in practice, so:
  - Need integration with the “outside world” — foreign functions, I/O.
  - Programs need to run sufficiently quickly.

Why Idris?

Why Idris rather than Agda, Coq, Epigram, . . . ?

- Useful to have freedom to experiment with high level language features.
- I want to see what we can achieve in practice, so:
  - Need integration with the "outside world" — foreign functions, I/O.
  - Programs need to run sufficiently quickly.
  - *(whisper: sometimes, in the short term, it's useful to cheat the type system)*

Why Idris?

Why Idris rather than Agda, Coq, Epigram, . . . ?

- Useful to have freedom to experiment with high level language features.
- I want to see what we can achieve in practice, so:
  - Need integration with the "outside world" — foreign functions, I/O.
  - Programs need to run sufficiently quickly.
  - *(whisper: sometimes, in the short term, it's useful to cheat the type system)*
- Making a programming language is fun. . .

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
```

Idris implements `%spec` and `%freeze` annotations which control the amount of evaluation at compile time.

[demo: `interp.idr` again]

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 good recipe for *efficient Domain Specific Language (DSL)* implementation:

- Define the language data type
- Write the interpreter
- Specialise the interpreter w.r.t. real programs

If we trust the host language's type checker and code generator — admittedly we still have to prove this, but only once! — then we can trust the DSL implementation.

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.

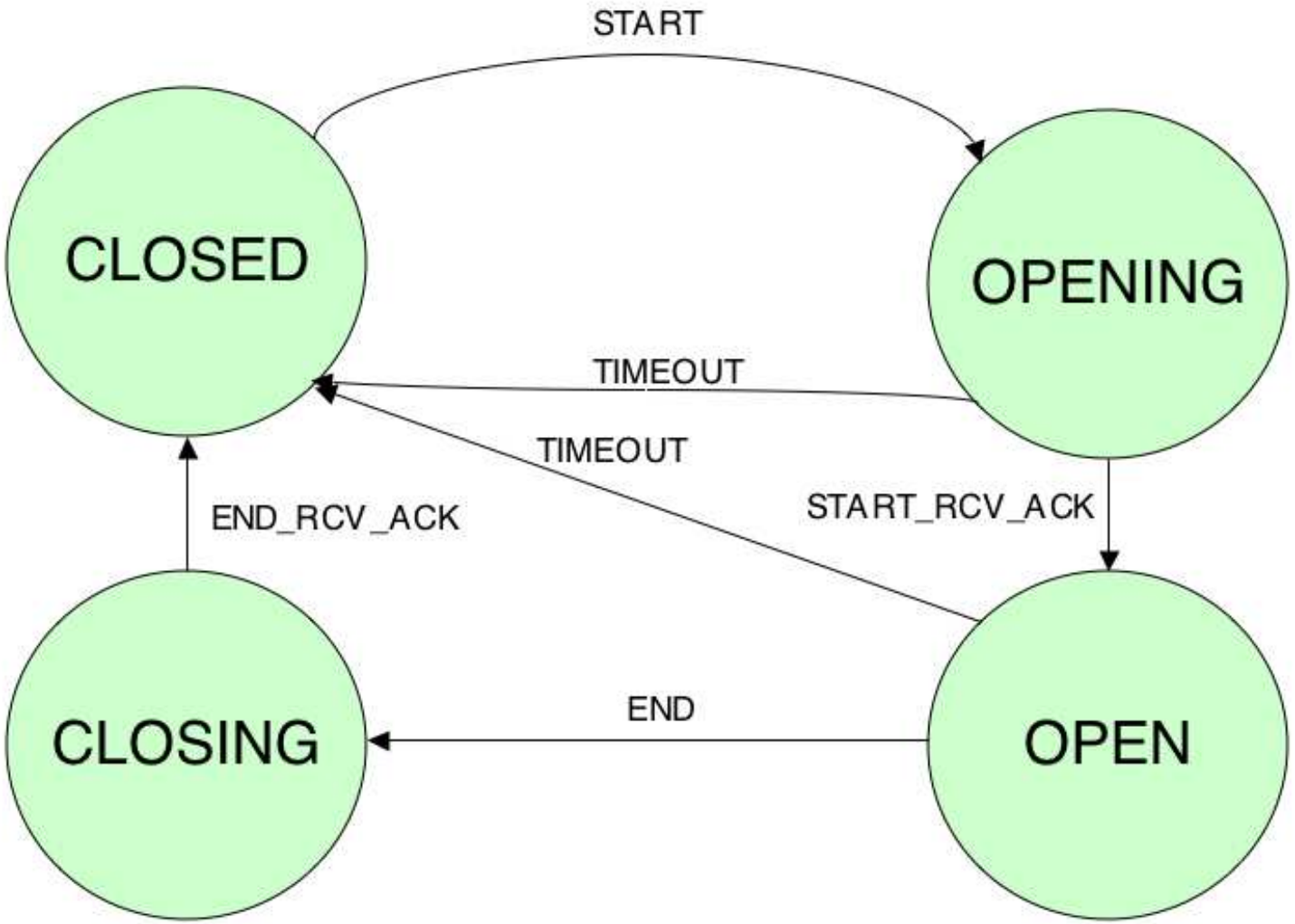
- We can express correctness properties *in the implementation itself*.
- We can express the *precise form of data* and *ensure it is validated*.
- We aim for *Correctness By Construction*.

ARQ

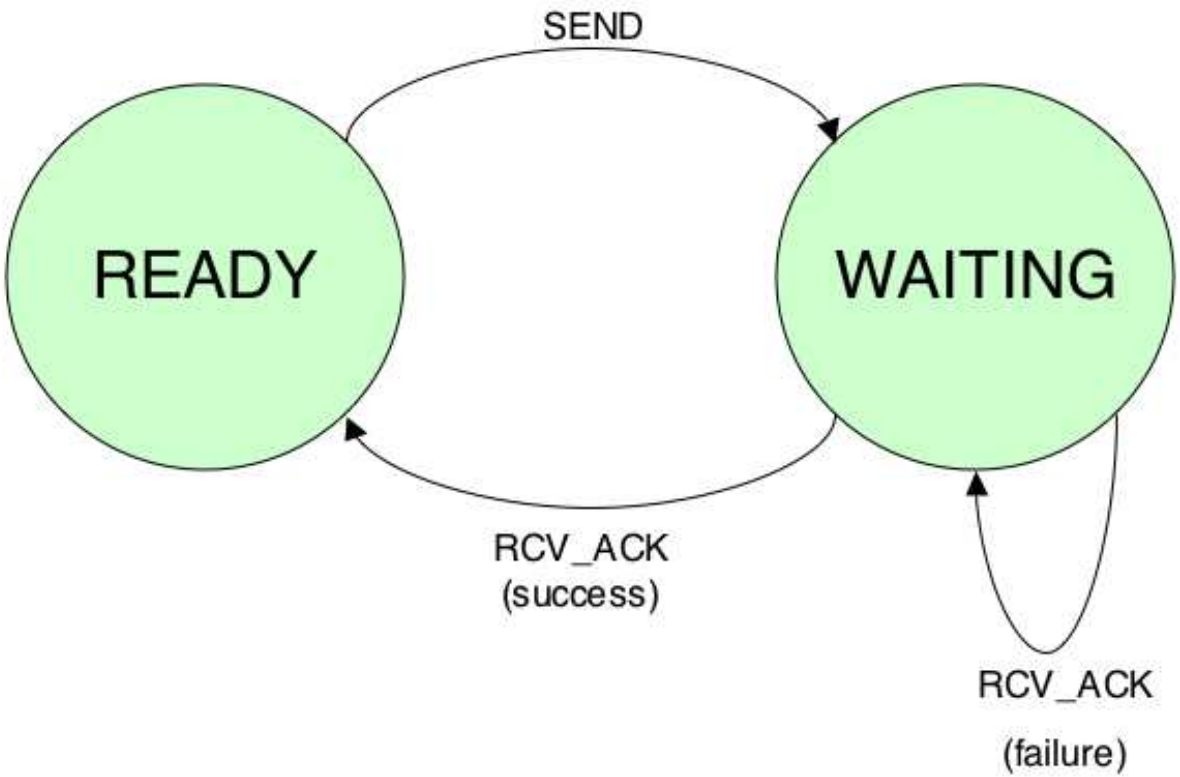
Our simple transport protocol:

- 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 PState -- transmission state
           | CLOSING
           | OPENING
```

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

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

Sessions, Formally

ARQLang is a data type defining the abstract syntax of our DSL, encoding state transitions in the type:

```
data ARQLang : State -> State -> Set -> Set where
   START : ARQLang CLOSED OPENING ()
 | START_RECV_ACK
     : (if_ok: ARQLang (OPEN (Ready First)) B Ty) ->
       (if_fail: ARQLang OPENING B Ty) ->
       (ARQLang OPENING B Ty)
   ...
```

[demo: ARQdsl.idr]

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`	0.149	0.240	—	—

Conclusion

Dependent types allow us to implement embedded DSLs *with rich specification/verification*. Also:

- We need an evaluator for type checking anyway, so why not use it for specialisation?
  - Related to MetaOCaml/Template Haskell, but free!
  - If (when?) we trust the Idris type checker and code generator, we can trust our DSL.
  - DSL programs will be as efficient as we can make Idris (i.e. no interpreter overhead).
- Lots of interesting (resource related) problems fit into this framework.

Further Reading

- "Scrapping your Inefficient Engine: using Partial Evaluation to Improve Domain-Specific Language Implementation"
  — E. Brady and K. Hammond,
  submitted 2009.
- "Domain Specific Languages (DSLs) for Network Protocols"
  — S. Bhatti, E. Brady, K. Hammond and J. McKinna,
  In Next Generation Network Architecture 2009.
- `http://www.cs.st-andrews.ac.uk/~eb/hacking/ARQdsl.html`
  — ARQ DSL implementation
- `http://www.cs.st-andrews.ac.uk/~eb/Idris`
- `http://www.cs.st-andrews.ac.uk/~eb/Idris/tutorial.html`