Domain Specific Languages by Overloading

Edwin Brady

eb@cs.st-andrews.ac.uk

University of St Andrews

GSDP Conference, 11th October 2011

Introduction

A constant problem:

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

Using strong type systems, we aim to write programs and *know* they are correct before running them.

Problem: Resource Management

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

Of course, resource management is also a problem in other, non Computer Science, contexts.

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, using DSLs

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

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

Dependent Types

Dependent types are types which are parameterised by *values*. For example:

- `Fin n` — finite set with `n` elements
- `Vect a n` — a vector of types `a` with `n` elements

By parameterising types by values, we can give more precise types (hence specifications) to programs. e.g.

```
lookup : Fin n    -> Vect a n -> a;
append : Vect a n -> Vect a m -> Vect a (n + m);
```


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

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 know how to construct syntax trees explicitly!

Syntax overloading: `dsl`-notation

To make such languages usable, we provide a *syntax overloading* construct:

```
dsl expr {
    lambda      = Lam
    variable    = Var  -- de Bruijn indexed variable
    index_first = fO   -- 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.

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

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

We define an EDSL which defines when resources are valid indexed over a set of *input* and *output* resources.

```
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

We define an EDSL which defines when resources are valid indexed over a set of *input* and *output* resources.

```
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

We define an EDSL which defines when resources are valid indexed over a set of *input* and *output* resources.

```
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

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 valu
   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

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"; });
             };
```


Conclusion

Using *dependent types*, and implementing domain specific languages by *syntax overloading*, we can construct programs with guaranteed resource consumption properties.

- Express *pre-conditions* and *post-conditions* on resource operations.
- Ensure *composability* of resource operations.

I'm a Computer Scientist, so for me, resources are:

- Files, network sockets, locks, memory . . .

But of course, "resource" has a more general meaning.

- Over to you! What to model, what properties are needed?