Compile-Time Views: Predictable Type-Directed Partial Evaluation Without Code Duplication

No Author Given

No Institute Given

Abstract.

Keywords: Partial Evaluation

1 Introduction

Partial evaluation [?] is an optimization technique that identifies *statically known* program parts and pre-computes them at compile time. The compile-time computation yields a *residual program* that does not contain the, previously identified, statically known parts of the program. Partial evaluation has been intensively studied and successfully applied for: removing abstraction overheads in high-level programs [?,?], domain-specific languages [?,?], and converting language interpreters into compilers [?,?,?]. Applying partial evaluation in these domains often improves program performance by several orders of magnitude [?,?].

Unlike other compiler optimizations partial evaluation is not *safe*: it might lead to *code explosion* and might not *terminate*. Due to compile-time execution, computing `fold`s and loops over data structures of static size can produce arbitrarily large programs. Furthermore, in a Turing-complete language assuring termination is undecidable.

Automatically assuring safety of partial evaluators necessarily leads to lack of *predictability*. To illustrate, let us define a function `dot` for computing a dot-product of two vectors that contain numeric values[1].

```
def dot[V:Numeric](v1: Vector[V], v2: Vector[V]): V =
  (v1 zip v2).foldLeft(zero[V]){ case (prod, (cl, cr)) =>
    prod + cl * cr
  }
```

When `dot` is called with vectors of static size (*e.g.* `dot(Vector(2, 4), Vector(1, 10))`) the abstraction overhead of `zip` and `foldLeft` can be completely removed. However, the partial evaluator must apply extensive analysis to conclude that vectors are static in size and that this can be later used to unroll the recursion inside `foldLeft`. Even if the analysis is successful the evaluator must be conservative about unrolling the `foldLeft`. The vector sizes, and thus the produced code,

[1] We use Scala for all code examples in this paper. In order to comprehend the paper the reader is required to know the mere basics of the language

can unacceptably large in a general case. What if we know that vector sizes are relatively small and we would like to predictably unroll `dot` into a flat sum of products?

Lack of predictability and danger of code explosion are the reason that successful partial evaluators [**?**,**?**,**?**,**?**,**?**] are programmer controlled. We categorize the existing solutions in three categories (for further discussion *c.f.* §**??**):

- Programming languages Idris and D provide allow placing the `static` annotation on function arguments. Since `static` is placed on terms, it denotes that the *whole term* is static. This restricts the number of programs that can be expressed, *e.g.* , we could not express in the signature of *dot* that vector parameters are static in size.
- Type-directed partial evaluation [**?**] and Lightweight Modular Staging (LMS) [**?**] use types to communicate the programmer's intent about partial evaluation. By changing the types of parameters to be (*e.g.* `Vector[Rep[T]]` these approaches can express that parameter vectors are statically known.However, they still require existence of two data structures (*e.g.* `Rep[Vector]` and `Vector`. This fosters, costly and hardly maintainable, code duplication.
- MetaOCaml [**?**] places terms in, possibly nested, quotes. Depth of the term in the quotes denotes the stage of the computation where it will be executed. In MetaOCaml we can express the `dot` function, but we have to modify the code of the `dot` function which might not be desirable.

Ideally, a programmer would with a minimal number of annotations be able to: *i)* require that input vectors are of statically known size but polymorphic in their elements, *ii)* without modifying the terms require that all operations on vector arguments are further partially evaluated, *iii)* allow elements of vectors to be generic, and *iv)* reuse the existing implementation of the `Vector` data structure.

The main idea of this paper is to provide a statically typed *compile-time view* of existing data types. The compile-time view makes all operations and non-generic fields partially evaluated on a type. The compile-time view allows programmers to define a single definition of a type. Then the existing types can be promoted to their compile-time duals with the `@ct` annotation at the type level, and with the `ct` function on the term level. Consequently, due to the integration with the type system, the control over partial evaluation is fine-grained and polymorphic and term level promotions obviate code duplication for static data structures.

With our partial evaluator, to require that vectors `v1` and `v2` are static and to partially evaluate the function, a programmer would need to make a simple modification of the `dot` signature:

```
def dot[V: Numeric](v1: Vector[V] @ct, v2: Vector[V] @ct): V
```

This, in effect, requires that only vector arguments (not their elements) are statically known and that all operations on vector arguments will be executed at compile time (partially evaluated). Since, values are polymorphic the result

of the function will either be a dynamic value or a compile-time value. Residual programs of `dot` application for different arguments:

```
// [el1, el2, el3, el4] are dynamic
dot(ct(Vector)(el1, el2), ct(Vector)(el3, el4))
  ↪ el1 * el3 + el2 * el4

dot(ct(Vector)(2, 4), ct(Vector)(1, 10))
  ↪ 2 * 1 + 4 * 10

// ct promotes static terms to compile-time
dot(ct(Vector)(ct(2), ct(4)), ct(Vector)(ct(1), ct(10)))
  ↪ 42
```

In this paper we make the following contributions to the state-of-the-art:

- By introducing the $F_{i<:}$ calculus (§**??**) that in a fine- grained way captures the user's intent about partial evaluation. The calculus is based on $F_{<:}$ with records which makes it suitable for representing modern multi-paradigm languages. Finally, we formally define a partial evaluator for $F_{i<:}$.
- By providing a translation scheme from data types in object oriented languages (polymorphic classes and methods into the dual views in the $F_{i<:}$ calculus (§**??**).
- By demonstrating the usefulness of compile-time views in four case studies (§**??**): partially evaluating recursion, removing overheads of variable arguments functions, removing overheads of type-classes [**?**], and simulating value classes [**?**].

We have implemented a partial evaluator according to the $F_{i<:}$ calculus and the desugaring rules (§**??**). The partial evaluator is implemented for Scala and open-sourced (`https://github.com/scala-inline/`). We have evaluated the performance gains and validity of the partial evaluator on all case studies (§**??**) and compared them to LMS. In all benchmarks our evaluator gives significant performance gains compared to original programs and performs equivalently to LMS.

2 The $F_{i<:}$ Calculus

$t ::=$	Terms:	$S,\ T,\ U ::=$	Types:
$x,\ y$	identifier	$iS \Rightarrow jT$	function type
$(x : iT) \Rightarrow t$	function	$\{\overline{x : iS}\}$	record type
$t(t)$	application	$[X <: iS] \Rightarrow jT$	universal type
$\{\overline{x = t}\}$	record	Any	top type
$t.x$	selection	$iT,\ jT,\ kT,\ lT ::=$	Binding-Time Types:
$[X <: iT] \Rightarrow t$	type abstraction	X	type identifier
$t[iT]$	type application	$T,\ dynamic\ T$	dynamic type
$inline\ t$	inline view	$static\ T$	static type
$v ::=$	Values:	$inline\ T$	inline type
$x \Rightarrow t$	function value	$\Gamma ::=$	Contexts:
$\{\overline{x = t}\}$	record value	$\emptyset$	empty context
		$\Gamma,\ x : iT$	term binding
		$\Gamma,\ X <: iT$	type binding

Fig. 1. Syntax of $F_{i<:}$

We formalize the essence of our inlining system in a minimalistic calculus based on $F_{<:}$ with lazy records. To accommodate predictable partial evaluation we introduce binding-time annotations into the type system as first-class types that represent three kinds of bindings:

1. **Dynamic binding**. These are the types which express computation at runtime. All types written in the end user code are considered to be dynamic by default if no other binding-time annotation is given.
2. **Static binding**. Values of static terms can be computed at compile-time (*e.g.* constant expressions) but at are still evaluated at runtime by default. All language literals are static by default.
3. **Inline binding**. And finally the types that correspond to terms that are hinted to be computed at compile-time whenever possible.

2.1 Composition

An interesting consequence of encoding of binding times as first-class types is ability to represent values which are partially static and partially dynamic.

For example lets have a look at simple record that describes a complex number with two possible representations encoded through *isPolar* flag:

$$complex : static\ \{isPolar : static\ Boolean,\ a : Double,\ b : Double\} \in \Gamma$$

This type is constructed out of a number of components with varying binding times. Representation encoding is known in advance and is static according to the signature. Coordinates a and b do not have any binding-time annotation meaning that they are dynamic.

Given this binding to *complex* in our environment Γ we can use *inline* to obtain a compile-time view to evaluate acess to *isPolar* field at compile-time:

$$inline\ complex.isPolar : inline\ Boolean$$

Any statically known expression can be promoted via *inline*. Selection of dynamic fields on the other hand will return dynamic values despite the fact that record is statically known. In practice this can be used to specialize a particular execution path in the application to a particular representation by selectively inlining statically known parts.

Once you have inline view of the term it's also possible to demote it back to runtime evaluation through *dynamic* view.

Not all type and binding time combinations are correct though. We restrict types to disallow nesting of more specific binding times into less specific ones.

$$\frac{}{\text{wff } iAny} \quad \text{(W-Any)}$$

$$\frac{i <: j \quad i <: k \quad \text{wff } jT_1 \quad \text{wff } kT_2}{\text{wff } i(jT_1 \Rightarrow kT_2)} \quad \text{(W-Abs)}$$

$$\frac{i <: j \quad i <: k \quad \text{wff } jS \quad \text{wff } kT}{\text{wff } i([X <: jS] \Rightarrow kT)} \quad \text{(W-TAbs)}$$

$$\frac{\forall j. \quad i <: j \quad \overline{\text{wff } jT}}{\text{wff } i\{\overline{x : jT}\}} \quad \text{(W-Rec)}$$

Fig. 2. Well formed types wff iT

This restiction allows us to reject programs that have inconsistent annotations. For example the following function has incorrectly annotated parameter binding time:

$$(x : inline\ Int) \Rightarrow x + 1$$

This is inconsistent because the body of the function might not be evaluated at compile-time (as the function is not inline.) As described in (W-Abs) functions may only have parameters that are at most as specific as the function binding-time. In our example this doesn't hold as *inline* is more specific than implicit *static* annotation on function literal.

2.2 Subtyping

Another notable feature of our binding-time analysis system is deep integration with subtyping. We believe that such integration is crucial for an object-oriented language that wants to incorporate partial evaluation.

At core of the subtyping relation we have a subtyping relation on binding-time information with *dynamic* as top binding-time.

$$i <: \mathit{dynamic} \ \text{(I-Dynamic)} \qquad \mathit{inline} <: \mathit{static} \ \text{(I-Static2)}$$
$$\mathit{static} <: \mathit{static} \ \text{(I-Static1)} \qquad \mathit{inline} <: \mathit{inline} \ \text{(I-Inline)}$$

Fig. 3. Binding-time subtyping.

We proceed by threading binding time information throughout regular $F_{<:}$ subtyping rules augmented with standard record types.

$$\Gamma \vdash iS <: \mathit{Any} \quad \text{(S-Top)}$$
$$\Gamma \vdash iS <: iS \quad \text{(S-Refl)}$$
$$\frac{\Gamma \vdash iS <: jU \qquad \Gamma \vdash jU <: kT}{\Gamma \vdash iS <: jU} \quad \text{(S-Trans)}$$
$$\frac{X <: iT \in \Gamma}{\Gamma \vdash X <: iT} \quad \text{(S-TVar)}$$
$$\{x_p : i_p T_p \ ^{p \in 1..n+m}\} <: \{x_p : i_p T_p \ ^{p \in 1..n}\} \quad \text{(S-Width)}$$
$$\frac{\Gamma \vdash kT_1 <: iS_1 \qquad \Gamma \vdash jS_2 <: lT_2}{\Gamma \vdash iS_1 \Rightarrow jS_2 <: kT_1 \Rightarrow lT_2} \quad \text{(S-Arrow)}$$
$$\frac{\forall p \in 1..n.\ i_p S_p <: j_p T_p}{\{x_p : i_p S_p \ ^{p \in 1..n}\} <: \{x_p : j_p T_p \ ^{p \in 1..n}\}} \quad \text{(S-Depth)}$$
$$\frac{i <: j \qquad \Gamma \vdash S <: T}{\Gamma \vdash iS <: jT} \quad \text{(S-Inline)}$$
$$\frac{\Gamma,\ X <: iU_1 \vdash jS_2 <: kT_2}{\Gamma \vdash [X <: iU_1] \Rightarrow jS_2 <: [X <: iU_1] \Rightarrow kT_2} \quad \text{(S-All)}$$
$$\frac{\{x_p : i_p S_p \ ^{p \in 1..n}\} \text{ is permutation of } \{y_p : j_p T_p \ ^{p \in 1..n}\}}{\{x_p : i_p S_p \ ^{p \in 1..n}\} <: \{y_p : j_p T_p \ ^{p \in 1..n}\}} \quad \text{(S-Perm)}$$

Fig. 4. Subtyping.

Integration between binding-time subtyping and subtyping on regular types is expressed through (S-Inline) rule that merges the two into one coherent relation on binding-time types.

2.3 Generics

Crucial consequence of our design choices made in the system manifests in ability to use regular generics as means to abstract over binding-time without any additional language constructs.

For example given a generic identity function:

$$\mathit{identity} : \mathit{static}\ ([X <: \mathit{Any}] \Rightarrow \mathit{static}\ (X \Rightarrow X)) \in \Gamma$$

We can instantiate it to both in static and dynamic contexts through corresponding type application:

$$\begin{gathered} identity[static\ Int] : static\ (static\ Int \Rightarrow static\ Int) \\ identity[Int] : static\ (Int \Rightarrow Int) \end{gathered} \tag{1}$$

In practice this allows us to write code that is polymorphic in the binding time without any code duplication which is quite common in other partial evaluation systems.

This is possible due to the fact that we've integrated binding time information into types and augmented subtyping relation with subtyping

2.4 Typing

To enforce well-formedness of types in a context of partial evaluation we customize standard typing rules with additional constraints with respect to binding time.

$$\frac{x : iT \in \Gamma}{\Gamma \vdash x : iT} \tag{T-Ident}$$

$$\frac{\forall t. \quad \Gamma \vdash t : jT \quad \text{wff } i\{x : jT\}}{\Gamma \vdash i\{\overline{x = t}\} : i\{\overline{x : jT}\}} \tag{T-Rec}$$

$$\frac{\Gamma \vdash t_1 : i(jT_1 \Rightarrow kT_2) \quad \Gamma \vdash t_2 : jT_1}{\Gamma \vdash t_1(t_2) : kT_2} \tag{T-App}$$

$$\frac{\Gamma \vdash t : i\{x = jT_1, \overline{y = kT_2}\}}{\Gamma \vdash t.x : jT_1} \tag{T-Sel}$$

$$\frac{t \text{ is not literal} \quad \Gamma \vdash t : static\ T}{\Gamma \vdash inline\ t : inline\ T} \tag{T-Inline}$$

$$\frac{t \text{ is not literal} \quad \Gamma \vdash t : iT}{\Gamma \vdash dynamic\ t : dynamic\ T} \tag{T-Dynamic}$$

$$\frac{\Gamma \vdash t : iS \quad \Gamma \vdash iS <: jT}{\Gamma \vdash t : jT} \tag{T-Sub}$$

$$\frac{\Gamma,\ x : jT_1 \vdash t : kT_2 \quad \text{wff } i(jT_1 \Rightarrow kT_2)}{\Gamma \vdash i((x : jT_1) \Rightarrow t) : i(jT_1 \Rightarrow kT_2)} \tag{T-Func}$$

$$\frac{\Gamma,\ X <: jT_1 \vdash t_2 : kT_2 \quad \text{wff } i([X <: jT_1] \Rightarrow kT_2)}{\Gamma \vdash i([X <: jT_1] \Rightarrow t_2) : i([X <: jT_1] \Rightarrow kT_2)} \tag{T-TAbs}$$

$$\frac{\Gamma \vdash t_1 : i([X <: jT_{11}] \Rightarrow kT_{12}) \quad \Gamma \vdash lT_2 <: jT_{11}}{\Gamma \vdash t_1[lT_2] : [X \mapsto lT_2]kT_{12}} \tag{T-TApp}$$

Fig. 5. Typing.

The most significant changes lie in:

- Additional checks in literal typing that ensure that constructed values correspond to well-formed types (T-Func, T-Rec, T-TAbs). To do this we typecheck literals together with possible binding-time term that might enclose it.
- New typing rules for binding-time views (T-Inline, T-Dynamic). These rules only cover non-literal terms as composition of binding-time view and literal itself is handled in corresponding typing rule for given literal.

2.5 Partial Evaluation

In order to simplify partial evaluation rules we erase all of the type information before partial evaluation. This means that all functions become function values, type abstraction and application are complete eliminated.

$$\frac{t \rightsquigarrow t'}{x \Rightarrow t \rightsquigarrow x \Rightarrow t'} \quad \text{(PE-Func)}$$

$$\frac{\overline{t} \rightsquigarrow \overline{t'}}{\{\overline{x = t}\} \rightsquigarrow \{\overline{x = t'}\}} \quad \text{(PE-Rec)}$$

$$\frac{t_1 \rightsquigarrow t_1' \quad t_1' \text{ is not inline} \quad t_2 \rightsquigarrow t_2'}{t_1(t_2) \rightsquigarrow t_1'(t_2')} \quad \text{(PE-App)}$$

$$\frac{t_1 \rightsquigarrow \mathit{inline}\ x \Rightarrow t \quad t_2 \rightsquigarrow t_2' \quad [x \mapsto t_2']t \rightsquigarrow t'}{t_1(t_2) \rightsquigarrow t'} \quad \text{(PE-IApp)}$$

$$\frac{t \rightsquigarrow t' \quad t' \text{ is not inline}}{t.x \rightsquigarrow t'.x} \quad \text{(PE-Sel)}$$

$$\frac{t \rightsquigarrow \mathit{inline}\ \{x = t_x,\ \overline{y = t_y}\} \quad t_x \rightsquigarrow t_x'}{t.x \rightsquigarrow t_x'} \quad \text{(PE-ISel)}$$

$$\frac{t \text{ is not literal} \quad t \rightsquigarrow t' \quad t' \Downarrow v}{\mathit{inline}\ t \rightsquigarrow \mathit{inline}\ v} \quad \text{(PE-Inline)}$$

Fig. 6. Partial evaluation $t \rightsquigarrow t'$

2.6 Evaluation

Once partial evaluation is complete we strip all binding-time terms and use regular untyped lambda calculus evaluation rules extended with lazy records.

$$v \Downarrow v \quad \text{(E-Value)}$$

$$\frac{t_1 \Downarrow x \Rightarrow t \quad t_2 \Downarrow v \quad [x \mapsto v]t \Downarrow v'}{t_1(t_2) \Downarrow v'} \quad \text{(E-App)}$$

$$\frac{t \Downarrow \{x = t_x,\ \overline{y = t_y}\} \quad t_x \Downarrow v}{t.x \Downarrow v} \quad \text{(E-Sel)}$$

Fig. 7. Evaluation $t \Downarrow v$

2.7 Conjectures

1. Progress.
2. Preservation.
3. Static terms are closed over statically bound variables.
4. Inline terms will be replaced with canonical value of corresponding type after partial evaluation.

3 Translating Object Oriented Features Into $F_{i<:}$

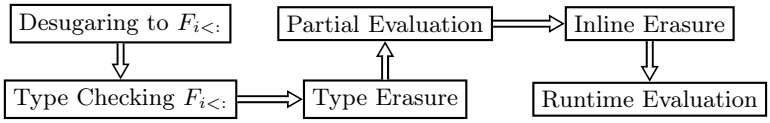

Fig. 8. Compilation pipeline.

The core calculus §?? captures the essence of user-controlled predictable partial-evaluation. In practice, though, it requires an inconveniently large number of `inline` calls. Moreover, the calculus does not provide a way to define data structures that would correspond to *classes* in modern multi-paradigm languages. In this section we formalize convenient implicit conversions for the calculus §??, a scheme for translating classes into the calculus how to promote constructs classes and *methods* into their partially-evaluated versions §??.

The core rules of the calculus do not support effect-full computations and each `inline` term is trivially converted to a dynamic term after erasure. In case of languages that do support mutable state and side-effects this needs to be treated specially. For simplicity, we omit side-effects from our discussion and assume that all partially evaluated code is side-effect free and that each `inline` term can be converted to dynamic code.

3.1 Desugaring Object Oriented Constructs to $F_{i<:}$

3.2 Compile-Time View of Terms

4 Case Studies

In this section we present selected use-cases for compile-time views that demonstrate the core functionality. We start with a canonical example of the power function (§??), then we demonstrate how variable argument functions can be desugared into the core functionality (§??). Finally, we demonstrate how the abstraction overhead of the `dot` function and all associated type-classes can be removed (§??).

$$[\![\textbf{let } x : T_x = t_x \textbf{ in } t]\!] = ((x : T_x) \Rightarrow t)(t_x)$$
$$[\![\textbf{let } type\ T_1 = T_2 \textbf{ in } t]\!] = ([T_1 <: T_2] \Rightarrow t)[T_2]$$
$$[\![\textbf{let } class\ C[A](x : T_x)\{def\ f[B](y : T_y) = t_f\} \textbf{ in } t]\!] =$$
$$\textbf{let } type\ C = [A] \Rightarrow \{x : T_x, f : [B] \Rightarrow T_y \Rightarrow T_f\} \textbf{ in}$$
$$\textbf{let } C : [A] \Rightarrow (x : T_x) \Rightarrow C[A] =$$
$$[A] \Rightarrow (x : T_x) \Rightarrow \{x = x, f = [B] \Rightarrow (y : T_y) \Rightarrow t_f\} \textbf{ in } t$$

Fig. 9. Desugaring of classes into $F_{i<:}$.

$$\frac{\Pi \vdash T \in \Pi}{\Pi \vdash\rightsquigarrow iTiT}\ (\text{CT-TVar}) \qquad \frac{\Pi \vdash T \notin \Pi}{\Pi \vdash\rightsquigarrow iTinline\ T}\ (\text{CT-T-Var})$$

$$\frac{\overline{\Pi \vdash\rightsquigarrow tt'}}{\Pi \vdash\rightsquigarrow i\{\overline{x = t}\}inline\ \{\overline{x = t'}\}}\ (\text{CT-Rec})$$

$$\frac{\overline{\Pi \vdash\rightsquigarrow iTjT}}{\Pi \vdash\rightsquigarrow \{\overline{x : iT}\}inline\ \{\overline{x : jT}\}}\ (\text{CT-T-Rec})$$

$$\frac{\Pi \vdash\rightsquigarrow iTjT \quad \Pi \vdash\rightsquigarrow kSlS}{\Pi \vdash\rightsquigarrow iT \Rightarrow kSjT \Rightarrow lS}\ (\text{CT-T-Arrow})$$

$$\frac{\Pi \vdash\rightsquigarrow jTkT}{\Pi \vdash\rightsquigarrow [X <: iS] \Rightarrow jT[X <: iS] \Rightarrow kT}\ (\text{CT-T-Univ})$$

$$\frac{\Pi \vdash\rightsquigarrow tt' \quad \Pi \vdash\rightsquigarrow iTjT}{\Pi \vdash\rightsquigarrow i(x : iT) \Rightarrow tinline\ (x : jT) \Rightarrow t'}\ (\text{CT-Func})$$

$$\frac{\Pi,\ X \vdash\rightsquigarrow tt'}{\Pi \vdash\rightsquigarrow i([X <: jT_1] \Rightarrow t)inline\ ([X <: jT_1] \Rightarrow t')}\ (\text{CT-TAbs})$$

$$\frac{\Pi \vdash\rightsquigarrow tt' \quad \Pi \vdash\rightsquigarrow iTjT}{\Pi \vdash\rightsquigarrow t[iT]t'[jT]}\ (\text{CT-TApp})$$

Fig. 10. Translation of a regular class to a compile-time version.

4.1 Recursion

The canonical example in partial evaluation is the computation of the integer power function:

```
def pow(base: Double, exp: Int): Double =
  if (exp == 0) 1 else base * pow(base, exp)
```

When the exponent (`exp`) is statically known this function can be partially evaluated into `exp` multiplications of the `base` argument, significantly improving performance [].

With compile-time views making `pow` partially evaluated requires adding two annotations:

```
@inline def pow(base: Double, exp: Int @ct): Double =
  if (exp == 0) 1 else base * pow(base, exp)
```

`@inline` denotes that the `pow` function it self must be inline and `@ct` requires that the `exp` argument is a compile-time view of `Int`.

4.2 Variable Argument Functions

- `@i?` in argument position is a macro that expands the function to an underlying function `@i? def min_underlying[T: Numeric](values:Seq[T]@i?): T` and a macro that will call it according to the input parameters.
- Comparison to other approaches.

```
@inline def min[T: Numeric](@ct values:T*): T =
  values.tail.foldLeft(values.head)((min, el) => if (el < min) el else min)
```

Fig. 11. Function for computing the non-negative power of a real number.

4.3 Inline Classes: Butterfly Networks

- Reference LMS. Discuss @i! annotation on classes. Works for both dynamic and static inputs.
- Comparison to LMS. Mention a pervasive number of annotations. Discuss duality of Exp[T] and @i!.

4.4 Dot Product

- Explain the removal of type classes together with inline. Explain how type classes are @i? and how they will completely evaluate if they are passed a static value.
- Comparison to other approaches.

```
object Numeric {
  @i! implicit def dnum: Numeric[Double] @i! = DoubleNumeric
  @i! def zero[T](implicit num: Numeric[T]): T = num.zero
  object Implicits {
    @i! implicit def infixNumericOps[T](x: T)(implicit num: Numeric[T]): Numeric[T]#Ops = new
  }
}
trait Numeric[T] {
  def plus(x: T, y: T): T
  def times(x: T, y: T): T
  def zero: T

  @i! class Ops(lhs: T) {
    @i! def +(rhs: T) = plus(lhs, rhs)
    @i! def *(rhs: T) = times(lhs, rhs)
  }
}

object DoubleNumeric extends Numeric[Double] {
  @i! def plus(x: Double @i?, y: Double @i?): Double = x + y
  @i! def times(x: Double @i?, y: Double @i?): Double = x * y
  @i! def zero: Double = 0.0
}
```

Fig. 12. Function for computing the non-negative power of a real number.

5 Evaluation

6 Related Work

7 Conclusion