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 [5] 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 [2, 10], domain-specific languages [1, 6], and converting language interpreters into compilers [4, 11, 14]. Applying partial evaluation in these domains often improves program performance by several orders of magnitude [12, 1].

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 `folds` and loops over data structures of static size can produce arbitrarily large residual programs. Furthermore, in a Turing-complete language assuring termination of partial-evaluation is undecidable. **[TODO: cite]**

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 of static size and that this information 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, 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 [1, 13, 10, 14, 7] are programmer controlled. We categorize the existing solutions in three categories (for further discussion *c.f.* §7):

– 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 that vectors in the signature of *dot* are partially static.
– Type-directed partial evaluation [3] and Lightweight Modular Staging (LMS) [10] 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 [13] 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 (§4) that in a fine- grained way captures the user's intent about partial evaluation. The calculus is based on $F_{<:}$ with lazy records which makes it suitable for representing modern multi-paradigm languages with object oriented features. 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 their dual compile-time views in the $F_{i<:}$ calculus (§5).
– By demonstrating the usefulness of compile-time views in four case studies (§3): inlining, partially evaluating recursion, removing overheads of variable argument functions, and removing overheads of type-classes [8].

We have implemented a partial evaluator according to the translation scheme (§5) from object oriented features of Scala to the $F_{i<:}$ calculus. The partial is implemented for Scala and open-sourced (`https://github.com/scala-inline/`). It has a minimal Scala interface (§2) based on type annotations. We have evaluated the performance gains and the validity of the partial evaluator on all case studies (§3) 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 Partial Evaluator for Scala

We have implemented a prototype partial evaluator, formally defined in §**??**, and according to the $F_{i<:}$ calculus (formally define in §4). The partial evaluator is a compiler plugin that executes in a phase after the Scala type checker. The plugin starts with pre-typed Scala programs and uses a type annotations **[TODO: cite]** to track and verify information about the biding-time of terms.

To the user, the partial evaluator exposes a minimal interface (Figure 2) with annotations `inline` and `ct` and the `ct` function.

Annotation ct is used at the type level and denotes that one expects a compile-time view of a type. The annotation is integrated in the Scala's type system and, therefore, can be arbitrarily nested in different variants of types. Table 2 shows how the `@ct` annotation can be placed on types and how it,

```
package object scalainline {

  final class ct extends StaticAnnotation
  final class inline extends StaticAnnotation

  @compileTimeOnly def ct[T](body: => T): T = ???
  @compileTimeOnly def inline[T](body: => T): T = ???

}
```

Fig. 1. Interface of the Scala partial evaluator.

due to the translation to the compile-time views (Figure **??**), changes method signatures.

Table 1. Types and corresponding method signatures after the translation to the compile-time view.

Annotated Type	Type's Method Signatures
`Int@ct`	`+(rhs: Int@ct): Int@ct`
`Vector[Int]@ct`	`map[U](f: (Int => U)@ct): Vector[U]@ct`
	`length: Int@ct`
`Vector[Int@ct]@ct`	`map[U](f: (Int@ct => U)@ct): Vector[U]@ct`
`Map[Int@ct, Int]@ct`	`get(key: Int@ct): Option[Int]@ct`

In Table 2, `Int@ct` is a non-polymorphic type and therefore according to the translation to the compile-time view (14) all arguments of all methods will be executed at compile-time. On the other hand, `Vector[Int]@ct` will have all arguments of all methods transformed except the generic ones. In effect, this, makes higher order combinators of `Vector` operate on dynamic values, thus, function `f` passed to `map` accepts the dynamic value as input. Type `Vector[Int@ct]@ct` is has all parts executed at compile-time. However, the return type of the function `map` can still be a compile-time view - due to the type parameter `U`.

Functions `ct` and `inline` is used at the term level for promoting Scala objects and functions into their compile-time views. Without `ct` we would not be able to instantiate compile-time views of the types. Table 2 shows how different types of terms are promoted to their compile-time views.

[TODO: footnote about Scala objects] [TODO: static promotion of lambdas] Function `ct` can be applied to objects (*e.g.* `Vector`) to provide a compile-time view over their methods. When those objects have generic parameters, `ct` be used to promote the arguments, and thus, the result types of these functions. When applied, on functions `ct` promotes the compile-time view as well as its arguments and the return type. **[TODO: inline]**

Annotation inline can be used only on methods and functions. This function uses partial evaluation to achieve inlining**[TODO: cite]**. This is not the

Table 2. Types and corresponding method signatures after the translation to the compile-time view.

Promoted Term	Term's Promoted Type
`ct(Vector)(1, 2, 3)`	`: Vector[Int]@ct`
`ct(Vector)(ct(1), ct(2), ct(3))`	`: Vector[Int@ct]@ct`
`new (Cons@ct)(1, Nil)`	`: Cons[Int]@ct`
`new (Cons@ct)(ct(1), ct(Nil))`	`: Cons[Int@ct]@ct`
`ct((x: Int) => x)`	`: (Int@ct => Int@ct)@ct`
`inline((x: Int) => x)`	`: (Int => Int)@ct`

first time that inlining is achieved through partial evaluation**[TODO: cite]**, however, partial evaluation is trivially added to the system. It directly corresponds to adding `inline` from $F_{i<:}$ in front of the function or method definition.

2.1 Interaction with the Scala Language

3 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 (§3.2), then we demonstrate how variable argument functions can be desugared into the core functionality (§3.3). Finally, we demonstrate how the abstraction overhead of the `dot` function and all associated type-classes can be removed (§3.5).

3.1 Inlining Expressed Through Partial-Evaluation

3.2 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 inlined at application and `@ct` requires that the `exp` argument is a compile-time view of `Int`. The application of the function `pow` with a constant exponent will produce:

```
pow(base, 4)
  ↪ base * base * base * base * 1
```

Here, in the function application, constant 4 is promoted to `ct` by the automatic conversions. **[TODO: ref]**

3.3 Variable Argument Functions

Variable argument functions appear in widely used languages like Java, C#, and Scala. Such arguments are typically passed in the function body inside of the data structure (*e.g.* `Seq[T]` in Scala). When applied with variable arguments the size of the data-structure is statically known and all operations on them can be partially evaluated. However, sometimes, the function is called with arguments of dynamic size. For example, function `min` that accepts multiple integers

```
def min(vs: Int*): Int =
  vs.tail.foldLeft(vs.head){ (min, el) => if (el < min) el else min }
```

can be called either with statically known arguments (*e.g.* `min(1,2)`) or with dynamic arguments:

```
val values: Seq[Int] = ... // dynamic value
min(values: _*)
```

Ideally, we would be able to achieve partial evaluation if the arguments are of statically known size and avoid partial evaluation in case of dynamic arguments. To this end we translate the method `min` into a partially evaluated version and a dynamic version. The call to these methods is dispatched, at compile-time, by the `min` method which checks if arguments are statically known. Desugaring of `min` is shown in Figure 2.

```
def min(vs: Int*): Int = macro
  if (isVarargs(vs)) q"min_CT(vs)"
  else q"min_D(vs)"

def min_CT(vs: Seq[Int] @ct): Int =
  vs.tail.foldLeft(vs.head){ (min, el) => if (el < min) el else min }
def min_D(vs: Seq[Int]): Int =
  vs.tail.foldLeft(vs.head){ (min, el) => if (el < min) el else min }
```

Fig. 2. Function `min` is desugared into a `min` macro that based on the binding time of the arguments dispatches to the partially evaluated version (`min_CT`) for statically known varargs or to the original min function for dynamic arguments `min_D`.

3.4 Removing Abstraction Overhead of Type-Classes

[TODO: not-sure how to achieve this!] [TODO: cite] Type-classes are omnipresent in everyday programming as they provide allow abstraction over generic parameters (*e.g.* Numeric abstracts over numeric values). Unfortunately, type-classes are a source of abstraction overheads during execution**[TODO: cite]**. Type-classes are in most of the cases statically known. Ideally, we would be able to deterministically remove abstraction overheads of type classes.

```
object Numeric {
  @inline implicit def dnum: Numeric[Double] = DoubleNumeric
  @inline def zero[T](implicit num: Numeric[T]): T = num.zero
}

trait Numeric[T] {
  def plus(x: T, y: T): T
  def times(x: T, y: T): T
  def zero: T
}

class DoubleNumeric[T <: Double] extends Numeric[Double] {
  @inline def plus(x: T, y: T): T = x + y
  @inline def times(x: T, y: T): T = x * y
  @inline def zero: T = 0.0
}
```

Fig. 3. Function for computing the non-negative power of a real number.

3.5 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.

4 The $F_{i<:}$ Calculus

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

Fig. 4. Syntax of $F_{i<:}$

We formalize the essence of our partial evaluation system in a minimalistic calculus based on kernel $F_{<:}$[9] with lazy records. To accommodate predictable partial evaluation we introduce binding-time annotations into the type system as types that represent two kinds of bindings:

1. **Dynamic binding**. Corresponds to terms that are expected to be evaluted at runtime.
2. **Inline binding**. Corresponds to terms that must be evaluated at compile-time.

To simplify judgments in our formalization we use concise iT syntax to abstract over binding-times of types. Here i signifies the bit of information that says if type is inline or not and T carries underlying type that is being annotated. So for example in $inline\ Any$ we get $i = inline$ and $T = Any$.

Similarly we abstract over binding time of terms through it notation that has analogous to the one we use for types.

4.1 Well-formed types

Even though binding-time information is represented as types, not all of the possible combinations of types and binding-times is correct. We restrict types to disallow nesting of more specific binding times into less specific ones.

$$\text{wff } i\mathit{Any} \qquad (\text{W-A}\textsc{ny})$$

$$\frac{i \le j \quad i \le k \quad \text{wff } jT_1 \quad \text{wff } kT_2}{\text{wff } i(jT_1 \Rightarrow kT_2)} \qquad (\text{W-A}\textsc{bs})$$

$$\frac{i \le k \quad \text{wff } [X \mapsto iS]kT}{\text{wff } i([X <: iS] \Rightarrow kT)} \qquad (\text{W-TA}\textsc{bs})$$

$$\frac{i \le \overline{j} \quad \overline{\text{wff } jT}}{\text{wff } i\{\overline{x : jT}\}} \qquad (\text{W-R}\textsc{ec})$$

Fig. 5. Well-formed types wff iT.

We represent notion of more specific binding-times through a simple partial order on binding time annotations.

$$\begin{gathered} \mathit{dynamic} \le \mathit{dynamic} \\ \mathit{inline} \le \mathit{dynamic} \\ \mathit{inline} \le \mathit{inline} \end{gathered}$$

Fig. 6. Partial order on binding-time $i \le j$

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

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

This is inconsistent because a dynamic function may not have any non-dynamic parameters. As described in W-Abs functions may only have parameters that are at most as specific as function binding-time. In our example this doesn't hold as *inline* is more specific than *dynamic*.

4.2 Subtyping

$F_{i<:}$ integrates binding-time annotation into subtyping relation on regular types by threading inlining information throughout all of the standard subtyping rules.

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

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

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

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

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

$$\frac{\{x_p : i_pS_p{}^{\ p\in 1..n}\} \text{ is permutation of } \{y_p : j_pT_p{}^{\ p\in 1..n}\}}{\{x_p : i_pS_p{}^{\ p\in 1..n}\} <: \{y_p : j_pT_p{}^{\ p\in 1..n}\}} \qquad \text{(S-Perm)}$$

$$\frac{\forall p \in 1..n.\ i_pS_p <: j_pT_p}{\{x_p : i_pS_p{}^{\ p\in 1..n}\} <: \{x_p : j_pT_p{}^{\ p\in 1..n}\}} \qquad \text{(S-Depth)}$$

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

Fig. 7. Subtyping $\Gamma \vdash T_1 <: T_2$.

Apart from that we also introduce subtyping on binding-time types.

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

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

Fig. 8. Subtyping of binding-time types $\Gamma \vdash iT_1 <: jT_2$.

Two binding-time types are subtypes if their underlying types are subtypes and if they have the same binding time.

4.3 Type polymorphism

Our system retains traditional type abstraction means inherited from $F_{<:}$. We extend it to accomodate encoding of binding-times into types. This allows us to specify binding type of the abstracted generic type:

$$[T <: dynamic\ Any] \Rightarrow (x : T) \Rightarrow x$$

For this particular identity function we need to restrict subset of all admissable types to only allow *dynamic* ones. Passing an *inline* type would not make sense as the resulting type would have not been well-formed.

4.4 Typing

$$\frac{x : iT \in \mathit{\Gamma}}{\mathit{\Gamma} \vdash x : iT} \qquad (\text{T-}\textsc{Ident})$$

$$\frac{\mathit{\Gamma} \vdash \overline{t} : \overline{jT} \quad \text{wff } i\{\overline{x : jT}\}}{\mathit{\Gamma} \vdash i\{\overline{x = t}\} : i\{\overline{x : jT}\}} \qquad (\text{T-}\textsc{Rec})$$

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

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

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

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

$$\frac{\mathit{\Gamma},\ X <: jT_1 \vdash t_2 : kT_2 \quad \text{wff } i([X <: jT_1] \Rightarrow kT_2)}{\mathit{\Gamma} \vdash i([X <: jT_1] \Rightarrow t_2) : i([X <: T_1] \Rightarrow kT_2)} \qquad (\text{T-TA}\textsc{bs})$$

$$\frac{\mathit{\Gamma} \vdash t : i([X <: T_1] \Rightarrow kT_2) \quad \mathit{\Gamma} \vdash T <: T_1}{\mathit{\Gamma} \vdash t[T] : [X \mapsto T]kT_2} \qquad (\text{T-TA}\textsc{pp})$$

$$\frac{\mathit{\Gamma} \vdash t : \mathit{inline}\ T}{\mathit{\Gamma} \vdash \mathit{dynamic}\ t : \mathit{dynamic}\ T} \qquad (\text{T-}\textsc{Dynamic})$$

Fig. 9. Typing $\mathit{\Gamma} \vdash t : iT$.

Similarly to the changes made to the subtyping relation we thread binding-time information throughout typing relation. Apart from that we also ensure that all literals produces by the user have well-formed types.

4.5 Partial Evaluation

$$\frac{t \rightsquigarrow t'}{(x : iT) \Rightarrow t \rightsquigarrow (x : T) \Rightarrow t'} \qquad (\text{PE-Func})$$

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

$$\frac{t \rightsquigarrow t'}{[X <: iT] \Rightarrow t \rightsquigarrow [X <: iT] \Rightarrow t'} \qquad (\text{PE-TAbs})$$

$$\frac{t_1 \rightsquigarrow t'_1 \quad t_1 \neq \mathit{inline}\ t_3 \quad t_2 \rightsquigarrow t'_2}{t_1(t_2) \rightsquigarrow t'_1(t'_2)} \qquad (\text{PE-App})$$

$$\frac{t \rightsquigarrow t' \quad t' \neq \mathit{inline}\ t_3}{t.x \rightsquigarrow t'.x} \qquad (\text{PE-Sel})$$

$$\frac{t_1 \rightsquigarrow t'_1 \quad t'_1 \neq \mathit{inline}\ t_3}{t_1[T_1] \rightsquigarrow t'_2} \qquad (\text{PE-TApp})$$

$$\frac{t_1 \rightsquigarrow \mathit{inline}\ (x : iT) \Rightarrow t \quad t_2 \rightsquigarrow t'_2 \quad [x \mapsto t'_2]t \rightsquigarrow t'}{t_1(t_2) \rightsquigarrow t'} \qquad (\text{PE-InlineApp})$$

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

$$\frac{t_1 \rightsquigarrow \mathit{inline}\ [X <: iT_2] \Rightarrow t_2 \quad [X \mapsto iT_1]t_2 \rightsquigarrow t'_2}{t_1[iT_1] \rightsquigarrow t'_2} \qquad (\text{PE-InlineTApp})$$

$$\mathit{inline}\ v \rightsquigarrow \mathit{inline}\ v \qquad (\text{PE-InlineValue})$$

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

Fig. 10. Partial evaluation $t \rightsquigarrow t'$

4.6 Evaluation

Once partial evaluation is complete we strip all binding-time annotations on types and convert inline terms into corresponding dynamic ones. After that we can use standard $F_{<:}$evaluation rules augmented with lazy records semantics (E-Sel).

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

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

$$\frac{t_1 \Downarrow [X <: T_2] \Rightarrow t_2 \quad [X \mapsto T_1]t_2 \Downarrow v}{t_1[T_1] \Downarrow v} \qquad (\text{E-TApp})$$

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

Fig. 11. Evaluation $t \Downarrow v$

4.7 Conjectures

1. Progress and preservation of partial evaluation.
2. Progress and preservation of evaluation.

5 Integrating $F_{i<:}$ with Object Oriented Languages

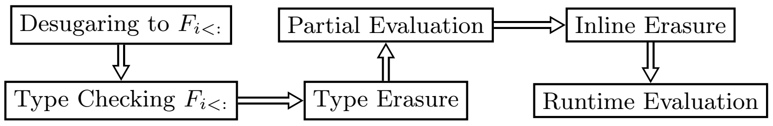

Fig. 12. Compilation pipeline.

The $F_{i<:}$ calculus §4 captures the essence of user-controlled predictable partial-evaluation. In practice, though, it is fairly low level and it is not obvious how to define *classes* and methods from in modern multi-paradigm programming languages. Furthermore, $F_{i<:}$ requires an inconveniently large number of `inline` calls in method invocations. In this section we a scheme for translating classes into $F_{i<:}$ (§5.1), show how to provide compile time views of classes and and *methods*§**??**, and formalize convenient implicit conversions for the calculus §5.3.

Furthermore, rules of $F_{i<:}$ 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.

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

$$
\begin{aligned}
&[\![\textbf{let } x : T_x = t_x \textbf{ in } t]\!] = ((x : T_x) \Rightarrow t)(t_x)\\
&[\![\textbf{let } \mathit{type}\ T_1 = T_2 \textbf{ in } t]\!] = ([T_1 <: T_2] \Rightarrow t)[T_2]\\
&[\![\textbf{let } \mathit{class}\ C[A](x : T_x)\{\mathit{def}\ f[B](y : T_y) = t_f\} \textbf{ in } t]\!] =\\
&\quad \textbf{let } \mathit{type}\ C = [A] \Rightarrow \mathit{inline}\ \{\mathit{fields} : \{x : T_x\}, \mathit{methods} : \mathit{inline}\ \{f : [B] \Rightarrow T_y \Rightarrow T_f\}\ \} \textbf{ in}\\
&\qquad \textbf{let } C : [A] \Rightarrow \mathit{inline}\ ((t_x : T_x) \Rightarrow C[A]) = [A] \Rightarrow \mathit{inline}\ ((t_x : T_x) \Rightarrow\\
&\qquad\quad \mathit{inline}\ \{\mathit{fields} = \{x = t_x\}, \mathit{methods} = \mathit{inline}\ \{f = [B] \Rightarrow (y : T_y) \Rightarrow t_f\}\}) \textbf{ in } t
\end{aligned}
$$

Fig. 13. Desugaring of classes into $F_{i<:}$.

5.2 Compile-Time View of the Terms

$$\frac{\mathit{\Pi} \vdash T \in \mathit{\Pi}}{\mathit{\Pi} \vdash iT \rightsquigarrow iT}\ (\text{CT-TV}\textsc{ar}) \qquad \frac{\mathit{\Pi} \vdash T \notin \mathit{\Pi}}{\mathit{\Pi} \vdash iT \rightsquigarrow \mathit{inline}\ T}\ (\text{CT-T-V}\textsc{ar})$$

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

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

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

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

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

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

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

Fig. 14. Translation of a type abstractions, function, and record values into a compile-time view. The translation is used for promoting types into their compile time versions.

5.3 Implicit Conversions

According to $F_{i<:}$ rules if method signatures contain compile-time views of a type the corresponding arguments in method application would always have to be promoted to `inline`. In practice this is not convenient as it requires an inconveniently large number of annotations. Partial evaluation is an optimization, and as such, it should not affect user code - users should not be aware of the internal operation of the library.

To address this issue we introduce implicit conversions from all language literals, and direct class constructor calls of non-inline type into their compile-time views. For example, for a factorial function

```
def fact(n: Int @ct) = if (n == 0) 1 else fact(n - 1)
```

we will not require annotations on literals `0`, and `1`. Furthermore, the function can be invoked without promoting the literal `5` into it's compile-time view:

```
fact(5)
  ↪ 120
```

6 Evaluation

6.1 Reduction of Code Duplication

6.2 Performance Comparison

Table 3. Performance comparison with LMS and hand optimized code.

Benchmark	Hand Optimized	LMS	Scala Inline
`pow`			
`min`			
`dot`			
`fft`			

7 Related Work

8 Conclusion

References

1. Edwin C. Brady and Kevin Hammond. Scrapping your inefficient engine: Using partial evaluation to improve domain-specific language implementation. In *International Conference on Functional Programming (ICFP)*, 2010.
2. Jacques Carette and Oleg Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. In *Generative Programming and Component Engineering (GPCE)*, 2005.
3. Olivier Danvy. *Type-directed partial evaluation*. Springer, 1999.
4. Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.
5. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
6. Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. Staged parser combinators for efficient data processing. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2014.
7. Anne-Françoise Le Meur, Julia L Lawall, and Charles Consel. Specialization scenarios: A pragmatic approach to declaring program specialization. *Higher-Order and Symbolic Computation*, 17(1-2):47–92, 2004.
8. Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *ACM Sigplan Notices*, volume 45, pages 341–360, 2010.
9. Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.
10. Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, 55(6):121–130, June 2012.

11. Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Kunle Olukotun, and Martin Odersky. Project Lancet: Surgical precision JIT compilers. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2013.
12. Amin Shali and William R. Cook. Hybrid partial evaluation. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011.
13. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 1997.
14. Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, 2013.