A Calculus for Compiling and Linking Classes

Kathleen Fisher[1], John Reppy[2], and Jon G. Riecke[2]

[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}@research.bell-labs.com

Abstract. We describe $\lambda ink\varsigma$ (pronounced "links"), a low-level calculus designed to serve as the basis for an intermediate representation in compilers for class-based object-oriented languages. The primitives in $\lambda ink\varsigma$ can express a wide range of class-based object-oriented language features, including various forms of inheritance, method override, and method dispatch. In particular, $\lambda ink\varsigma$ can model the object-oriented features of MOBY, OCAML, and LOOM, where subclasses may be derived from unknown base classes. $\lambda ink\varsigma$ can also serve as the intermediate representation for more conventional class mechanisms, such as JAVA's. In this paper, we formally describe $\lambda ink\varsigma$, give examples of its use, and discuss how standard compiler transformations can be used to optimize programs in the $\lambda ink\varsigma$ representation.

1 Introduction

Class-based object-oriented languages provide mechanisms for factoring code into 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, compilers for such languages need an intermediate representation (IR) that allows them to represent code fragments (*e.g.*, the code for each class) 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 trickier (Section 2 illustrates the complications).

We are interested in a compiler IR that can handle inheritance from non-manifest base classes. In addition, the IR should satisfy a number of other important criteria. The IR should be expressive enough to support a wide range of statically typed surface languages from JAVA to LOOM. The IR should be reasonably close to the machine and should be able to express efficient object representations (*e.g.*, shared method suites) and both static and dynamic method dispatch. The IR should enable optimizations based on simple and standard transformations. Lastly, the IR should be amenable to formal reasoning about compiler transformations and class linking.

This paper presents λ*ink*ς, which is an extension of the untyped λ-calculus that meets these design goals. λ*ink*ς extends the λ-calculus with *method suites,* which are ordered collections of methods; *slots,* which index into method suites; and *dictionaries,* which map method labels to slots. In λ*ink*ς, 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. λ*ink*ς can support true private names and avoid the fragile base class problem by dynamically changing the dictionary associated with an object when the object's view of itself changes [RS98,FR99b]. Separating dynamic dispatch into two pieces also enables more compiler optimizations. In this paper, we treat λ*ink*ς as a compiler IR, although the reader should think of it as more of a framework or basis for a compiler's IR.

By design, λ*ink*ς satisfies our goals. Because of the abstractions in λ*ink*ς, it can express a wide range of surface class mechanisms, from the static classes found in JAVA through the dynamic inheritance of LOOM (Section 5). By making λ*ink*ς untyped, we avoid limiting the applicability of λ*ink*ς to languages with incompatible type systems. The operations in the calculus allow compilers to leverage static information to optimize message dispatch. For example, the type system in C++ guarantees the slot at which each method may be found at run-time. In λ*ink*ς, we may use this information to evaluate the dictionary lookup operation associated with message dispatch at compile time — providing the expected efficiency for message dispatch to C++ programmers. Because λ*ink*ς is based on the λ-calculus, familiar λ-calculus optimizations apply immediately to λ*ink*ς (Section 6), and these optimizations yield standard object-oriented optimizations when applied to λ*ink*ς programs. Consequently, ad-hoc optimizations for the object-oriented pieces of a compiler based on λ*ink*ς are not necessary. Because λ*ink*ς is a formal language, it is amenable to formal reasoning. For example, one can show that λ*ink*ς is confluent and that the reductions tagged as linking redexes are strongly normalizing (Section 4).

In the next section, we discuss the challenges involved in implementing inheritance from an unknown base-class. In Section 3, we present the syntax, operational semantics, and rewrite systems of λ*ink*ς. To keep the discussion focused, we restrict the technical presentation to a version of λ*ink*ς with methods, but no fields (instance variables). The techniques used to handle methods apply directly to fields (see Section 5.1). Section 4 defines a simple class language SCL and shows how it can be translated to λ*ink*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 5, we sketch how λ*ink*ς can serve as an IR for MOBY, LOOM, a mixin extension for SCL, and C++. Section 6 further demonstrates the utility of the rewriting system for λ*ink*ς by showing how method dispatch can be optimized in the calculus. We conclude with a discussion of related and future work.

2 Inheritance from Unknown Classes

One of our principal design goals is to support inheritance from unknown base classes. Figure 1 shows where difficulties can arise when compiling languages with such a feature. 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

```
signature PT {
  class Point : {
    public meth getX : Unit -> Int
    public meth getY : Unit -> Int
    public meth move : (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

does not know in what order these methods appear in the internal representation of the `Point` class, nor what other private methods and fields the `Point` class might have. As an example of inheritance from such a 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 and allowing their names to be reused for other, independent methods in `ColorPt` and its descendants. 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 its code must have access to methods that might not be directly available in its eventual subclasses.

3 λ*inkς*

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

3.1 Syntax

The syntax of λ*inkς* 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 $\langle e_1, \ldots, e_n \rangle$ constructs a method suite from the expressions $e_1, \ldots, e_n$, where each e_i is assigned slot i. The expression $e@e'$ extracts the value stored in the slot denoted by

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

Fig. 2. The syntax of $\lambda\mathit{ink}\varsigma$

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

Fig. 3. Reduction rules for $\lambda\mathit{ink}\varsigma$

e' from the method suite denoted by e. The method suite extension $e||e'$ concatenates the suites e and 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 by a slot expression, which is either an integer i or the addition of two slot expressions. 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, since it maps m to two different values. To simplify notation, we use the following shorthands:

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

3.2 Operational Semantics

We specify the operational semantics of $\lambda\mathit{ink}\varsigma$ 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 λ*ink*ς 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:

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

The second grammar describes the set of evaluation contexts.

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

The primitive reduction rules for λ*ink*ς 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.

3.3 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 λ*ink*ς 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 $\rightarrow$.

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 it 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.

4 A Simple Class Language

To give evidence of the expressivity of λ*ink*ς, we now give a translation of a simple class-based language into λ*ink*ς. Simpler translations may be possible, but the translation here illustrates some techniques that are useful for more complex languages.

The source language is called SCL for “simple class language.” The syntax of SCL appears in Figure 4. A program consists of a sequence of one or more class declarations followed by an expression; class declarations may only use those declarations that appear before and may not be recursive. There are two forms of class declaration. The first is a *base-class declaration*, which defines a class as a collection of methods. The

$$\begin{array}{rcll}
\textit{prog} & ::= & \textit{dcl prog} & \text{Programs} \\
 & \mid & \textit{exp} & \\
\textit{dcl} & ::= & \texttt{class}\, C\, \texttt{\{}\ \textit{meths}\ \texttt{\}} & \text{Class declarations} \\
 & \mid & \texttt{class}\, C\, \texttt{\{ inherit}\, C' : \texttt{\{}\ m^*\ \texttt{\}}\ \textit{meths}\ \texttt{\}} & \\
\textit{meths} & ::= & \epsilon & \\
 & \mid & \textit{meth meths} & \\
\textit{meth} & ::= & m(x)\textit{exp} & \text{Methods} \\
\textit{exp} & ::= & x & \text{Expressions} \\
 & \mid & \texttt{self} & \\
 & \mid & \textit{exp} \Leftarrow m(\textit{exp}) & \\
 & \mid & \texttt{super} \Leftarrow m(\textit{exp}) & \\
 & \mid & \texttt{new}\, C &
\end{array}$$

Fig. 4. The syntax for SCL

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^* `}`. Other method names can be called only by superclass, not subclass, methods. This operation—in essence, a restriction operation—resembles MOBY's support for private members [FR99b,FR99a] and subsumes mechanisms found in JAVA and other languages.

At the expression level, SCL contains only those features relevant to linking. Methods take exactly one argument and have expressions for bodies; expressions include `self`, method dispatch, super-method dispatch, and object creation. A more complete language would include other expression forms, *e.g.*, integers, booleans, and conditionals.

The translation from SCL into λ*ink*ς fixes representations for 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:

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

These functions take a *class environment* Γ as a parameter, which maps the name of a class to its λ*ink*ς representation. A class environment is tuple of fully-linking classes. The symbol $\Gamma(C)$ denotes the position in the tuple associated with class C, and $\Gamma \pm \{C \mapsto e\}$ denotes the tuple with e bound to class C.

The translation of methods and expressions require more parameters than the translation of programs and classes. 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 λink_ς 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:

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

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. 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 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*.

$$
\begin{array}{l}
\mathcal{C}[\![\texttt{class}\, C\, \texttt{\{ inherit}\, B : \texttt{\{}\ m^*\ \texttt{\}}\ \mathit{meths}\ \texttt{\}}]\!]_\Gamma = \\
\quad \lambda(\sigma_B, \phi_B, \mu_B). \\
\qquad\qquad \texttt{let}^L\ \sigma_{n_1} = \sigma_B +^L 1 \ldots\ \texttt{let}^L \sigma_{n_k} = \sigma_B +^L k \\
\qquad\qquad \texttt{let}^L\ \sigma_{\mathit{ov}_1} = \phi_B !^L \mathit{ov}_1\ \ \ldots\ \texttt{let}^L\ \sigma_{\mathit{ov}_j} = \phi_B !^L \mathit{ov}_j \\
\qquad\qquad \texttt{let}^L\ \sigma_{\mathit{inh}_1} = \phi_B !^L \mathit{inh}_1\ \ \ldots\ \texttt{let}^L\ \sigma_{\mathit{inh}_i} = \phi_B !^L \mathit{inh}_i \\
\qquad\qquad \texttt{let}^L\ \sigma_C = \sigma_B +^L k \\
\qquad\qquad \texttt{let}^L\ \phi_C = \{\, n_1 \mapsto \sigma_{n_1}, \ldots, n_k \mapsto \sigma_{n_k}, \\
\qquad\qquad\qquad\qquad \mathit{ov}_1 \mapsto \sigma_{\mathit{ov}_1}, \ldots, \mathit{ov}_j \mapsto \sigma_{\mathit{ov}_j}, \\
\qquad\qquad\qquad\qquad \mathit{inh}_1 \mapsto \sigma_{\mathit{inh}_1}, \ldots, \mathit{inh}_i \mapsto \sigma_{\mathit{inh}_i} \} \\
\qquad\qquad \texttt{let}^L\ \mu_0 = \mu_B \\
\qquad\qquad \texttt{let}^L\ \mu_1 = \mu_0 @^L \sigma_{\mathit{ov}_1} \leftarrow \mathcal{M}[\![\mathit{meth}_{\mathit{ov}_1}]\!]_{\mu_B, \phi_B, \phi_C, \Gamma} \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad \texttt{let}^L\ \mu_j = \mu_{j-1} @^L \sigma_{\mathit{ov}_j} \leftarrow \mathcal{M}[\![\mathit{meth}_{\mathit{ov}_j}]\!]_{\mu_B, \phi_B, \phi_C, \Gamma} \\
\qquad\qquad \texttt{let}^L\ \mu_C = \mu_j \mid\mid^L \langle \mathcal{M}[\![\mathit{meth}_{n_1}]\!]_{\mu_B, \phi_B, \phi_C, \Gamma}, \ldots, \mathcal{M}[\![\mathit{meth}_{n_k}]\!]_{\mu_B, \phi_B, \phi_C, \Gamma} \rangle \\
\qquad\qquad \texttt{in}\ (\sigma_C,\ \phi_C,\ \mu_C)
\end{array}
$$

where

$$
\begin{array}{l}
\textsc{NewNames} = \{n_1, \ldots, n_k\} = \text{Names}(\mathit{meths}) \setminus \{\ m^*\ \} \\
\textsc{OvNames} = \{\mathit{ov}_1, \ldots, \mathit{ov}_j\} = \{\ m^*\ \} \cap \text{Names}(\mathit{meths}) \\
\textsc{InhNames} = \{\mathit{inh}_1, \ldots, \mathit{inh}_i\} = \{\ m^*\ \} \setminus \textsc{OvNames} \\
\{\mathit{meth}_{n_1}, \ldots, \mathit{meth}_{n_k}\} = \{m(x)\mathit{exp} \mid m(x)\mathit{exp} \in \mathit{meths} \text{ and } m \in \textsc{NewNames}\} \\
\{\mathit{meth}_{\mathit{ov}_1}, \ldots, \mathit{meth}_{\mathit{ov}_j}\} = \{m(x)\mathit{exp} \mid m(x)\mathit{exp} \in \mathit{meths} \text{ and } m \in \textsc{OvNames}\}
\end{array}
$$

Fig. 5. Translating SCL classes to λ*ink*ς

For base-class declarations, the translation is similar, except that there are no inherited or overridden methods. Furthermore, we use a special class $(0, \{\,\}, \langle\,\rangle)$ for the base-class argument. We omit the details for space reasons. Finally, we translate programs as follows:

$$
\begin{array}{rl}
\mathcal{P}[\![\mathit{dcl\ prog}]\!]_\Gamma = \mathcal{P}[\![\mathit{prog}]\!]_{\Gamma'} & \text{where } \Gamma' = \Gamma \pm \{C \mapsto \mathcal{C}[\![\mathit{dcl}]\!]_\Gamma(\Gamma(B))\} \\
\mathcal{P}[\![\mathit{exp}]\!]_\Gamma = \mathcal{E}[\![\mathit{exp}]\!]_{\langle\,\rangle, \{\,\}, \{\,\}, \Gamma} &
\end{array}
$$

The B stands for the base class in the definition of *dcl*.

The language SCL enjoys the property that for a *well-ordered program* — one in which all classes have been defined, and every class is defined before it is used — all linking operations labeled L can be eliminated statically. More formally,

Theorem 2 *If prog is a well-ordered program and* $\mathcal{P}[\![\mathit{prog}]\!]_\Gamma = e$*, then there is a term* e' *such that* $e \to^* e'$ *and* e' *contains no linking operations labeled* L.

This theorem can probably be proven using a size argument, but we use a strong-normalization approach instead. The proof of strong normalization is a bit subtle because expressions in λ*ink*ς can loop. We use a simple type system to show that a *fragment* of λ*ink*ς is strongly normalizing. The proof of strong normalization relies upon Tait's method [GLT89]. One may show that the translation of a well-ordered program is well-typed in the system, and hence all linking reductions can be done statically. We omit the proof for space reasons.

5 Other Examples

We now sketch how $\lambda ink\varsigma$ can be used to compile class mechanisms found in various programming languages.

5.1 Moby Classes

We originally designed $\lambda ink\varsigma$ to support Moby's class mechanism in a compiler that we are writing. Section 4'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 2). 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 are fields (*a.k.a.* instance variables), which require a richer version of $\lambda ink\varsigma$. 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 are 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.

5.2 OCaml Classes

Like Moby, OCaml is a language with both parameterized modules and classes [Ler98]. For the most part, translating OCaml classes to $\lambda ink\varsigma$ is similar to translating Moby classes. The one difference is that OCaml supports a simple form of *multiple inheritance*, whereas Moby only has single inheritance. A class in OCaml can inherit from several base classes, but there is no sharing between base classes — the methods of the base classes are just concatenated. The one subtlety that we must address is that when compiling a class definition, we cannot assume that access to its methods will be zero-based in its subclasses. To solve this problem, we λ-abstract over the initial slot index. Otherwise, translating OCaml classes to $\lambda ink\varsigma$ is essentially the same as for Moby classes.[1]

5.3 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 4, 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.

[1] To the best of our knowledge, the implementation techniques used for classes in the OCaml system have not been formalized or described in print, so we are not able to compare approaches.

5.4 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 $\lambda\mathit{ink}_\varsigma$ 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 $\Leftarrow$ print(self $\Leftarrow$ show()) }
}
```

This mixin adds a `print` method to any class `C` that has a `show` method already. The translation of this mixin to $\lambda\mathit{ink}_\varsigma$ is similar to that of subclasses given in Section 4:

```
$\lambda$($\sigma_C$,$\phi_C$,$\mu_C$,mkDict).
  let $\sigma_{print}$ = $\sigma_C$+1
  let $\phi_{Print}$ = mkDict($\phi_C$, $\sigma_{print}$)
  let pre_print = $\lambda$(self).
      let print = ($\pi_1$ stdOut)@(($\pi_2$ stdOut)!print)
      let show = ($\pi_1$ self)@($\phi_{Print}$!show)
      in print(stdOut, show(self))
  let $\mu_{Print}$ = $\mu_C$ $||$ $\langle$pre_print$\rangle$
  in ($\sigma_{print}$, $\phi_{Print}$, $\mu_{Print}$)
```

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 $\lambda\mathit{ink}_\varsigma$. 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.

5.5 C^{++} and Java Classes

For a language with a manifest class hierarchy, such as C^{++} or Java, the language's static type system provides substantial information about the representation of dictionaries and method suites. 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. We should note that we do not know how to handle C^{++}'s form of multiple inheritance in $\lambda\mathit{ink}_\varsigma$ because of the object layout issues related to sharing of virtual base classes [Str94].

6 Optimization

Many compilers for higher-order languages use some form of λ-calculus as their intermediate representation (IR). 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 $\lambda\mathit{ink}_\varsigma$. Because $\lambda\mathit{ink}_\varsigma$ allows reuse of standard optimizations, the optimizer is simpler and more likely to be correct. 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 and method-suite operations, which are *pure*. Consequently, the compiler is free to move these operations, subject only to the constraints of their data dependencies.

To make the discussion concrete, we consider the λ*ink*$_\varsigma$ representation of SCL programs and their optimization. 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 transformations to reduce or eliminate this cost. 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 4, 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).

6.1 Applying CSE and Hoisting

Common subexpression elimination (CSE) is a standard optimization whereby two identical pure expressions are replaced by a single expression. When method invocations are expanded into the λ*ink*$_\varsigma$ representation, there are many opportunities for CSE optimizations. For example, if there are two method invocations to the same object, fetching its dictionary will be a common subexpression. If the method calls are to the same method, then the dictionary application and method suite indexing operations will be common subexpressions.

Another standard transformation is to hoist invariant expressions out of functions. When applied to method dispatch, this transformation amortizes the cost of a dictionary application over multiple function applications or loop iterations.[2]

6.2 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 λ*ink*$_\varsigma$ 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] gives the following:

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

[2] Note that loops are represented as tail-recursive functions in this style of IR.

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 μ = π₁(self)
let obj = (μ, φ_self)
let meth = μ@σ
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.

6.3 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 λ*ink*ς is

```
(μ_super @ (φ_super!m)) (self, exp)
```

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

```
let σ = φ_super!m
let meth = μ_super@σ
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.

6.4 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 type information, such as `final` annotations, that can enable optimizations, such as static method resolution.

6.5 Final Code Generation

We intentionally left the implementation of dictionaries abstract in λ*ink*ς 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].

7 Related Work

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. λink_ς, on the other hand, breaks these operations into smaller primitives. By working at a finer level of granularity, λink_ς 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 λink_ς 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.

In other related work, Bono, *et. al.* have designed a class calculus, based on the λ-calculus, for evaluating single and mixin inheritance [BPS99]. The focus of their work differs from ours, in that their calculus describes the core functionality of a particular surface language, whereas we provide the basic building blocks with which to implement a myriad of surface designs. Essentially, their language could be implemented in λink_ς; the translation from their calculus to λink_ς would capture the implementation information encoded in their operational semantics.

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 [FF98].

8 Conclusions

We have presented λ*inkς*, a low-level calculus for representing class-based object-oriented languages. λ*inkς* satisfies the goals we set in designing an IR. In particular, it provides support for inheritance from non-manifest base classes, such as occurs in MOBY, OCAML, and LOOM. It is amenable to formal reasoning, such as in the proof of termination of linking in Section 4. As illustrated in Section 5, λ*inkς* is expressive enough to support a wide-range of surface languages, from the concrete representations of JAVA to the dynamic classes of LOOM. Finally, simple λ-calculus optimizations, such as common subexpression elimination and hoisting, yield standard object-oriented optimizations, such as method caching, when applied to λ*inkς* terms.

We are currently implementing a compiler for MOBY that uses λ*inkς* as the basis of the object fragment of its IR. One refinement that we use in our implementation is to syntactically distinguish between the link-time and run-time forms of λ*inkς*. In the future, we plan to explore the use of λ*inkς* to support dynamic class loading and mobile code, and to develop a typed IR based on λ*inkς*.

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.

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.

BPS99. Bono, V., A. Patel, and V. Shmatikov. A core calculus of classes and mixins. In *ECOOP'99*, vol. 1628 of *LNCS*, New York, NY, June 1999. Springer-Verlag.

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.

FF98. Flatt, M. and M. Felleisen. Units: Cool modules for HOT languages. In *PLDI'98*, June 1998, pp. 236–248.

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.

Rém92. 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 *Information and Computation*.

RV98. Rémy, D. and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *TAPOS*, **4**, 1998, pp. 27–50.

Str94. Stroustrup, B. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA, 1994.

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.