Scrapping your Inefficient Engine: Using Partial Evaluation to Improve Domain-Specific Language Implementation

Edwin C. Brady Kevin Hammond

School of Computer Science, University of St Andrews, St Andrews, Scotland.
Email: eb,kh@cs.st-andrews.ac.uk

Abstract

Partial evaluation aims to improve the efficiency of a program by specialising it with respect to some known inputs. In this paper, we show that partial evaluation can be an effective *and, unusually, easy to use* technique for the efficient implementation of *embedded domain-specific languages*. We achieve this by exploiting *dependent types* and by following some simple rules in the definition of the interpreter for the domain-specific language. We present experimental evidence that partial evaluation of programs in domain-specific languages can yield efficient residual programs whose performance is competitive with their Java and C equivalents and which are also, through the use of dependent types, *verifiably resource-safe*. Using our technique, it follows that a verifiably correct and resource-safe program can also be an *efficient* program.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) Languages; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Languages, Verification, Performance

Keywords Dependent Types, Partial Evaluation

1. Introduction

This paper reconsiders the use of *partial evaluation* [20] for implementing *embedded domain-specific languages* (EDSLs) [19]. Partial evaluation is a well-known technique that aims to improve the efficiency of a program by automatically specialising it with respect to some known inputs. Embedded domain-specific languages embed specialist languages for some problem domain in a general-purpose host language. By reusing features from the host language, EDSLs can be implemented much more rapidly than their standalone equivalents, and can take advantage of compiler optimisations and other implementation effort in the host language.

A common approach to EDSL implementation in functional languages such as Haskell is to design an abstract syntax tree (AST) which captures the required operations and properties of the domain [14, 24], and then either to implement an interpreter on this AST or to use a code generator targeting e.g. C or LLVM [23]. Directly interpreting a syntax tree is a simple, lightweight approach, and it is relatively straightforward to verify the functional correctness of the interpreter. However, the resulting implementation is unlikely to be efficient. In contrast, code generation gives an efficient implementation, but it is harder to verify its correctness, harder to add new features to the EDSL, and can be harder to exploit features of the host language.

In this paper, we consider how partial evaluation can be used to achieve an EDSL implementation that is simple to write, straightforward to verify, *and efficient*. While the *theoretical* benefits of partial evaluation have been extensively covered in the literature e.g. [8, 16, 20, 37], there are very few *practical* examples (the work of Seefried et el. on Pantheon [34] is a notable exception). This is because it can be difficult to use partial evaluation effectively — several issues must be dealt with, including binding-time improvements, function calls, recursion, code duplication, and management of side-effects. Since these issues must usually be dealt with by the applications programmer, they limit the practical benefits of partial evaluation and so limit its widespread adoption. We argue here that partial evaluation can be a highly effective technique for implementing efficient EDSLs, allowing us to specialise the EDSL interpreter with respect to the EDSL source program. We also argue that this approach allows us to reason easily about the *correctness* of our implementation.

1.1 Contributions

Our main contribution is a new study of the practical limits of partial evaluation as a technique for realistic EDSL implementation. It is folklore that we have a choice between writing verifiably correct, but inefficient, code and more efficient, but potentially incorrect, code. Sadly, at this point in time, pragmatic software developers will generally make the latter choice. In this paper, we make the following key claim and support it with experimental evidence:

There is no need for correctness to be at the expense of efficiency.

We make the following specific contributions:

- we give experimental evidence that by partially evaluating an interpreter for the EDSL, the EDSL implementation compares favourably with Java, and is not significantly worse than C;
- we give concrete rules for defining an interpreter for an EDSL so as to gain the maximum benefit from partial evaluation;
- we describe the implementation of realistic, state-aware EDSLs using *dependent types*.

Although the techniques we describe apply equally to other dependently typed languages, we will use IDRIS[1] as a host language. IDRIS is a pure functional programming language with full-spectrum dependent types. Throughout this paper, we will identify

[1] `http://www.idris-lang.org/`

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

the features of IDRIS that assist both with EDSL implementation and with partial evaluation, and will also identify how the presence of dependent types affects standard partial evaluation techniques. In particular, in a language with full dependent types, there is a different view of the *phase distinction* between compile-time and run-time. Traditionally, this distinction is observed *syntactically* between types (which are easily erasable) and values. With dependent types, the distinction is *semantic*, between compile-time values (which are erasable by analysing data types [6]) and run-time values. This affects several aspects of the implementation, which we will identify here.

Partial evaluation has been known for many years, and Futamura's paper [16] on interpreter specialisation is now 39 years old. However, the technique is still not widely applied, because several problems arise when putting it into practice. We have found that these problems are either easily handled or simply do not arise when embedding DSLs in IDRIS, largely due to its type system.

1.2 Research Motivation: Resource-Aware Programming

The underlying motivation for our research is a desire to reason about *extra-functional* properties of programs, that is how programs behave in terms of their usage of finite resources such as memory and execution time, and how that behaviour affects the end-user, e.g. through proper management of files and exceptions. Ensuring that essential extra-functional properties are met is vitally important to writing software that is useful in practice. In fact, a respected industrial language designer once confided to us that "nobody in industry really cares whether a program does what it's supposed to – what they are really concerned with is how the program behaves when it is run"[2]. This concern is reflected in the major software failures that we see reported in the national and international press. Many of the most significant problems with software behaviour boil down to poor usage of resources: software can fail because of buffer overflows, because of deadlocks, because of memory leaks, because of inadequate checking (e.g. that a file handle has the correct mode), because it fails to meet hard real-time deadlines, and in many other ways that are not direct consequences of the *functional properties* of the software.

The EDSL approach, using a *dependently-typed* host language, allows us to define notations in which extra-functional resource usage properties are stated *explicitly* in a program's type. Since we are primarily concerned with ease of *reasoning* and *verification*, we take the simplest possible approach to EDSL implementation, via an interpreter. Our hypothesis is that first defining an interpreter and then specialising it with respect to EDSL programs is efficient enough for practical purposes. The definition of "efficient enough" is, of course, open to interpretation and it may be hard to evaluate whether it has been met. For the purposes of this paper, we will call a program "efficient enough" if it is similar in speed and memory footprint to an equivalent hand-written Java program

2. Idris and its Type Theory

IDRIS is an experimental functional programming language with dependent types, similar to Agda [29] or Epigram [26]. It has a Haskell-like syntax, but is evaluated eagerly. It compiles to C via a supercombinator compiler. IDRIS has monadic I/O in the style of Hancock and Setzer [18], and a simple foreign function interface. It is implemented on top of the IVOR theorem proving library [7], giving direct access to an interactive tactic-based theorem prover. This section explores the relationship between type checking and evaluation in dependently-typed languages, such as IDRIS.

[2] Joe Armstrong, Erlang designer, *personal communication*.

2.1 The Core Type Theory

The type theory underlying IDRIS is implemented by the IVOR theorem proving library. IDRIS provides a front-end (syntactic sugar), a back-end (a compiler), some primitive types and an interface to the IVOR theorem proving tools. IDRIS programs consist of functions over inductive data types, defined by dependent pattern-matching. Terms, t, are defined as follows:

t	::=	`Set`	Type of types
	|	x	Variables
	|	$\backslash x : t$ `=>` t	Abstraction
	|	$(x\ :\ t)$ `->` t	Function space
	|	$t\ t$	Function application
	|	`c`	Data constructor
	|	`D`	Inductive family

The type theory possesses **full-spectrum dependent types**, where the definition of terms also captures the definition of types. As a convention, we use T to stand for a term which is to be used as a type. A function definition consists of a type declaration, followed by a number of pattern-matching clauses. We use p to stand for a term which is to be used as a pattern:

$$\begin{array}{l}
\texttt{f} : (x_1\ :\ T_1) \texttt{->} (x_2\ :\ T_2\ x_1) \texttt{->} \ldots \texttt{->} \\
\quad (x_n\ :\ T_n\ x_1\ x_2 \ldots x_{n-1}) \texttt{->} T\ x_1\ x_2\ \ldots\ x_n \\
\texttt{f}\ \ p_{1,1}\ \ p_{2,1}\ \ldots\ p_{n,1}\ \ = t_1 \\
\texttt{f}\ \ p_{1,2}\ \ p_{2,2}\ \ldots\ p_{n,2}\ \ = t_2 \\
\ldots \\
\texttt{f}\ \ p_{1,m}\ p_{2,m} \ldots p_{n,m}\ \ = t_m
\end{array}$$

The *type* of each argument may be predicated on the *values* of previous arguments, and the return type may be predicated on the values of any of the arguments. A pattern, $p_{i,j}$, may be an arbitrary term, but can only be matched if it is either a variable or is in head normal form.

2.2 Type Checking Dependent Types

Since full dependent types may include *values*, which may need to be reduced to normal form, type checking exploits the following rules[3], where `Set` represents the type of types:

$$\frac{\Gamma \vdash A, B\ :\ \texttt{Set} \qquad \Gamma \vdash A \rightsquigarrow_\beta A' \qquad \Gamma \vdash B \rightsquigarrow_\beta A'}{\Gamma \vdash A \simeq B}$$

$$\frac{\Gamma \vdash A, B\ :\ \texttt{Set} \qquad \Gamma \vdash x\ :\ A \qquad \Gamma \vdash A \simeq B}{\Gamma \vdash x\ :\ B}$$

The context Γ records the types and values of all names in scope, as is standard. The first rule defines the **conversion** relation. If two terms A and B have a common redex A', then they are β-convertible. The second rule states that β-convertible terms can be interchanged within types. Two key implications of these rules are:

1. Implementing a type checker for a dependently-typed language *requires* an evaluator which can compute under binders.
2. In order to ensure termination of type checking (and therefore decidability), we must distinguish terms for which evaluation definitely terminates, and those for which it may not.

We take a simple but effective approach to termination checking: any functions that do not satisfy a simple syntactic constraint on recursive calls will not be reduced by the type checker. The constraint we use is that each recursive call must have an argument that is structurally smaller than the input argument in the same position, and that these arguments must belong to a strictly positive data type. We check for *totality* by additionally ensuring that the patterns cover all possible cases.

[3] The full type-checking rules for core IDRIS may be found elsewhere [8]. The type checker follows standard methods for implementing a dependently typed lambda calculus [11, 25].

$$
\begin{array}{lcl}
\mathcal{E}[\![\mathtt{Set}]\!] & = & \mathtt{Set} \\
\mathcal{E}[\![x]\!] & = & x \\
\mathcal{E}[\![\backslash x : T \texttt{ => } t]\!] & = & \backslash x : \mathcal{E}[\![T]\!] \texttt{ => } \mathcal{E}[\![t]\!] \\
\mathcal{E}[\![(x \ : \ T_1) \texttt{ -> } T_2]\!] & = & (x \ : \ \mathcal{E}[\![T_1]\!]) \texttt{ -> } \mathcal{E}[\![T_2]\!] \\
\mathcal{E}[\![t_1 \ t_2]\!] & = & \mathcal{F}[\![\mathcal{E}[\![t_1]\!] \ t_2]\!] \\
\mathcal{E}[\![\mathtt{c}]\!] & = & \mathtt{c} \\
\mathcal{E}[\![\mathsf{D}]\!] & = & \mathsf{D}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{F}[\![(\backslash x : T \texttt{ => } t_1) \ t_2]\!] & = & \mathcal{E}[\![t_1[t_2/x]]\!] \\
\mathcal{F}[\![\mathtt{f} \ \vec{t}]\!] & = & \mathcal{E}[\![\phi(e)]\!] \\
\quad \underline{\text{if}} \quad \mathtt{f} \ \vec{p} & = & e \qquad \text{defined in } \Gamma \\
\quad \mathsf{Yes} \ \phi & = & \text{MA}\vec{\text{TCH}}(\vec{p}, \mathcal{E}[\![\vec{t}]\!]) \\
\mathcal{F}[\![t]\!] & = & t
\end{array}
$$

$$
\begin{array}{lcl}
\text{MATCH} \ (\mathtt{c} \ \vec{p}) \ (\mathtt{c} \ \vec{t}) & = & \text{MA}\vec{\text{TCH}}(\vec{p}, \vec{t}) \\
\text{MATCH} \ x \quad t & = & \mathsf{Yes} \ [t/x] \\
\text{MATCH} \ \_ \quad \_ & = & \mathsf{No}
\end{array}
$$

Figure 1. Sketch of the evaluation function $\mathcal{E}[\![\cdot]\!]$

2.3 Evaluation

The evaluator used by the type checker implements β-reduction and pattern matching. A sketch of the evaluation function $\mathcal{E}[\![\cdot]\!]$ is given in Figure 1. To keep the presentation simple, the sketch uses a substitution-based approach. In practice, however, for efficiency reasons we will use an environment and *de Bruijn* indexed variables. In the evaluator we use $\vec{t}$ to denote a telescope of arguments $t_1 \ldots t_n$. We define a function $\mathcal{F}[\![\cdot]\!]$, which evaluates a function application, using β-reduction and pattern-matching definitions, where possible, and an overall evaluation function $\mathcal{E}[\![\cdot]\!]$ which effectively implements structural closure of function application. MATCH implements pattern matching of an argument against a pattern, returning a substitution if matching succeeds, with $\vec{\text{MATCH}}$ being the obvious lifting across a telescope of arguments.

2.4 Implicit Arguments

IDRIS adds a layer of syntactic sugar to the core type theory, including **implicit arguments**. Where the value of an argument can be determined from its type, or from another argument, it can be omitted. For example, vectors (lists with the length expressed in the type) are defined as follows:

```
data Nat = O | S Nat;

infixr 5 ::;
data Vect : Set -> Nat -> Set where
   VNil : Vect A O
 | (::) : A -> Vect A k -> Vect A (S k);
```

The `VNil` constructor has an implicit argument `A`, and the `::` constructor has two implicit arguments, `A` and `k`. Written out in full, the types are:

```
    VNil : (A:Set) -> Vect A O
    (::) : (A:Set) -> (k:Nat) -> A -> Vect A k ->
           Vect A (S k);
```

Similarly, we can leave arguments implicit in function definitions:

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

Written in full, using the prefix form of `::`, this would be:

```
vadd : (n:Nat) -> Vect Int n -> Vect Int n ->
       Vect Int n;
vadd O (VNil Int) (VNil Int) = VNil Int;
vadd (S k) ((::) Int k x xs) ((::) Int k y ys)
     = (::) Int k (x + y) (vadd k xs ys);
```

The values for implicit arguments, in types, patterns and function applications, are inferred from explicit values by unification. For each implicit argument IDRIS adds a place holder (_) to the term in the type theory, which is then filled in by IVOR's type checker. It is important to consider *fully explicit* terms when type checking and reasoning about meta-theory. A programmer, however, need not be concerned that these additional arguments affect performance: the back end *erases* computationally irrelevant information [6, 9]. In the case of `Vect` and `vadd` above, all implicit arguments are erased from the definition of `Vect`, and implicit arguments to `vadd` are marked as unused, so a dummy value is passed instead.

This is an instance of the phase distinction between compile-time and run-time affecting the implementation. Implicit arguments, although they are available at run-time, are normally present primarily for type correctness and unused at run-time. In a function $\mathtt{f} \ : \ (x \ : \ \vec{T}) \texttt{ -> } T'$ we consider an argument x_i **unused** if both of the following conditions hold:

1. It is implicit (i.e. its value can be determined by the value of another argument at compile-time by unification).
2. It is not used on the right hand side of the definition, except in an unused argument position.

The first condition ensures that the argument's value will not affect case distinction — another argument suffices. The second condition ensures that it will not be used for case distinction elsewhere.

2.5 Type Checking and Partial Evaluation

There is an important implication of the typing rules:

If a language has full dependent types, it requires an evaluator.

This evaluator could take several forms, but it must provide compile-time β-reduction and pattern matching to implement conversion, from which it is simple to extend to full normalisation. So, if we have a function `f` with statically-known arguments `s` and dynamic arguments `d`, we can create a specialised version (the **residual** program) by normalising an expression of the form:

```
\d => f s d
```

A standard example (e.g. [38]) is the `power` function:

```
power : Int -> Nat -> Int;
power x O = 1;
power x (S k) = x * power x k;
```

We can specialise the `power` function for a particular exponent. For example, if the first argument `x` is dynamic, and the second argument has the statically-known value `S (S (S (S O)))` — i.e. 4 — the IDRIS built-in evaluator, using β-reduction and pattern matching alone, gives us the residual program:

```
     \ x => power x (S (S (S (S O))))
==> \ x : Int => x*(x*(x*(x*1))) : Int -> Int
```

In the rest of this paper, we will make extensive use of this built-in evaluator, and introduce some simple methods that can be used to control its behaviour in order to make partial evaluation effective.

3. Embedding Languages in Idris

In this section, we demonstrate how we can implement EDSLs efficiently using partial evaluation by showing the implementation of a simple expression language embedded in IDRIS.

```
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
     f0 : 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;
}
```

Figure 2. The Simple Functional Expression Language, **Expr**.

3.1 A Simple Expression EDSL, Expr

A common introductory example for dependently-typed languages is a well-typed interpreter [2, 8, 31], where the type system ensures that only well-typed source programs can be represented and interpreted. Figure 2 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, `Val : (x:Int) -> Expr G TyInt` indicates that literal values have integer types (`TyInt`), and `Var : (i:Fin n) -> Expr G (vlookup i G)` 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.

In order to evaluate this language, we will need to keep track of the *values* of all variables that are in scope. Environments, `Env`, allow us to link a vector of types with instances of those types. They are indexed by a vector of types `Vect Ty n`. Each element in the environment corresponds to an element in the vector:

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

We provide operations to lookup/update an environment, corresponding to lookup and update on vectors:

```
vlookup   : (i:Fin n) -> Vect A n -> A;
envLookup : (i:Fin n) -> Env G -> interpTy (vlookup i G);

update    : (i:Fin n) -> A -> Vect A n -> Vect A n;
updateEnv : Env G -> (i:Fin n) -> interpTy T ->
            Env (update i T G);
```

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

Figure 3. Expression Language Interpreter

```
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;

interp env (If v t e) =
       if (interp env v) then (interp env t)
                         else (interp env e);
```

Figure 4. Booleans and `If` construct

The full interpreter for **Expr** is given in Figure 3. Note that its return type depends on the type of the expression to be interpreted. This is a significant benefit of dependent types for language implementation — there is no need to *tag* the result of the interpreter with a type, because its type is *known* from the input program.

3.2 Example Programs

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 using the IDRIS `+` primitive.

```
add : Expr G (TyFun TyInt (TyFun TyInt TyInt));
add = Lam (Lam (Op (+) (Var (fS f0)) (Var f0)));
```

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

```
double : Expr G (TyFun TyInt TyInt);
double = Lam (App (App add (Var f0)) (Var f0));
```

Now, `runDouble` applies `double` to an argument:

```
runDouble : Expr VNil TyInt;
runDouble = App double (Val 21);
```

We run this program by interpreting it in an empty environment:

```
    interp Empty runDouble
==> 42 : Int
```

Running the interpreter yields a host language representation of the EDSL program. In the example above, the program had type `TyInt`, so the value returned was of type `Int`. The value to be returned is computed from the representation type of the expression. So, if we were to evaluate something with a function type, we would obtain an IDRIS *function*. For example, for `add`:

```
    interp Empty add
==> \ x : Int => \ x0 : Int => x+x0 : Int -> Int -> Int
```

3.3 Control structures and recursion

To make **Expr** more realistic, we will add boolean values and an `If` construct, and attempt to write a recursive function. Our extensions are shown in Figure 4. We can now define a factorial function:

```
fact : Expr G (TyFun TyInt TyInt);
fact = Lam (If (Op (==) (Val 0) (Var f0)) (Val 1)
           (Op (*) (Var f0)
             (App fact (Op (-) (Var f0) (Val 1)))));
```

Unfortunately, we cannot specialise an interpreter with respect to this definition: it is not structurally recursive, and if we try to evaluate it, the recursive call to `fact` will unfold forever. Evaluation of the interpreter with this definition will only terminate if it is given a concrete argument and the recursive call is evaluated lazily.

4. Partial Evaluation

In general, a partial evaluator produces specialised versions of functions where some arguments are statically known. We are interested in the instance where the function to specialise is an EDSL interpreter, and the statically known argument is an input program. As we have seen, evaluating the interpreter with specific input programs yields specialised versions of those programs in the host language. This is, of course, not surprising: it is the first *Futamura projection* [16] — specialising an interpreter for given source code yields an executable. However, two problems arise if we simply use the standard evaluator:

1. Recursive programs with dynamically known inputs cannot be specialised since recursive calls are unfolded arbitrarily deeply;
2. Evaluating completely, unfolding every function definition, can lead to loss of sharing.

The first problem is illustrated by `fact`, in which the number of times to unfold the recursive definition is dynamically known. The second problem is illustrated by an application of `double` to a complex expression:

```
doubleBig : Expr VNil TyInt;
doubleBig = App double complexFn;
```

Assuming `complexFn` evaluates to `complexExpr`, evaluating `doubleBig` gives:

```
complexExpr + complexExpr
```

The large expression `complexFn` is evaluated twice in the residual code, where it would make more sense to evaluate it once, and then pass its result to the evaluated version of `double`. This is a similar problem to that which arises when inlining functions [32] — there is needless duplication of work in the evaluated code.

4.1 A Partial Evaluator for Idris

We can significantly improve the results of partial evaluation by taking care with function application. A sketch of the partial evaluator, $\mathcal{P}[\![\cdot]\!]$, is given in Figure 5. This mostly follows the standard evaluator rules given previously, but differs in the $\mathcal{F}'[\![\cdot]\!]$ function used to evaluate function definitions. The partial evaluator $\mathcal{P}[\![\cdot]\!]$ is implemented relative to a context Γ, but unlike the regular evaluator $\mathcal{E}[\![\cdot]\!]$ it *updates* Γ during evaluation.

When applying a function `f`, we separate its arguments $\vec{t}$ into those which are *statically* known (i.e. known at compile-time), $\vec{t}_s$, and those which are *dynamic*, $\vec{t}_d$ (i.e. which will be known at run-time). In IDRIS, we make this distinction through programmer annotations. These annotations define which functions should be partially evaluated, and which arguments of those functions should be treated as static. For `interp`, we declare the function as follows:

```
interp : Env G -> Expr G T [static] -> interpTy T;
```

The `[static]` annotation on the expression argument indicates that the expression may be static in any application of `interp`. Additionally, any implicit argument which can be uniquely inferred

$$
\begin{array}{lcl}
\mathcal{P}[\![\mathsf{Set}]\!] & = & \mathsf{Set} \\
\mathcal{P}[\![x]\!] & = & x \\
\mathcal{P}[\![\backslash x : T \Rightarrow t]\!] & = & \backslash x : \mathcal{P}[\![T]\!] \Rightarrow \mathcal{P}[\![t]\!] \\
\mathcal{P}[\![(x\ :\ T_1) \to T_2]\!] & = & (x\ :\ \mathcal{P}[\![T_1]\!]) \to \mathcal{P}[\![T_2]\!] \\
\mathcal{P}[\![t_1\ t_2]\!] & = & \mathcal{F}'[\![\mathcal{P}[\![t_1]\!]\ t_2]\!] \\
\mathcal{P}[\![\mathsf{c}]\!] & = & \mathsf{c} \\
\mathcal{P}[\![\mathsf{D}]\!] & = & \mathsf{D}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{F}'[\![(\backslash x : T \Rightarrow t_1)\ t_2]\!] & = & \mathcal{P}[\![t_1[t_2/x]]\!] \\
\mathcal{F}'[\![\mathtt{f}\ \vec{t}]\!] & = & \langle \mathtt{f}, \vec{t}_s \rangle\ \vec{t}_d
\end{array}
$$

$$
\begin{array}{lcll}
\underline{\text{if}} & \mathtt{f} : \vec{T} \to T' & & \\
& \mathtt{f}\ \vec{p} & = e & \text{defined in } \Gamma \\
& \mathsf{Yes}\ \phi & = \textsc{match}(\vec{p}_s, \mathcal{P}[\![\vec{t}_s]\!]) & \\
\underline{\text{add}} & \langle \mathtt{f}, \vec{t}_s \rangle\ :\ \phi(\vec{T}_d) \to \phi(\vec{T}') & & \\
& \langle \mathtt{f}, \vec{t}_s \rangle\ \phi(\vec{p}_d) & = \mathcal{P}[\![\phi(e)]\!] & \text{to } \Gamma \\
\mathcal{F}'[\![t]\!] & & = t &
\end{array}
$$

$$
\begin{array}{lcl}
\textsc{match}\ (\mathsf{c}\ \vec{p})\ (\mathsf{c}\ \vec{t}) & = & \textsc{match}(\vec{p}, \vec{t}) \\
\textsc{match}\ x\ \ t & = & \mathsf{Yes}\ [t/x] \\
\textsc{match}\ \_\ \ \_ & = & \mathsf{No}
\end{array}
$$

Figure 5. Partial Evaluator

from a static argument may also be static. In this case, `T` can be uniquely inferred from the expression. `G` may not be considered static, because it can also be inferred from the (dynamic) environment. We therefore define the **static** arguments in an application to be those in a position annotated as `[static]` or uniquely inferrable from an argument in a position annotated as `[static]`, *and* in head normal form, or a global definition.

The evaluator constructs a new version of `f`, $\langle \mathtt{f}, \vec{t}_s \rangle$, that is specialised with the static arguments $\vec{t}_s$, reusing this definition if it already exists. The type of $\langle \mathtt{f}, \vec{t}_s \rangle$ is constructed by specialising the type according to the statically known values — these values may appear in the dynamic portion of the type. This new definition is added, permanently, to the global context Γ. Effectively, this caches the intermediate result of a computation with specific static arguments. In this way, we abstract away multiple calls to a given specialised definition. In particular, this means that specialising the interpreter with `fact` will result in a recursive IDRIS program.

The method we use for specialising function applications, namely extending the environment with cached versions of partially evaluated functions, is a standard technique [13]. The new definitions are well-typed and preserve the semantics of the original program. However, some care is required in the presence of fully explicit dependently typed programs, because the values of implicit arguments may make a definition less generic than required. When constructing a new function $\langle \mathtt{f}, \vec{t}_s \rangle$ we aim to produce the *most generic* definition possible. We achieve this by replacing any implicit, unused arguments with a place holder before adding the definition, as demonstrated by the example below.

Example — Factorial

When the interpreter is partially evaluated with `fact` as a static argument, according to the scheme in Figure 5, we obtain:

```
interp : (n:Nat) -> (T:Ty) -> (G:Vect Ty n) ->
         Env G -> Expr G T -> interpTy T
fact   : (n:Nat) -> (G:Vect Ty n) ->
        Expr G (TyFun TyInt TyInt)
```

For the remainder of this section, we write applications in fully explicit form. `interp` has implicit arguments for the expression type and context, and `fact` has implicit arguments for its initial context.

Partially evaluating the interpreter happens as follows. The type and expression are in static argument positions, so the evaluator creates a new definition `interpfact` that has been specialised according to these arguments and adds it to the context:

```
    interp 0 (TyFun TyInt TyInt) VNil Empty (fact 0 VNil)
==> interpfact 0 VNil Empty
```

The new definition replaces arguments in implicit positions in the original application with place holders:

```
interpfact : (n:Nat) -> (G:Vect Ty n) -> Env G ->
             Int -> Int;
interpfact n G e = interp _ _ _ _ (fact _ _);
```

Type checking this leads to the following definition with the implicit arguments filled in:

```
interpfact : (n:Nat) -> (G:Vect Ty n) -> Env G ->
             Int -> Int;
interpfact n G e
     = interp n (TyFun TyInt TyInt) G e (fact n G);
```

The next step is to partially evaluate the definition of `interpfact`. Eventually, this reaches another call to `interp`:

```
interpfact : (n:Nat) -> (G:Vect Ty n) -> Env G ->
             Int -> Int;
interpfact n G e = \ x : Int =>
   if (0==x) then 1 else
      interp (S n) (TyFun TyInt TyInt) (TyInt::G)
             (Extend x e) (fact (S n) G) (x-1);
```

When creating a specialised version of `interp` there is already a suitable definition of `interpfact` that can be reused:

```
interpfact : (n:Nat) -> (G:Vect Ty n) -> Env G ->
             Int -> Int;
interpfact n G e = \ x : Int =>
    if (0==x) then 1 else
       interpfact (S n) (TyInt::G) (Extend x e) (x-1);
```

This definition builds a context and an environment, which are *unused* (as defined in Section 2.4). IDRIS notes this and replaces these arguments with dummy place holder values:

```
interpfact : (n:Nat) -> (G:Vect Ty n) -> Env G ->
             Int -> Int;
interpfact _ _ _ = \ x : Int =>
    if (0==x) then 1 else interpfact _ _ _ (x-1);
```

4.2 Why Taglessness Matters

As mentioned above, one important feature of an interpreter that has been defined in this style is that there is no need to *tag* the return value with its type, because the type can be computed in advance. This is a common feature of interpreters in dependently-typed languages [2, 8, 31]. Effectively, we use the type checker for the host language to check the terms in the object language, so that there is no need to check types dynamically. This leads us to a concrete rule, to be followed by the EDSL author, for maximising the effect of partial evaluation:

> **Rule 1:** Index the EDSL representation by its type, to avoid needing to tag the result.

The tag elimination problem, in which an evaluator aims to move type checking of intermediate results in the interpreter from run-time to compile-time, has been extensively studied in the partial evaluation literature [10, 21, 39, 40]. The presence of dependent types simplifies the implementation of a tagless interpreter greatly, in that we are able to write it in a natural style (avoiding, for example, continuation passing style), with no post-processing required.

```
data Ty = TyUnit | TyBool | TyLift Set;

interpTy : Ty -> Set;
interpTy TyBool = Bool;
interpTy TyUnit = ();
interpTy (TyLift A) = A;

data Imp : Ty -> Set where
   ACTION : IO a -> Imp (TyLift a)
 | RETURN : interpTy a -> Imp a
 | WHILE  : Imp TyBool -> Imp TyUnit -> Imp TyUnit
 | IF     : Bool -> Imp a -> Imp a -> Imp a
 | BIND   : Imp a -> (interpTy a -> Imp b) -> Imp b
```

Figure 6. A Simple Imperative EDSL, **Imp**.

```
interp : Imp a [static] -> IO (interpTy a);
interp (ACTION io) = io;
interp (RETURN val) = return val;
interp (WHILE add body) =
    while (interp add) (interp body);
interp (IF v thenp elsep) =
    if v then (interp thenp) else (interp elsep);
interp (BIND code k) =
   do { v <- interp code; interp (k v); };
```

Figure 7. Interpreter for **Imp**.

5. EDSLs with State

The **Expr** language and its interpreter demonstrate our general approach to EDSL implementation:

1. Define the data type for the EDSL in the host language;
2. Write an interpreter to evaluate this type, in the host language;
3. Specialise the interpreter with respect to concrete programs.

In practice, however, the languages that interest us will not be as simple as **Expr**. Real programs have state, they may communicate across a network, they may need to read and write files, allocate and free memory, or spawn new threads and processes. To be usable in practice we need to ensure that we can deal with such issues. In this section, we implement a simple EDSL for file manipulation, which demonstrates how our approach can deal with external state, side effects such as I/O, and imperative features in general.

5.1 A Simple Imperative EDSL, Imp

Figure 6 describes a simple imperative EDSL, **Imp**, which includes a means of embedding arbitrary I/O operations (`ACTION`) and pure values (`RETURN`), `WHILE` and `IF` statements, and a monadic bind operation (`BIND`) for sequencing statements. Types in **Imp** are the unit type, `TyUnit`, for operations which do not return any value, such as writing to a file; a boolean type, `TyBool` for intermediate results in control structures, and lifted host language types, `TyLift`, which allow host language functions and arbitrary I/O actions to be embedded in EDSL programs. Figure 7 gives an interpreter.

5.2 A File Management EDSL, File

So far, we have indexed our language representations by the *type* of the programs they represent. This allows us to exploit the host language's type checker to ensure that EDSL programs are well-typed. We can take this idea much further with a dependently-typed host language such as IDRIS, and index EDSL representations not only by the program's type, but also by other properties such as the states of the resources that it uses. To demonstrate this, we implement an EDSL for file management, designed so that type-correct programs have the following informally-stated properties:

```
data File : Vect FileState n ->
            Vect FileState n' -> Ty -> Set  where
...

-- Updated control structures

 | WHILE : (File ts ts TyBool) ->
           (File ts ts TyUnit) -> (File ts ts TyUnit)
 | IF    : (a:Bool) -> (File ts ts b) -> (File ts ts b) ->
           (File ts ts b)

-- File management operations

 | OPEN    : (p:Purpose) -> (fd:Filepath) ->
             (File ts (snoc ts (Open p)) (TyHandle (S n)))
 | CLOSE   : (i : Fin n) -> (OpenH i (getPurpose i ts) ts) ->
             (File ts (update i Closed ts) TyUnit)
 | GETLINE : (i:Fin n) -> (p:OpenH i Reading ts) ->
             (File ts ts (TyLift String))
 | EOF     : (i:Fin n) -> (p:OpenH i Reading ts) ->
             (File ts ts TyBool)
 | PUTLINE : (i:Fin n) -> (str:String) ->
             (p:OpenH i Writing ts) ->
             (File ts ts (TyUnit));
```

Figure 8. The File-Handling Language, **File**

```
interp : Env ts -> File ts ts' T [static] ->
         IO (Env ts' & interpTy T);
...

interp env (WHILE test body) =
     do { while (ioSnd (interp env test))
                (ioSnd (interp env body));
          return (env, II); };

-- File operations

interp env (OPEN p fpath)
   = do { fh <- fopen (getPath fpath) (pMode p);
          return (addEnd env (OpenFile fh), bound); };
interp env (CLOSE i p)
   = do { fclose (getFile p env);
          return (updateEnv env i ClosedFile, II); };
interp env (GETLINE i p)
   = do { str <- fread (getFile p env);
          return (env, str); };
interp env (EOF i p)
   = do { e <- feof (getFile p env); return (env, e); };
interp env (PUTLINE i str p)
   = do { fwrite (getFile p env) str;
          fwrite (getFile p env) "\n";
          return (env, II); };
```

Figure 9. Interpreter for **File**.

- Files must be open before they are read or written;
- Files that are open for reading cannot be written, and files that are open for writing cannot be read;
- Only open files can be closed;
- All files must be closed on exit.

Our first attempt extends **Imp**. The language definition is given in Figure 8, and its interpreter in Figure 9. The interpreter returns a pair of the modified environment, and the value resulting from interpretation, where `(S & T)` is IDRIS notation for a pair of types `S` and `T`. Each command in the language has a more or less direct translation into a host language function. We index programs in **File** over their input and output states, as well as their type. This state is a vector that holds information about whether file handles are open or closed:

```
data Purpose = Reading | Writing;
data FileState = Open Purpose | Closed;
data File : Vect FileState n ->
            Vect FileState n' -> Ty -> Set  where
```

Environments carry concrete file handles, if the file state indicates that a file is open:

```
data FileHandle : FileState -> Set where
   OpenFile : (h:File) -> -- actual file
              FileHandle (Open p)
 | ClosedFile : FileHandle Closed;

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

Since interpreting **File** threads an environment through the program, we also modify the interpreter so that imperative constructs such as while loops manage the environment. Since the type of `WHILE` indicates that the test and body of the loop cannot modify an environment, we call the interpreter recursively and discard the resulting environment:

```
ioSnd : IO (a & b) -> IO b;
ioSnd p = do { p' <- p; return (snd p'); };

interp env (WHILE test body) =
     do { while (ioSnd (interp env test))
                (ioSnd (interp env body));
          return (env, II); };
```

Types in **File** include the types from **Imp**, extended with a file handle type, `TyHandle`, to carry the number of available file handles.

```
data Ty = TyUnit        -- unit type
        | TyBool        -- booleans
        | TyLift Set    -- host language type
        | TyHandle Nat; -- a file handle
```

These types can be converted into host language types using an `interpTy` function, as before. In the interpreter, we will need to lookup concrete file handles from the environment, so it is convenient to use elements of finite sets as a concrete representation.

```
interpTy : Ty -> Set;
interpTy TyBool = Bool;
interpTy TyUnit = ();
interpTy (TyLift A) = A;
interpTy (TyHandle n) = Fin n;
```

A program which is correct with respect to its file management operations will begin and end with no open file handles. We use `FileSafe T` as a notational convenience to represent a safe program which returns a value of type `T`:

```
FileSafe T = File VNil VNil T;
```

We use `handles` to declare, in advance, the number of file handles we will create, also as a notational convenience. The function `allClosed` returns a list of closed file handles, so that `handles` requires a program which closes all of the file handles it opens:

```
handles : (x:Int) -> File VNil (allClosed x) T ->
                     FileSafe T;
```

```
copyLine : Filepath -> Filepath -> FileSafe TyUnit;
copyLine inf outf = handles 2
   do { fh1 <- OPEN Reading inf;
        fh2 <- OPEN Writing outf;
        str <- GETLINE fh1 ?;
        PUTLINE fh2 str ?;
        CLOSE fh2 ?;
        CLOSE fh1 ?; };
```

Figure 10. Simple **File** program.

```
interpCopyLine : Filepath -> Filepath -> IO ();
interpCopyLine inf outf
                  = IODo (fopen (getPath inf)  "r")
      (\ x : Ptr => IODo (fopen (getPath outf) "w")
     (\ x0 : Ptr => IODo (freadStr x)
   (\ x1 : String => IODo (fputStr x0 x1)
      (\ x2 : () => IODo (fputStr x0 "\n")
      (\ x3 : () => IODo (fclose x0)
      (\ x4 : () => IODo (fclose x)
      (\ x5 : () => IOReturn (Empty, II)))))))));
```

Figure 11. Simple **File** program (interpreted).

5.3 Example 1: Copying a line

Our first example is a very simple program which reads a line from one file and writes it to another (Figure 10). IDRIS provides rebindable `do`-notation, which here uses the `BIND` and `RETURN` operators provided by `File`:

```
do using (BIND, RETURN) {  ...  }
```

`GETLINE`, `PUTLINE` and `CLOSE` each take an additional argument as a proof that the file is open. Since the files are known statically, these proofs are all straightforward. IDRIS provides hooks to the IVOR theorem prover [7] to allow these proofs to be completed, as well as a means to implement decision procedures. As before, we can specialise the interpreter with respect to this program, and obtain a version which calls the I/O operations directly, as in Figure 11. Despite adding an environment for *external* resources, and using side-effecting I/O operations, this partial evaluation proceeds smoothly. Since the interpreter returns a pair of the final environment and a value, the specialised version returns an empty environment. This is a single constructor, so not expensive, but it can easily be removed with a call to `ioSnd`, which can itself be specialised.

Input/Output implementation

The reason we do not have any difficulties with partial evaluation of I/O is because the evaluator does not *execute* I/O operations itself, but rather constructs an *I/O tree* explaining which operations will be executed at run-time. Like Haskell, IDRIS provides an `IO` monad. This is implemented in the style of Hancock and Setzer [18], where an I/O operation consists of a command followed by a continuation that defines how to process the response to that command:

```
data IO : Set -> Set where
   IOReturn : a -> (IO a)
 | IODo : (c:Command) -> (Response c -> IO a) ->
          (IO a);
```

IDRIS defines default `Command` and `Response` structures which allow simple interaction with the outside world, plus calls to C functions. We define a `bind` operation for sequencing I/O operations:

```
bind : IO a -> (a -> IO b) -> IO b;
bind (IOReturn a) k = k a;
bind (IODo c p) k = IODo c (\x => (bind (p x) k));
```

```
copy : Filepath -> Filepath -> FileSafe TyUnit;
copy i o = handles 2
   do { fh1 <- OPEN Reading i;
        fh2 <- OPEN Writing o;
        WHILE (do { e <- EOF (handle fh1) ?;
                    return (not e); })
              (do { str <- GETLINE (handle fh1) ?;
                    PUTLINE (handle fh2) str ?; });
        CLOSE (handle fh1) ?;
        CLOSE (handle fh2) ?; };
```

Figure 12. Copying a file line by line.

We consider `IO` to be an EDSL for describing interaction with the operating system, and the run-time system to be its interpreter. This gives a clean separation between pure values and external operations, and means we can treat IDRIS evaluation as pure, even with side-effecting code.

Why Partial Evaluation Worked

As with the functional language example in Section 3.1, partial evaluation of our example program above yielded a residual program without *any* trace of the interpreter or environment. Firstly, in the host language, I/O operations remain pure, so there is no need to treat them specially. Secondly, we followed a simple rule:

> **Rule 2:** The interpreter must *only* pattern match on the EDSL program to be translated.

The reason for this rule is that the EDSL program is the only static argument, i.e. the *only* thing we know at compile-time. Everything else (including the environment and any additional arguments to the EDSL program) is dynamic, i.e. unknown until the program is run. Therefore, we write the interpreter so that it does not need to match dynamic values. Auxiliary functions *may* match them (indeed, they may *need to*), as long as we follow another rule:

> **Rule 3:** Auxiliary functions which match on *dynamic* data must not be passed EDSL code unless it has been interpreted first.

These rules are *necessary* to eliminate any trace of the interpreter in the residual code, in that they prevent situations which will cause partial evaluation to get stuck, but they are not *sufficient* to guarantee the best results from partial evaluation, as we will see.

5.4 Example 2: Copying a file

In order to increase expressivity, **File** also includes *while*-loops and conditional expressions. For example, Figure 12 shows how our previous example can be extended to copy an entire file.

Partial Evaluation — First attempt

Unfortunately, there is a problem. Using our initial implementation of the interpreter to evaluate the `copy` program, the specialised program includes the fragment shown in Figure 13. At first, partial evaluation removes all of the interpreter and environment overhead. After translating the while-loop, however, the residual code still carries an environment. We have faithfully followed our three rules, so why does this happen? To understand this, we observe that there are several instances of `bind` in the residual program, including the following call: `bind (while (IODo (feof x) ...) ...)`. However, `bind` matches on its first argument, so can be reduced only when its first argument is known. Since `while` will never be reduced by the partial evaluator, because it could loop forever, reduction cannot continue! But we know, both from the type of `WHILE`

```
interpCopy : Filepath -> Filepath -> IO ();
interpCopy i o = IODo (fopen (getPath i) "r")
   (\ x : Ptr => IODo (fopen (getPath o) "w")
  (\ x0 : Ptr => bind (bind (bind (bind (bind
                  (while (IODo (feof x)

     {- ... while loop translation omitted ... -}

   (\ k0 : () => IOReturn
                  (Extend (OpenFile (FHandle x))
                  (Extend (OpenFile (FHandle x0))
                   Empty), II)))

     {- ... further residual code omitted ... -}
```

Figure 13. First (unsuccessful) attempt at partial evaluation.

and the behaviour of the interpreter, exactly what the result of the loop will be (namely, a unit value and an unchanged environment):

```
WHILE : File ts ts (TyLift Bool) ->
        File ts ts TyUnit -> File ts ts TyUnit

interp env (WHILE test body) =
     do { while (ioSnd (interp env test))
                (ioSnd (interp env body));
          return (env,II); };
```

We should be able to persuade the evaluator to continue with what *we* know will be the result of the loop. The solution is to observe that `bind` can reduce if the I/O operation is in constructor form. We therefore include a lifting operation in the `Command` type:

```
data Command : Set where
    ...
  | IOLift : (IO a) -> Command;
```

We also introduce an alternative bind operation:

```
ibind : (IO a) -> (a -> (IO b)) -> (IO b);
ibind c p = IODo (IOLift c) p;
```

Using `ibind` allows evaluation to continue *as long as either the result of the operation is known statically, or it is unused.* If we have an expression `bind c k`, for some arbitrary `c` which is not in constructor form, evaluation cannot proceed. On the other hand, using `ibind` to bind the result of the `c` which we know will not reduce, expands and evaluates as follows:

```
     bind (ibind c p) k
==> bind (IODo (IOLift c) p) k
==> IODo (IOLift c) (\x => bind (p x) k)
```

Evaluation then proceeds with the inner `bind`, with `x` standing for the value returned by the action `c`. As long as `x` is never used in `p`, as with `WHILE` where we already know the environment will be unchanged, the inner `bind` can be reduced. We change the interpreter for `WHILE` as below, and note that the variable `x` is unused in the continuation:

```
interp env (WHILE test body) =
     ibind (while (ioSnd (interp env test))
                  (ioSnd (interp env body)))
           (\x => return (env, II));
```

Partial Evaluation — Successfully

After this change, specialising the `copy` program yields the residual program shown in Figure 14, with the environment eliminated entirely. To understand what has happened, observe that `IO` itself is an EDSL describing *execution*, interpreted by the run-time system, and `bind` is a program transformation operation. We can only

```
interpCopy : Filepath -> Filepath -> IO ();
interpCopy i o = IODo (fopen (getPath i) "r")
   (\ x : Ptr => IODo (fopen (getPath o) "w")
   (\ x : Ptr => IODo (IOLift (while
                         (IODo (feof x)
            (\ x1 : Int => IOReturn (x1==0)))
                         (IODo (freadStr x)
         (\ x2 : String => IODo (fputStr x0 x2)
             (\ x3 : () => IODo (fputStr x0 "\n")
             (\ x4 : () => IOReturn II)))))
   (\ x5 : () => IODo (fclose x)
   (\ x6 : () => IODo (fclose x0)
   (\ x7 : () => IOReturn (Empty, II)))))));
```

Figure 14. Second (successful) attempt at partial evaluation.

successfully partially evaluate an EDSL program if the program itself is *statically* known, and we have broken this by allowing `bind` to transform a program dynamically, based on the environment. Using `ibind`, the environment can be discarded statically. This leads to a further rule:

> **Rule 4:** Ensure that EDSL program construction, generation and transformation can be evaluated statically.

Adding `IOLift` allows more work to be done statically by giving `bind` a constructor form to evaluate in static position where it would not otherwise be available. Therefore this rule has a consequence specific to IDRIS programs using the `Command`/`Response` I/O system:

> **Consequence:** The result of interpreting a control structure should be bound with `ibind` rather than the default `bind`.

5.5 A note on modularity

It is worth observing that the safety of the **File** EDSL requires that only `File`, its constructors and `interp` are exposed as file manipulation operations. If this were not the case, an EDSL programmer would be able to bypass the safety mechanisms given by the EDSL by invoking `fopen`, `fread` and other file manipulation functions directly. This can be achieved, as normal, by not exporting these names to the EDSL programmer.

6. Experimental Results

To assess the value of our partial evaluation approach, we implemented several example programs as EDSLs, and measured their execution time and memory footprint before and after partial evaluation. The example programs that we used were:

`fact` : In **Expr**, the factorial program, repeatedly calculating the sum of all factorials from 1! to 20!. We implemented this both using direct recursion (as previously described) and using tail recursion, and timed both 20,000 and 2,000,000 iterations.

`sumlist` : In **Expr** extended with list processing, calculating the sum of a list of 10,000 elements (iterating 20,000 times).

`copy` : In **File**, copying the contents of a large file, line by line.

`copy_dynamic` : In **File**, copying the contents of several large files, reading the file names from another file.

`copy_store` : In **File**, copying the contents of a large file by storing the entire contents in memory.

`sort_file` : In **File**, sorting the contents of a large file using a tree sort, and writing a new file.

Program	Idris (gen.)		Idris (spec)		Java		C (`gcc -O3`)	
	Time (s)	Space (kb)	Time (s)	Space (kb)	Time (s)	Space (kb)	Time (s)	Space (kb)
`fact` (20K tail-recursive)	8.598	1892	0.017	816	0.081	11404	0.007	292
`fact` (2M tail-recursive)	877.2	1900	1.650	816	1.937	11388	0.653	292
`fact` (2M recursive)	538.7	1888	3.154	816	N/A†	N/A†	N/A†	N/A†
`sumlist` (10K elements)	1148.0	155616	3.181	1604	4.413	12092	0.346	504
`copy`	1.974	1944	0.589	1896	1.770	12764	0.564	296
`copy_dynamic`	1.763	1940	0.507	1900	1.673	12796	0.512	304
`copy_store`	7.650	59872	1.705	51488	3.324	46364	1.159	24276
`sort_file`	7.510	42228	5.205	42180	2.610	32560	1.728	15832

† Java and C versions implemented iteratively

Table 1. Experimental Results

We would, of course, prefer to use real programs, as might be found in the *nofib* suite for Haskell [30], for example. Since IDRIS is a new, experimental, language, however, such a suite does not yet exist, and we have therefore tried to implement a variety of simple benchmarks covering both functional and imperative features, as well as examples which use the host language extensively.

In order to compare our approach with mainstream programming methods, we implemented equivalent programs in Java (version 1.5.0) and C (`gcc` 4.0.1, using `-O3`), following the same algorithms as far as possible and appropriate. Clearly, these results should be treated with some caution, since comparing different language implementations does not always produce completely fair results: different languages are optimised for different tasks; different algorithms work better in some languages than others; and, to some extent, we are also comparing library implementations. Nevertheless, the results provide an indication of the feasibility of our approach for more realistic tasks. The source code for our examples is available at `http://www.cs.st-and.ac.uk/~eb/icfp10/`.

Analysis of our Results

Table 1 gives absolute run times for our example programs. These results were obtained on an Apple MacBook Pro with a 2.8GHz Intel Core 2 Duo processor and 4Gb memory, running Mac OS X 10.5.8, using `time -l`. The times are the reported CPU times (i.e. the actual processing time, rather than wall clock time or system time), and the space is the maximum *resident set size* (i.e. the maximum portion of the process's memory held in RAM).

For each example, specialising the interpreter provides both a significant speedup and a reduction in space usage. The speedup is particularly dramatic for **Expr** programs. There are two likely reasons for this (established using Apple's Shark profiler[4]): firstly, **Expr** is far more fine-grained than **File**, in that it has syntactic forms for variables, values and application, whereas **File** takes advantage of host language constructs; and secondly, recursive calls in non-specialised **Expr** programs must be evaluated *lazily*, with some associated overhead, in order to avoid expanding the abstract syntax tree indefinitely. The speedup is less dramatic, but still significant, for programs in **File**, even for `sort_file` which spends much of its time evaluating host language code. It is worth noting that, for **File**, the main benefit of partial evaluation is in improved execution time rather than reducing space usage.

In most cases, the results of specialising the interpreter produces a program that runs faster than its Java equivalent and also uses significantly less memory. For `sort_file`, this is not the case, however, because the Java version of tree sort allows in-place update (this is safe since we discard the intermediate trees) whereas the IDRIS version, being purely functional, does not. Few of the programs are close to the efficiency of the C equivalents, due to the overhead of the run-time system, but in every case the results are well within an order of magnitude, and in one case (`copy_dynamic`), the partially evaluated interpreter is actually slightly faster than the C version. It is worth noting that the IDRIS compiler and run-time system are at an early stage of development, and do not yet apply well-known optimisations that have been used in production systems. For example, deforestation [17, 43] might improve the performance of `sort_file` and similar programs that build and destroy intermediate structures. Our results are therefore highly encouraging.

[4] `http://developer.apple.com/tools/sharkoptimize.html`

Program	Idris (gen.)		Idris (spec.)	
	Time	Space	Time	Space
`fact` (tail-rec, 2M)	992.15	10904	1.642	816
`sumlist`	709.6	70824	3.161	1604
`copy`	2.048	10976	0.587	10916
`copy_dynamic`	1.847	10948	0.521	10920
`copy_store`	7.576	57944	1.708	57936
`sort_file`	7.593	49840	5.223	40604

Table 2. Space and time results with default heap size 10 Mb

6.1 Garbage Collection

IDRIS compiles to C, using the *Boehm-Demers-Weiser* conservative garbage collector [5] with an initial heap size of 1Mb. As an experiment, we increased the initial heap size to 10Mb, hypothesising that this would lead to faster run times due to fewer calls to the garbage collector, at the expense of a larger memory footprint. The results are shown in Table 2. In fact, they suggest little more than that it is difficult to predict the effect of changing garbage collector parameters. Increasing the heap-size has little positive effect on either execution time or heap usage, other than `sumlist` where there is a significant benefit for the generic version. In general, with a larger heap, while the programs collect less frequently, each collection takes longer. Further (informal) experiments with a hand-written allocator for IDRIS suggest that we could significantly improve execution times by implementing a special-purpose collector, with specific knowledge of the IDRIS memory structure.

6.2 A Larger Example — Network Protocols

Given our encouraging results, we have begun research into implementing verified network protocols, using the EDSL approach and partial evaluation. We have implemented a simple transport protocol [4] as a DSL embedded in IDRIS. Space does not permit a full explanation, but the DSL encodes valid transitions of a state machine, where transitions represent actions such as sending a packet to a remote machine and receiving an acknowledgment, and its representation is parameterised over start and end states. Programs are therefore guaranteed to use valid transitions, and terminate in an

Program	Idris (gen.)		Idris (spec.)	
	Time	Space	Time	Space
`protocol` (20K packets)	0.990	24285	0.751	24011

Table 3. Space and time results for Network Protocol

expected state. The state machine handles error conditions such as timeout and dropped packets. Table 3 gives CPU time and space usage for a test run sending 20,000 packets to a remote machine. Once again, specialisation improves the run time. Profiling suggests that much of the time spent in this example, at this stage of development, is involved in validating packet data. Nevertheless, a significant interpreter overhead is eliminated.

7. Related Work

Domain-specific languages are a recognised technique for improving programmer productivity by providing appropriate notation and abstractions for a particular problem domain [42]. Recently, the Embedded Domain Specific Language (EDSL) approach, in which a DSL is implemented by embedding in a host language, has been gaining in popularity, with Haskell a popular choice as the host language [3, 14, 24]. A common approach to executing these languages is to use a code generation library such as LLVM [23, 41]. We prefer to use partial evaluation, for two reasons: firstly, if we aim for *verification*, a general purpose partial evaluator need only be verified once, rather than verifying a specialised code generator for every compiled program; and secondly, we can produce efficient language implementations more rapidly.

Partial evaluation [20] has been studied for many years, along with related methods such as *multi-stage programming* as implemented in e.g. MetaOCaml [38], Template Haskell [35], or Concoqtion [15]. The idea that an interpreter can be specialised to generate an efficient executable has been known since at least 1971 [16]. It may therefore seem surprising that the technique is not more widely used[5]. However, there are several issues that must be addressed when using partial evaluation in a general setting, e.g. *tag elimination* [39], *termination analysis* [1], *code duplication* [36], and it is also difficult to use with imperative programming and side effects [12]. In contrast, our approach is specifically targeted towards the efficient execution of EDSLs in a dependently-typed purely functional host language, where partial evaluation is effectively β-normalisation. We therefore avoid many of the complexities involved with general solutions.

Our approach makes extensive use of *tagless interpreters*. An alternative approach is to use combinators, rather than data constructors, to build an object language, which can then be partially evaluated [10]. Combined with representation of stronger invariants [22], it is possible that efficient, correct EDSLs could be implemented in a more mainstream functional language. However, the stronger the invariants required, the more likely it is that a stronger type system with full dependent types will be needed.

Finally, *supercompilation* [27, 28] is closely related to partial evaluation. This technique aims to reduce abstraction overhead through compile-time evaluation. We believe that it may be particularly effective when combined with tagless interpreters, and we hope to explore it in future work.

[5] One example where partial evaluation *has* been applied to a realistic problem is Pantheon [34], an implementation of Pan [14] using Template Haskell.

8. Conclusions and Further Work

The underlying motivation for our research is to be able to reason about extra-functional properties of programs, while ensuring good efficiency. We have found that if EDSLs are based on a dependently-typed host language, then they present a promising basis for such reasoning. Moreover, we have established the feasibility of producing *"efficient enough"* implementations this way. Our methods apply not only to EDSLs embedded in IDRIS, but would also apply to those embedded in *any* language with a suitably rich type system, if extended with the `[static]` annotation and caching partial evaluator described in Section 4.1. This includes languages such as Agda, Coq and Concoqtion, or even GHC with recent extensions such as GADTs and open type families [33]. Given the current trend for EDSL development in Haskell, our results suggest that extending GHC's compile-time evaluation machinery to support full partial evaluation would be beneficial.

In particular, through developing several EDSL programs using a variety of language features, we have found that dependently-typed languages such as IDRIS represent a "sweet spot" where partial evaluation is particularly effective. With minimal modification to the evaluator, and minimal annotations to direct the process of evaluation, the efficiency of our programs compares favourably with similar hand-written programs in Java (in fact, they are generally faster with better memory usage), and is not significantly worse than C (generally around 2–3 times slower, and about 3 times the memory usage). Since we have not yet applied standard optimisation techniques, this is highly promising.

The main reason partial evaluation is so effective in our setting is that we can state precisely what the type of a residual program should be, and can allow that type to vary according to the input program. This removes any need for tagging of intermediate values. I/O and side effects also pose no problem because we distinguish computation and execution. Code duplication, which might arise as a result of carelessly expanding definitions, is easily avoided by caching the result of function applications.

There are some obvious limitations to our approach which we hope to address in future work. In particular, an EDSL with higher-order functions would still retain an interpreter in the residual program, because higher-order functions accept functions as *dynamic* arguments, violating our Rule 4. Possible solutions include the use of multi-stage languages, defunctionalisation or run-time type-safe transformation rules.

A more serious limitation is that generated programs can only be as efficient as the underlying host language constructs. For purely functional EDSLs this is not a problem, as there are equivalent constructs in IDRIS, but it would be a problem for a language with, e.g., mutable local variables. For the resource-safe EDSLs that currently interest us, such as those for safe network protocols, we do not anticipate that this will be an issue, but if it is, then it may be possible to remove the overhead, for example by compiling environments specially.

To conclude, partial evaluation in a dependently-typed language enables inefficient EDSL interpretation engines to be scrapped, so achieving both efficiency *and* verifiability. We hope that our techniques and the new opportunities afforded by dependently-typed languages will lead to a renewed interest in partial evaluation.

Acknowledgments

This work was partly funded by the Scottish Informatics and Computer Science Alliance (SICSA), by EU Framework 7 Project No. 248828 (ADVANCE) and by EPSRC grant EP/F030657 (Islay). We thank our colleagues, notably William Cook, James McKinna and Anil Madhavapeddy for several helpful discussions, and the anonymous reviewers for their constructive suggestions on this paper.

References

[1] P. H. Andersen and C. K. Holst. Termination analysis for offline partial evaluation of a higher order functional language. In *Proc. SAS '96: Intl. Symp. on Static Analysis*, pages 67–82. Springer, 1996.

[2] L. Augustsson and M. Carlsson. An exercise in dependent types: A well-typed interpreter. In *Workshop on Dependent Types in Programming*, Gothenburg, 1999. Available from `http://www.cs.chalmers.se/~augustss/cayenne/interp.ps`.

[3] L. Augustsson, H. Mansell, and G. Sittampalam. Paradise: a two-stage DSL embedded in Haskell. In *Proc. ICFP 2008: International Conf. on Functional Programming*, pages 225–228. ACM, 2008.

[4] 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.

[5] H.-J. Boehm, A. J. Demers, Xerox Corporation Silicon Graphic, and Hewlett-Packard Company. A garbage collector for C and C++. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`, 2001.

[6] E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.

[7] E. Brady. Ivor, a proof engine. In *Implementation and Application of Functional Languages 2006*, volume 4449 of *LNCS*, pages 145–162. Springer, 2007.

[8] E. Brady and K. Hammond. A verified staged interpreter is a verified compiler. In *Proc. GPCE '06: Conf. on Generative Programming and Component Engineering*, 2006.

[9] E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs 2003*, volume 3085, pages 115–129. Springer, 2004.

[10] J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

[11] T. Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1-3):167–177, 1996.

[12] S. Debois. Imperative program optimization by partial evaluation. In *Proc. PEPM '04: ACM Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, pages 113–122. ACM, 2004.

[13] B. Delaware and W. R. Cook. Generic operations and partial evaluation using models, 2009. Draft.

[14] C. Elliott, S. Finne, and O. De Moor. Compiling embedded languages. *J. Funct. Program.*, 13(3):455–481, 2003.

[15] S. Fogarty, E. Pasalic, J. Siek, and W. Taha. Concoqtion: indexed types now! In *Proc. PEPM '07: ACM Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, pages 112–121, 2007.

[16] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Comps., Controls*, 2(5):45–50, 1971.

[17] A. Gill. *Cheap deforestation for non-strict functional languages*. PhD thesis, University of Glasgow, January 1996.

[18] P. Hancock and A. Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Proc. CSL 2000: 14th Ann. Conf. of EACSL, Fischbau, Germany, 21–26 Aug 2000, LNCS 1862*, pages 317–331. 2000.

[19] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28A(4), December 1996.

[20] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[21] N. D. Jones. Challenging problems in partial evaluation and mixed computation. *New Gen. Comput.*, 6(2-3):291–302, 1988.

[22] O. Kiselyov and C.-c. Shan. Lightweight monadic regions. In *Proc. Haskell '08: ACM SIGPLAN Symp. on Haskell*, pages 1–12, 2008.

[23] C. Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, December 2002.

[24] S. Lee, M. M. T. Chakravarty, V. Grover, and G. Keller. GPU kernels as data-parallel array computations in Haskell. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPAHM 2009)*, 2009.

[25] A. Löh, C. McBride, and W. Swierstra. A tutorial implementation of a dependently typed lambda calculus, 2010. To appear in Fundam. Inf.

[26] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[27] N. Mitchell. *Transformation and Analysis of Functional Programs*. PhD thesis, University of York, June 2008.

[28] N. Mitchell and C. Runciman. A supercompiler for core Haskell. In *Implementation and Application of Functional Languages 2007*, volume 5083 of *LNCS*, pages 147–164. Springer, May 2008.

[29] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[30] W. Partain. The nofib benchmark suite of Haskell programs. In J. Launchbury and P. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 195–202. Springer, 1992.

[31] E. Pašalíc, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. In *Proc. 2002 International Conf. on Functional Programming (ICFP 2002)*. ACM, 2002.

[32] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4):393–434, September 2002.

[33] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *International Conf. on Functional Programming (ICFP 2008)*, pages 51–62, New York, NY, USA, 2008. ACM.

[34] S. Seefried, M. Chakravarty, and G. Keller. Optimising embedded DSLs using Template Haskell. In *Proc. GPCE '04: Conf. Generative Prog. and Component Eng.*, LNCS. Springer, 2004.

[35] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In *ACM Haskell Workshop*, pages 1–16, Oct. 2002.

[36] K. Swadi, W. Taha, O. Kiselyov, and E. Pasalic. A monadic approach for avoiding code duplication when staging memoized functions. In *Proc. PEPM '06: ACM Symp. on Partial Evaluation and Semantics-based Program Manipulation*, pages 160–169, 2006.

[37] W. Taha. *Multi-stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Inst. of Science and Technology, 1999.

[38] W. Taha. A Gentle Introduction to Multi-stage Programming, 2003. Available from `http://www.cs.rice.edu/~taha`.

[39] W. Taha and H. Makholm. Tag elimination – or – type specialisation is a type indexed effect. In *Subtyping and Dependent Types in Programming, APPSEM Workshop*, 2000.

[40] W. Taha, H. Makholm, and J. Hughes. Tag elimination and jones-optimality. In *PADO '01: Proceedings of the Second Symposium on Programs as Data Objects*, pages 257–275, London, UK, 2001. Springer-Verlag.

[41] D. Terei. Low level virtual machine for Glasgow Haskell Compiler. Bachelor's Thesis, Computer Science and Engineering Dept., The University of New South Wales, Sydney, Australia, 2009.

[42] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages — an annotated bibliography. `http://homepages.cwi.nl/~arie/papers/dslbib/`, 2000.

[43] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.