

# *IDRIS: Systems Programming with Dependent Types*

Edwin Brady

`eb@cs.st-andrews.ac.uk`

University of St Andrews

DTP 2011, August 27th 2011



# Introduction

---

A constant problem:

- Writing a **correct** computer program is hard
- **Proving** that a program is correct is even harder

**Dependent Types**, we claim, allow us to write programs and *know* they are correct before running them.

## Introduction

This talk is about building correct *systems software* by implementing *domain specific languages* using **IDRIS**, a dependently typed functional programming language.

- `cabal install idris`
- `http://www.idris-lang.org/`
- `http://www.idris-lang.org/tutorial`

## Part 1

# Domain Specific Languages for Correctness

## Resource Correctness — Preview

Our goal is to set things up so that programs such as the following are *guaranteed correct* (w.r.t. resource usage) because *type checking succeeds*:

```
dumpFile : String -> RES ();
dumpFile filename
  = res do { let h = open filename Reading;
              Check h
                (rputStrLn "File open error")
                (do { rreadH h;
                      rclose h;
                      rputStrLn "DONE" ; } ) ;
              } ;
```

## What is correctness?

---

- What does it mean to be “correct”?
  - Depends on the application domain, but could mean one or more of:

## What is correctness?

---

- What does it mean to be “correct”?
  - Depends on the application domain, but could mean one or more of:
    - **Functionally** correct  
(e.g. arithmetic operations on a CPU)

## What is correctness?

---

- What does it mean to be “correct”?
  - Depends on the application domain, but could mean one or more of:
    - **Functionally** correct  
(e.g. arithmetic operations on a CPU)
    - **Resource** safe  
(e.g. runs within memory bounds, no memory leaks, no accessing unallocated memory, no deadlock. . . )

## What is correctness?

---

- What does it mean to be “correct”?
  - Depends on the application domain, but could mean one or more of:
    - **Functionally** correct  
(e.g. arithmetic operations on a CPU)
    - **Resource** safe  
(e.g. runs within memory bounds, no memory leaks, no accessing unallocated memory, no deadlock. . . )
    - **Secure**  
(e.g. not allowing access to another user’s data)

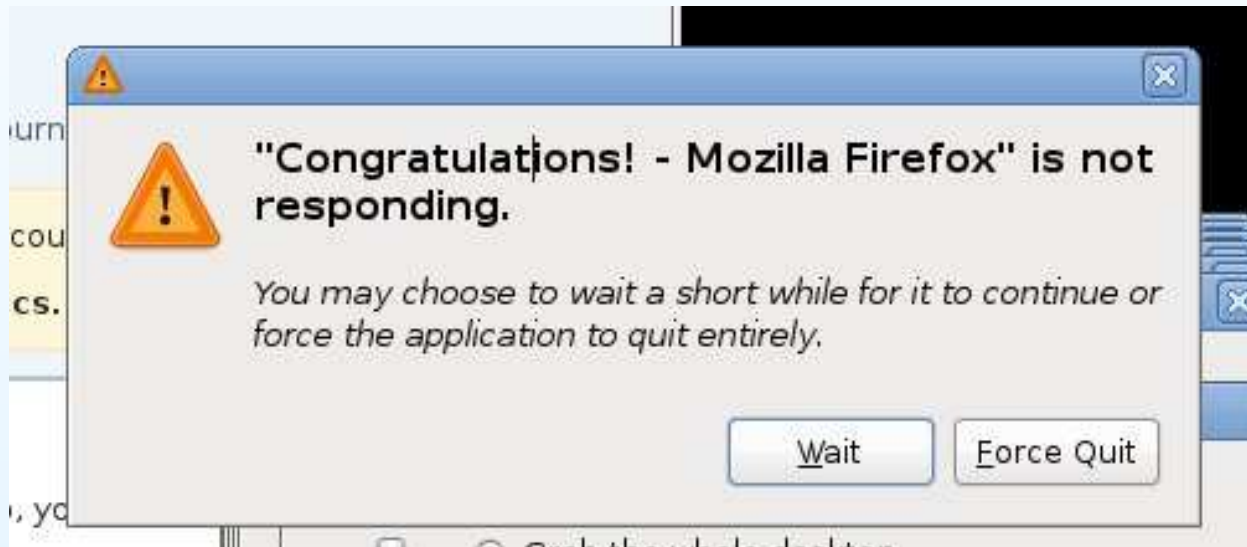
## Why do we care about correctness?

---

- On the desktop, we can, and usually do, tolerate software failures:

## Why do we care about correctness?

- On the desktop, we can, and usually do, tolerate software failures:



# Why do we care about correctness?

- On the desktop, we can, and usually do, tolerate software failures:



## Why do we care about correctness?

---

- However, software is everywhere, not just the desktop. In other contexts incorrect programs can be:
  - **Dangerous**
    - Control systems: aircraft, nuclear reactors, . . .
  - **Costly**
    - Intel Pentium bug (estimated \$475 million)
    - Ariane 5 failure (more than \$370 million)
  - **Inconvenient** on a large scale
    - February 2009 Gmail failure
    - Debian OpenSSL bug

## Correctness, with dependent types

---

We know that we can use dependent types to reason about correctness of functional programs. But...

- Real world programs are rarely pure
  - State, network communication, reading/writing files and other resources, spawn threads and processes...
- Systems may fail, data may be corrupted or untrusted
- Do systems programming experts need to be type theorists?

Proposed solution: Embedding *Domain Specific Languages* in a dependently typed *host language*

## Domain Specific Languages

---

- A *Domain Specific Language* (DSL) is a language designed for a particular problem domain.
  - User can focus on the *high level* of the domain, rather than the *low level* implementation details.
- Many Unix examples:
  - regular expressions, sed, awk, lex, yacc, sendmail.cf, procmail, bash, ...
- Databases, internet applications:
  - SQL, PHP, XPath, XQuery, ...

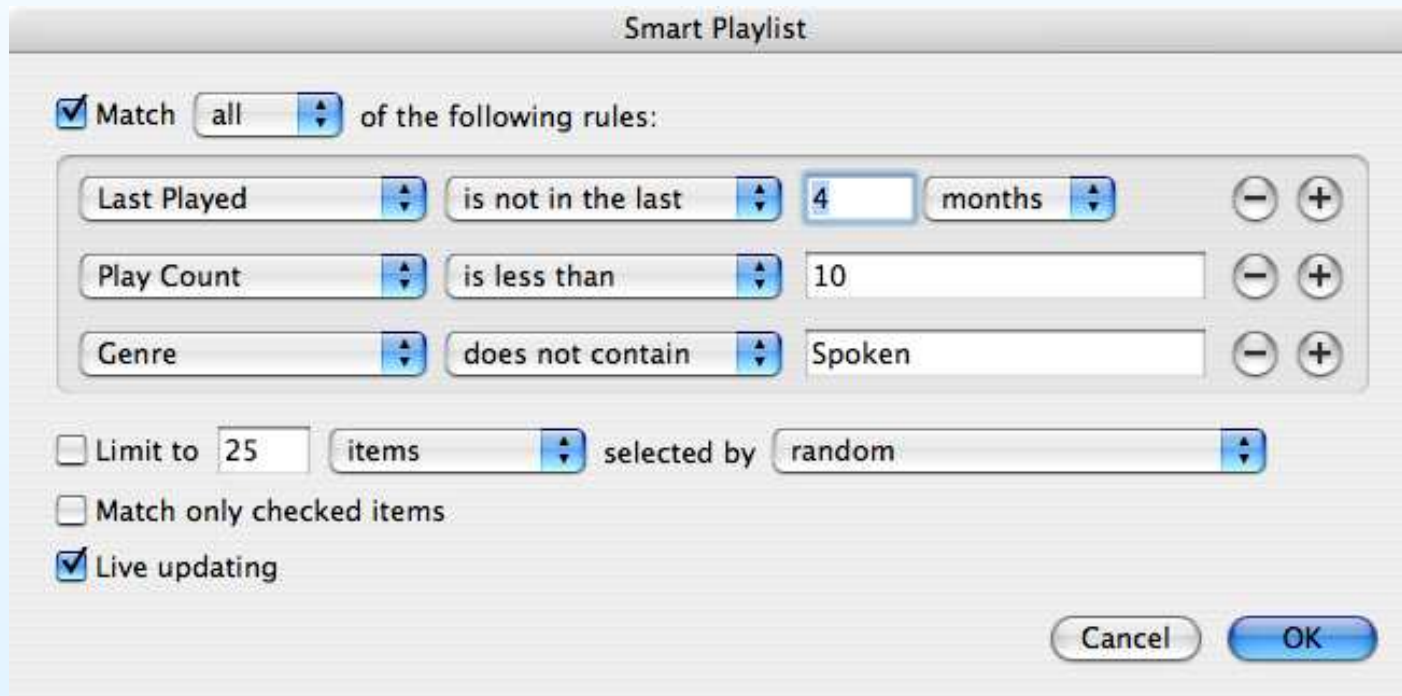
# Domain Specific Languages

- A *Domain Specific Language* (DSL) is a language designed for a particular problem domain.
  - User can focus on the *high level* of the domain, rather than the *low level* implementation details.
- Email filtering:

The image shows a screenshot of an email filtering rule configuration dialog box. At the top, there is a text field labeled "Description:" containing the word "Haskell". Below this, there is a section labeled "If" followed by a dropdown menu set to "any" and the text "of the following conditions are met:". Underneath, there is a single condition row: "Subject" (dropdown), "Contains" (dropdown), and "[Haskell]" (text field), with minus and plus buttons on the right. Below this is a section labeled "Perform the following actions:". Underneath, there is a single action row: "Move Message" (dropdown), "to mailbox:" (text), "Haskell" (dropdown), and minus/plus buttons on the right. At the bottom right, there are "Cancel" and "OK" buttons.

# Domain Specific Languages

- A *Domain Specific Language* (DSL) is a language designed for a particular problem domain.
  - User can focus on the *high level* of the domain, rather than the *low level* implementation details.
- Music playlists



## Embedded Domain Specific Languages

---

An *Embedded Domain Specific Language* (EDSL) is a DSL implemented by embedding in a *host* language.

- Identify the general properties, requirements and operations in the domain
- Using a dependently typed host, give precise constraints on valid programs

# Embedded Domain Specific Languages

---

An *Embedded Domain Specific Language* (EDSL) is a DSL implemented by embedding in a *host* language.

- Identify the general properties, requirements and operations in the domain
- Using a dependently typed host, give precise constraints on valid programs

IDRIS aims to support hosting EDSLs for *correct systems programming*. Key features to support this are:

- Compile-time evaluation
- Overloadable syntax
- Interfacing with C libraries, efficient code generation

## Part 2

# A Brief Introduction to IDRIS

## Dependent Types in IDRIS

IDRIS is loosely based on Haskell, and has similarities with Agda and Epigram. Some data types:

```
data Nat = 0 | S Nat;
```

```
infixr 5 :: ; -- Define an infix operator
```

```
data Vect : Set -> Nat -> Set where -- List with size
```

```
  VNil : Vect a 0
```

```
  | (::) : 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` (by unification).

## Syntax overloading: `do`-notation

Like Haskell, IDRIS allows `do`-notation to be rebound, e.g.:

```
data Maybe a = Nothing | Just a;
```

```
maybeBind : Maybe a -> (a -> Maybe b) -> Maybe b;
```

```
do using (maybeBind, Just) {  
  m_add : Maybe Int -> Maybe Int -> Maybe Int;  
  m_add x y = do { x' <- x;  
                  y' <- y;  
                  return (x' + y'); };  
}
```

## Classic example: The well-typed interpreter

```
data Ty = TyInt | TyFun Ty Ty;
```

```
evalTy : Ty -> Set;
```

```
using (G:Vect Ty n) {
```

```
  data Expr : Vect Ty n -> Ty -> Set where
```

```
    Var : (i:Fin n) -> Expr G (vlookup i G)
```

```
  | Val : (x:Int) -> Expr G TyInt
```

```
  | Lam : Expr (A :: G) T -> Expr G (TyFun A T)
```

```
  | App : Expr G (TyFun A T) -> Expr G A -> Expr G T
```

```
  | Op : (evalTy A -> evalTy B -> evalTy C) ->
```

```
        Expr G A -> Expr G B -> Expr G C;
```

```
}
```

## Classic example: The well-typed interpreter

```
data Env : Vect Ty n -> Set where
  Empty   : Env VNil
  | Extend : (res:evalTy T) -> Env G -> Env (T ::: G);
```

```
eval : Env G -> Expr G T -> evalTy T;
eval env (Var i)      = envLookup i env;
eval env (Val x)      = x;
eval env (Lam sc)     = \x => eval (Extend x env) sc;
eval env (App f a)    = eval env f (eval env a);
eval env (Op op l r) = op (eval env l) (eval env r);
```

## Classic example: The well-typed interpreter

We use the IDRIS type checker to check `Expr` programs, e.g.:

```
add : Expr G (TyFun TyInt (TyFun TyInt TyInt));  
add = Lam (Lam (Op (+) (Var (fS fO)) (Var fO)))
```

```
double : Expr G (TyFun TyInt TyInt);  
double = Lam (App (App add (Var fO)) (Var fO))
```

Unfortunately, this approach is not entirely suitable for an EDSL  
— we have to construct syntax trees explicitly!

## Classic example: The well-typed interpreter

We use the IDRIS type checker to check `Expr` programs, e.g.:

```
add : Expr G (TyFun TyInt (TyFun TyInt TyInt));  
add = Lam (Lam (Op (+) (Var (fS fO)) (Var fO)))
```

```
double : Expr G (TyFun TyInt TyInt);  
double = Lam (App (App add (Var fO)) (Var fO))
```

Unfortunately, this approach is not entirely suitable for an EDSL  
— we have to construct syntax trees explicitly!

(...no, I'm not a LISP programmer.)

## Syntax overloading: `dsl`-notation

We have seen overloadable `do`-notation, which is useful for EDSL construction. In IDRIS, we also provide a more general overloading construct:

```
dsl expr {  
  lambda      = Lam  
  variable    = Var  -- de Bruijn indexed variable  
  index_first = f0   -- most recently bound variable  
  index_next  = fS   -- earlier bound variable  
  apply       = App  
  pure        = id  
}
```

This allows IDRIS syntactic constructs to be used to build `Expr` programs. (Open question: is there a type class explanation?)

## Syntax overloading: DSL-notation

Some `Expr` programs, revisited:

```
test    = expr (\x, y => Op (+) x y );  
double = expr (\x => [ | test x x | ] );
```

The *idiom brackets* `[ | . | ]` allow an alternative form of application.

```
fact : Expr G (TyFun TyInt TyInt);  
fact = expr (\x =>  
    If (Op (==) x (Val 0))  
        (Val 1)  
        (Op (*) x [ | fact (Op (-) x (Val 1)) | ] ));
```

Can we now apply the well-typed interpreter approach to more interesting problems?

# An EDSL for Generic Resource Correctness

## The Problem

Consider a simple file handling API (following Haskell's):

```
open   : String -> Purpose -> IO File;
read   : File -> IO String;
close  : File -> IO ();
```

The following program type checks, but fails at run-time:

```
fprog filename =
  do { h <- open filename Writing;
      content <- read h;
      close h; };
```

## The Problem

Adding some dependent types makes things slightly better:

```
open   : String -> (p:Purpose) -> IO (File p);
read   : File Reading -> IO String;
close  : File p -> IO ();
```

The following program no longer type checks:

```
fprog filename =
  do { h <- open filename Writing;
      content <- read h;    -- Type error!
      close h; };
```

## The Problem

Adding some dependent types makes things slightly better:

```
open   : String -> (p:Purpose) -> IO (File p);  
read   : File Reading -> IO String;  
close  : File p -> IO ();
```

The following program corrects this error:

```
fprog filename =  
  do { h <- open filename Reading;  
      content <- read h;  
      close h; };
```

...but we still have some problems.

## The Problem

Adding some dependent types makes things slightly better:

```
open   : String -> (p:Purpose) -> IO (File p);
read   : File Reading -> IO String;
close  : File p -> IO ();
```

This program type checks, but fails at run-time:

```
fprog filename =
  do { h <- open filename Reading;
      content <- read h;
      close h;
      read h; -- It's closed, but h still in scope
    };
```

(Not to mention that we didn't check that `open` succeeded!)

## Managing State

---

Resource management (file handling, network protocols, memory, ...) is a common problem in systems programming. Some difficulties:

- *Time dependence* — need to reason about a given state while it is valid
- *Aliasing* — must not retain references to earlier invalid states
- *Errors* — some operations (e.g. opening a file) may not execute correctly

## Resource Aware EDSL

Our solution: Embedded DSL to capture resource state *in the type* (c.f. linear types). Recall our motivating example:

```
dumpFile : String -> RES ();
dumpFile filename
  = res do { let h = open filename Reading;
              Check h
              (rputStrLn "File open error")
              (do { rreadH h;
                    rclose h;
                    rputStrLn "DONE"; } ) };
              };
```

## Resource Aware EDSL

---

A File is an instance of a *resource*, with a state, which can be:

- *Created*: by an `open` (which might fail)
- *Updated*: changing the state, e.g. by closing the file
- *Used*: accessing without updating, e.g. by reading

File operations conform to a *resource usage protocol* which explain which operations are valid, and when.

## Resource Aware EDSL

First, we categorise operations into resource *Creators*, *Updaters*, and *Users*, lifting from IO:

```
data Creator : Set -> Set where
    MkCreator : IO a -> Creator a;

ioc : IO a -> Creator a;
ioc = MkCreator;

open  : String -> (p:Purpose) ->
        Creator (Either () (File p));
close : File p -> Updater ();
read  : File Reading -> Reader String;
```

## Resource Aware EDSL

Next, we define an EDSL which captures scoping rules for resources, indexed over a set of *input* and *output* resources, and the return type:

```
data Ty = R Set | Val Set | Choice Set Set;
```

```
data Res : Vect Ty n -> Vect Ty n -> Ty -> Set where
```

```
  Let      : Creator (evalTy a) ->
             Res (a :: gam) (Val () :: gam') (R t) ->
             Res gam gam' (R t)
```

```
  | Update : (a -> Updater b) ->
             (p:HasType gam i (Val a)) ->
             Res gam (update gam p (Val b)) (R ())
```

```
  | Use     : (a -> Reader b) -> HasType gam i (Val a) ->
             Res gam gam (R b)
```

```
  . . .
```

## Resource Aware EDSL

Next, we define an EDSL which captures scoping rules for resources, indexed over a set of *input* and *output* resources, and the return type:

```
data Ty = R Set | Val Set | Choice Set Set;
```

```
data Res : Vect Ty n -> Vect Ty n -> Ty -> Set where
```

```
  Let      : Creator (evalTy a) ->
             Res (a :: gam) (Val () :: gam') (R t) ->
             Res gam gam' (R t)
```

```
  | Update : (a -> Updater b) ->
             (p:HasType gam i (Val a)) ->
             Res gam (update gam p (Val b)) (R ())
```

```
  | Use    : (a -> Reader b) -> HasType gam i (Val a) ->
             Res gam gam (R b)
```

```
  ...
```

## Resource Aware EDSL

Next, we define an EDSL which captures scoping rules for resources, indexed over a set of *input* and *output* resources, and the return type:

```
data Ty = R Set | Val Set | Choice Set Set;
```

```
data Res : Vect Ty n -> Vect Ty n -> Ty -> Set where
```

```
  Let      : Creator (evalTy a) ->
```

```
            Res (a :: gam) (Val () :: gam') (R t) ->
```

```
            Res gam gam' (R t)
```

```
  | Update : (a -> Updater b) ->
```

```
            (p:HasType gam i (Val a)) ->
```

```
            Res gam (update gam p (Val b)) (R ())
```

```
  | Use    : (a -> Reader b) -> HasType gam i (Val a) ->
```

```
            Res gam gam (R b)
```

```
  ...
```

## Resource Aware EDSL

Next, we define an EDSL which captures scoping rules for resources, indexed over a set of *input* and *output* resources, and the return type:

```
eval : Env gam -> Res gam gam' t ->  
      (Env gam' -> evalTy t -> IO u) -> IO u;
```

```
run : Res VNil VNil (R t) -> IO t;
```

```
run prog = interp Empty prog (\env, res => res);
```

## Resource Aware EDSL

We can give this language some usable syntax with a `dsl` declaration:

```
dsl res {  
    bind          = Bind  
    return       = Return  
    variable     = id  
    let          = Let -- as lambda overloading, plus value  
    index_first  = stop  
    index_next   = pop  
}
```

(Note that we also use the `dsl` construct to overload `do`-notation — **Bind** composes DSL operations, **Return** injects values into a resource.)

## Resource Aware EDSL

Returning to our motivating example:

```
syntax RES x      = {gam:Vect Ty n} -> Res gam gam (R x);  
syntax rclose h = Update close h;  
...
```

```
dumpFile : String -> RES ();  
dumpFile filename  
  = res do { let h = open filename Reading;  
             Check h  
             (rputStrLn "File open error")  
             (do { rreadH h;  
                   rclose h;  
                   rputStrLn "DONE"; } )};  
};
```

## Resources and State Machines

---

The `Res` EDSL allows us to write functions with signatures corresponding directly to *state machine transitions*, and guarantee that functions are composed correctly. How widely applicable is this?

- Any API which relies on operations being applied in a certain order
  - Network sockets, OpenSSL, AI planning. . .
- Any API with state/resource constraints
  - Files, Threads/locks, Hardware interfaces, . . .
- Any protocol which can be described by a state machine
  - TCP/IP, Needham-Schroeder-Lowe, . . .
- Also, `Res` programs are *composable*
  - Using de Bruijn indices for resources makes this easy

## Part 4

# Conclusions

## Conclusions

- 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
- Applied to Files, but other APIs could benefit
  - Networks, OpenSSL, Threads, ...
- What is different about IDRIS that makes this possible?
  - Simple FFI, `dsl` syntax.
  - (Some features I haven't had time to mention are useful too — e.g. embedded theorem prover)
- Complete example available online
  - <https://github.com/edwinb/ResIO>

## Final Thoughts

- IDRIS allows us to write (now!) *systems software* with *guaranteed* properties.
  - (Caveat: “Research quality” meaning of “guaranteed” :-))
- We’ve seen lots of interesting dependently typed functions, programs and models...
  - ... and I expect to see more today!
  - But what about *systems* and *applications*?
  - In other words — how well do our languages and tools scale? What about *software engineering* considerations?
  - Don’t just model it, implement it!
- Things I’d like to see (and are possible!) in a DT language:
  - Network transport and routing, type safe web server and application DSL, device drivers, embedded systems, ...

Part n+1

Extras

## Resource Aware EDSL, completed

```
data HasType : Vect Ty n -> Fin n -> Ty -> Set where
  stop : HasType (a :: gam) f0 a
  | pop  : HasType gam i b ->
          HasType (a :: gam) (fS i) b;
```

```
data Env : Vect Ty n -> Set where
  Empty : Env VNil
  | Extend : evalTy a -> Env gam -> Env (a :: gam);
```

```
envLookup : HasType gam i a -> Env gam -> evalTy a;
update    : (gam : Vect Ty n) -> HasType gam i b ->
            Ty -> Vect Ty n;
```

## Resource Aware EDSL, completed

```
data Res : Vect Ty n -> Vect Ty n -> Ty -> Set where
  ...
  | Check   : (p:HasType gam i
               (Choice (evalTy a) (evalTy b))) ->
               Res (update gam p a) (update gam p c) T ->
               Res (update gam p b) (update gam p c) T ->
               Res gam (update gam p c) T
  | While   : Res gam gam (R Bool) ->
               Res gam gam (R ()) -> Res gam gam (R ())
  | Return  : a -> Res gam gam (R a)
  | Bind    : Res gam gam' (R a) ->
               (a -> Res gam' gam'' (R t)) ->
               Res gam gam'' (R t);
```