Type-Driven 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. Partial evaluation has been intensively studied and successfully applied: for removing abstraction overheads in high-level programs [?,?], for domain-specific languages [?,?], and for converting language interpreters into compilers [?,?,?]. Applying partial evaluation in these domains often improves program performance by several orders of magnitude [?,?].

To achieve *predictable* and *safe* partial evaluation, however, a partial evaluator must be instructed by the programmer [?,?]. Unlike other compiler optimizations, due to compile-time execution, partial evaluation might not *terminate*. Furthermore, *code explosion* is possible as the final program can be arbitrarily large due to compile-time execution. Lack of predictability and danger of code explosion are the reason that successful partial evaluators [?,?,?,?] are programmer controlled.

To show how programmers can control partial evaluation we define a function `dot` for computing a dot-product of two vectors that contain numeric values.

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

In order to remove the abstraction overhead of `zip` and `foldLeft` 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` as vector sizes, and thus the produced code, can be very large. What

[0] 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

if we know that vector sizes are relatively small and we would like to predictably unroll `dot` into a flat sum of products?

Ideally, the programmer would with a minimal number of annotations be able to: *i)* require that input vectors are of statically known size, *ii)* require that all operations on vector arguments should be further partially evaluated, *iii)* allow elements of vectors to be generic, and *iv)* reuse the existing implementation of the `Vector` data structure.

[TODO: expand] The main idea of this paper is to explicitly capture the user intent about partial evaluation in the types. We annotate every type in the language with one of the three values:

- *dynamic* signifies that the value of the type is not known at compile-time. In code *dynamic* is represented as `@d`.
- *static* signifies that the value is known at compile-time. In code *static* is represented as `@s`.
- *inline* requires that the type is statically known and guarantees that operations on the term will be partially evaluated. In code *inline* is represented as `@i`.

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] @i, v2: Vector[V] @i): V
```

This, in effect, requires that only vector arguments (not their elements) are statically known and that all operations will be inlined and further partially evaluated. Residual programs of `dot` application in different cases are:

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

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

// inline promotes static terms into inline
dot(Vector(inline(2), inline(4)), Vector(inline(1), inline(10)))
  ↪ 42
```

Predictable partial evaluation has been a goal of many projects (§**??**) which can be roughly categorized into following categories:

- Programming languages Idris and D provide annotations `static` that can be placed on function arguments. In these systems static parameters are required to be deeply static and, thus, the `dot` function could not be expressed.
- Type-directed partial evaluation [**?**] and LMS [**?**] use types to communicate the programmers intent about partial evaluation and, thus, can express `dot` function. These approaches, however, require the programmer to implement two versions of `Vector` and other operations. This fosters, costly and hardly maintainable, code duplication.

– MetaOCaml [**?**]

Contributions:
Evaluation:
Sections:

2 Formalization $F_{i<:}$

$t ::=$	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
$dynamic\ t$	dynamic view	$inline\ T$	inline type
$v ::=$	Values:	$\Gamma ::=$	Contexts:
$x \Rightarrow t$	function value	$\emptyset$	empty context
$\{\overline{x = t}\}$	record value	$\Gamma,\ x : iT$	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.

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

$$\frac{i <: j \quad i <: k \quad \text{wff } jS \quad \text{wff } kT}{\text{wff } i([X <: jS] \Rightarrow kT)}\ (\text{W-TAbs}) \qquad \frac{\forall j. \quad i <: j \quad \overline{\text{wff } jT}}{\text{wff } i\{\overline{x : jT}\}}\ (\text{W-Rec})$$

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.

$$\Gamma \vdash iS <: \mathit{Any}\ \text{(S-Top)} \qquad \Gamma \vdash iS <: iS\ \text{(S-Refl)}$$

$$\frac{\Gamma \vdash iS <: jU \quad \Gamma \vdash jU <: kT}{\Gamma \vdash iS <: jU}\ \text{(S-Trans)}$$

$$\frac{X <: iT \in \Gamma}{\Gamma \vdash X <: iT}\ \text{(S-TVar)}$$

$$\{x_p : i_p T_p\ ^{p \in 1..n+m}\} <: \{x_p : i_p T_p\ ^{p \in 1..n}\}\ \text{(S-Width)}$$

$$\frac{\Gamma \vdash kT_1 <: iS_1 \quad \Gamma \vdash jS_2 <: lT_2}{\Gamma \vdash iS_1 \Rightarrow jS_2 <: kT_1 \Rightarrow lT_2}\ \text{(S-Arrow)}$$

$$\frac{\forall p \in 1..n.\ i_p S_p <: j_p T_p}{\{x_p : i_p S_p\ ^{p \in 1..n}\} <: \{x_p : j_p T_p\ ^{p \in 1..n}\}}\ \text{(S-Depth)}$$

$$\frac{i <: j \quad \Gamma \vdash S <: T}{\Gamma \vdash iS <: jT}\ \text{(S-Inline)}$$

$$\frac{\Gamma,\ X <: iU_1 \vdash jS_2 <: kT_2}{\Gamma \vdash [X <: iU_1] \Rightarrow jS_2 <: [X <: iU_1] \Rightarrow kT_2}\ \text{(S-All)}$$

$$\frac{\{x_p : i_p S_p\ ^{p \in 1..n}\} \text{ is permutation of } \{y_p : j_p T_p\ ^{p \in 1..n}\}}{\{x_p : i_p S_p\ ^{p \in 1..n}\} <: \{y_p : j_p T_p\ ^{p \in 1..n}\}}\ \text{(S-Perm)}$$

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:

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

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.

$$\frac{x : iT \in \Gamma}{\Gamma \vdash x : iT} \text{ (T-Ident)} \qquad \frac{t \text{ is not literal} \quad \Gamma \vdash t : \mathit{static}\ T}{\Gamma \vdash \mathit{inline}\ t : \mathit{inline}\ T} \text{ (T-Inline)}$$

$$\frac{\forall t. \quad \Gamma \vdash t : jT \quad \text{wff } i\{x : jT\}}{\Gamma \vdash i\{\overline{x = t}\} : i\{\overline{x : jT}\}} \text{ (T-Rec)} \qquad \frac{t \text{ is not literal} \quad \Gamma \vdash t : iT}{\Gamma \vdash \mathit{dynamic}\ t : \mathit{dynamic}\ T} \text{ (T-Dynamic)}$$

$$\frac{\Gamma \vdash t_1 : i(jT_1 \Rightarrow kT_2) \quad \Gamma \vdash t_2 : jT_1}{\Gamma \vdash t_1(t_2) : kT_2} \text{ (T-App)} \qquad \frac{\Gamma \vdash t : iS \quad \Gamma \vdash iS <: jT}{\Gamma \vdash t : jT} \text{ (T-Sub)}$$

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

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

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

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

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.

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

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

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

$$\frac{t \rightsquigarrow t'}{\mathit{inline}\ t \rightsquigarrow \mathit{inline}\ t'} \quad \text{(PE-Inline)}$$

$$\frac{t \rightsquigarrow \mathit{inline}\ t'}{\mathit{dynamic}\ t \rightsquigarrow t'} \quad \text{(PE-Dynamic1)}$$

$$\frac{t \rightsquigarrow t' \quad t' \neq \mathit{inline}\ t''}{\mathit{dynamic}\ t \rightsquigarrow \mathit{dynamic}\ t'} \quad \text{(PE-Dynamic2)}$$

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

$$\frac{t_1 \rightsquigarrow \mathit{inline}\ x \Rightarrow b \quad t_2 \rightsquigarrow t'_2 \quad [x \to t'_2]\, b \rightsquigarrow b'}{t_1(t_2) \rightsquigarrow b'} \quad \text{(PE-IApp)}$$

$$\frac{t_1 \rightsquigarrow t'_1 \quad t_2 \rightsquigarrow t'_2 \quad t'_1 \neq \mathit{inline}\ x \Rightarrow b}{t_1(t_2) \rightsquigarrow t'_1(t'_2)} \quad \text{(PE-App)}$$

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.

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

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

$$\frac{t \Downarrow \{x = t_x,\ \overline{y = t_y}\} \quad t_x \Downarrow t'_x}{t.x \Downarrow t'_x} \quad \text{(E-Sel)}$$

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 Scala to the Core Calculus

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 Implicit Conversions

One of the core assumptions of the calculus is that all `static` terms can always be promoted to `inline`. In case of function interfaces, e.g. the `dot` function in §??, that user should not manually promote all arguments to `inline`. As a convenience for a library user we provide a type-driven conversion by default (C-StaticInline in Figure ??).

$$\frac{\Gamma \vdash t_1 : i(\mathit{inline}\ T_1 \Rightarrow kT_2) \qquad \Gamma \vdash t_2 : \mathit{static}\ T_1}{\Gamma \vdash t_1(t_2) : kT_2 \rightsquigarrow t_1(\mathit{inline}\ t_2) : kT_2}\ (\text{C-StaticInline})$$

Fig. 8. Type-driven conversions from `static` terms to `inline` terms.

3.2 Making Object Oriented Constructs Partially Evaluated

$[[\mathit{let}\ x : T_x = t_x\ \mathit{in}\ t]] =$
$((x : T_x) \Rightarrow t)(t_x)$

$[[\mathit{let\ type}\ T_1 = T_2\ \mathit{in}\ t]] =$
$([T_1 <: T_2] \Rightarrow t)[T_2]$

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

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

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

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

$$\frac{\Pi \vdash iT \rightsquigarrow jT \quad \Pi \vdash kS \rightsquigarrow lS}{\Pi \vdash iT \Rightarrow kS \rightsquigarrow jT \Rightarrow lS}\ \text{(CT-T-Arrow)}$$

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

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

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

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

Fig. 9. Translation of a regular class to a compile-time version.

4 Case Studies

4.1 Integer Power Function

- Explain what happens.
- Typical partial evaluation example. Can be handled by D and Idris and not without duplication with type-driven partial evaluation.

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

Fig. 10. Function for computing the non-negative power of a real number.

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.

```
@i? def min[T: Numeric](@i? 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 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.