Resource-safe Systems Programming with Embedded Domain Specific Languages

Edwin Brady and Kevin Hammond

University of St Andrews, KY16 9SX, Scotland/UK,
{eb,kh}@cs.st-andrews.ac.uk

Abstract. We introduce a new overloading notation that facilitates programming, modularity and reuse in Embedded Domain Specific Languages (EDSLs), and use it to reason about safe resource usage and state management. We separate the structural language constructs from our primitive operations, and show how precisely-typed functions can be lifted into the EDSL. In this way, we implement a generic framework for constructing state-aware EDSLs for systems programming.

Keywords: Dependent Types, Resource Usage, (Embedded) Domain-Specific Languages, Program Verification.

1 Introduction

Domain Specific Languages (DSLs) are designed to solve problems in specific domains (e.g. Matlab/Simulink for real-time systems or SQL for database queries). One popular implementation technique is to embed a DSL in a *host* language, so creating an Embedded Domain Specific Language (EDSL) [12]. This allows rapid development of a DSL by exploiting host language features, such as parsing/code generation. However, host-language specific information, such as details of host language constructs, often "leaks" into the DSL, inhibiting usability and reducing abstraction. In order to be truly *practical*, we must address such issues so that our EDSL is modular, composable and reusable. This paper introduces a new overloading notation that allows EDSLs to be more easily used in practice, and shows how it can be used to develop an EDSL for reasoning about safe resource usage and state management. We make the following specific contributions:

1. We present the `dsl` construct, a modest extension to the dependently-typed language Idris that allows host language syntax, in particular variable binding, to be overloaded by an Embedded DSL (Section 3).
2. Using the `dsl` construct, we show how to embed languages with alternative forms of binding: we embed an *imperative* language, which manages mutable local variables in a type-safe way, and extend this to a *state-aware* language which manages linear resources (Section 4).
3. We show how to convert a protocol described by state transitions into a verified implementation (Sections 5 and 6).

By embedding the DSL within a dependently-typed language, we obtain the key advantage of *correctness by construction*: the host language type system *automatically* verifies the required DSL properties without needing to first translate into an equivalent set of state transitions and subsequently checking these. As Landin said, "Most programming languages are partly a way of expressing things in terms of other things, and partly a basic set of given things" [14]. In our state-aware DSL, the basic set of given things explains how resources are created and how states interact. Like Landin's ISWIM, this DSL can be problem-oriented by providing functions for creating, updating and using primitive values. The embedding then *composes* these constructs into a complete and verifiable EDSL. Example code for the resource language presented in this paper is available from `http://www.cs.st-andrews.ac.uk/~eb/padl12-resources.tgz`.

2 The Well-typed Interpreter

Dependent types, in which *types* may be predicated on *values*, allow us to express a program's specification and constraints precisely. In the context of EDSLs, this allows us to express a precise type system, describing the exact properties that EDSL programs must satisfy, and have the host language check those properties. The well-typed interpreter [1, 7, 20] for a simple functional language is commonly used to illustrate the key concepts of dependently-typed programming. Here, the type system ensures that only well-typed source programs can be represented and interpreted.

In this section, we use the well-typed interpreter example to introduce Domain Specific Language implementation in IDRIS. IDRIS [5] is an experimental functional programming language with dependent types, similar to Agda [19] or Epigram [9, 16]. It is eagerly evaluated and compiles to C via the Epic compiler library [4]. It is implemented on top of the IVOR theorem proving library [3], giving direct access to an interactive tactic-based theorem prover. A full tutorial is available online at `http://idris-lang.org/tutorial/`.

2.1 Language Definition

Figure 1 defines a simple functional expression language, **Expr**, with integer values and operators. The `using` notation indicates that `G` is an implicit argument to each constructor, with type `Vect Ty n`. Terms of type `Expr` are indexed by i) a context (of type `Vect Ty n`), which records types for the variables that are in scope; and ii) the type of the term (of type `Ty`). The valid types (`Ty`) are integers (`TyInt`) or functions (`TyFun`). We define terms to represent variables (`Var`), integer values (`Val`), lambda-abstractions (`Lam`), function calls (`App`), and binary operators (`Op`). Types may either be integers (`TyInt`) or functions (`TyFun`), and are translated to IDRIS types using `interpTy`. Our definition of `Expr` also states its typing rules, in some context, by showing how the type of each term is constructed. For example:

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

interpTy : Ty -> Set;
interpTy TyInt = Int;
interpTy (TyFun A T) = interpTy A -> interpTy T;

data Fin : Nat -> Set where
     fO : Fin (S k)
   | fS : Fin k -> Fin (S k);

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  : (interpTy A -> interpTy B -> interpTy C) ->
           Expr G A -> Expr G B -> Expr G C;
}
```

Fig. 1. The Simple Functional Expression Language, **Expr**.

```
   Val : (x:Int)   -> Expr G TyInt
   Var : (i:Fin n) -> Expr G (vlookup i G)
```

The type of `Val` indicates that values have integer types (`TyInt`), and the type of `Var` indicates that the type of a variable is obtained by looking up `i` in context `G`. For any term, `x`, we can read `x : Expr G T` as meaning "`x` has type `T` in the context `G`". Expressions in this representation are *well-scoped*, as well as *well-typed*. Variables are represented by *de Bruijn* indices, which are guaranteed to be bounded by the size of the context, using `i:Fin n` in the definition of `Var`. A value of type `Fin n` is an element of a finite set of `n` elements, which we use as a reference to one of `n` variables. Evaluation is via an interpretation function, which takes an expression and and environment corresponding to the context in which that expression is defined. The definition can be found in [6].

```
   interp : Env G -> Expr G T -> interpTy T;
```

We can now define some simple example functions. We define each function to work in an arbitrary context `G`, which allows it to be applied in any subexpression in any context. Our first example function adds its integer inputs:

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

We can use `add` to define the `double` function:

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

```
data Ty = TyInt | TyBool | TyFun Ty Ty;
interpTy TyBool = Bool;

data Expr : (Vect Ty n) -> Ty -> Set where
    ...
  | If : Expr G TyBool -> Expr G A -> Expr G A -> Expr G A;
```

Fig. 2. Booleans and `If` construct

2.2 Control structures and recursion

To make **Expr** more realistic, we add boolean values and an `If` construct. These extensions are shown in Figure 2. Using these extensions, we can define a (recursive) factorial function:

```
fact : Expr G (TyFun TyInt TyInt);
fact = Lam (If (Op (==) (Val 0) (Var fO)) (Val 1)
            (Op (*) (Var fO)
              (App fact (Op (-) (Var fO) (Val 1)))));
```

We have all the fundamental features of a full programming language here: a type system, variables, functions and control structures. While **Expr** itself is clearly too limited to be of practical use, we could use similar methods to represent more complex systems, e.g. capturing sizes, resource usage or linearity constraints. In the rest of this paper, we will explore how to achieve this.

3 Syntax Overloading

We would like to use the well-typed interpreter approach to implement *domain specific* type systems capturing important properties of a particular problem domain, such as *resource correctness*. Unfortunately, the need to write programs as syntax trees, and in particular the need to represent variables as *de Bruijn* indices, at first appears to make this impractical. In this section, we present a new host language construct that allows host language syntax to be used when constructing programs in the EDSL, and use it to implement a practical embedded DSL for resource- and state-aware programs.

3.1 `do`-notation

In Haskell, we can overload `do`-notation to give alternative interpretations of variable binding in monadic code. We have implemented a similar notation in Idris using syntactic overloading. For example, we can use `do`-notation for `Maybe` by declaring which bind and return operators to use:

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

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

```
dsl expr {
    lambda      = Lam, variable = Var,
    index_first = f0,  index_next = fS,
    apply       = App, pure = id
}
```

Fig. 3. Overloading syntax for **Expr**

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

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

Fig. 4. Expr programs after overloading

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

Overloading `do`-notation is useful for EDSL implementation, in that it allows us to use a different binding construct provided by the EDSL. However, `do`-notation provides only one kind of binding. What if we need e.g. λ and `let` binding? What if we need a different notion of application, for example with effects [17]?

3.2 The `dsl` Construct

To allow multiple kinds of binding and application, we introduce a new construct to Idris. A `dsl` declaration gives a name for a language and explains how each *host* language construct is translated into the required EDSL construct. Figure 3 shows, for example, how Idris's binding syntax is overloaded for **Expr**. We give a language name, `expr`, and say that Idris lambdas correspond to `Lam`, and variables correspond to `Var` applied to a *de Bruijn* index, which is constructed from `f0` and `fS`. Applications are built using `App`, with the pure, functional part of the application built with `id`.

The programs we presented in the previous section can now be written using Idris's binding construct, as in Figure 4. Since we called the DSL `expr`, an expression `expr e` applies the syntactic overloading to the sub-expression `e`. Application overloading applies only under explicit "idiom brackets" [17]. Intuitively, `expr e` translates `e` according to the following rules:

- Any expression `\x => a` is translated to `Lam a'`, where `a'` is `a` with instances of the variable `x` is translated to a de Bruijn indexed `Var i`. The index `i` is built from `f0` and `fS` counting the number of names bound since `x`.

$e ::= x$ (Variable) | $e\ e$ (Application)
| $\backslash\ x$ `=>` e (`lambda` binding) | `let` x `=` e_1 `in` e_2 (`let` binding)
| `[|` e `|]` (Idiomatic application) | `do` { ds } (`do` block)
| *return* (return keyword) | $dsl\ e$ (Expression under overloading)

$d ::= x$ `<-` e (Binding) | e (Expression)

$ds ::= d; ds \mid e$

Fig. 5. Core Idris expressions

– Any application under idiom brackets `[| f a1 a2 ... an |]`is translated to `App (App (App (id f) a1) a2) ... an`

Within a `dsl` declaration, we can provide several overloadings:

– `bind` and `return`, for overloading `do`-notation.
– `pure` and `apply`, for overloading application under idiom brackets.
– `lambda`, `let`, `variable`, `index_first` and `index_next`, for overloading lambda and let bindings.

It is not necessary to define all of these overloadings. However, if either `lambda` or `let` is defined, all of `variable`, `index_first` and `index_next` must be defined, otherwise there is no valid translation for the bound variable.

3.3 Formal definition

To give a precise definition of the `dsl` construct, we define four translation schemes on core Idris expressions as defined in Figure 5.

– $\mathcal{D}[\![\cdot]\!]\ dsl$, defined in Figure 6, transforms an Idris expression by a given set of overloadings *dsl*.
– $\mathcal{V}[\![\cdot]\!]\ x\ i$, defined in Figure 7, converts a variable name x to de Bruijn index i in an expression.
– $\mathcal{I}[\![\cdot]\!]$, defined in Figure 8, converts an application under idiom brackets
– $\mathcal{M}[\![\cdot]\!]$, also defined in Figure 8, converts a `do`-block.

Mostly, these schemes are a straightforward traversal of the structure of Idris expressions. In $\mathcal{D}[\![\cdot]\!]$, we can nest `dsl` declarations, updating the set of overloadings. We leave the overloading parameter o implicit in $\mathcal{V}[\![\cdot]\!]$, $\mathcal{I}[\![\cdot]\!]$ and $\mathcal{M}[\![\cdot]\!]$. The definition of each of the overloadable names is extracted from this parameter. Note that $\mathcal{D}[\![\cdot]\!]$ combines the other translation schemes, which each do a specific job. This means in particular that lambda bindings generated by $\mathcal{M}[\![\cdot]\!]$ can further be translated to an overloaded `lambda`.

$$
\begin{array}{lll}
\mathcal{D}[\![x]\!]\ o & \mapsto\ x & \\
\mathcal{D}[\![e_1\ e_2]\!]\ o & \mapsto\ (\mathcal{D}[\![e_1]\!]\ o)\ (\mathcal{D}[\![e_2]\!]\ o) & \\
\mathcal{D}[\![\backslash\ x\ \texttt{=>}\ e]\!]\ o & \mapsto\ \mathcal{D}[\![\texttt{lambda}\ (\mathcal{V}[\![e]\!]\ x\ 0)]\!]\ o & \text{(if } \texttt{lambda} \text{ defined)} \\
 & \mapsto\ \backslash\ x\ \texttt{=>}\ \mathcal{D}[\![e]\!]\ o & \text{(otherwise)} \\
\mathcal{D}[\![\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!]\ o & \mapsto\ \mathcal{D}[\![\texttt{let}\ e_1\ (\mathcal{V}[\![e_2]\!]\ x\ 0)]\!]\ o & \text{(if } \texttt{let} \text{ defined)} \\
 & \mapsto\ \texttt{let}\ x\ \texttt{=}\ \mathcal{D}[\![e_1]\!]\ o\ \texttt{in}\ \mathcal{D}[\![e_2]\!]\ o & \text{(otherwise)} \\
\mathcal{D}[\![\texttt{[|}\ e\ \texttt{|]}]\!]\ o & \mapsto\ \mathcal{D}[\![\mathcal{I}[\![e]\!]]\!]\ o & \\
\mathcal{D}[\![\texttt{do}\ \{\ ds\ \}]\!]\ o & \mapsto\ \mathcal{D}[\![\mathcal{M}[\![ds]\!]]\!]\ o & \\
\mathcal{D}[\![return]\!]\ o & \mapsto\ \texttt{return} & \\
\mathcal{D}[\![dsl\ e]\!]\ o & \mapsto\ \mathcal{D}[\![e]\!]\ dsl &
\end{array}
$$

Fig. 6. The $\mathcal{D}[\![\cdot]\!]$ translation schemes

$$
\begin{array}{lll}
\mathcal{V}[\![x_1]\!]\ x_2\ i & \mapsto\ \texttt{variable}\ (\textsc{MkVar}\ i) & \text{(if } x_1 = x_2\text{)} \\
 & \mapsto\ x_1 & \text{(otherwise)} \\
\mathcal{V}[\![e_1\ e_2]\!]\ x\ i & \mapsto\ (\mathcal{V}[\![e_1]\!]\ x\ i)\ (\mathcal{V}[\![e_2]\!]\ x\ i) & \\
\mathcal{V}[\![\backslash\ x_1\ \texttt{=>}\ e]\!]\ x_2\ i & \mapsto\ \backslash\ x_1\ \texttt{=>}\ \mathcal{V}[\![e]\!]\ x_2\ (i+1) & \text{(if } \texttt{lambda} \text{ defined)} \\
 & \mapsto\ \backslash\ x_1\ \texttt{=>}\ \mathcal{V}[\![e]\!]\ x_2\ i & \text{(otherwise)} \\
\mathcal{V}[\![\texttt{let}\ x_1\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!]\ x_2\ i & \mapsto\ \texttt{let}\ x_1\ \texttt{=}\ \mathcal{V}[\![e_1]\!]\ x_2\ i\ \texttt{in}\ \mathcal{V}[\![e_2]\!]\ x_2\ (i+1) & \text{(if } \texttt{let} \text{ defined)} \\
 & \mapsto\ \texttt{let}\ x_1\ \texttt{=}\ \mathcal{V}[\![e_1]\!]\ x_2\ i\ \texttt{in}\ \mathcal{V}[\![e_2]\!]\ x_2\ i & \text{(otherwise)} \\
\mathcal{V}[\![\texttt{[|}\ e\ \texttt{|]}]\!]\ x\ i & \mapsto\ \texttt{[|}\ \mathcal{V}[\![e]\!]\ x\ i\ \texttt{|]} & \\
\mathcal{V}[\![\texttt{do}\ \{\ ds\ \}]\!]\ x\ i & \mapsto\ \texttt{do}\ \{\ \mathcal{V}[\![ds]\!]\ x\ i\ \} & \\
\mathcal{V}[\![return]\!]\ x\ i & \mapsto\ return & \\
\mathcal{V}[\![dsl\ e]\!]\ x\ i & \mapsto\ dsl\ (\mathcal{V}[\![e]\!]\ x\ i) &
\end{array}
$$

$$
\begin{array}{lll}
\textsc{MkVar}\ 0 & \mapsto\ \texttt{index\_first} & \\
\textsc{MkVar}\ (n+1) & \mapsto\ \texttt{index\_next}\ (\textsc{MkVar}\ n) &
\end{array}
$$

Fig. 7. The $\mathcal{V}[\![\cdot]\!]$ translation scheme

4 Resource Management

In a typical file management API, such as that in Haskell, we might find the following typed operations:

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

Unfortunately, it is easy to construct programs which are well-typed, but nevertheless fail at run-time, for example:

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

$$
\begin{array}{lll}
\mathcal{I}[\![e_1\ e_2]\!] & \mapsto \texttt{apply}\ (\mathcal{I}[\![e_1]\!])\ e_2 & \text{(top level application)} \\
\mathcal{I}[\![e]\!] & \mapsto \texttt{pure}\ e & \text{(all other expressions)} \\
\\
\mathcal{M}[\![x\ \texttt{<-}\ e; ds]\!] & \mapsto \texttt{bind}\ e\ (\backslash\ x\ \texttt{=>}\ \mathcal{M}[\![ds]\!]) & \\
\mathcal{M}[\![e; ds]\!] & \mapsto \texttt{bind}\ e\ (\backslash\ \_\ \texttt{=>}\ \mathcal{M}[\![ds]\!]) & \\
\mathcal{M}[\![e]\!] & \mapsto e &
\end{array}
$$

Fig. 8. The $\mathcal{I}[\![\cdot]\!]$ and $\mathcal{M}[\![\cdot]\!]$ translation schemes

If we make the types more precise, parameterising open files by purpose, `fprog` is no longer well-typed, and will therefore be rejected at compile-time.

```
data Purpose = Reading | Writing;

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

However, there is still a problem. The following program is well-typed, but fails at run-time — although the file has been closed, the handle `h` is still in scope:

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

Furthermore, we did not check whether the handle `h` was created successfully. Resource management problems such as this are common in systems programming — we need to deal with files, memory, network handles, etc, ensuring that operations are executed only when valid and errors are handled appropriately.

4.1 An EDSL for Generic Resource Correctness

To tackle this problem, we present an EDSL which tracks the *state* of resources at any point during program execution, and ensures that any resource protocol is correctly executed. We begin by categorising resource operations into creation, update and usage operations, by lifting them from `IO`. We illustrate this using `Creator`; `Updater` and `Reader` can be defined similarly.

```
data Creator a = MkCreator (IO a);

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

The `MkCreator` constructor is left abstract, so that a programmer can lift an operation into `Creator` using `ioc`, but cannot run it directly. `IO` operations can be converted into resource operations, tagging them appropriately:

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

```
data Res : Vect Ty n -> Vect Ty n -> Ty -> Set where
   Let    : Creator (interpTy 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)
   ...
```

Fig. 9. Resource constructs

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

Fig. 10. Control constructs

Here: `open` creates a resource, which may be an error (for simplicity here, the unit type) or a file handle opened for the appropriate purpose; `close` updates a resource from a `File p` to a unit (i.e., it makes the resource unavailable); and `read` accesses a resource (i.e., it reads from it, and the resource remains available). They are implemented by using the relevant `IO` functions and lifting. Resource operations are executed through a resource management EDSL, **Res**, with resource constructs (Figure 9), and control constructs (Figure 10).

As we did with **Expr** in Section 2, we index **Res** over the variables in scope (which represent resources), and the expression's type. This means firstly that we can refer to resources by *de Bruijn* indices, and secondly we can express precisely how operations may be combined. Unlike **Expr**, however, we allow types of variables to be updated. Therefore, we index over the input set of resource states, and the output set:

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

We can read `Res gam gam' T` as, "an expression of type `T`, with input resource states `gam` and output resource states `gam'`". Expression types can be resources, values, or a choice type:

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

The distinction between *resource* types, `R a`, and *value* types, `Val a`, is that resource types arise from `IO` operations. A choice type corresponds to `Either`:

```
interpTy : Ty -> Set;
interpTy (R t) = IO t;
interpTy (Val t) = t;
interpTy (Choice x y) = Either x y;
```

We represent variables by proofs of context membership, rather than directly by *de Bruijn* indices. As we will see shortly, this allows a neater representation of some language constructs:

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

envLookup : HasType gam i a -> Env gam -> interpTy a;
envUpdate : (p:HasType gam i a) -> (val:interpTy b) ->
            Env gam -> Env (update gam p b);
```

The type of the `Let` construct explicitly shows that, in the scope of the `Let` expression a new resource of type `a` is added to the set, having been made by a `Creator` operation. Furthermore, by the end of the scope, this resource must have been consumed (i.e. its type must have been updated to `Val ()`):

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

The `Update` construct applies an `Updater` operation, changing the type of a resource in the context. Here, using `HasType` to represent resource variables allows us to write the required type of the update operation simply as `a -> Updater b`, and put the operation first, rather than the variable.

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

The `Use` construct simply executes an operation without updating the context, provided that the operation is well-typed:

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

Finally, we provide a small set of control structures: `Choice`, effectively an `if...then...else` construct that guarantees that resources are correctly defined in each branch; `While`, a loop construct that guarantees that there are no state changes during the loop; `Lift`, a lifting operator for `IO` functions[1]; and `Bind` and `Return` to support `do`-notation. The type of `Bind` captures updates in the resource set. We use `dsl`-notation to overload the IDRIS syntax, in particular providing a `let`-binding to bind a resource and give it a human-readable name:

```
dsl res {
   let         = Let,  variable  = id,
   bind        = Bind, return    = Return,
   index_first = stop, index_next = pop
}
```

[1] This requires us to hide the resource operations, e.g. in a module

The interpreter for **Res** is written in continuation-passing style, where each operation passes on a result and an updated environment (containing resources):

```
interp : Env gam -> Res gam gam' t ->
         (Env gam' -> interpTy t -> IO u) -> IO u;

run : Res VNil VNil (R t) -> IO t;
run prog = interp Empty prog (\env, res => res);
```

5 First Example: File Management

We can use **Res** to implement a safe file-management protocol, where each file must be opened before use, opening a file must be checked, and files must be closed on exit. We define the following operations for opening, closing, reading a line[2], and testing for the end of file.

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

Simple example Returning to our simple example from Section 4, we now write the file-reading program as follows:

```
fprog : {gam:Vect Ty n} -> String -> Res gam gam (R String);
fprog filename =
   res do { let h = open filename Reading;
            Check h
              (Lift (putStrLn "File error"))
              (do { content <- Use read h;
                    Update close h; }); };
```

This is well-typed because the file is opened for reading, and by the end of the scope, the file has been closed. Syntax overloading allows us to name the resource `h` rather than using a *de Bruijn* index or context membership proof. Although this is a big improvement, the syntax is still somewhat unsatisfactory:

- The type of `fprog` is hard to read and write (and for practical use, we need programmers to write these signatures!)
- The need to apply `Use`, `Read` and `Lift` explicitly is a little ugly.

Fortunately, both problems can be addressed using Idris's syntax macros:

```
syntax RES x = {gam:Vect Ty n} -> Res gam gam (R x);

syntax rclose    h = Update close h;
syntax rread     h = Use read h;
syntax reof      h = Use eof h;
syntax rputStrLn x = Lift putStrLn x;
```

[2] Reading a line may fail, but we consider this harmless and return an empty string.

We now use `RES x` as the type of *any* resource safe program which returns an `x`, and `rclose` and `rread` as the file operations:

```
fprog : String -> RES String;
fprog filename =
   res do { let h = open filename Reading;
            Check h
              (rputStrLn "File error")
              (do { content <- rread h;
                    rclose h; }) };
```

Using loops In the following program, we open a file, read each line of the file and output it using a `While` loop, then close it:

```
dump : String -> RES String;
dump filename =
   res do { let h = open filename Reading;
            Check h
              (rputStrLn "File error")
              (do { While (do { end <- reof h;
                                return (not end); })
                          (do { str <- rread h;
                                rputStrLn str; });
                    rclose h; }) };
```

This program has a similar structure to the equivalent Haskell program written using the `IO` monad However, here the Idris type system guarantees that each operation is executed only when it is valid. We *cannot*, for instance, close the file during the loop, or try to read from the file in the branch where opening has failed. We have achieved this by writing ordinary monadic IO functions and lifting them intro a general framework which guarantees linear use of resources.

Embedding functions We can improve the program by lifting out the `While` loop into a function. Since this is an EDSL, we can use a host language function, but its type must refer to the EDSL's context. A **Res** function which uses a resource `a` but does not update it, and returns a value `b` is denoted by `a :-> b`:

```
syntax (:->) a b = {gam:Vect Ty n} ->
    HasType gam i (Val a) -> Res gam gam b;

readFile : FILE Reading :-> R ();
readFile h = res (While (do { end <- reof h;
                              return (not end); })
                        (do { str <- rreadLine h;
                              rputStrLn str; }));
```

We can use this function directly in a **Res** program:

```
dump filename = res do { let h = open filename Reading;
                         Check h
                           (rputStrLn "File error")
                           (do { readFile h; rclose h; }) };
```

Correspondingly, updating a resource variable is denoted by `a |-> b`:

```
syntax (|->) a b = {gam:Vect Ty n} ->
    (p:HasType gam i (Val a)) -> Res gam (update gam p b) (R ()) ;
```

6 Second Example: Network Transport

As well as defining high-level APIs, we can also use **Res** to implement low-level operating systems components such as reliable network transport. Let us briefly consider a simple automatic repeat request (ARQ) protocol, in which a machine S attempts to send packets reliably to a machine R.

1. S opens a connection to R and waits for R to acknowledge the connection.
2. For each packet, with a sequence number n:
   (a) S sends a packet with sequence number n, and waits for an acknowledgement from R.
   (b) If an acknowledgement is not received within a timeout period, retry.
3. S requests that the connection be closed.

Each operation may have pre-/post-conditions on the state of the connection.

```
connect    : Receiver        -> Creator (Either () (Net (Ready 0)));
send       : Net (Ready n)   -> Updater (Net (Waiting n));
recvAck    : Net (Waiting n) -> Updater (Either (Net (Ready n))
                                                (Net (Ready (S n))));
disconnect : Net (Ready n)   -> Updater ();
```

We implement the protocol by lifting these functions into **Res**, defining a function `sendList` which iterates across a list of packets, either sending them successfully or timing out:

```
sendList : List Packet -> (Net (Ready n) |-> R ());

arq : Receiver -> List Packet -> RES ();
arq r pkts = res do { let h = connect r;
                      Check h (rputStrLn "Couldn't open connection")
                              (do { sendList pkts h; }); };
```

Note that `sendList`'s type requires that it also closes the connection. It is written as a combination of `send` and `recvAck`, retrying if an acknowledgement is not received. As before, the primitives are composed using the constructs in **Res** to guarantee that resources are managed according to the protocol.

7 Related Work

We have previously explored the use of Idris for implementing EDSLs in domains such as networking [2, 5] and concurrency [8]. The work described here is similar to that of [8]. However, by using *de Bruijn* indices we obtain the key advantage of compositionality, a neat way to build contexts, etc. Unlike other approaches to resource usage verification based on e.g. model-checking [13, 15,

21], which translate the program into a (hopefully) equivalent set of state transitions that can subsequently be checked, the EDSL approach we have used here relates the *actual* program to the abstract state machine model, so guaranteeing correctness *by construction*. **Res** is inspired by work on linear types for resource management [10, 11], and an alternative approach would have been to add linear types to Idris's type system. We have avoided this for two reasons: firstly, we prefer to keep the core type theory of Idris as small and as easy to reason about as possible; secondly, as **Res** demonstrates, dependent types alone are strong enough to capture the linearity property. Finally, Hoare Type Theory has also been used in the Ynot system [18] to reason about imperative programs with side effects, as we have done in **Res**. However, our approach is much lighter weight: it involves writing the state transition functions directly, as normal Idris functions, then "promoting" them into the resource language. This makes it much easier to plug in new functionality, for example, in our system.

8 Conclusion

We have shown a new way to write resource-safe systems programs using domain-specific languages embedded in a dependently-typed host language. The `dsl`-notation introduced in this paper raises the abstraction level when programming EDSLs, adding the important properties of compositionality and modularity over previous approaches. Using this notation over a dependently-typed host language, we are able to produce automatically verified EDSL programs, provided the primitive state transitions are correctly written. We have also demonstrated the applicability and generality of the notation by developing a generic resource usage framework and applying it to two realistic systems programming scenarios. Like Landin's Iswim, the EDSL can be problem-oriented, providing functions for creating, updating and using primitive values. These primitives are then embedded into a generic composition framework, here exemplified by **Res**. Although not shown here, the approach can easily handle other constructs such as (higher-order) functions, further extending its applicability. We have not considered how to e.g. embed resources within data structures, although we expect this to be achievable by indexing larger data structures over a resource context. This may be important for some examples, particularly where we have long-lived resources, or a collection of live resources such as a list of open file handles.

There are a number of obvious future applications of this work in systems programming. In particular, we intend to study larger applications in network protocols, and consider how to capture and reason about security issues. The use of dependent types simplifies the task of producing verifiable systems programs as EDSLs, providing a lightweight, extensible and composable framework that is tightly integrated with the actual systems program.

References

1. L. Augustsson and M. Carlsson. An exercise in dependent types: A well-typed interpreter, 1999.

2. S. Bhatti, E. Brady, K. Hammond, and J. McKinna. Domain specific languages (DSLs) for network protocols. In *International Workshop on Next Generation Network Architecture (NGNA 2009)*, 2009.
3. E. Brady. Ivor, a proof engine. In *Implementation and Application of Functional Languages 2006*, LNCS. Springer-Verlag, 2007.
4. E. Brady. Epic — a library for generating compilers. In *Trends in Functional Programming (TFP '11)*, 2011. To appear.
5. E. Brady. Idris — systems programming meets full dependent types. In *Programming Languages meets Program Verification (PLPV 2011)*, pages 43–54, 2011.
6. E. Brady and K. Hammond. Scrapping your Inefficient Engine: using Partial Evaluation to Improve Domain-Specific Language Implementation. In *Proc. ICFP '10: ACM Intl. Conf. on Functional Programming*.
7. E. Brady and K. Hammond. A Verified Staged Interpreter is a Verified Compiler. In *Proc. GPCE '06: Conf. on Generative Prog. and Component Eng.*, 2006.
8. E. Brady and K. Hammond. Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. *Fundamenta Informaticae*, 102(2):145–176, 2010.
9. J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The Gentle Art of Levitation. In *Proc. ICFP '10: ACM Intl. Conf. on Funct. Prog.*, pages 3–14, 2010.
10. C. Hawblitzel. Linear types for aliased resources. Technical Report MSR-TR-2005-141, Microsoft Research, 2005.
11. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order function al programs. In *Proc. POPL 2003 — 2003 ACM Symp. on Principles of Programming La nguages*, pages 185–197. ACM, 2003.
12. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28A(4), December 1996.
13. A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Trans. Program. Lang. Syst.*, 27:264–313, March 2005.
14. P. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3), March 1966.
15. K. Marriott, P. Stuckey, and M. Sulzmann. Resource usage verification. In *In Proc. of First Asian Symposium, APLAS 2003*, pages 212–229. Springer-Verlag, 2003.
16. C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
17. C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18:1–13, January 2008.
18. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the Awkward Squad. In *Proc. ICFP '08: 2008 ACM Intl. Conf. on Functional programming*, pages 229–240. ACM, 2008.
19. U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, September 2007.
20. E. Pašalíc, W. Taha, and T. Sheard. Tagless Staged Interpreters for Typed Languages. In *Proc. ICFP 2002: Intl. Conf. on Functional Programming*. ACM, 2002.
21. D. Walker. A Type System for Expressive Security Policies. In *Proc. POPL 2000: ACM Intl. Symp. on Principles of Programming Languages*, pages 254–267, 2000.