Dynamic Partial Evaluation

Gregory T. Sullivan
gregs@ai.mit.edu

Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Abstract. Dynamic partial evaluation performs partial evaluation as a side effect of evaluation, with no previous static analysis required. A completely dynamic version of partial evaluation is not merely of theoretical interest, but has practical applications, especially when applied to dynamic, reflective programming languages. Computational reflection, and in particular the use of meta-object protocols (MOPs), provides a powerful abstraction mechanism, providing programmatic "hooks" into the interpreter semantics of the host programming language. Unfortunately, a runtime MOP defeats many optimizations based on static analysis (for example, the applicable methods at a call site may change over time, even for the same types of arguments). Dynamic partial evaluation allows us to apply partial evaluation techniques even in the context of a meta-object protocol. We have implemented dynamic partial evaluation as part of a Dynamic Virtual Machine intended to host dynamic, reflective object-oriented languages. In this paper, we present an implementation of dynamic partial evaluation for a simple language – a lambda calculus extended with dynamic typing, subtyping, generic functions and multiple dispatch.

1 Introduction

Our goal is the efficient implementation of dynamic, higher-order, reflective object-oriented languages. Language features that we must support include dynamic typing, runtime method definition, first class types (with subtyping), first class (and higher order) functions, and reflection.

The concept of a *meta-object protocol* (MOP) [KdR91] subsumes many of the above-listed features. A MOP is based upon computational reflection [Mae87] – giving a program access to its internal structure and behavior and allowing programmatic manipulation of that structure and future behavior. A MOP entails that the entities being returned by reflective operations and manipulated programmatically be first class – thus the need for first class functions, types, classes, etc. We want to at once support the power and abstraction of meta-object protocols while at the same time providing efficient execution, especially when MOP-related features are not being used. For example, if the MOP provides programmer control over method dispatch, but the programmer has not used that feature, the implementation should not instrument every method call in order to support the unused abstraction. The research described in this paper is part of a project to implement a *Dynamic Virtual Machine* (DVM) – a VM well-suited to hosting dynamic, reflective languages. The small formal language Λ_{DVM} presented in this paper is a simplified version of the DVM's native language, DVML.

Partial evaluation [JGS93] is a general technique for specializing parts of a program with respect to known values. For example, if a function has multiple arguments, and that function is called frequently with the same values for some of the arguments, it may be worthwhile to create a specialized version of that function for those common argument values. Within the body of the specialized version of the function, we may be able to optimize away many computations that depend solely on the values of the arguments against which we are specializing.

Experience has shown that many of the elements of a program exposed and made mutable by a MOP (e.g. the class structure, methods of a virtual function, method dispatch algorithm, etc.) in fact change only rarely at runtime. Thus we may guardedly treat these aspects of an application as constant and then apply partial evaluation techniques to remove computations that depend on these mostly-constant program properties.

2 Related Work

A number of researchers have been drawn to using partial evaluation techniques to eliminate overhead due to reflective operations. This is not surprising, as a recurrent theme in partial evaluation research has been the elimination of "interpretive overhead" (especially by using self application), and reflective operations can be viewed as exposing an interpreter semantics to programs.

Masuhara et al. [MMAY95, MY98] use partial evaluation to eliminate interpretive/reflective overhead in an object-oriented concurrent language. Partial evaluation is performed as part of compilation, and impressive results are recorded, with nearly all interpretive overhead removed in some cases. A limitation of their system is that all possible effects on meta-level functionality must be known at compile time. For example, if modification to the evaluation of variables will be effected at runtime, which modifications must be known at compile time.

In [BN00], Braux and Noyé use partial evaluation techniques to eliminate reflection overhead in Java [GJSB00]. Java provides limited reflection functionality, but no fine-grained MOP to manipulate the features covered by reflection. For example, a Java program may query what methods belong to a class but may not add or remove a method. Java does support coarse-grained redefinition via dynamic class loading. The partial evaluation rules presented in [BN00] are specific to Java and would not work with a different language or runtime.

Runtime partial evaluation, as in [CN96, VCC97], defers some of the partial evaluation process until actual data is available at runtime. However, the scope and actions related to partial evaluation are largely decided at compile time.

Dynamic partial evaluation goes further, deferring all partial evaluation activity to runtime.

The technique of specializing a function on finer types than originally declared has been pursued by several research efforts – notably the Self [Cha92] and Cecil [DCG95] projects. In [VCC97] and [SLCM99], Volanchi, Schultz, et al. use declarations and partial evaluation to achieve similar specialization for object-oriented programs. The implementation of dynamic partial evaluation presented in this paper also produces specialized versions of functions.

The runtime partial evaluation in [VCC97] includes the notion of guards against future violation of invariants, and dynamic partial evaluation against "likely invariants" also requires such guards. The idea of *optimistic optimization* with respect to *quasi-invariants* has been pursued by Pu et al. in the Sythesis [PMI88] and then Synthetix [PAB^{+}95] projects in the context of operating systems.

3 Overview

We present an overview of the main concepts used in this paper, including dynamic partial evaluation, generic functions, and multiple dispatch.

3.1 Dynamic Partial Evaluation

Dynamic partial evaluation happens as a side-effect of evaluation. At runtime, an expression is evaluated with respect to an environment that contains both the usual *dynamic bindings* of identifiers to values, and also *static bindings* from identifiers to types. The static component of the environment corresponds to the symbolic environments of compile-time partial evaluation. In addition to producing a value for the expression, dynamic partial evaluation produces a residual version of the original expression based on the types in the environment. The folding that occurs to produce a residual expression is the same as in online partial evaluation – if the environment indicates that an identifier maps to a fully static value (i.e. has a singleton type), then an expression based on that identifier may be folded. Optimization may occur if a value is not fully static, but its type is known. For example, if accurate types are known for argument values before a call, dynamic type checks may be avoided.

Note that dynamic partial evaluation does not suffer from the "infinite unfolding" issues of static partial evaluation. Because dynamic partial evaluation happens during evaluation, partial evaluation only loops if the application loops.

While dynamic partial evaluation is defined at the expression level, control and collection of the results of dynamic partial evaluation happen at the function level. For example, suppose a function `f(int x, Object y)` is called repeatedly with a value of 42 for `x` and with (different) instances of the `Point` class for `y`. For one such invocation of `f`, the decision is made to evaluate the body with dynamic partial evaluation enabled. For the duration of this call to `f`, every expression is evaluated in an environment that maps `x` and `y` to both their actual concrete values (42 for `x`, some `Point` instance for `y`), and also to the types *eq*(42)[1] and *Point*. When the execution of `f`'s body is complete, we have both a concrete value for this call, and also a new version of `f`'s body expression, specialized to the signature (*eq*(42), *Point*). Within the body of the specialized version, we will have performed any possible optimizations assuming that `x` is 42 and that `y` is an instance of the `Point` class. The new, specialized version of `f` is then added to `f`'s *generic function* (explained in the next section) and will be selected whenever `f` is called with its first argument 42 and its second argument an instance of `Point`.

Dynamic partial evaluation is intended to be used in conjunction with more static techniques. However, the dynamic features of the languages we are targeting often preclude optimizations based on static analysis, and dynamic partial evaluation gives us a valuable tool for optimizing in the face of extreme dynamism.

3.2 Generic Functions and Multiple Dispatch

The virtual machine in which we have implemented dynamic partial evaluation provides *generic functions* and *multiple dispatch*. Generic functions and multiple dispatch are used not just to model corresponding features in source programming languages, but are also an integral part of our implementation of dynamic partial evaluation. The key insight is that adding specialized methods to a generic function at runtime corresponds to *polyvariant specialization* in compile-time partial evaluation, and the static notion of *sharing* is handled at runtime via multiple dispatch.

A *generic function* is a set of "regular" functions and selection criteria for choosing one of those functions (or signalling an error) for any given tuple of arguments. In a *single dispatch* language, such as C++ or Java, a generic function corresponds to a virtual method, and the selection criteria is to find the method defined in the class nearest above the receiver's concrete class in the class hierarchy. *Multiple dispatch* generalizes single dispatch in that more than one argument may be used in the method selection process. For example, we may define a generic function `foo` consisting of the following methods:

```
void foo(A this, A that);     // method 1
void foo(B this, B that);     // method 2
void foo(C this, B that);     // method 3
```

Suppose that `B` is a subclass of `A` and `C` is also a subclass of `A` (and all classes are instantiable). Then the following sequence of code:

```
A anA = new A();  B aB = new B(); C aC = new C();
foo(aC, anA); foo(aB, aC);  foo(aB, aB); foo(aC, aB);
```

invokes methods 1, 1, 2, and 3 in order.

Support for generic functions and multiple dispatch serves two distinct purposes in our system. First of all, we are interested in supporting languages with "interesting" dispatch mechanisms, including multiple dispatch such as in Dylan

[1] The notation *eq*(v) denotes the *singleton type* containing exactly one value, namely v.

and CLOS. Secondly, it is via the generic function and multiple dispatch mechanisms that we both cache and also invoke specialized versions of functions at runtime. Consider the first call to `foo`, above. If the first call, `foo(aC, anA)`, is executed with dynamic partial evaluation enabled, we will get a new version of method 1,

```
void foo(C this, A that);  // method 4 (specialized version of method 1)
```

Within the body of the new method, dynamic dispatch based on, or dynamic type checking of, the `this` argument may now be optimized under the assumption that `this` is an instance of class `C`. After adding the new method to the `foo` generic function, later calls to `foo` with arguments of class `C` and class `A` will resolve to this newly created method.

4 Λ_{DVM}, A Dynamically-typed Lambda Calculus with Subtyping and Generic Functions

To clarify the mechanism of dynamic partial evaluation, we introduce Λ_{DVM}, a dynamically-typed lambda calculus with subtypes and generic functions. Figure 1 gives an operational semantics for this simple functional language. Λ_{DVM} is modeled after, but much simpler than, DVML, the “native” language of the Dynamic Virtual Machine. Among other things, DVML supports recursive functions, predicate types, and more complicated function signatures. The syntax of Λ_{DVM} is given by the following grammar:

$$\begin{array}{rl} \mathit{Exp} ::= & x \ \mid\ n \ \mid\ (\mathsf{if}\ \mathit{Exp}\ \mathit{Exp}\ \mathit{Exp}) \\ \mid & (\mathsf{call}\ \mathit{Exp}\ \mathit{Exp}\ +) \ \mid\ (\mathsf{gf\text{-}call}\ \mathit{Exp}\ \mathit{Exp}\ +) \\ \mid & (\mathsf{lambda}\ ([x\ \ \mathit{Exp}]+)\ \ \mathit{Exp}\ .\ \mathit{Exp}) \end{array}$$

where x ranges over identifiers and n ranges over integers. Note the *Exp* phrases in lambda expressions for specifying the argument types and result type of a function.

The evaluation relation $\Rightarrow$ takes an expression e, an environment ρ, and a type τ to an extended value. *Extended values* are triples of a tagged value, an expression, and a type. A *tagged value* is a value for which the function *type-of* returns a type. In Figure 1, upper case V's range over extended values, and lower case v's over tagged values. There is syntax for creating integer, boolean, and closure tagged values, and there are predefined functions for creating other tagged values, including types, generic functions, mutable cells, and lists. A type may be one of the predefined types or built from a type constructor – see Figure 2 for some predefined values. Creating an extended value with tagged value v, expression e and type τ is denoted $\langle v, e, \tau\rangle_{val}$.

Types are ordered as follows: all types are subtypes of $\top$, $\bot$ is a subtype of all types, subtyping between types constructed using logical connectives is based on implication, function types have the usual contravariant subtyping, a singleton type $eq(v)$ is a subtype of $\mathit{type\text{-}of}(v)$, and there is a predefined function, *subtype* for creating subtypes of (multiple) other types.

$\Rightarrow \subseteq (\mathit{Exp} \times \mathit{Env} \times \mathit{Type}) \times \mathit{ExtValue}, \quad \mathit{ExtValue}{=}\mathit{TagValue} \times \mathit{Exp} \times \mathit{Type}$
e refers to the expression being evaluated in the current rule., "–" means "don't care"
$v = \mathit{some\text{-}operation}$ binds v to the result of *some-operation.*
$v \simeq e'$ deconstructs value v into its component parts.

$$\frac{\rho(\llbracket x \rrbracket) = V \simeq \langle v, -, - \rangle_{val}\ ;\quad \mathit{check}(v, \tau)}{\llbracket x \rrbracket\ \rho\ \tau \Rightarrow \mathcal{F}(e, \langle V \rangle, \rho, \tau)} \qquad \frac{\mathit{check}(n, \tau)}{\llbracket n \rrbracket\ \rho\ \tau \Rightarrow \mathcal{F}(e, \langle \langle n, e, \mathit{eq}(n) \rangle_{val} \rangle, \rho, \tau)}$$

$$\frac{\begin{array}{l} e_0\ \rho\ \tau_{bool} \Rightarrow V_0 \simeq \langle \mathit{true}, -, - \rangle_{val} \\ e_1\ \rho\ \tau \Rightarrow V \end{array}}{\llbracket (\mathsf{if}\, e_0\ e_1\ e_2) \rrbracket\ \rho\ \tau \Rightarrow \mathcal{F}(e, \langle V, V_0 \rangle, \rho, \tau)} \qquad \frac{\begin{array}{l} e_0\ \rho\ \tau_{bool} \Rightarrow V_0 \simeq \langle \mathit{false}, -, - \rangle_{val} \\ e_2\ \rho\ \tau \Rightarrow V \end{array}}{\llbracket (\mathsf{if}\, e_0\ e_1\ e_2) \rrbracket\ \rho\ \tau \Rightarrow \mathcal{F}(e, \langle V, V_0 \rangle, \rho, \tau)}$$

$$\frac{\begin{array}{l} e_0\ \rho\ \tau_{fun} \Rightarrow V_f \simeq \langle \mathit{closure}(\langle \llbracket x_1 \rrbracket, \ldots, \llbracket x_n \rrbracket \rangle, \langle \tau_{arg_1}, \ldots, \tau_{arg_n} \rangle, \tau_{res}, e_f, \rho_f), -, - \rangle_{val} \\ e_i\ \rho\ \tau_{arg_i} \Rightarrow V_i \simeq \langle v_i, e'_i, \tau_i \rangle_{val},\ i \in 1 \ldots n \\ e_f\ \rho_f[x_i \mapsto \langle v_i, e'_i, \mathit{glb}(\tau_i, \tau_{arg_i}) \rangle_{val}],\ \mathit{glb}(\tau, \tau_{res}) \Rightarrow V,\ i \in 1 \ldots n \end{array}}{\llbracket (\mathsf{call}\, e_0\ e_1\ \ldots\ e_n) \rrbracket\ \rho\ \tau \Rightarrow \mathcal{F}(e, \langle V, V_f, \langle V_1, \ldots, V_n \rangle \rangle, \rho, \tau)}$$

$$\frac{\begin{array}{l} e_0\ \rho\ \tau_{gf} \Rightarrow V_g \simeq \langle \mathit{generic}(ms, \langle \tau_{g_1}, \ldots, \tau_{g_n} \rangle, \tau_{g_{res}}), -, - \rangle_{val} \\ e_i\ \rho\ \tau_{g_i} \Rightarrow V_i \simeq \langle v_i, e'_i, \tau_i \rangle_{val},\ i \in 1 \ldots\ n \\ \mathit{find\text{-}mam}(ms, \langle V_1, \ldots, V_n \rangle) \simeq \langle V_f, \mathit{static?} \rangle \\ V_f \simeq \langle \mathit{closure}(\langle \llbracket x_1 \rrbracket, \ldots, \llbracket x_n \rrbracket \rangle, \langle \tau_{arg_1}, \ldots, \tau_{arg_n} \rangle, \tau_{res},\ e_f,\ \rho_f), -, - \rangle_{val} \\ (\mathit{specialize?}, \langle \mathit{spec\text{-}type}_1, \ldots \rangle) = \mathit{choose\text{-}specialization}(V_g, V_f, \langle V_1, \ldots, V_n \rangle) \\ \mathit{arg\text{-}type}_i = (\mathit{specialize?} \Rightarrow \mathit{spec\text{-}type}_i;\ \mathit{glb}(\tau_i, \tau_{arg_i})),\ i \in 1 \ldots\ n \\ e_f\ \rho_f[x_i \mapsto \langle v_i, e'_i, \mathit{arg\text{-}type}_i \rangle_{val}]\ \mathit{glb}(\tau, \tau_{res}) \Rightarrow V,\ i \in 1 \ldots\ n \end{array}}{\begin{array}{l} \llbracket (\mathsf{gf\text{-}call}\, e_0\ e_1\ \ldots\ e_n) \rrbracket\ \rho\ \tau \Rightarrow \mathcal{F}(e, \mathit{F\text{-}vals}, \rho, \tau),\ \text{where} \\ \quad \mathit{F\text{-}vals} = \langle V, V_g, \langle V_1, \ldots, V_n \rangle, V_f, \mathit{static?}, \mathit{specialize?}, \langle \mathit{spec\text{-}type}_1, \ldots \rangle \rangle \end{array}}$$

$$\frac{\begin{array}{l} e_{\tau_i}\ \rho\ \tau_{type} \Rightarrow V_{\tau_i} \simeq \langle v_{\tau_i}, -, - \rangle_{val},\ i \in 1 \ldots\ n \\ e_{\tau_{res}}\ \rho\ \tau_{type} \Rightarrow V_{\tau_{res}} \simeq \langle v_{\tau_{res}}, -, - \rangle_{val} \\ v_f = \mathit{closure}(\langle \llbracket x_1 \rrbracket, \ldots, \llbracket x_n \rrbracket \rangle, \langle v_{\tau_1}, \ldots, v_{\tau_n} \rangle, v_{\tau_{res}}, \mathrm{e}_0, \rho) \\ V_f = \langle v_f, e, \mathit{eq}(v_f) \rangle_{val}\ ;\quad \mathit{check}(v_f, \tau) \end{array}}{\llbracket (\mathsf{lambda}(x_1\ e_{\tau_1}, \ldots, x_n\ e_{\tau_n}) e_{\tau_{res}}\ .\ \mathrm{e}_0) \rrbracket\ \rho\ \tau \Rightarrow \mathcal{F}(e, \langle V_f, \langle V_{\tau_1}, \ldots, V_{\tau_n} \rangle, V_{\tau_{res}} \rangle, \rho, \tau)}$$

Fig. 1. Λ_{DVM}, A Dynamically typed Λ Calculus with Subtyping and Generic Functions

Environments ρ map from identifiers to extended values. The *dynamic context* is the projection of the environment as a map from identifiers to tagged values (i.e. the tagged value component of the mapped-to extended value). The *static context* is the projection of the environment as a map from identifiers to types (i.e. the type component of the mapped-to extended value).

When evaluation of an expression is complete, the *finish* function $\mathcal{F}$ is called with all the values relevant to the just-finished evaluation. $\mathcal{F}$ has type:
$(\mathit{Exp}, \mathit{Vector}(\mathit{Value}), \mathit{Env}, \mathit{Type}) \to \mathit{ExtValue}$, and $\mathcal{F}$ must maintain the invariant that if $e\ \rho\ \tau \Rightarrow V \simeq \langle v, e', \tau' \rangle_{val}$, the tagged value v satisfies type τ (that is,

Predefined Types: $\top, \bot, \tau_{int}, \tau_{bool}, \tau_{fun}, \tau_{gf}$ for top, bottom, integers, booleans, functions (closures), and generic functions, respectively.
Predefined Functions:

- *and*(*conjunct-types*), *or*(*disjunct-types*), *not*(*type*), *fun*(*arg-types*, *result-type*) – build types from other types.
- *type-of*(v) returns the (concrete) type for a given tagged value v.
- *check*(v, τ) returns true if *type-of*(v) is equal to or a subtype of τ; otherwise, halts with an error.
- *static?*(*ext-val*) For an extended value $\langle v, e, \tau \rangle_{val}$, returns true if $\tau \leq eq(v)$ – that is, if the value is completely static. We use $\leq$ rather than $=$ for type comparison because our type system includes conjunctive types $\tau \& \tau'$ such that $\tau \leq (\tau \& \tau')$ and $\tau' \leq (\tau \& \tau')$.
- *list*($v_1, \ldots, v_n$), *closure*(*var*, *arg-type*, *result-type*, *body-exp*, *closure-env*), *generic*(*list-of-methods*, *arg-type*, *return-type*), *subtype*(*list-of-supertypes*), *eq*(*val*): constructors for lists, closures, generic functions, subtypes, and singleton types, respectively.
- *add-method*(*generic*, *fun*) adds a function (method) to a generic function.
- *glb*(*list-of-types*) constructs the greatest lower bound of its type arguments.
- *find-mam*(*list-of-funs*, *arg-vals*) selects the most applicable method given a set of function ("methods") and a vector of argument values. If there are no applicable methods, *find-mam* halts with a "no applicable methods" error. If there are multiple (non-comparable) most applicable methods, *find-mam* halts with an "ambiguous methods" error. Otherwise, it returns the most applicable method and also a flag indicating whether method selection was static (more on this in Section 4.2).

Fig. 2. Predefined values for Λ_{DVM}

check(v, τ)). For simple interpretation, we instantiate $\mathcal{F}$ as the finish function $\mathcal{F}_{\text{simp}}$:

$$\mathcal{F}_{\text{simp}}(e, \langle V, \ldots \rangle, \rho, \tau) = V$$

$\mathcal{F}_{\text{simp}}$ simply returns the first value in its value vector. Later, we will define a finish function $\mathcal{F}_{\text{pe}}$ that implements dynamic partial evaluation.

4.1 Discussion of Evaluation Rules

In Figure 1, the symbol e always refers to the expression being evaluated. A rule subexpression of the form v = *some-operation* binds v to the result of *some-operation* for use elsewhere in the rule. An expression of the form $v \simeq e'$ deconstructs the value v, binding the variables mentioned in e'. We use the symbol $-$ to indicate that we will not make use of the corresponding component value.

var-ref: Evaluation of a variable reference x looks up the identifier x in the environment, checks that it satisfies the current type context τ, and sends the value to $\mathcal{F}$ (which, in the basic interpreter, simply returns the value).

integer: Evaluation of a numeric literal n checks the value n against the current type context τ, constructs an extended value with a fully-static type, and then sends the value to $\mathcal{F}$.

if-true, if-false: Evaluation of an if expression first evaluates the test expression e_0 in a boolean type context, producing V_0. Either the true branch, e_1, or the false branch, e_2, is evaluated, depending on the truth value of the tagged value component of V_0, and then the resulting value is sent to $\mathcal{F}$, along with V_0.

call: Evaluation of a function call first evaluates the function expression e_0 in a τ_{fun} type context. We then destructure the function value (closure) into its bound variables $\langle [\![x_1]\!], \ldots, [\![x_n]\!] \rangle$, argument types $\langle \tau_{\mathit{arg}_1}, \ldots, \tau_{\mathit{arg}_n} \rangle$, result type τ_{res}, body e_f, and the closure's creation environment ρ_f. Then the call's argument expressions $e_i, i \in 1 \ldots n$ are evaluated in type contexts of τ_{arg_i}. For the values to which the function arguments will be bound, we create new extended values with types that are the greatest lower bound of the static type τ_i of the argument value and the function's argument type τ_{arg_i} for each argument position i . Next the body of the function, e_f, is evaluated with the appropriately extended closure creation environment and with a type context that is the greatest lower bound of the function's return type (τ_{res}) and the current type context (τ). Finally, all relevant values are sent to $\mathcal{F}$.

gf-call: Evaluation of a generic function call first evaluates and destructures its generic function argument. A generic function consists of a triple: a set *ms* of functions, aka "methods", a vector of the argument types τ_{g_i}, and the result type $\tau_{g_{res}}$. All functions $\mathit{closure}(\langle [\![x_1]\!], \ldots, [\![x_n]\!] \rangle, \langle \tau_{\mathit{arg}_1}, \ldots, \tau_{\mathit{arg}_n} \rangle, \tau_{\mathit{res}},\ e_f,\ \rho_f)$ in *ms* must satisfy the following constraints:

1. $\tau_{\mathit{arg}_i} \leq \tau_{g_i}$ for each index i, and
2. $\tau_{\mathit{res}} \leq \tau_{g_{res}}$

The generic function call arguments are then evaluated with respect to the argument types. Then the helper function *find-mam*, discussed in the next section, is called to select the *most applicable method* from the set *ms* given the actual argument values V_i. The function *choose-specialization* decides whether or not to produce a new method for this generic function (using the results of dynamic partial evaluation). If so, *choose-specialization* returns *true* and a vector of argument types. If method specialization is not chosen, the arguments are assigned types as in normal function call – the greatest lower bounds of the declared argument types (of the most applicable method) and the static types of the argument values. For simple interpretation, *choose-specialization* always returns *false*. We discuss other scenarios in Section 5.1. The body e_f of the most applicable method is then invoked as in a normal function call.

abstraction: Evaluation of a lambda expression first evaluates the expressions for the argument types, e_{τ_i}, and result type, $e_{\tau_{res}}$, all of which must satisfy the τ_{type} type. Then a closure is created, the value is checked against the current type context τ, and the closure is sent to $\mathcal{F}$.

4.2 Generic Function Method Selection

The helper function *find-mam*$(ms, \langle V_1, \ldots, V_n\rangle)$ first finds the subset of ms that are applicable given the argument values, $V_i \simeq \langle v_i, e_i', \tau_i\rangle_{val}$:

$$\begin{aligned} ms_{app} = \{f \mid f \in ms \ \&\ f \simeq closure(\langle [\![x_1]\!], \ldots, [\![x_n]\!]\rangle, \langle \tau_{arg_1}, \ldots, \tau_{arg_n}\rangle, \tau_{res}, e_f,\ \rho_f) \\ \&\ \ check(v_i, \tau_{arg_i}), i \in 1 \ldots n\} \end{aligned}$$

If ms_{app} is empty, a "no applicable method" error is flagged and execution halts. Next a set of candidates for the most applicable method is derived (ideally a singleton set):

$$\begin{aligned} mams = \{f \mid\ & f \in ms_{app} \ \&\ f \simeq closure(\langle [\![x_1]\!], \ldots, [\![x_n]\!]\rangle, \langle \tau_{arg_1}, \ldots, \tau_{arg_n}\rangle, \tau_{res}, e_f,\ \rho_f) \\ & \&\ (\not\exists f' \in ms_{app} \text{ s.t.} \\ & \quad f' \simeq closure(\langle [\![x_1']\!], \ldots, [\![x_n']\!]\rangle, \langle \tau'_{arg_1}, \ldots, \tau'_{arg_n}\rangle, \tau'_{res}, e_f',\ \rho_f') \\ & \quad \&\ \tau'_{arg_i} \le \tau_{arg_i} \text{ for any } i \in 1 \ldots n)\} \end{aligned}$$

For any two applicable methods in ms_{app}, if the argument types of one of the methods are all $\le$ the corresponding argument types of the other, the second (less specific) method is removed from consideration. It is not allowed for two methods in a generic function to have identical argument type vectors. The set of most applicable method candidates, *mams*, consists of applicable methods each of which has an argument type vector that is incomparable to the argument type vector of any other method in *mams*. If the set of most applicable method candidates, *mams*, has exactly one element, that element is returned; otherwise, an "ambiguous methods" error is signalled and execution halts.

find-mam must also keep track of whether the selection of the most applicable method can be accomplished using only the type information in the extended value tuples. For example, suppose the relevant type hierarchy is $\tau_2 \le \tau_1$, the single argument value V has a concrete type of τ_2, and there are methods for the generic specialized on τ_2 and τ_1. If the static type of the argument value is τ_1 (that is, $V \simeq \langle v, -, \tau_1\rangle_{val}$, and $check(v, \tau_2)$), the choice of method cannot be determined statically (based on the static type alone). However, if the static type is $eq(v)$ or τ_2, then the method selection is static.

5 Instrumenting for Dynamic Partial Evaluation

Recall that extended values are triples $\langle v : TaggedValue, e : Exp, \tau : Type\rangle_{val}$. In the basic evaluator, only the tagged value component, v, is explicitly used. When dynamic partial evaluation is in effect, the expression and type components of an extended value become meaningful. In particular, the following holds (we write $\Rightarrow_{\mathrm{pe}}$ to indicate $\Rightarrow [\mathcal{F}_{\mathrm{pe}}/\mathcal{F}]$ and $\Rightarrow_{\mathrm{simp}}$ to indicate $\Rightarrow [\mathcal{F}_{\mathrm{simp}}/\mathcal{F}]$, we use a dash $(-)$ for values we do not care about, and we write $v : \tau$ for $check(v, \tau) = true$):

[dpe constraints] If $e\ \rho\ \tau \Rightarrow_{\mathrm{pe}} \langle v, e', \tau'\rangle_{val}$, then

1. $v : \tau'$ (as well as $v : \tau$ from the specification of $\mathcal{F}$),
2. $e\ \rho\ \tau \Rightarrow_{\mathrm{simp}} \langle v, -, -\rangle_{val}$, and
3. For every environment ρ' that *statically matches* ρ, if $e\ \rho'\ \tau'' \Rightarrow_{\mathrm{simp}} \langle v', -, -\rangle_{val}$, then
   (a) $e'\ \rho'\ \tau'' \Rightarrow_{\mathrm{simp}} \langle v', -, -\rangle_{val}$, and
   (b) $v' : \tau'$

An environment ρ' *statically matches* an environment ρ if $dom(\rho) \subseteq dom(\rho')$ and for all $x \in dom(\rho)$, if $\rho(x) = \langle -, -, \tau \rangle_{val}$, then $check(\rho'(x), \tau) = true$. That is, the types of the bound variables match, though the values may be different.

The statement [**dpe constraints**] above specifies that if $e\ \rho\ \tau \Rightarrow_{\mathrm{pe}} \langle v, e', \tau' \rangle_{val}$, then the value v is of type τ' and evaluating the expression with the simple evaluator $\Rightarrow_{\mathrm{simp}}$ will return the same value v. Furthermore, evaluating the *residualized expression* e' in a statically matching environment ρ' will produce the same value as evaluating the original expression e in ρ'. In other words, any optimizations that were done to produce e' from e depended only on the types of the values in the environment ρ – that is, the static context. Finally, all values produced by evaluating e' in any statically matching environment will be of type τ'.

5.1 $\mathcal{F}_{\mathrm{pe}}$ – a finish function that implements dynamic partial evaluation

The finish function $\mathcal{F}_{\mathrm{pe}}$ implements dynamic partial evaluation and is defined by structural induction on its expression argument. In the following, we present each case for $\mathcal{F}_{\mathrm{pe}}$ along with some discussion.

Variable Reference: $\mathcal{F}_{\mathrm{pe}}([\![x]\!], \langle V \rangle, \rho, \tau)$, where $V \simeq \langle v, e', \tau' \rangle_{val}$

```
= if static?(V) & expressible-as-literal?(v)   – if static and a literal
    ⟨v, [[v]], τ'⟩_val   – we can fold to a constant
    V
```

Both the value and the static type for a variable reference come directly from the environment. If the value is completely static and expressible as a literal (that is, an integer or a boolean), the variable reference may be replaced by the corresponding literal expression.

Literals: $\mathcal{F}_{\mathrm{pe}}([\![n]\!], \langle V \rangle, \rho, \tau) = V$

A literal is always completely static – that is, a given literal expression will always return the same value, no matter in what static context it is evaluated. The reduction relation $\Rightarrow$ ensures that for literals, $\mathcal{F}$ will be called with a fully-static value.

Conditionals: $\mathcal{F}_{\mathrm{pe}}([\![(\mathsf{if}\, e_0\ e_1\ e_2)]\!], \langle V, V_0 \rangle, \rho, \tau)$, where $V \simeq \langle v, e', \tau' \rangle_{val}$ and $V_0 \simeq \langle v_0, e_0', \tau_0 \rangle_{val}$

```
= if static?(V_0)   – if test val is a constant,
    V   – we can eliminate the conditional
    if v_0      – otherwise, rebuild the conditional
      ⟨v, [[(if e'_0 e' e_2)]], ⊤⟩_val
      ⟨v, [[(if e'_0 e_1 e')]], ⊤⟩_val
```

The decision whether or not to fold a conditional expression depends on whether or not the test expression is completely static. If the test expression is completely static, the if expression folds away. Otherwise, we rebuild the conditional with the residuals of the the test expression and the chosen branch. Note that the static type returned for the value is only $\top$.

An important optimization for conditionals is the case when e_0 is a variable reference. In that case, the chosen branch may be evaluated with the environment mapping the test variable to the singleton type of either *eq*(*true*) or *eq*(*false*). This is in fact always the case for the Dynamic Virtual Machine, where expressions are all in essentially static single assignment form, but we do not present that optimization here.

Function call: $\mathcal{F}_{\text{pe}}(\llbracket(\mathsf{call}\, e_0\ e_1\ \ldots\ e_n)\rrbracket, \langle V, V_f, \langle V_1, \ldots, V_n\rangle\rangle, \rho, \tau)$, where $V \simeq \langle v, e', \tau'\rangle_{val}$, $V_f \simeq \langle v_f, e'_f, \tau_f\rangle_{val}$, $V_i \simeq \langle v_i, e'_i, \tau_i\rangle_{val}, i \in 1 \ldots n$, and $v_f \simeq closure(\langle\llbracket x_1\rrbracket, \ldots, \llbracket x_n\rrbracket\rangle, \langle\tau_{arg_1}, \ldots, \tau_{arg_n}\rangle, \tau_{res},\ \rho_f)$

```
= if static?(V_f)   – if fun is a constant,
    if static?(V)   – and return val is a constant,                          finish-known-function
      ⟨v, val2exp(v), τ'⟩_val   – fold call
      if inline?(v_f)   – if fun is a constant, may choose to inline
        inline-call(V, V_f, ⟨V_1, ..., V_n⟩, ρ, τ)
        – here if fun is constant, but not inlining
        ⟨v, [[(call e'_f e'_1 ... e'_n)]], τ_res⟩_val
    – here if fun value is not constant
    ⟨v, [[(call e'_f e'_1 ... e'_n)]], ⊤⟩_val
```

Finishing a function call involves choosing one of several options:

1. **Fold the call to either a literal or a variable reference expression.** This can only be done if the value of the call is completely static. Folding to a variable reference may involve extending the current environment with a new binding. Residualizing a constant is handled by the function *val2exp*, whose logic is outside the scope of this paper.
2. **Inline the residualized body of the closure at the call site;** preceded by any argument type checks that did not statically succeed. Inlining may also involve extending the current environment, to handle references to variables closed over by the inlined function. Inlining is handled by the function *inline-call*, again outside the scope of this paper.
3. **Replace the call by an unchecked call;** preceded by any argument type checks that did not statically succeed.
4. **Leave the call as is.**

For option 3, the language needs to be extended with operations that do less type checking than base Λ_{DVM}, be we do not discuss those in this paper. Note that if the function value is static, but we choose not to inline, we may still use the the declared return type of the function for the static type of the returned value.

Generic Function Call: $\mathcal{F}_{\text{pe}}(\llbracket(\mathsf{gf\text{-}call}\, e_0\ e_1\ \ldots\ e_n)\rrbracket, F\text{-}vals, \rho, \tau)$, where $F\text{-}vals = \langle V, V_g, \langle V_1, \ldots, V_n\rangle, V_f, static?, specialize?, \langle spec\text{-}type_1, \ldots\rangle\rangle$, $V \simeq \langle v, e', \tau'\rangle_{val}$, $V_g \simeq \langle v_g, -, -\rangle_{val}$, $v_g \simeq generic(ms, \langle\tau_{g_1}, \ldots, \tau_{g_n}\rangle, \tau_{g_{res}})$, $V_f \simeq \langle closure(\langle\llbracket x_1\rrbracket, \ldots, \llbracket x_n\rrbracket\rangle, \langle\tau_{arg_1}, \ldots, \tau_{arg_n}\rangle, \tau_{res},\ e_f,\ \rho_f), -, -\rangle_{val}$, and $V_i \simeq \langle v_i, e'_i, \tau_i \rangle_{val}$, $i \in 1 \dots n$

```
= $\textit{if specialize?}$
    $\textit{add-method}(v_g, \textit{closure}(\langle [\![x_1]\!], \dots, [\![x_n]\!] \rangle, \langle \textit{spec-type}_1, \dots \rangle, e', \rho_f))$
    $\textit{if static?}(V_g)$  – if generic is a constant,
      $\textit{if}$ ($\textit{static?}$)  – if can statically dispatch
        $\dots \textit{finish-known-function} \dots$
        $\langle v, [\![(\mathsf{gf\text{-}call}\ e'_g\ e'_1 \dots e'_n)]\!], \tau_{g_{res}} \rangle_{val}$  – can't statically dispatch
      – here if generic is not a constant
      $\langle v, e, \top \rangle_{val}$
```

If method specialization was chosen, finishing a generic function call adds a new method to the generic function[2], using the residualized body of the applied method for the closure body, the argument types determined by *choose-specialization*, and other attributes from the original closure.

Technical note: Actually, the closure environment is extended with bindings for any static values exposed during inlining that are not expressible as literals and instead bound to fresh variables.

In the Dynamic Virtual Machine, the logic abstracted by *choose-specialization* simply uses programmer-defined rules, in the spirit of [VCC97], to decide when to create specialized versions of generic function methods.

If method specialization was not chosen, then we test whether the generic function value is constant and the most applicable method can be chosen based strictly on the static types of its arguments. In that case, finishing proceeds as in the case for a regular function call when the function argument is static. Recall from Section 4.2 that method selection is considered not static if at least one of the methods of the generic function is not applicable according to the concrete argument types, but is *potentially* applicable according to the static types of the arguments.

Abstraction: $\mathcal{F}_{\text{pe}}([\![(\mathsf{lambda}(x_1\ e_{\tau_1}, \dots, x_n\ e_{\tau_n}) e_{\tau_{res}}\ .\ e_0)]\!], \langle V_f, \langle V_{\tau_1}, \dots, V_{\tau_n} \rangle, V_{\tau_{res}} \rangle, \rho, \tau)$, where $V_f \simeq \langle v_f, e, \tau_f \rangle_{val}$, $V_{\tau_i} \simeq \langle v_{\tau_i}, e'_{\tau_i}, - \rangle_{val}$ for $i \in 1 \dots n$, $V_{\tau_{res}} \simeq \langle v_{\tau_{res}}, e'_{\tau_{res}}, - \rangle_{val}$

$$= \langle v_f, [\![(\mathsf{lambda}(x_1\ e'_{\tau_1}, \dots, x_n\ e'_{\tau_n}) e'_{\tau_{res}}\ .\ e_0)]\!], \tau_f \rangle_{val}$$

Lambda abstraction produces a closure value. To finish an abstraction, we return the closure, a rebuilt abstraction expression using the residual expressions from the type expressions, and the singleton type constructed by $\Rightarrow$.

6 Examples of Dynamic Partial Evaluation

We give a few examples of how dynamic partial evaluation works in practice.

6.1 A Contrived Example

Suppose we are dynamically partially evaluating the following expression (where $(\mathsf{let}(x\ \tau) = e_0\ \mathsf{in}\ e_1)$ is a macro for $((\mathsf{lambda}(x\ \tau)\top\ .\ e_1)\ e_0)$):

[2] In the DVM, method addition is contingent on there being some useful optimization during specialization.

(let $(a\ \top) = $ (if $(>\ b\ 0)\ 3\ 4)$ in
  (let $(c\ \tau_{int}) = $ (if $(>\ d\ 0)\ 5\ 6)$ in
    (gf-call $g\ a\ b$)))

in the following environment:

$b \mapsto \langle 1, 1, \mathit{eq}(1)\rangle_{val}$

$d \mapsto \langle 1, 1, \tau_{int}\rangle_{val}$

$g \mapsto \langle \mathit{generic}(\langle g_1, g_2\rangle, \langle\top\rangle, \top), -, \mathit{eq}(g)\rangle_{val}$,

  where $g_1 = \mathit{closure}(\langle x, y\rangle, \langle\top, \top\rangle, \top, [\![(+\ x\ y)]\!], \rho_{g_1})$

    $g_2 = \mathit{closure}(\langle x, y\rangle, \langle\top, \tau_{int}\rangle, \top, [\![x]\!], \rho_{g_2})$

The identifier b is statically bound to the integer 1. d is also bound to 1, but has static type τ_{int}. The identifier g is statically bound[3] to a generic function of two methods – one with the most general specializers, and one specialized on integer values for its second argument. The first expression to evaluate is $(>\ b\ 0)$. Because b is completely static, the result value, *true* is completely static and the if expression can be folded. The result of the if expression is the fully static value 3. The identifier a is bound to the value 3, with residual expression 3, and gets static type $\mathit{eq}(3)$, which is the greatest lower bound (glb) of the declared type $\top$ and the result type $\mathit{eq}(3)$ of the expression. The comparison $(>\ d\ 0)$ also evaluates to *true*, but the value is not completely static because the static type of d is τ_{int}. The identifier c is bound to the value 5, residual expression (if $(>\ d\ 0)\ 5\ 6$), and gets static type τ_{int}, which is the glb of the declared type τ_{int} and the type of the result of the expression, namely $\top$. The most applicable method for the call to g is g_2. Furthermore, static type τ_{int} of variable c is sufficient to statically select the most applicable method at the call, so the gf-call can be replaced by a simple call to g_2. When the body of g_2 is executed, it returns the value of its first argument, a, which is fully static. Because the method can be statically selected, and because the result of the call is a fully static value, the whole gf-call expression can be folded to the literal expression 3. Thus the expression residualizes to:

$$\text{let } (c\ \tau_{int}) = (\text{if } (>\ d\ 0)\ 5\ 6) \text{ in } 3$$

Dead variable elimination may eliminate the now useless let construct.

6.2 Example: Dynamic Partial Evaluation of Reflection in Java

In [BN00], Braux and Noyé use partial evaluation techniques to eliminate reflection overhead in Java. The rules they introduce are specific to the reflection API of Java. Dynamic partial evaluation provides a general mechanism that automatically eliminates the reflection overhead addressed by Braux and Noyé. Following is the main example from [BN00]:

```
public static void dumpFields(Object anObj) throws java.lang.IllegalAccessException {
    Field[] fields = anObj.getClass().getFields();
    for (int i = 0; i < fields.length; i++)
        System.out.println(fields[i].getName() + ":  " + fields[i].get(anObj));
}
```

If `dumpFields` is called often on a specific class, say `Point`, it is worthwhile to create a specialized version of `dumpFields` specific to `Point`. Assume for

[3] $g \mapsto \langle -, -, \mathit{eq}(g)\rangle_{val}$ means that g is completely static.

now that the `Point` class has no subclasses. Within the specialized version of `dumpFields`, most of the reflection overhead can be folded away – the call to `getClass` will always return the `Point` class, and `getFields` will always return an array containing the `x` and `y` Fields. Of course, the actual values of x and y are dynamic – that is, they will will vary between invocations of the method. After partial evaluation, the specialized method should be something like:

```
public static void dumpFieldsPoint(Point anObj) {
    System.out.println("x: "+anObj.x); System.out.println("y: "+anObj.y); }
```

We have an implementation of the relevant parts of the Java runtime, and a translation from Java into the Dynamic Virtual Machine (DVM). The above example, in the case that `Point` has no subclasses, folds to DVM code analogous to that given above, and a method specialized on `Point` is added to the `dumpFields` generic function. Thereafter, calls to `dumpFields` with `Point` arguments automatically select the optimized version.

Furthermore, if later calls to `dumpFields` take place within the context of specializing some other generic function, and the argument is statically bound to `Point`, the optimized code may be inlined into the calling method, and so on.

7 Conclusions and Future Work

Dynamic partial evaluation is a technique for instrumenting interpretation in order to perform partial evaluation actions as a side effect of evaluation. This is accomplished by interpreting expressions in an environment that maps identifiers not only to values but also to types. The type of a variable can be understood as "how much information dynamic partial evaluation is allowed to assume about this binding."

Dynamic partial evaluation has been implemented as part of a *Dynamic Virtual Machine* designed to host dynamic, reflective, higher-order languages with subtyping. In the current implementation, dynamic partial evaluation is always "on" – that is, evaluation always creates residual expressions and tracks static types. We would like to be able to dynamically switch between dynamic partial evaluation and simple interpretation – suffering the overhead of dynamic partial evaluation only when we know we will use the results.

As far as when to enable dynamic partial evaluation of a method, we currently specify rules by hand, in the spirit of [VCC97]. We plan to use dynamically generated profile data to decide when and where to do dynamic partial evaluation. Note that we are focusing on highly reflective runtime environments, so profile data should be readily available. We also plan on using more sophisticated techniques for deciding when to inline function bodies.

The current implementation of dynamic partial evaluation includes "optimistic" optimization with respect to rarely mutable values. We give a brief description of this technique in Appendix A, but leave in depth presentation to another forum.

Acknowledgements

Many thanks to Julia Lawall for reading initial drafts of this paper – the paper is much better thanks to her insightful comments. Thanks also to Mitch Wand for valuable discussions.

References

[BN00] Mathias Braux and Jacques Noyè. Towards partially evaluating reflection in java. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Evaluation and Semantics-Based Program Manipulation (PEPM-00)*, pages 2–11, N.Y., January 22–23 2000. ACM Press.

[Cha92] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Computer Science Department, Stanford University, March 1992.

[CN96] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 145–156, New York, NY, USA, 1996. ACM Press.

[DCG95] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 93–102, La Jolla, California, 18–21 June 1995. *SIGPLAN Notices* 30(6), June 1995.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* The Java Series. Addison-Wesley, Boston, Mass., 2000.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[KdR91] Gregor Kiczales and Jim des Rivieres. *The art of the metaobject protocol.* MIT Press, Cambridge, MA, USA, 1991.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In Norman Meyrowitz, editor, *Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 147–155, Orlando, FL, USA, October 1987. ACM Press.

[MMAY95] Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai, and Akinori Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In *OOPSLA '95 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 300–315. ACM Press, 1995.

[MY98] Hidehiko Masuhara and Akinori Yonezawa. Design and partial evaluation of meta-objects for a concurrent reflective language. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 418–439. Springer, 1998.

[PAB$^+$95] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proc. 15th ACM Symposium on Operating Systems Principles*, Copper Mountain CO (USA), December 1995. http://www.irisa.fr/EXTERNE/projet/lande/consel/papers/spec-sosp.ps.gz.

[PMI88] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. In USENIX Association, editor, *Computing Systems, Winter, 1988.*, volume 1, pages 11–32, Berkeley, CA, USA, Winter 1988. USENIX.

[SLCM99] U. P. Schultz, J. L. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. *Lecture Notes in Computer Science*, 1628:367–??, 1999.

[VCC97] Eugen N. Volanschi, Charles Consel, and Crispin Cowan. Declarative specialization of object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*, volume 32, 10 of *ACM SIGPLAN Notices*, pages 286–300, New York, October 5–9 1997. ACM Press.

A Optimistic Dynamic Partial Evaluation

As was mentioned in the introduction, we want to optimize with respect to "quasi-invariants" – in particular, elements of the meta-object protocol (MOP) that are technically mutable but rarely modified in practice. In the interest of simplicity, we have omitted from this paper the machinery to perform *optimistic dynamic partial evaluation*, but the Dynamic Virtual Machine does implement dynamic partial evaluation with respect to quasi-invariants. We briefly describe our implementation of optimistic dynamic partial evaluation.

In the Dynamic Virtual Machine, there are two mutable datatypes: cells and generic functions. A cell contains a single value that may be changed, and a generic function may be modified by updating its method list. As dynamic partial evaluation proceeds, each optimization (folding, inlining) notes any cells or generic functions that have been referenced. When a new method is added to a generic function as a result of dynamic partial evaluation, all referenced cells and referenced generic functions are instrumented to undo the optimization if mutated. For example, suppose we are evaluating a generic function call `(gf-call g x)`, and let us call the most applicable method `g1`. Further suppose that method `g1` has a body that calls generic function `h`. When the generic function call finishes, we have a specialized version of `g1` – let's call it `g2` – that is added to the method list of `g`. The dependency on generic function `h` is registered, and if at some later point the methods of `h` are modified, then `g2` will be removed from the method list of `g`. In the Dynamic Virtual Machine, cells and generic functions exposed by the MOP are considered "quasi-invariant" and dynamic partial evaluation tracks references to them.