

A Calculus for Compiling and Linking Classes

Kathleen Fisher¹, John Reppy², and Jon G. Riecke²

¹ AT&T Labs — Research
180 Park Avenue, Florham Park, NJ 07932 USA
kfisher@research.att.com

² Bell Laboratories, Lucent Technologies
700 Mountain Avenue, Murray Hill, NJ 07974 USA
{jhr, riecke}@bell-labs.com

Abstract. We describe a low-level calculus, called $\lambda\textit{ink}\zeta$ (pronounced “links”), designed to serve as an intermediate representation in compilers for class-based object-oriented languages. The calculus fills two roles. First, its primitives can express a wide range of class-based object-oriented language features, such as class construction and various forms of method dispatch. Second, it allows the compiler to specify class linking directly in $\lambda\textit{ink}\zeta$. In particular, the calculus can model the class systems of MOBY [FR99a], OCAML [RV98,Ler98], and LOOM [BFP97], where subclasses may be derived from unknown base classes, or *just-in-time* linking schemes like that of JAVA [AG98], where an application loads and links classes dynamically.

1 Introduction

Class-based object-oriented languages provide mechanisms for factoring implementations over a hierarchy of classes. For example, the implementation of a text window may be split into a base class that implements windows and a subclass that supports drawing text. Since these classes may be defined in separate compilation units, the compiler needs an internal representation (IR) that allows it to represent code fragments (e.g., the code for a subclass) and to generate linkage information to assemble the fragments. For languages with manifest class hierarchies (*i.e.*, languages where subclass compilation requires the superclass representation, as is the case in C++ [Str97] and JAVA [AG98]), representing code fragments and linkage information is straightforward. But for languages that allow classes as module parameters, such as MOBY [FR99a] and OCAML [RV98,Ler98], or languages that have classes as first-class values, such as LOOM [BFP97], the design of an IR becomes less clear. This paper introduces an untyped IR for these languages, called $\lambda\textit{ink}\zeta$, describes how to translate languages into the IR, presents theorems about linking, and discusses optimizations based on the IR.

Figure 1 gives an example of where difficulties can arise in compiling languages with classes as module parameters. The example is written in MOBY, although similar examples can be written in LOOM and OCAML. The module in Figure 1 defines a class `ColorPt` that extends an unknown base class `Pt.Point` by inheriting its `getX` and `getY` methods, overriding its `move` method, and adding a `color` field. When compiling the module, the compiler knows only that the `Pt.Point` superclass has three methods (`getX`, `getY`, and `move`). The compiler does not know what order these methods appear in the `Point` class’s method suite, nor what other private methods and fields the `Point` class might have.

```

signature PT {
  class Point : {
    public meth getX : Unit -> Int
    public meth getY : Unit -> Int
    public meth mv : (Int, Int) -> Unit
  }
}

module ColorPtFn (Pt : PT) {
  class ColorPt {
    inherits Pt.Point
    field c : Color
    public meth move (x : Int, y : Int) -> Unit {
      if (self.c == Red)
        then super.move(2*x, 2*y)
        else super.move(x, y)
    }
  }
}

```

Fig. 1. Inheriting from an unknown superclass

To handle this situation, the compiler’s IR must provide support for an *abstract* representation of the superclass. This paper presents $\lambda\text{ink}\zeta$, which provides a foundation for compiler IRs that support non-manifest superclasses. $\lambda\text{ink}\zeta$ is an untyped λ -calculus, extended with *method suites*, which are ordered collections of methods, *slots*, which index into method suites, and *dictionaries*, which map method labels to slots. In $\lambda\text{ink}\zeta$, method dispatch is implemented by first using a dictionary to find the method’s slot and then using the slot to index into a method suite. This separation of dynamic dispatch into two pieces also enables more compiler optimizations.

As an example of inheritance from a non-manifest base class, suppose we have a class `PolarPt` that implements the `Pt.Point` interface and has additional polar-coordinate methods `getTheta` and `getRadius`. When we apply the `ColorPtFn` module to `PolarPt`, we effectively hide the polar-coordinate methods, making them private. Such private methods, while hidden, are not forgotten, since they may be indirectly accessible from other visible methods (*e.g.*, the `PolarPt` class might implement `getX` in terms of polar coordinates). This hiding is a problem when compiling the `PolarPt` class, since the code for this class must have access to methods that might not be in the dictionaries of its eventual subclasses. We support this access in $\lambda\text{ink}\zeta$ by dynamically changing the dictionary used by an object when its view of itself changes [RS98,FR99b].

The main contribution of this paper is its formalization of data structures that are used in compilers for object-oriented languages. $\lambda\text{ink}\zeta$ codifies many of the ad-hoc optimizations made in such compilers. Reductions in $\lambda\text{ink}\zeta$ can, for instance, express the linking of classes from code fragments and allow us to support a variety of class-linking policies, ranging from static linkage to dynamic class loading. The reductions can also be used to

optimize method invocations through **super**, and even eliminate some of the calculations needed to send messages in more dynamic settings.

We deliberately chose to make *link_ς* untyped. Our goal is to use *link_ς* as an IR for languages with different type systems—e.g., JAVA and MOBY—and fixing a type system prematurely can limit the applicability of the IR. The type system we use for proving strong normalization (Section 3 below) could provide a basis for a type system for *link_ς*.

In the next section, we present the syntax, operational semantics, and rewrite systems of *link_ς*. To keep the discussion focused, we restrict the technical presentation to a version of *link_ς* with methods, but no fields (instance variables). The techniques used to handle methods apply directly to fields (see Section 4.1). Section 3 defines a simple class language SCL and shows how it can be translated to *link_ς*. We prove that the translation of any “well-ordered” SCL program has the property that all linking steps can be reduced statically. In Section 4, we sketch how *link_ς* can serve as an IR for MOBY, LOOM, a mixin extension for SCL, and C++. Section 5 further demonstrates the utility of the rewriting system for *link_ς* by showing how method dispatch can be optimized in the calculus. We conclude with a discussion of related and future work.

2 *link_ς*

link_ς is a λ -calculus extended with method suites, slots, and dictionaries, which provides a rich notation for class assembly, inheritance, dynamic dispatch, and other object-oriented features.

2.1 Syntax

The syntax of *link_ς* is given by the grammar in Figure 2. In addition to the standard λ -calculus forms, there are eight expression forms for supporting objects and classes. The term $\langle e_1, \dots, e_n \rangle$ constructs a method suite from the expressions e_1, \dots, e_n , where each e_i is assigned slot i . The expression $e@e'$ extracts the value stored in the slot denoted by e' from the method suite denoted by e . The method suite extension $e||\langle e_1, \dots, e_n \rangle$ adds n slots to the suite denoted by e . The last method suite operation is override, which functionally updates a slot in a given suite to produce a new suite. A slot is specified either as an integer i or as a positive offset from another slot. The expression $\{m_1 \mapsto e_1, \dots, m_n \mapsto e_n\}$ denotes a dictionary where each label m_i is mapped to the slot denoted by e_i . Application of a dictionary to a label m is written $e!m$.

We identify terms up to the renaming of bound variables and use $e[x \mapsto e']$ to denote the capture-free substitution of e' for x in e . We assume that dictionaries are unordered and must represent finite functions. For instance, the dictionary $\{m \mapsto 1, m \mapsto 2\}$ is an ill-formed expression. To simplify notation, we use the following shorthands:

$$\begin{aligned} \mathbf{let } x = e \mathbf{ in } e' \quad \mathbf{for} \quad & (\lambda x.e')(e) \\ \lambda(x_1, \dots, x_n).e \quad \mathbf{for} \quad & \lambda p.((\lambda x_1. \dots \lambda x_n.e) (\pi_1 p) \dots (\pi_n p)) \end{aligned}$$

$e ::= x$	variable
$\lambda x.e \mid e(e')$	function abstraction/application
$(e_1, \dots, e_n) \mid \pi_i e$	tuple creation/projection
$\langle e_1, \dots, e_n \rangle$	method suite construction
$e@e'$	method suite indexing
$e \parallel \langle e_1, \dots, e_n \rangle$	method suite extension
$e@e' \leftarrow e''$	method override
i	slot
$e + i$	slot addition
$\{m_1 \mapsto e_1, \dots, m_n \mapsto e_n\}$	dictionary construction
$e!m$	dictionary application

Fig. 2. The syntax of $\lambda ink\zeta$

2.2 Operational semantics

We specify the operational semantics of $\lambda ink\zeta$ using an evaluation-context based rewrite system [FF86]. Such systems rewrite terms step-by-step until no more steps can be taken. At each step, the term to be reduced is parsed into an evaluation context and a redex. The redex is then replaced, and evaluation begins anew with another parsing of the term. Note that since $\lambda ink\zeta$ is untyped there are legal expressions, such as $\pi_i (\lambda x.e)$, that cannot be reduced.

Two grammars form the backbone of the semantics. The first describes values, a subset of expressions that are in reduced form:

$$v ::= x \mid \lambda x.e \mid (v_1, \dots, v_n) \mid \langle v_1, \dots, v_n \rangle \mid i \mid \{m_1 \mapsto v_1, \dots, m_n \mapsto v_n\}$$

The second grammar describes the set of evaluation contexts.

$$\begin{aligned} E ::= & [\cdot] \mid E(e) \mid v(E) \mid \pi_i E \\ & \mid \langle v_1, \dots, E, \dots, e_n \rangle \mid E \parallel \langle e_1, \dots, e_n \rangle \mid v \parallel \langle v_1, \dots, E, \dots, e_n \rangle \\ & \mid E@e \leftarrow e \mid v@E \leftarrow e \mid v@v \leftarrow E \mid E@e \mid v@E \\ & \mid E + i \mid \{m_1 \mapsto v_1, \dots, m_i \mapsto E, \dots, m_n \mapsto e_n\} \mid E!m \end{aligned}$$

The primitive reduction rules for $\lambda ink\zeta$ are given in Figure 3. We write $e \mapsto e'$ if $e = E[e_0]$, $e' = E[e'_0]$, and $e_0 \hookrightarrow e'_0$ by one of the rules above.

2.3 Reduction system

Under the operational semantics, there is no notion of transforming a program before it is run: all reductions happen when they are needed. We want, however, a method for rewriting $\lambda ink\zeta$ terms to equivalent, optimized versions. The basis of the rewrite system is the relation \hookrightarrow . We write \rightarrow for the congruence closure of this relation; *i.e.*, for the system in which rewrites may happen anywhere inside a term. For example, reductions like $(\lambda x.\pi_1 (v_1, x))(e) \rightarrow (\lambda x.v_1)(e)$ are possible, whereas in the operational semantics they are not. We write \rightarrow^* for the reflexive, transitive closure of \rightarrow .

$$\begin{aligned}
& (\lambda x.e)(v) \hookrightarrow e[x \mapsto v] \\
& \pi_i(v_1, \dots, v_n) \hookrightarrow v_i \quad \text{where } 1 \leq i \leq n \\
& i + j \hookrightarrow k \quad \text{where } k = i + j \\
& \{m_1 \mapsto v_1, \dots, m_n \mapsto v_n\}!m_i \hookrightarrow v_i \quad \text{where } 1 \leq i \leq n \\
& \langle v_1, \dots, v_n \rangle || \langle v'_1, \dots, v'_n \rangle \hookrightarrow \langle v_1, \dots, v_n, v'_1, \dots, v'_n \rangle \\
& \langle v_1, \dots, v_i, \dots, v_n \rangle @i \leftarrow v' \hookrightarrow \langle v_1, \dots, v', \dots, v_n \rangle \quad \text{where } 1 \leq i \leq n \\
& \langle v_1, \dots, v_n \rangle @i \hookrightarrow v_i \quad \text{where } 1 \leq i \leq n
\end{aligned}$$

Fig. 3. Reduction rules for link_ζ

The reduction system will be used in the next two sections when we discuss static linking for a simple class language and optimizations. The reduction relation \rightarrow is non-deterministic: multiple paths may emanate from a single expression, but like many λ -calculi the relation \rightarrow is confluent:

Theorem 1 If $e \rightarrow^* e'$ and $e \rightarrow^* e''$, there is an e''' such that $e' \rightarrow^* e'''$ and $e'' \rightarrow^* e'''$.

The proof uses the Tait-Martin-Löf parallel moves method [Bar84]; we omit the proof.

3 A simple class language

We illustrate the utility of link_ζ by showing how it can serve as the IR for an idealized class language called SCL. We give a formal translation from SCL into link_ζ . In this translation, certain operations in link_ζ are labeled as linking operations. Intuitively, such operations may be eliminated at link time. We show that the linking operations for SCL can be eliminated by a procedure that always terminates.

The syntax of SCL is given in Figure 4. A program consists of a sequence of one or more class declarations followed by an expression. There are two forms of class declaration. The first is a *base-class declaration*, which defines a class as a collection of methods. The second form is a *subclass declaration*, which defines a class by inheriting methods from a *superclass*, overriding some of them, and then adding new methods. The subclass constrains the set of methods it visibly inherits from its superclass by listing the names of such methods as $\{ m^* \}$. Unmentioned method names defined in the superclass are inherited by the subclass, but such methods are not visible in the subclass. This restriction operation models the way that MOBY supports private members [FR99b,FR99a]; it subsumes the more restricted annotation mechanisms found in JAVA and other languages. Methods in this language take exactly one argument and have expressions for bodies. For simplicity our expression language has only those features relevant to the linking of classes: variables including **self**, method dispatch, super-method dispatch, and object creation.

$prog ::= dcl\ prog$	Programs
exp	
$dcl ::= \mathbf{class}\ C\ \{\ meths\ \}$	Class declarations
$\mathbf{class}\ C\ \{\ \mathbf{inherit}\ C' : \{\ m^*\ \}\ meths\ \}$	
$meths ::= \epsilon$	
$meth\ meths$	
$meth ::= m(x)exp$	Methods
$exp ::= x$	Expressions
\mathbf{self}	
$exp \leftarrow m(exp)$	
$\mathbf{super} \leftarrow m(exp)$	
$\mathbf{new}\ C$	

Fig. 4. The syntax for SCL

3.1 Translation to $\lambda ink\varsigma$

To translate SCL into $\lambda ink\varsigma$, we must translate classes, objects, and methods. Each fully-linked class is translated to a triple (σ, ϕ, μ) , where σ is the size of the class (*i.e.*, the number of slots in its method suite), ϕ is a dictionary for mapping method names to method-suite indices, and μ is the class's method suite. Each object is translated to a pair of the object's method suite and a dictionary for resolving method names. Each method is translated into a *pre-method* [AC96]; *i.e.*, a function that takes **self** as its first parameter. The translation is defined by the following functions:

$\mathcal{P}[\![prog]\!]_{\Gamma}$	Program translation
$\mathcal{C}[\![dcl]\!]_{\Gamma}$	Class translation
$\mathcal{M}[\![meth]\!]_{\mu_{super}, \phi_{super}, \phi_{self}, \Gamma}$	Method translation
$\mathcal{E}[\![exp]\!]_{\mu_{super}, \phi_{super}, \phi_{self}, \Gamma}$	Expression translation

Each of these functions take a *class environment* Γ as a parameter mapping the name of a class to its $\lambda ink\varsigma$ representation. A class environment is simply a tuple of class triples, intuitively indexed by class names (other implementations might be possible as well). We will write $\Gamma(C)$ to denote the position in the tuple associated with class C , and $\Gamma \pm \{C \mapsto e\}$ to denote the tuple with the class tuple e bound to class C . The program, method, and expression translation functions map the corresponding SCL form into a $\lambda ink\varsigma$ expression, while the class translation function returns a modified class environment.

In addition to a class environment, the method and expression translation functions take additional parameters to translate **self** and **super**. In particular, the dictionary ϕ_{self} is used to resolve message sends to **self**, and the method suite μ_{super} and dictionary ϕ_{super} are used to translate **super** invocations. Each method is translated to a $\lambda ink\varsigma$ pre-method as follows:

$$\mathcal{M}[\![m(x)exp]\!]_{\mu_{super}, \phi_{super}, \phi_{self}, \Gamma} = \lambda(self, x). \mathcal{E}[\![exp]\!]_{\mu_{super}, \phi_{super}, \phi_{self}, \Gamma}$$

Expressions are translated as follows:

$$\begin{aligned}
\mathcal{E}[[x]]_{\mu_{super}, \phi_{super}, \phi_{self}, \Gamma} &= x \\
\mathcal{E}[[\mathbf{self}]]_{\mu_{super}, \phi_{super}, \phi_{self}, \Gamma} &= (\pi_1 self, \phi_{self}) \\
\mathcal{E}[[exp_1 \leftarrow m(exp_2)]]_{\mu_{super}, \phi_{super}, \phi_{self}, \Gamma} &= \mathbf{let} \ obj = \mathcal{E}[[exp_1]]_{\mu_{super}, \phi_{super}, \phi_{self}, \Gamma} \\
&\quad \mathbf{let} \ meth = (\pi_1 obj) @ ((\pi_2 obj)!m) \\
&\quad \mathbf{in} \ meth(obj) (\mathcal{E}[[exp_2]]_{\mu_{super}, \phi_{super}, \phi_{self}, \Gamma}) \\
&\quad \text{where } obj \text{ and } meth \text{ are fresh} \\
\mathcal{E}[[\mathbf{super} \leftarrow m(exp)]]_{\mu_{super}, \phi_{super}, \phi_{self}, \Gamma} &= (\mu_{super} @ (\phi_{super}!m))(self, \mathcal{E}[[exp]]_{\mu_{super}, \phi_{super}, \phi_{self}, \Gamma}) \\
\mathcal{E}[[\mathbf{new} C]]_{\mu_{super}, \phi_{super}, \phi_{self}, \Gamma} &= \mathbf{let} \ (\sigma, \phi, \mu) = \Gamma(C) \ \mathbf{in} \ (\mu, \phi)
\end{aligned}$$

To translate **self**, we extract the method suite of *self* and pair it with the current self dictionary, ϕ_{self} . Note that because of method hiding, ϕ_{self} may have more methods than $(\pi_2 self)$ [RS98,FR99b]. To translate message sends, we first translate the receiver object and bind its value to *obj*. We then extract from this receiver its method suite $(\pi_1 obj)$ and its dictionary $(\pi_2 obj)$. Using dictionary application, we find the slot associated with method *m*. Using that slot, we index into the method suite to extract the desired pre-method, which we then apply to *obj* and the translated argument. We resolve **super** invocations by selecting the appropriate code from the superclass method suite according to the slot indicated in the superclass dictionary. Notice that this translation implements the standard semantics of super-method dispatch; *i.e.*, future overrides do not affect the resolution of super-method dispatch. We translate the **super** keyword to the ordinary variable *self*. In the translation of **new**, we look up the class to instantiate in the class environment. In our simple language, the new object is a pair of the class's method suite and dictionary.

The translation for subclasses appears in Figure 5. Before describing the translation, we first introduce some meta-notation. In the translation, certain subterms are annotated by a superscript *L*; these subterms denote link-time operations that are reduced during class linking. In addition, we use the function $\text{Names}(meths)$ to extract the names of the methods in *meths*.

A subclass *C* is translated to a function *f* that maps any fully-linked representation of its base class *B* to a fully-linked representation of *C*. The body of the linking function *f* has three phases: slot calculation, dictionary definition, and method suite construction. In the first phase, fresh slot numbers are assigned to new methods (σ_n), while overridden (σ_{ov}) and inherited methods (σ_{inh}) are assigned the slots they have in *B*. The size of the subclass method suite (σ_C) is calculated to be the size of *B*'s suite plus the number of new methods. In the dictionary definition phase, each visible method name is associated with its slot number. During method suite construction, the definitions of overridden methods are replaced in the method suite for *B*. The function then extends the resulting method suite with the newly defined methods to produce the method suite for *C*.

For base-class declarations, the translation is similar, except that there are no inherited or overridden methods. We omit the details for space reasons.

Finally, we translate programs as follows:

$$\begin{aligned}
\mathcal{P}[[dcl prog]]_{\Gamma} &= \mathcal{P}[[prog]]_{\Gamma'} \quad \text{where } \Gamma' = \mathcal{C}[[dcl]]_{\Gamma} \\
\mathcal{P}[[exp]]_{\Gamma} &= \mathcal{E}[[exp]]_{\langle \rangle, \{ \}, \{ \}, \Gamma}
\end{aligned}$$

4 Other examples

In this section, we sketch how *link ζ* can be used to compile a number of more substantial class designs.

4.1 MOBY classes

We originally designed *link ζ* to support MOBY’s class mechanism in a compiler that we are writing. Section 3’s SCL models many of the significant parts of MOBY’s class mechanism, including one of its most difficult features to compile, namely its treatment of private names. In particular, MOBY relies on signature matching in its module mechanism to hide private methods and fields [FR99a] (we illustrated this feature with the example in Section 1). Because MOBY signatures define opaque interfaces, the MOBY compiler cannot rely on complete representation information for the superclass of any class it is compiling. Instead, it must use the *class interface* of the superclass (*e.g.*, the `Pt` class in the `PT` signature) when compiling the subclass. SCL models this situation by requiring each subclass to specify in the `inherits` clause which superclass methods are visible.

The main piece missing from SCL is fields (*a.k.a.* instance variables), which requires a richer version of *link ζ* . While fields require extending the representation of objects with per-object instance variables, the details of instance variable access are very similar to those of method dispatch. As with methods, fields require dictionaries to map labels to slots and slot assignment. Dictionary creation and application is the same as for methods. When we create an object using `new`, we use the size of the class’s instance variables as the size of the object to create — object initialization is done imperatively.

4.2 LOOM classes

In the language LOOM [BFP97], the class construct is an expression form, and a deriving class may use an arbitrary expression to specify its base class. Thus, unlike the translation in Section 3, a translation of LOOM to our calculus cannot have the phase distinction between class link-time and run-time. In a translated LOOM program, computation of slots, dictionary construction, method overrides, and method suite extensions can all happen at run-time. The fact that we can use one representation to handle both static and dynamic classes demonstrates the flexibility of our approach.

4.3 Mixins

Mixins are functions that map classes to classes [FKF98], and unlike parameterized modules, mixins properly extend the class that they are applied to (recall that applying `ColorPtFn` to `PolarPt` hid the polar-coordinate interface). Supporting this kind of class extension in *link ζ* requires a bit of programming. The trick is to include a *dictionary constructor function* as an argument to the translated mixin. For example, consider the following mixin, written in an extension of SCL syntax:

```
mixin Print (C <: {show}) {  
  meth print () { stdout <- print(self <- show()) }  
}
```

This mixin adds a `print` method to any class `C` that has a `show` method already. The translation of this mixin to λink_ζ is similar to that of subclasses given in Section 3.1:

```

λ(σC, φC, μC, mkDict) .
  let σprint = σC+1
  let φPrint = mkDict(φC, σprint)
  let pre_print = λ(self) .
    let print = (π1 stdout)@((π2 stdout)!print)
    let show = (π1 self)@(φPrint!show)
    in print(stdout, show(self))
  let μPrint = μC || ⟨pre_print⟩
  in (σprint, φPrint, μPrint)

```

The main difference is that we use the `mkDict` function, supplied at the linking site, to create the extended dictionary. An alternative to this approach is to add a dictionary extension operation to λink_ζ . For purposes of this example, we assume that the surface language does not permit method-name conflicts between the argument class and the mixin, but it is possible to support other policies, such as C++-style qualified method names to resolve conflicts.

4.4 C++ and JAVA classes

For a language with a manifest class hierarchy, such as C++ (without multiple inheritance) or JAVA (without reflection), the language’s static type system provides substantial information about the representation of dictionaries and method suites. (Multiple inheritance and reflection seem to require additional operations beyond those in λink_ζ .) By exploiting this representation information, we can optimize away all of the dictionary-related overhead in such programs, which results in the efficiency of method dispatch that C++ and JAVA programmers expect. The disadvantage of this approach is that it introduces representation dependencies that lead to the so-called *fragile base class* problem, in which changing the private representation of a base class forces recompilation of its subclasses.

5 Optimization

Many compilers for higher-order languages use some form of λ -calculus as their intermediate representation (IR). We designed λink_ζ to serve as the IR for MOBY, a higher-order object-oriented language [FR99a]. In this section, we show that the techniques commonly used in λ -calculus-based compilers can be used to optimize our encoding of method dispatch in λink_ζ . In general, method dispatch in SCL requires an expensive lookup operation to map a method’s label to its method-suite slot. Often, however, it is possible to apply optimizations to reduce or eliminate this cost. It is important to note that the optimizations described in this section also apply to objects with instance variables. Even though instance variables are mutable, the optimizations focus on the dictionary operations, which are pure.

For the purposes of this section, we assume that we are optimizing well-typed programs that do not have run-time type errors (see Fisher and Reppy [FR99b] for an appropriate type system). We also assume that we produce the IR from SCL as described in Section 3, with the further step of normalizing the terms into a *direct-style* representation [FSDF93, Tar96, OT98] (a *continuation-passing style* representation [App92] is also

possible). In this IR, all intermediate results are bound to variables, and the right-hand side of all bindings involve a single function application or primitive operation applied to *atomic* arguments (*i.e.*, either variables or constants).

Note that dictionary application and method-suite indexing are *pure* operations; *i.e.*, they are guaranteed to terminate and have no side effects. Consequently, the compiler is free to move these operations, subject only to the constraints of their data dependencies.

5.1 Applying CSE and hoisting

Common subexpression elimination (CSE) is a standard optimization whereby two identical pure expressions are replaced by a single expression. In λink_{ζ} , opportunities for this optimization arise when methods are invoked on the same object. Consider the following code fragment, for example:

```
stdOut ← print "hello";
stdOut ← print " world";
stdOut ← newline ();
```

Figure 6 gives the λink_{ζ} representation of this code and the code that results from applying the CSE optimization. Because the first two method statements invoke the same method (`print`) on the same object (`stdOut`), CSE eliminates the redundant occurrences of the operations to extract the dictionary and method suite from `stdOut`, look up `print`'s slot, and select the `print` method body. Similarly, because the third statement also invokes a method on `stdOut`, we are able to eliminate the dictionary and method-suite extraction operations, although in this case we must still lookup the slot for `newline` and select its method.

<pre>let $\phi_1 = \pi_2(\text{stdOut})$ let $\sigma_1 = \phi_1!\text{print}$ let $\mu_1 = \pi_1(\text{stdOut})$ let $m_1 = \mu_1@\sigma_1$ let $x_1 = m_1(\text{stdOut}, \text{"hello"})$ let $\phi_2 = \pi_2(\text{stdOut})$ let $\sigma_2 = \phi_2!\text{print}$ let $\mu_2 = \pi_2(\text{stdOut})$ let $m_2 = \mu_2@\sigma_2$ let $x_2 = m_2(\text{stdOut}, \text{" world"})$ let $\phi_3 = \pi_3(\text{stdOut})$ let $\sigma_3 = \phi_3!\text{newline}$ let $\mu_3 = \pi_2(\text{stdOut})$ let $m_3 = \mu_3@\sigma_3$ let $x_3 = m_3(\text{stdOut})$ in ...</pre>	<pre>let $\phi_1 = \pi_2(\text{stdOut})$ let $\sigma_1 = \phi_1!\text{print}$ let $\mu_1 = \pi_1(\text{stdOut})$ let $m_1 = \mu_1@\sigma_1$ let $x_1 = m_1(\text{stdOut}, \text{"hello"})$ let $x_2 = m_1(\text{stdOut}, \text{" world"})$ let $\sigma_3 = \phi_1!\text{newline}$ let $m_3 = \mu_1@\sigma_3$ let $x_3 = m_3(\text{stdOut})$ in ...</pre>
(a) unoptimized	(b) optimized

Fig. 6. Using CSE to optimize method dispatch

Another standard transformation is to hoist invariant expressions out of functions. When applied to method dispatch, this transformation can amortize the cost of a dictionary application over multiple function applications or loop iterations.

5.2 Self-method dispatch

While CSE and hoisting apply to any method dispatch, we can do significantly better when we have a message sent to **self**. Recall that the translation of the self-method dispatch **self** $\Leftarrow m(exp)$ into *link* ζ is

```

let obj = ( $\pi_1$ (self),  $\phi_{self}$ )
let meth =  $\pi_1$ (obj) @ ( $\pi_2$ (obj)!m)
in meth(obj, exp)

```

Normalizing to our IR and applying the standard *contraction* phase [App92,AJ97] gives the following:

```

let  $\mu$  =  $\pi_1$ (self)
let obj = ( $\mu$ ,  $\phi_{self}$ )
let  $\sigma$  =  $\phi_{self}$ !m
let meth =  $\mu$ @ $\sigma$ 
in meth(obj, a)

```

where a is the atom resulting from normalizing the argument expression. The expression ϕ_{self} !m is invariant in its containing premethod, and thus the binding of σ can be lifted out of the premethod. This transformation has the effect of moving the dictionary application from run-time to link-time and leaves the following residual:

```

let  $\mu$  =  $\pi_1$ (self)
let obj = ( $\mu$ ,  $\phi_{self}$ )
let meth =  $\mu$ @ $\sigma$ 
in meth(obj, a)

```

While it is likely that a compiler will generate this reduced form directly from a source-program self-method dispatch, this optimization is useful in the case where other optimizations (e.g., inlining) expose self-method dispatches that are not present in the source.

5.3 Super-method dispatch

Calls to superclass methods can be resolved statically, so there should be no run-time penalty for superclass method dispatch. While it is possible to “special-case” such method calls in a compiler, we can get the same effect by code hoisting. Recall that the translation of the super-method dispatch **super** $\Leftarrow m(exp)$ into *link* ζ is

```

( $\mu_{super}$  @ ( $\phi_{super}$ !m)) (self, exp)

```

As before, we normalize to our IR and contract, which produces the following:

```

let  $\sigma$  =  $\phi_{super}$ !m
let meth =  $\mu_{super}$ @ $\sigma$ 
in meth(self, a)

```

where a is the atom resulting from normalizing the argument expression. In this case, we can hoist both the dictionary application and the method-suite indexing out of the containing method, which leaves the term “meth(self, a).” Thus, by using standard λ -calculus transformations, we can resolve super-method dispatch statically. Furthermore, if the superclass’s method suite is known at compile time, then the standard optimization

of reducing a selection from a known record can be applied to turn the call into a direct function call. This reduction has the further effect of enabling the call to be inlined.

5.4 Using static analysis

The optimizations that we have described so far require only trivial analysis. More sophisticated analyses can yield better optimizations [DGC95]. For example, *receiver-class prediction* [GDGC95] may permit us to eliminate some dictionary applications in method dispatches (as we do already for self-method dispatch). There may also be source-language information, such as **final** annotations, that can help the optimizer.

5.5 Final code generation

We intentionally left the implementation of dictionaries abstract in $\lambda\text{ink}\zeta$ so that the optimization techniques described above can be used independently of their concrete representation. Depending on the properties of the source language, dictionaries might be tables [Rém92,DH95], a graph structure [CC98], or a simple list of method names. We might also use caching techniques to improve dispatch performance when there is locality [DS84]. We might also maintain information in the compiler as to the origin of the dictionary and use multiple representations, each tailored to a particular dictionary origin. For example, a JAVA compiler can distinguish between dictionaries that correspond to classes and dictionaries that correspond to interfaces. In the former case, the dictionary is known at class-load time and dictionary applications can be resolved when the class is loaded and linked. For interfaces, however, a dictionary might be implemented as an indirection table [LST99].

6 Discussion and related work

We have presented $\lambda\text{ink}\zeta$, a low-level calculus for representing class-based object-oriented languages. The calculus is intended to fill two roles in a compiler: first, to serve as the object-oriented fragment of the compiler's intermediate representation and second, to allow the compiler to specify class linking. We have illustrated $\lambda\text{ink}\zeta$'s utility for these purposes by describing a translation from a simple class language SCL to $\lambda\text{ink}\zeta$ and by describing how standard λ -calculus optimizations can be used to improve the efficiency of method dispatch. Furthermore $\lambda\text{ink}\zeta$'s flexibility in supporting a range of class mechanisms, from the concrete representations of C++ to the dynamic classes of LOOM, supports our belief that $\lambda\text{ink}\zeta$ provides good abstractions for compiling classes. Evidence that $\lambda\text{ink}\zeta$ is suitable for expressing linking includes the fact that we can define different linking policies for SCL and that the linking redices in the resulting $\lambda\text{ink}\zeta$ programs are strongly normalizing.

There is other published research on IRs for compiling class-based languages. The Vortex project at the University of Washington, for instance, supports a number of class-based languages using a common optimizing back-end [DDG⁺95]. The Vortex IR has fairly high-level operations to support classes: class construction and method dispatch are both monolithic primitives. $\lambda\text{ink}\zeta$, on the other hand, breaks these operations into smaller primitives. By working at a finer level of granularity, $\lambda\text{ink}\zeta$ is able to support a wider range of class mechanisms in a single framework (*e.g.*, Vortex cannot support the dynamic classes found

in LOOM). Another approach pursued by researchers is to encode object-oriented features in typed λ -calculi. While such an approach can support any reasonable surface language design, its effectiveness as an implementation technique depends on the character of the encoding. For example, League, *et. al.*, have recently proposed a translation of a JAVA subset into the FLINT intermediate representation extended with row polymorphism [LST99]. Although they do not have an implementation yet, their encoding seems efficient, but it is heavily dependent on the semantics of JAVA. For example, their translation relies on knowing the exact set of interfaces that a class implements. The encoding approach has also been recently tried by Vanderwaart for LOOM [Van99]. In this case, because of the richness of LOOM's feature set, the encoding results in an inefficient implementation of operations like method dispatch. We believe that a compiler based on $\lambda ink\zeta$ can do at least as well for JAVA as the encoding approach, while doing much better for languages like MOBY and LOOM that do not have efficient encodings in the λ -calculus.

There are other formal linking frameworks [Car97,Ram96,GM99,AZ99,DEW99]. Of particular relevance here are uses of β -reduction to implement linking of modules, as we do for the linking of classes. From the very beginning, the Standard ML of New Jersey compiler has used the λ -calculus to express module linking [AM87]. More recently, Flatt and Felleisen describe a calculus for separate compilation that maps *units* to functions over their free variables. As we do with SCL, they are able to model different linking policies in their framework.

Currently, we are implementing a compiler for the MOBY programming language. We intend to use $\lambda ink\zeta$ as the object fragment of our optimizer's IR. We also plan to explore the use of $\lambda ink\zeta$ to support dynamic class loading and mobile code, and to develop a typed IR based on $\lambda ink\zeta$.

References

- [AC96] Abadi, M. and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, NY, 1996.
- [AG98] Arnold, K. and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 2nd edition, 1998.
- [AJ97] Appel, A. W. and T. Jim. Shrinking lambda expressions in linear time. *JFP*, 7(5), September 1997, pp. 515–540.
- [AM87] Appel, A. W. and D. B. MacQueen. A Standard ML compiler. In *FPCA'87*, vol. 274 of *LNCS*, New York, NY, September 1987. Springer-Verlag, pp. 301–324.
- [App92] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.
- [AZ99] Ancona, D. and E. Zucca. A primitive calculus for module systems. In *PPDP'99*, *LNCS*. Springer-Verlag, September 1999, pp. 62–79.
- [Bar84] Barendregt, H. P. *The Lambda Calculus*, vol. 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.
- [BFP97] Bruce, K. B., A. Fiech, and L. Petersen. Subtyping is not a good “match” for object-oriented languages. In *ECOOP'97*, vol. 1241 of *LNCS*, New York, NY, 1997. Springer-Verlag, pp. 104–127.
- [Car97] Cardelli, L. Program fragments, linking, and modularization. In *POPL'97*, January 1997, pp. 266–277.
- [CC98] Chambers, C. and W. Chen. Efficient predicate dispatching. *Technical report*, Department of Computer Science, University of Washington, 1998.

- [DDG⁺95] Dean, J., G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *OOPSLA'96*, October 1995, pp. 83–100.
- [DEW99] Drossopoulou, S., S. Eisenbach, and D. Wragg. A fragment calculus — towards a model of separate compilation, linking and binary compatibility. In *LICS-14*, June 1999, pp. 147–156.
- [DGC95] Dean, J., D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95*, August 1995.
- [DH95] Driesen, K. and U. Hölzle. Minimizing row displacement dispatch tables. In *OOPSLA'95*, October 1995, pp. 141–155.
- [DS84] Deutsch, L. P. and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *POPL'84*, January 1984, pp. 297–302.
- [FF86] Felleisen, M. and D. P. Friedman. Control operators, the SECD-machine, and the λ -calculus. In M. Wirsing (ed.), *Formal Description of Programming Concepts – III*, pp. 193–219. North-Holland, New York, N.Y., 1986.
- [FKF98] Flatt, M., S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL'98*, January 1998, pp. 171–183.
- [FR99a] Fisher, K. and J. Reppy. The design of a class mechanism for Moby. In *PLDI'99*, May 1999, pp. 37–49.
- [FR99b] Fisher, K. and J. Reppy. Foundations for MOBY classes. *Technical Memorandum*, Bell Labs, Lucent Technologies, Murray Hill, NJ, February 1999.
- [FSDF93] Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI'93*, June 1993, pp. 237–247.
- [GDGC95] Grove, D., J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *OOPSLA'95*, October 1995, pp. 108–123.
- [GLT89] Girard, J.-Y., Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, Cambridge, England, 1989.
- [GM99] Glew, N. and G. Morrisett. Type-safe linking and modular assembly language. In *POPL'99*, January 1999, pp. 250–261.
- [Ler98] Leroy, X. *The Objective Caml System (release 2.00)*, August 1998. Available from <http://pauillac.inria.fr/caml>.
- [LST99] League, C., Z. Shao, and V. Trifonov. Representing Java classes in a typed intermediate language. In *ICFP'99*, September 1999, pp. 183–196.
- [OT98] Oliva, D. P. and A. P. Tolmach. From ML to Ada: Strongly-typed language interoperability via source translation. *JFP*, **8**(4), July 1998, pp. 367–412.
- [Ram96] Ramsey, N. Relocating machine instructions by currying. In *PLDI'96*, May 1996, pp. 226–236.
- [Rémy92] Rémy, D. Efficient representation of extensible records. In *ML'92 Workshop*, San Francisco, USA, June 1992. pp. 12–16.
- [RS98] Riecke, J. G. and C. Stone. Privacy via subsumption. In *FOOL5*, January 1998. A longer version will appear in TAPOS.
- [RV98] Rémy, D. and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *TAPOS*, **4**, 1998, pp. 27–50.
- [Str97] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 3rd edition, 1997.
- [Tar96] Tarditi, D. *Design and implementation of code optimizations for a type-directed compiler for Standard ML*. Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1996. Available as Technical Report CMU-CS-97-108.
- [Van99] Vanderwaart, J. C. Typed intermediate representations for compiling object-oriented languages, May 1999. Williams College Senior Honors Thesis.