# Type-Driven Partial Evaluation without Code Duplication

No Author Given

No Institute Given

**Abstract.** [1]

**Keywords:** Partial Evaluation

## 1 Introduction

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

**Fig. 1.** Function for computing the dot product over generic `Numeric` values.

## 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 | $iT,\ jT,\ kT,\ lT ::=$ | Inlineable Types: |
| $inline\ t$ | | $X$ | type identifier |
| $dynamic\ t$ | | $T,\ dynamic\ T$ | dynamic type |
| $[X <: iT] \Rightarrow t$ | type abstraction | $static\ T$ | static type |
| $t[iT]$ | type application | $inline\ T$ | must inline type |
| $\Gamma ::=$ | Contexts: | | |
| $\emptyset$ | empty context | | |
| $\Gamma,\ x : iT$ | term binding | | |
| $\Gamma,\ X <: iT$ | type binding | | |

**Fig. 2.** Syntax of $F_{i<:}$

We formalize the essence of our inlining system in a minimalistic calculus based on $F_{<:}$ with records. To accommodate inlining we introduce binding-time annotations into the type system as first-class type production rules 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 $\Gamma$ we can use *inline* to obtain a compile-time view to evaluate acess to *isPolar* field at compile-time:

$$\mathit{inline}\ \mathit{complex.isPolar} : \mathit{inline}\ \mathit{Boolean}$$

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.

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.

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

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

**Fig. 3.** 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:

$$(x : \mathit{inline}\ \mathit{Int}) \Rightarrow x + 1$$

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 natural subtyping 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. 4.** Binding-time subtyping.

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

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

$$\frac{\Gamma \vdash iS <: jU \quad \Gamma \vdash jU <: kT}{\Gamma \vdash iS <: jU} \text{(S-Trans)} \qquad \frac{X <: iT \in \Gamma}{\Gamma \vdash X <: iT} \text{(S-TVar)}$$

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

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

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

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

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

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

**Fig. 5.** Subtyping.

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

### 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:

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

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

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

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

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

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

$$\frac{\Gamma \vdash t_1 : i(jT_1 \Rightarrow kT_2) \quad \Gamma \vdash t_2 : jT_1}{\Gamma \vdash t_1(t_2) : kT_2} \text{ (T-App)} \qquad \frac{t \text{ is not literal} \quad \Gamma \vdash t : iT}{\Gamma \vdash \mathit{dynamic}\ t : \mathit{dynamic}\ T} \text{ (T-Inline)}$$

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

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

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

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

**Fig. 6.** Typing.

### 2.5 Partial Evaluation

Assumption is that types are erased and therefore there is no type abstraction or type application and there are no types in lambdas.

$$\frac{t \rightsquigarrow t'}{x \Rightarrow t \rightsquigarrow x \Rightarrow t'} \text{ (PE-Func)} \qquad \frac{t \rightsquigarrow t'}{\mathit{inline}\ t \rightsquigarrow \mathit{inline}\ t'} \text{ (PE-Inline)}$$

$$\frac{\overline{t} \rightsquigarrow \overline{t'}}{\{\overline{x = t}\} \rightsquigarrow \{\overline{x = t'}\}} \text{ (PE-Rec)} \qquad \frac{t \rightsquigarrow \mathit{inline}\ t'}{\mathit{dynamic}\ t \rightsquigarrow t'} \text{ (PE-Dynamic1)}$$

$$\frac{t \rightsquigarrow \mathit{inline}\ \{x = f, \overline{y = f_i}\}}{t.x \rightsquigarrow f} \text{ (PE-ISel)} \qquad \frac{t \rightsquigarrow t' \quad t' \neq \mathit{inline}\ t''}{\mathit{dynamic}\ t \rightsquigarrow \mathit{dynamic}\ t'} \text{ (PE-Dynamic2)}$$

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

$$\frac{t_1 \rightsquigarrow \mathit{inline}\ x \Rightarrow b \quad t_2 \rightsquigarrow t_2' \quad [x \to t_2']\, b \rightsquigarrow b'}{t_1(t_2) \rightsquigarrow b'} \text{ (PE-IApp)}$$

$$\frac{t_1 \rightsquigarrow t_1' \quad t_2 \rightsquigarrow t_2' \quad t_1' \neq \mathit{inline}\ x \Rightarrow b}{t_1(t_2) \rightsquigarrow t_1'(t_2')} \text{ (PE-App)}$$

**Fig. 7.** Reduction rules for the partial evaluator.

# 3 Translating Scala to the Core Calculus

The core calculus (§2) 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 (§3.1), a scheme for translating classes into the calculus how to promote constructs classes and *methods* into their partially-evaluated versions (§3.2).

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 8).

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

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

## 3.2 Making Object Oriented Constructs Partially Evaluated

# 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.

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

**Fig. 9.** Function for computing the non-negative power of a real number.

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

**Fig. 10.** 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.

## 5 Evaluation

## 6 Related Work

## 7 Conclusion

## References

1. Eugene Burmako and Martin Odersky. Scala Macros, a Technical Report. In *Third International Valentin Turchin Workshop on Metacomputation*, 2012.

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