Type-Driven Partial Evaluation without Code Duplication

No Author Given

No Institute Given

Abstract.

Keywords: Partial Evaluation

1 Introduction

Partial evaluation [4] 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 [2, 7], for domain-specific languages [1, 5], and for converting language interpreters into compilers [3, 8, 11]. Applying partial evaluation in these domains often improves program performance by several orders of magnitude [9, 1].

To achieve *predictable* and *safe* partial evaluation, however, a partial evaluator must be controlled by the programmer [1, 6]. Unlike other compiler optimizations, due to compile-time execution, partial evaluation might not *terminate*. Furthermore, the *code explosion* is a concern as the final program is a result of compile-time execution and thus can be arbitrarily large. Lack of predictability and danger of code explosion are the reason that successful partial evaluators [1, 10, 7, 11] are programmer controlled.

To illustrate 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
  }
```

State the problem:

- Minimal number of annotations
- No code duplication
- Allow generics

The main idea of this paper is to explicitly capture user intent in the types.

Modified signature:

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

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

Others have tried:

- type-directed/LMS
- MetaML
- Idris/D

Contributions:

Evaluation:

Sections:

2 Formalization $F_{i<:}$

$t ::=$	Terms:
x, y	identifier
$(x : iT) \Rightarrow t$	function
$t(t)$	application
$\{\overline{x = t}\}$	record
$t.x$	selection
$\mathit{inline}\ t$	inlining request
$[X <: iT] \Rightarrow t$	type abstraction
$t[iT]$	type application
$S,\ T,\ U ::=$	Types:
$iS \Rightarrow jT$	function type
$\{\overline{x : iS}\}$	record type
$[X <: iS] \Rightarrow jT$	universal type
$\top$	top type
$iT,\ jT,\ kT,\ lT ::=$	Inlineable Types:
$T,\ \mathit{dynamic}\ T$	dynamic type
$\mathit{static}\ T$	static type
$\mathit{inline}\ T$	must inline type
$\Gamma ::=$	Contexts:
$\emptyset$	empty context
$\Gamma,\ x : iT$	term binding
$\Gamma,\ X <: iT$	type binding

Fig. 1. Syntax

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

$$\frac{\Gamma,\ x : iT_1 \vdash t : jT_2}{\Gamma \vdash (x : iT_1) \Rightarrow t : static\ iT_1 \Rightarrow jT_2} \quad \text{(T-Func)}$$

$$\frac{\Gamma \vdash \overline{t : iT}}{\Gamma \vdash \{\overline{x = t}\} : static\ \{\overline{x : iT}\}} \quad \text{(T-Rec)}$$

$$\frac{\Gamma \vdash t_1 : i(jT_1 \Rightarrow kT_2) \quad \Gamma \vdash t_2 : jT_2}{\Gamma \vdash t_1(t_2) : (i \wedge j \wedge k)T_2} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash t : i\{x = jT_1, \overline{y = kT_2}\}}{\Gamma \vdash t.x : (i \wedge j)T_1} \quad \text{(T-Sel)}$$

$$\frac{\Gamma \vdash t : static\ T}{\Gamma \vdash inline\ t : inline\ T} \quad \text{(T-Inline)}$$

$$\frac{\Gamma,\ X <: iT_1 \vdash t_2 : jT_2}{\Gamma \vdash [X <: iT_1] \Rightarrow t_2 : static\ [X <: iT_1] \Rightarrow jT_2} \quad \text{(T-TAbs)}$$

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

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

Fig. 2. typing $\Gamma \vdash t : iT$

$$\frac{\forall i.\ i <: dynamic}{} \quad \text{(IS-Dynamic)}$$

$$\frac{\forall i \in \{static,\ inline\}.\ i <: static}{} \quad \text{(IS-Static)}$$

$$\frac{inline <: inline}{} \quad \text{(IS-Inline)}$$

Fig. 3. Inlinity Subtyping $i <: j$

$$\frac{\forall i,\ \forall j,\ i <: j.\ i \wedge j = j}{}$$

Fig. 4. Inlinity Intersection $i \wedge j$

$$\frac{}{\Gamma \vdash S <: S} \quad \text{(S-Refl)}$$

$$\frac{\Gamma \vdash S <: U \quad \Gamma \vdash U <: T}{\Gamma \vdash S <: U} \quad \text{(S-Trans)}$$

$$\frac{}{\Gamma \vdash S <: \top} \quad \text{(S-Top)}$$

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

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

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

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

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

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

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

Fig. 5. Subtyping $\Gamma \vdash iS <: jT$

3 Translating Scala to the Core Calculus

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. 6. 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. 7. 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. 8. Function for computing the non-negative power of a real number.

5 Evaluation

6 Related Work

7 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. Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.
4. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
5. 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.
6. 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.
7. 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.
8. 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.
9. Amin Shali and William R. Cook. Hybrid partial evaluation. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011.
10. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 1997.
11. 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.