Formalization of Generics for the .NET Common Language Runtime

Dachuan Yu *
Yale University
New Haven, CT, U.S.A.
yu@cs.yale.edu

Andrew Kennedy
Microsoft Research
Cambridge, U.K.
akenn@microsoft.com

Don Syme
Microsoft Research
Cambridge, U.K.
dsyme@microsoft.com

Abstract

We present a formalization of the implementation of generics in the .NET Common Language Runtime (CLR), focusing on two novel aspects of the implementation: mixed specialization and sharing, and efficient support for run-time types. Some crucial constructs used in the implementation are dictionaries and run-time type representations. We formalize these aspects type-theoretically in a way that corresponds in spirit to the implementation techniques used in practice. Both the techniques and the formalization also help us understand the range of possible implementation techniques for other languages, e.g., ML, especially when additional source language constructs such as run-time types are supported. A useful by-product of this study is a type system for a subset of the polymorphic IL proposed for the .NET CLR.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features—*Polymorphism, Classes and Objects*; D.3.1 [Programming Languages]: Formal Definitions and Theory

General Terms: Languages

Keywords: Polymorphism, Generics, Run-time Types, CLR, .NET

1 Introduction

Parametric polymorphism, also known as "generics", is an important new feature of Version 2.0 of the C♯ programming language [10, 11] and also the .NET Common Language Runtime (CLR) [18, 6] that underpins C♯ and other languages. In previous work, two of the authors presented informally the design and implementation of generics for C♯ and the .NET CLR [14]. The primary novelty of the design is the integration of parameterized types and polymorphic methods into the type system of the intermediate language (IL) implemented by a virtual machine, or runtime. The implementation techniques described there are novel in many ways: they include "just-in-time" specialization of classes and code, code-sharing between distinct instantiations of generic classes, and the efficient implementation of runtime types using dictionaries of type representations and computations.

There has also been much recent work [1, 20, 22, 21, 3, 4, 27] on adding generics to the Java programming language [8], most of which "compiles away" generics by translation into the JVM [17]. The fact that type loopholes were identified [13] in Generic Java suggests we pay close attention not only to genericity itself as a language feature [12], but also the underlying implementation techniques. In particular, the generics implementation for the CLR is of unique interest because better expressivity is achieved with extended support from the virtual machine, which contrasts with most work on Java genericity whose design space is limited by targeting the JVM.

In this paper we formalize the generics design of the CLR via a type system and operational semantics for a subset of IL with generics, and formalize two novel aspects of the implementation: specialization of generic code up to data representation, and efficient support for run-time types. We do not attempt to formalize dynamic class loading or just-in-time compilation (which is a challenging topic on its own), instead presenting a static compilation scheme, thus demonstrating that some of the techniques employed by the implementation could be utilized by more conventional compilers for polymorphic languages.

As far as possible, the formalization is faithful to the implementation whose prototype was described in [14]; this contrasts with other work on IL (*sans* generics) [29] in which the target of a translation from IL is a foundational calculus based on F_ω. Our target language corresponds (in spirit, at least) to the output of the first stage of JIT-compilation. It does differ, however, in that the translation preserves static types, and at the same time the use of types as runtime entities (*e.g.* in checked casts) is separated off using type-reps à la Crary *et al.* [5]. This lets us prove an erasure theorem for the target ("static types do not affect evaluation") and makes it a good basis for further work on optimization. Indeed it could form the core of a typed intermediate language used by an optimizing JIT compiler for the CLR. We also discuss briefly how an alternative target language that is more foundational in flavour (*e.g.* with structural subtyping and recursive types) would offer both opportunities for expressing interesting optimizations and challenges for

*This work was undertaken during Dachuan Yu's internship at Microsoft Research, Cambridge, UK. Dachuan Yu's research is supported in part by DARPA OASIS grant F30602-99-1-0519 and NSF grant CCR-0208618.

POPL'04, January 14–16, 2004, Venice, Italy.

$$
\begin{array}{rrl}
\textit{(type)} & T,U & ::= X \mid \texttt{int32} \mid \texttt{int64} \mid I \\
\textit{(inst type)} & I & ::= C\texttt{<}\overline{T}\texttt{>} \\
\textit{(class)} & cd & ::= \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : I\ \{\overline{T}\ \overline{f}\texttt{;}\ \overline{md}\} \\
\textit{(method)} & md & ::= \texttt{static}\ T\ m\texttt{<}\overline{X}\texttt{>(}\overline{T}\ \overline{x}\texttt{)}\ \{\ e\} \\
 & & \mid \texttt{virtual}\ T\ m\texttt{<}\overline{X}\texttt{>(}\overline{T}\ \overline{x}\texttt{)}\ \{\ e\} \\
\textit{(mdesc)} & M & ::= T\ I\texttt{::}m\texttt{<}\overline{T}\texttt{>(}\overline{U}\texttt{)} \\
\textit{(exp)} & e & ::= \texttt{ldc.i4}\ i4 \mid \texttt{ldc.i8}\ i8 \mid \texttt{ldarg}\ x \\
 & & \mid \overline{e}\ \texttt{newobj}\ I \mid e\ \texttt{ldfld}\ T\ I\texttt{::}f \\
 & & \mid \overline{e}\ \texttt{call}\ M \mid e\ \overline{e}\ \texttt{callvirt}\ M \\
 & & \mid e\ \texttt{isinst}\ I\ \texttt{or}\ e \\
\textit{(value)} & v,w & ::= i4 \mid i8 \mid I(\overline{f} \mapsto \overline{v}) \\
\textit{(type env)} & E & ::= \overline{X},\overline{x} : \overline{T}
\end{array}
$$

Figure 1. Syntax of BILG.

efficient implementation in a virtual machine.

The paper is structured as follows. In Sections 2 and 3 we present our source and target languages, and in Section 4 we formalize the translation between them. To better demonstrate the key ideas for supporting specialization and run-time types, we make simplifications with regard to the modelling of other features, such as sharing and cycles in dictionaries of types, the possibility of an infinite graph of dictionaries, and support for polymorphic virtual methods. The handling of these features, together with other language constructs including value classes, is discussed in detail in Section 5 along with related work. Section 6 concludes. Interested readers are refered to our companion technical report [28] for a type-erasure semantics of the target language and the soundness proof of the translation.

2 Source Language

Our formalization starts from a purely-functional core of the verifiable IL with generics, which we call Baby IL with Generics (BILG)[1]. Following Baby IL [7], BILG specifies the instruction set as tree-structured applicative expressions. In BILG, we support primitive types of various sizes, parameterized reference types with general instantiations, polymorphic static and virtual methods, and exact run-time types. The support for mutable fields, managed pointers and value types (structs) is largely orthogonal from the translation's point of view, hence it is omitted, although extensions for supporting value types will be discussed later.

The syntax of BILG is shown in Figure 1. The types (T or U) include formal type parameters (X or Y), primitive types, and type instantiations of classes (I). The type arguments of class instantiations are uncurried and denoted using vectors $\overline{T}$.

A class definition (cd) contains the name of the class and its superclass, a sequence of field declarations, and a sequence of method declarations. FJ-like systems [12] usually define for every class a single constructor whose behavior is fixed. In BILG this behavior is built-in, hence the constructor is omitted from the syntax. We take the liberty of using vector notation when there is no confusion. For example, the list of field declarations is abbreviated as $\overline{T}\ \overline{f}$. For simplicity we assume no field hiding, *i.e.,* all fields in a class are distinct from those in its superclass. The top of the class hierarchy

[1] :-)

Field Lookup: $\boxed{\mathit{fields}(I) = \ldots}$

$$\frac{}{\mathit{fields}(\texttt{object}) = \{\}}$$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : I\ \{\overline{U_1}\ \overline{f_1}\texttt{;}\ \overline{md}\} \quad \mathit{fields}([\overline{T}/\overline{X}]I) = \overline{U_2}\ \overline{f_2}}{\mathit{fields}(C\texttt{<}\overline{T}\texttt{>}) = \overline{U_2}\ \overline{f_2}, [\overline{T}/\overline{X}]\overline{U_1}\ \overline{f_1}}$$

Method Lookup: $\boxed{\mathit{mtype}(I.m) = \ldots \quad \mathit{mbody}(I.m\texttt{<}\overline{T}\texttt{>}) = \ldots}$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : I\ \{\overline{U}\ \overline{f}\texttt{;}\ \overline{md}\} \quad m \text{ not defined in } \overline{md}}{\begin{array}{l}\mathit{mtype}(C\texttt{<}\overline{T_1}\texttt{>}.m) = \mathit{mtype}([\overline{T_1}/\overline{X}]I.m) \\ \mathit{mbody}(C\texttt{<}\overline{T_1}\texttt{>}.m\texttt{<}\overline{T_2}\texttt{>}) = \mathit{mbody}([\overline{T_1}/\overline{X}]I.m\texttt{<}\overline{T_2}\texttt{>})\end{array}}$$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : I\ \{\overline{U}\ \overline{f}\texttt{;}\ \overline{md}\} \quad \texttt{static/virtual}\ U_2\ m\texttt{<}\overline{Y}\texttt{>(}\overline{U_1}\ \overline{x}\texttt{)}\ \{\ e\} \in \overline{md}}{\begin{array}{l}\mathit{mtype}(C\texttt{<}\overline{T_1}\texttt{>}.m) = [\overline{T_1}/\overline{X}](\texttt{<}\overline{Y}\texttt{>}\overline{U_1} \to U_2) \\ \mathit{mbody}(C\texttt{<}\overline{T_1}\texttt{>}.m\texttt{<}\overline{T_2}\texttt{>}) = \langle \overline{x}, [\overline{T_1}/\overline{X}, \overline{T_2}/\overline{Y}]e\rangle\end{array}}$$

Figure 2. Macros on fields and methods look-up.

is handled specially and referred to as `object`.

Methods are either static (statically dispatched based on the class instantiation) or virtual (dynamically dispatched based on the object). A method descriptor (M) provides the signature of a method. As is the case of the implementation [14], we require that the type instantiation of the class is specified, and the signature must be exactly that of the definition. This is only to maintain the flavor of the actual IL. For BILG, the argument and return types in the method descriptors are of little importance due to the omission of overloading. Method descriptors are used in method invocation expressions (`call` and `callvirt`).

Field access expressions (`ldfld`) refer to field descriptors which also specify the type instantiations of the classes. Since field descriptors are simpler constructs, we inline them in the corresponding expressions.

The actual IL accesses method arguments by specifying an offset (`ldarg` j). In BILG we use argument names (`ldarg` x) for ease of understanding (this syntax is also supported by the .NET IL assembler).

The run-time type test e `isinst` I `or` e' returns e if e is an instance of I; otherwise it returns e', which is of type I. In contrast, the actual IL returns `null` when the test fails. We use the default branch e' to avoid introducing `null`.

The remaining expressions are standard; they include loading of constants (the `ldc` family) and object creation (`newobj`). The values (v or w) are either integers or objects. Besides the labelled fields, an object value contains also a name of the class instantiation for dynamic dispatch.

A typing environment E has the form $\overline{X},\overline{x} : \overline{T}$ where free type variables in $\overline{T}$ are drawn from $\overline{X}$. Our semantics are in the style of Featherweight GJ [12]. We use the following judgment forms to define the static semantics:

Type Formation: $\boxed{E \vdash T\ ok}$

$$\frac{}{\overline{X}, \overline{x} : \overline{T} \vdash X_i\ ok} \qquad \frac{}{E \vdash \texttt{int32}\ ok} \qquad \frac{}{E \vdash \texttt{int64}\ ok}$$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : I\ \{\ldots\} \qquad E \vdash \overline{T}\ ok}{E \vdash C\texttt{<}\overline{T}\texttt{>}\ ok}$$

Subtyping: $\boxed{E \vdash I <: I'}$

$$\frac{}{E \vdash I <: I} \qquad \frac{E \vdash I_1 <: I_2 \qquad E \vdash I_2 <: I_3}{E \vdash I_1 <: I_3}$$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : I\ \{\ldots\} \qquad E \vdash C\texttt{<}\overline{T}\texttt{>}\ ok}{E \vdash C\texttt{<}\overline{T}\texttt{>} <: [\overline{T}/\overline{X}]I}$$

Typing: $\boxed{E \vdash e : T}$

$$\frac{E \vdash e : I \qquad E \vdash I <: I'}{E \vdash e : I'} \qquad \frac{}{E \vdash \texttt{ldc.i4}\ i4 : \texttt{int32}}$$

$$\frac{}{E \vdash \texttt{ldc.i8}\ i8 : \texttt{int64}} \qquad \frac{}{\overline{X}, \ldots x : T \ldots \vdash \texttt{ldarg}\ x : T}$$

$$\frac{E \vdash I\ ok \qquad \mathit{fields}(I) = \overline{T}\ \overline{f} \qquad E \vdash \overline{e} : \overline{T}}{E \vdash \overline{e}\ \texttt{newobj}\ I : I}$$

$$\frac{E \vdash \overline{e} : [\overline{U}/\overline{X}]\overline{T} \qquad \mathit{mtype}(I.m) = \texttt{<}\overline{X}\texttt{>}\overline{T} \to T'}{E \vdash \overline{e}\ \texttt{call}\ T'\ I\texttt{::}m\texttt{<}\overline{U}\texttt{>}(\overline{T}) : [\overline{U}/\overline{X}]T'}$$

$$\frac{E \vdash e_0 : I \qquad E \vdash \overline{e} : [\overline{U}/\overline{X}]\overline{T} \qquad \mathit{mtype}(I.m) = \texttt{<}\overline{X}\texttt{>}\overline{T} \to T'}{E \vdash e_0\ \overline{e}\ \texttt{callvirt}\ T'\ I\texttt{::}m\texttt{<}\overline{U}\texttt{>}(\overline{T}) : [\overline{U}/\overline{X}]T'}$$

$$\frac{E \vdash e : I \qquad \mathit{fields}(I) = \overline{T}\ \overline{f}}{E \vdash e\ \texttt{ldfld}\ T_i\ I\texttt{::}f_i : T_i} \qquad \frac{E \vdash e : I' \qquad E \vdash e' : I}{E \vdash e\ \texttt{isinst}\ I\ \texttt{or}\ e' : I}$$

Method and Class Typing: $\boxed{\vdash \mathit{md}\ ok\ \mathit{in}\ C\texttt{<}\overline{X}\texttt{>}}$ $\boxed{\vdash \mathit{cd}\ ok}$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : I\ \{\overline{U_1\ f_1}\texttt{;}\ \overline{md}\} \qquad \overline{X}, \overline{Y}, \overline{x} : \overline{T} \vdash e : T}{\vdash \texttt{static}\ T\ m\texttt{<}\overline{Y}\texttt{>}(\overline{T}\ \overline{x})\ \{\ e\}\ ok\ in\ C\texttt{<}\overline{X}\texttt{>}}$$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : I\ \{\overline{U_1\ f_1}\texttt{;}\ \overline{md}\} \qquad \overline{X}, \overline{x} : \overline{T}, \texttt{this} : C\texttt{<}\overline{X}\texttt{>} \vdash e : T}{\vdash \texttt{virtual}\ T\ m\texttt{<}\overline{Y}\texttt{>}(\overline{T}\ \overline{x})\ \{\ e\}\ ok\ in\ C\texttt{<}\overline{X}\texttt{>}}$$

$$\frac{\mathit{fields}(I) = \overline{U}\ \overline{g} \qquad \overline{f}\ \text{and}\ \overline{g}\ \text{disjoint} \qquad \vdash \overline{md}\ ok\ in\ C\texttt{<}\overline{X}\texttt{>}}{\vdash \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : I\ \{\overline{T}\ \overline{f}\texttt{;}\ \overline{md}\}\ ok}$$

Value Typing: $\boxed{\vdash v : T}$

$$\frac{}{\vdash i4 : \texttt{int32}} \qquad \frac{}{\vdash i8 : \texttt{int64}}$$

$$\frac{\mathit{fields}(I) = \overline{T}\ \overline{f} \qquad \vdash \overline{v} : \overline{T}}{\vdash I(\overline{f} \mapsto \overline{v}) : I}$$

Figure 3. Typing rules of BILG.

Evaluation: $\boxed{fr \vdash e \Downarrow v}$

(frame) $fr ::= (\overline{x} = \overline{v})$

$$\frac{}{fr \vdash \texttt{ldc.i4}\ i4 \Downarrow i4} \qquad \frac{}{fr \vdash \texttt{ldc.i8}\ i8 \Downarrow i8}$$

$$\frac{}{(\overline{x} = \overline{v}) \vdash \texttt{ldarg}\ x_i \Downarrow v_i}$$

$$\frac{fr \vdash \overline{e} \Downarrow \overline{v} \qquad \mathit{fields}(I) = \overline{T}\ \overline{f}}{fr \vdash \overline{e}\ \texttt{newobj}\ I \Downarrow I(\overline{f} \mapsto \overline{v})}$$

$$\frac{\mathit{mbody}(I.m\texttt{<}\overline{U}\texttt{>}) = \langle \overline{x}, e' \rangle \qquad fr \vdash \overline{e} \Downarrow \overline{v} \qquad (\overline{x} = \overline{v}) \vdash e' \Downarrow v}{fr \vdash \overline{e}\ \texttt{call}\ T'\ I\texttt{::}m\texttt{<}\overline{U}\texttt{>}(\overline{T}) \Downarrow v}$$

$$\frac{fr \vdash e_0 \Downarrow I'(\overline{f} \mapsto \overline{w}) \qquad \mathit{mbody}(I'.m\texttt{<}\overline{U}\texttt{>}) = \langle \overline{x}, e' \rangle \qquad fr \vdash \overline{e} \Downarrow \overline{v} \qquad (\texttt{this} = I'(\overline{f} \mapsto \overline{w}), \overline{x} = \overline{v}) \vdash e' \Downarrow v}{fr \vdash e_0\ \overline{e}\ \texttt{callvirt}\ T'\ I\texttt{::}m\texttt{<}\overline{U}\texttt{>}(\overline{T}) \Downarrow v}$$

$$\frac{fr \vdash e \Downarrow I'(\overline{f} \mapsto \overline{v})}{fr \vdash e\ \texttt{ldfld}\ T\ I\texttt{::}f_i \Downarrow v_i}$$

$$\frac{fr \vdash e \Downarrow I'(\overline{f} \mapsto \overline{v}) \qquad \{\} \vdash I' <: I}{fr \vdash e\ \texttt{isinst}\ I\ \texttt{or}\ e' \Downarrow I'(\overline{f} \mapsto \overline{v})}$$

$$\frac{fr \vdash e \Downarrow I'(\overline{f} \mapsto \overline{v}) \qquad \{\} \vdash I' \not<: I \qquad fr \vdash e' \Downarrow v'}{fr \vdash e\ \texttt{isinst}\ I\ \texttt{or}\ e' \Downarrow v'}$$

Figure 4. Evaluation rules of BILG.

$E \vdash T\ ok$	(T *is well-formed in context* E)
$E \vdash I <: I'$	(I *is subtype of* I' *in* E)
$E \vdash e : T$	(e *has type* T *in* E)
$\vdash \mathit{md}\ ok\ \mathit{in}\ C\texttt{<}\overline{X}\texttt{>}$	(*md is well-formed in* $C\texttt{<}\overline{X}\texttt{>}$)
$\vdash \mathit{cd}\ ok$	(*cd is well-formed*)
$\vdash v : T$	(v *has type* T)

The typing rules are largely standard and shown in Figure 3. Figure 2 defines some macros to help manipulate fields and methods.

In the interest of modelling the observable behaviours of BILG programs, as opposed to a particular implementation, we use a big-step evaluation semantics as shown in Figure 4. The variable `this` is reserved for self pointers; it corresponds to argument 0 in the actual IL. A single stack frame suffices in defining the evaluation (most interestingly for the cases of method invocations) because of the simplicity of BILG. A generalization using a full-fledged stack of frames appeared in Gordon and Syme's Baby IL [7].

All of the judgment forms and helper definitions assume a class table $\mathcal{D}$. When we wish to be more explicit, we annotate judgments and helpers with $\mathcal{D}$. We say that $\mathcal{D}$ is a valid class table if $\vdash^{\mathcal{D}} \mathit{cd}\ ok$ for each class definition cd in $\mathcal{D}$.

Theorem 1 (BILG evaluation preserves typing) Suppose that $\mathcal{D}$ is a valid class table, $\overline{X}, \overline{x} : \overline{T} \vdash^{\mathcal{D}} e : T$ and $\vdash^{\mathcal{D}} v_i : [\overline{U}/\overline{X}]T_i$ holds for all $v_i \in \overline{v}$ and $T_i \in \overline{T}$. If $(\overline{x} = \overline{v}) \vdash^{\mathcal{D}} e \Downarrow w$ then $\vdash^{\mathcal{D}} w : [\overline{U}/\overline{X}]T$.

Proof sketch. By induction on the structure of the evaluation derivation. □

$$
\begin{array}{rrl}
\textit{(type)} & T,U & ::= X \mid \texttt{int32} \mid \texttt{int64} \mid I \\
\textit{(inst ty)} & I & ::= C\texttt{<}\overline{T}\texttt{>} \\
\textit{(ext ty)} & \tau & ::= T \mid \texttt{Rep(}T\texttt{)} \mid \texttt{Rep(}M\texttt{)} \\
\textit{(cnstrnt)} & s & ::= \texttt{ref} \mid \texttt{i4} \mid \texttt{i8} \\
\textit{(class)} & cd & ::= \texttt{class}\ C\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>}\ \texttt{:}I\,\{\overline{T}\,\overline{f}\,\texttt{;}\,\overline{md}\}\ \texttt{with}\ \overline{\tau} \\
\textit{(meth)} & md & ::= \texttt{static}\ T\ m\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>(}\overline{\tau}\,\overline{x}\texttt{)}\ \{\ e\}\ \texttt{with}\ \overline{\tau} \\
 & & \mid \texttt{virtual}\ T\ m\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>(}\overline{T}\,\overline{x}\texttt{)}\ \{\ e\} \\
\textit{(mdesc)} & M & ::= I\texttt{::}m\texttt{<}\overline{T}\texttt{>} \\
\textit{(desc)} & D & ::= T \mid M \\
\textit{(exp)} & e & ::= i4 \mid i8 \mid x \mid I\texttt{(}e,\overline{e}\texttt{)} \mid \overline{e}\ \texttt{call}\ M \\
 & & \mid e\,\overline{e}\ \texttt{callvirt}\ M \mid e\ \texttt{ldfld}\ I\texttt{::}f \\
 & & \mid e\ \texttt{isinst}_I\ e\ \texttt{or}\ e \mid R_T \mid R_M \mid \texttt{mkrep}_I\overline{e} \\
 & & \mid \texttt{mkrep}_M\texttt{(}\overline{e},\overline{e}\texttt{)} \mid \texttt{objdict}_i e \mid \texttt{mdict}_i e \\
\textit{(value)} & v & ::= i4 \mid i8 \mid I\texttt{(}v,\overline{v}\texttt{)} \mid R_T \mid R_M \\
\textit{(ty env)} & E & ::= \overline{X} \triangleright \overline{s}, \overline{x} : \overline{\tau}
\end{array}
$$

Figure 5. Syntax of BILC.

3 Target Language

What makes the implementation of generic IL unique is that it combines exact run-time types, shared code and code specialization for non-uniform instantiations. To focus on these issues without being distracted by the largely orthogonal handling of object-oriented features [29], we designed our target language, Baby IL with Constraints (BILC), to contain counterparts for all the expressions of BILG. The novelty of BILC lies in its support for *constraints*, term representations of types (or *type-reps* for short), and *dictionaries*.

The syntax of BILC is shown in Figure 5. Its type language is stratified into types (T or U), which consist of all BILG types, and extended types (τ), which support type-reps. Extended types occur explicitly in the syntax of static method definition, because our translation passes a type-rep as an extra argument in static method calls. Type environments map type variables to constraints, and term variables to extended types. The remainder of BILC is best learnt by comparison with BILG, with an emphasis on the new features.

Constraints The implementation handles specialization and sharing based on compatibility of instantiations. Two instantiations are *compatible*, and their code can be shared, if for any parameterized class its compilation at these instantiations gives rise to identical code and other execution structures (*e.g.,* field layout and GC tables), except for the instantiation-specific dictionaries described below. A piece of specialized code is generated for each set of compatible instantiations reached in the program. To prevent specialized code from being instantiated with improper type arguments, constraints on the representation of terms inhabited by types (not to be confused with type-reps) are introduced. For instance, a natural choice is to let all reference types be compatible with each other, because objects of reference types are represented as pointers, so no distinction is made for field layout or code generation; hence a natural constraint `ref` for all (terms of) reference types is useful.

In BILC, type parameters in generic class and method definitions are qualified with constraints (s) to limit the type arguments with which the class or method can be instantiated. Besides the constraint `ref` used for all reference types, there are two "singleton constraints" `i4` and `i8`, satisfied by `int32` and `int64` respectively.

Type-reps To support exact run-time types without using a type-passing interpretation, we represent run-time type information by ordinary terms. These type-reps are analyzed by those expressions that require run-time type information, such as object creation and type test. Value R_T is the type-rep for T, for closed T. For a type $C\texttt{<}\overline{U}\texttt{>}$ where $\overline{U}$ are open, the type-rep can be constructed using expression $\texttt{mkrep}_{C\texttt{<}\overline{U}\texttt{>}}\overline{e}$, where $\overline{e}$ are the type-reps of $\overline{U}$.

We also introduce type-reps for method descriptors (M). They simply provide a means for referring to method descriptors at run-time. Value R_M is the type-rep for closed M, and the type-rep of open method descriptors can be constructed using a `mkrep` expression.

We use the `Rep` type constructor to assign types to type-reps. A type-rep of T or M has the type `Rep(`T`)` or `Rep(`M`)` respectively.

Dictionaries Type-reps are used by operations that require run-time type information. Inside a piece of shared code, the type-reps of open types cannot be determined statically — they can only be constructed after the type arguments are given. To avoid costly repetition of these constructions, the implementation uses dictionaries to store type-reps for open types, which are pre-computed when a generic class or method is instantiated.

Syntactically, a dictionary is simply a vector of values. In BILC, dictionaries for classes and methods are accessed using expressions $\texttt{objdict}_i e$ and $\texttt{mdict}_i e$ respectively. Given e as an object of the class instantiation or a type-rep of the method descriptor, the above two expressions fetch the ith element out of the corresponding dictionary.

The type of the dictionary elements are explicitly specified by the `with` clause in the class and method definitions. The type system allows these definitions to specify arbitrary dictionary types, as long as the dictionary elements are type-rep values. A dictionary map δ (mapping instantiated types and method descriptors to dictionaries) contains all the dictionaries that the program may refer to, and their types match those specified by the corresponding class and method definitions. Virtual methods do not take dictionaries, simply because we do not make use of dictionaries to translate virtual methods.

The remainder of BILC is relatively easy to understand. The types of BILC do not affect evaluation, and run-time type information is acquired from type-reps. An object creation expression $I\texttt{(}e,\overline{e'}\texttt{)}$ creates a new object whose type I has the type-rep e. A type test expression $e\ \texttt{isinst}_T\ e'\ \texttt{or}\ e''$ tests e against the type-rep e' of type T. Types still occur inside this type test and some other expressions but only to enable BILC to be statically typed. For conciseness, we safely omit the argument and return types from method descriptors since we do not support overloading. Field types are also omitted from field access expressions. A new syntactic category of *descriptors* is introduced to cover both types and method descriptors.

As in BILG, the top of the class hierarchy `object` is handled specially. We use R_{obj} as the type-rep of `object`. Aside from the extra handling of constraints, type-reps, and dictionaries, the semantics are still in the style of Featherweight GJ. We use the following judgment forms to define the static semantics.

Field Lookup: $\boxed{\mathit{fields}(I) = \ldots}$

$$\frac{}{\mathit{fields}(\texttt{object}) = \{\}}$$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>} : I\ \{\overline{U}\ \overline{f}\texttt{;}\ \overline{md}\}\ \texttt{with}\ \overline{\tau} \quad \mathit{fields}([\overline{T}/\overline{X}]I) = \overline{U_2}\ \overline{f_2}}{\mathit{fields}(C\texttt{<}\overline{T}\texttt{>}) = \overline{U_2}\ \overline{f_2}, [\overline{T}/\overline{X}]\overline{U_1}\ \overline{f_1}}$$

Method Lookup: $\boxed{\begin{array}{l}\mathit{mtype}(I.m) = \ldots \\ \mathit{mbody}(I.m\texttt{<}\overline{T}\texttt{>}) = \ldots\end{array}}$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>} : I\ \{\overline{U}\ \overline{f}\texttt{;}\ \overline{md}\}\ \texttt{with}\ \overline{\tau} \quad m \text{ not defined in } \overline{md}}{\begin{array}{l}\mathit{mtype}(C\texttt{<}\overline{T_1}\texttt{>}.m) = \mathit{mtype}([\overline{T_1}/\overline{X}]I.m) \\ \mathit{mbody}(C\texttt{<}\overline{T_1}\texttt{>}.m\texttt{<}\overline{T_2}\texttt{>}) = \mathit{mbody}([\overline{T_1}/\overline{X}]I.m\texttt{<}\overline{T_2}\texttt{>})\end{array}}$$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X} \triangleright \overline{s_1}\texttt{>} : I\ \{\overline{U}\ \overline{f}\texttt{;}\ \overline{md}\}\ \texttt{with}\ \overline{\tau'} \quad \texttt{static/virtual}\ U'\ m\texttt{<}\overline{Y} \triangleright \overline{s_2}\texttt{>(}\overline{\tau}\ \overline{x}\texttt{)}\ \{\ e\}\ \texttt{with}\ \overline{\tau''} \in \overline{md}}{\begin{array}{c}\mathit{mtype}(C\texttt{<}\overline{T_1}\texttt{>}.m) = [\overline{T_1}/\overline{X}](\texttt{<}\overline{Y} \triangleright \overline{s_2}\texttt{>}\overline{\tau} \to U') \\ \mathit{mbody}(C\texttt{<}\overline{T_1}\texttt{>}.m\texttt{<}\overline{T_2}\texttt{>}) = \langle \overline{x}, [\overline{T_1}/\overline{X}, \overline{T_2}/\overline{Y}]e\rangle\end{array}}$$

Dict Type: $\boxed{\begin{array}{l}\mathit{dictty}(I) = \overline{\tau} \\ \mathit{dictty}(M) = \overline{\tau}\end{array}}$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>} : I\ \{\overline{U}\ \overline{f}\texttt{;}\ \overline{md}\}\ \texttt{with}\ \overline{\tau}}{\mathit{dictty}(C\texttt{<}\overline{T}\texttt{>}) = [\overline{T}/\overline{X}]\overline{\tau}}$$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X} \triangleright \overline{s_1}\texttt{>} : I\ \{\overline{T'}\ \overline{f}\texttt{;}\ \overline{md}\}\ \texttt{with}\ \overline{\tau'} \quad \texttt{static}\ U'\ m\texttt{<}\overline{Y} \triangleright \overline{s_2}\texttt{>(}\overline{\tau}\ \overline{x}\texttt{)}\ \{\ e\}\ \texttt{with}\ \overline{\tau''} \in \overline{md}}{\mathit{dictty}(C\texttt{<}\overline{T}\texttt{>::}m\texttt{<}\overline{U}\texttt{>}) = [\overline{T}/\overline{X}, \overline{U}/\overline{Y}]\overline{\tau''}}$$

Figure 6. Macros on fields and methods lookup.

$E \vdash T \triangleright s$	(T *satisfies constraint* s *in* E)
$E \vdash D\ ok$	(D *well-formed in* E)
$E \vdash I <: I'$	(I *is subtype of* I' *in* E)
$E \vdash e : T$	(e *has type* T *in* E)
$\vdash md\ ok\ in\ C\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>}$	(md *well-formed in* $C\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>}$)
$\vdash cd\ ok$	(cd *well-formed*)

Macros for manipulating fields and methods (Figure 6) are formulated in largely the same way as those of BILG. The definitions are based on substitutions of types. Since types do not affect evaluation, these substitutions can be viewed as no-ops at run-time. The macro *dictty* simply collects the types of the dictionary elements to help ease the presentation of the typing rules.

The typing rules are shown in Figure 7 and Figure 8. A type is well-formed in an environment only if the environment contains all the type variables used in the type. An instantiated type or method descriptor is well-formed if the type arguments satisfy the constraints as specified by the judgment of constraint satisfaction. Class typing makes sure that any type argument of a class which satisfies the constraints can be safely used to instantiate the superclass; it also enforces that dictionaries only contain type-reps. Method typings are straightforward extensions of those of BILG.

The expression typing takes into account the constraint satisfaction for method instantitations. Types of type-reps are checked to ensure that they indeed represent the types of interest. Typing the

Constraint Satisfaction: $\boxed{E \vdash T \triangleright s}$

$$\frac{}{\overline{X} \triangleright \overline{s}, \overline{x} : \overline{\tau} \vdash X_i \triangleright s_i} \qquad \frac{}{E \vdash I \triangleright \texttt{ref}}$$

$$\frac{}{E \vdash \texttt{int32} \triangleright \texttt{i4}} \qquad \frac{}{E \vdash \texttt{int64} \triangleright \texttt{i8}}$$

Descriptor Formation: $\boxed{E \vdash D\ ok}$

$$\frac{}{\overline{X} \triangleright \overline{s}, \overline{x} : \overline{\tau} \vdash X_i\ ok} \qquad \frac{}{E \vdash \texttt{int32}\ ok} \qquad \frac{}{E \vdash \texttt{int64}\ ok}$$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>} : I\ \{\ldots\}\ \texttt{with}\ \overline{\tau} \quad E \vdash \overline{T}\ ok \qquad E \vdash \overline{T} \triangleright \overline{s}}{E \vdash C\texttt{<}\overline{T}\texttt{>}\ ok}$$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X} \triangleright \overline{s_1}\texttt{>} : I\ \{\overline{T'}\ \overline{f}\texttt{;}\ \overline{md}\}\ \texttt{with}\ \overline{\tau'} \quad \texttt{static/virtual}\ U'\ m\texttt{<}\overline{Y} \triangleright \overline{s_2}\texttt{>(}\overline{\tau}\ \overline{x}\texttt{)}\ \{\ e\}\ \texttt{with}\ \overline{\tau''} \in \overline{md} \quad E \vdash C\texttt{<}\overline{T}\texttt{>}\ ok \qquad E \vdash \overline{U}\ ok \qquad E \vdash \overline{U} \triangleright \overline{s_2}}{E \vdash C\texttt{<}\overline{T}\texttt{>::}m\texttt{<}\overline{U}\texttt{>}\ ok}$$

Class and Method Typing: $\boxed{\begin{array}{l}\vdash cd\ ok \\ \vdash md\ ok\ in\ C\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>}\end{array}}$

$$\frac{\overline{X} \triangleright \overline{s} \vdash I\ ok \qquad \mathit{fields}(I) = \overline{U}\ \overline{g} \qquad \overline{f} \text{ and } \overline{g} \text{ disjoint} \quad \vdash \overline{md}\ ok\ in\ C\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>} \qquad \tau_i = \texttt{Rep(}\ldots\texttt{)}}{\vdash \texttt{class}\ C\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>} : I\ \{\overline{T}\ \overline{f}\texttt{;}\ \overline{md}\}\ \texttt{with}\ \overline{\tau}\ ok}$$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>} : I\ \{\ldots\}\ \texttt{with}\ \overline{\tau'} \quad \overline{X} \triangleright \overline{s}, \overline{Y} \triangleright \overline{s'}, \overline{x} : \overline{\tau} \vdash e : T \qquad \tau''_i = \texttt{Rep(}\ldots\texttt{)}}{\vdash \texttt{static}\ T\ m\texttt{<}\overline{Y} \triangleright \overline{s'}\texttt{>(}\overline{\tau}\ \overline{x}\texttt{)}\ \{\ e\}\ \texttt{with}\ \overline{\tau''}\ ok\ in\ C\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>}}$$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>} : I\ \{\ldots\}\ \texttt{with}\ \overline{\tau} \quad \overline{X} \triangleright \overline{s}, \overline{Y} \triangleright \overline{s'}, \overline{x} : \overline{T}, \texttt{this} : C\texttt{<}\overline{X}\texttt{>} \vdash e : T}{\vdash \texttt{virtual}\ T\ m\texttt{<}\overline{Y} \triangleright \overline{s'}\texttt{>(}\overline{T}\ \overline{x}\texttt{)}\ \{\ e\}\ ok\ in\ C\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>}}$$

Figure 7. Class and method typing.

dicitionary access is straightforward given the macro *dictty*. As in the implementation, dictionaries of objects are accessed through the object record. Hence we choose to let $\texttt{objdict}_i e$ obtain the dictionary through the object e, instead of directly taking the corresponding type-rep.

To obtain a modelling closer to the implementation and achieve more appealing erasure properties, we give a small-step reduction semantics for BILC in Figure 9. The judgment $\delta \vdash e \mapsto e'$ means "under dictionary map δ the expression e reduces to expression e'". Substitutions of terms should be viewed as local bindings in an implementation. We omit the usual congruence rules due to space constraints.

A well-formed dictionary map is required for the safe execution of BILC programs. A dictionary map δ is well-formed, written $\vdash \delta$, if it maps every closed descriptor D to a vector of values $\delta(D) = \overline{v}$ such that $\{\} \vdash \overline{v} : \mathit{dictty}(D)$. In practice, of course, one cannot implement such an infinite map directly; but observe that any particular terminating program execution will only touch a finite number of entries in the map. This can be implemented by lazy lookup and creation of dictionaries, as is the case in the CLR.

Subtyping: $\boxed{E \vdash I <: I'}$

$$\frac{}{E \vdash I <: I} \qquad \frac{E \vdash I <: I' \qquad E \vdash I' <: I''}{E \vdash I <: I''}$$

$$\frac{\mathcal{D}(C) = \texttt{class } C\texttt{<}\overline{X} \triangleright \overline{s}\texttt{>} : I\,\{\ldots\} \qquad E \vdash C\texttt{<}\overline{T}\texttt{>}\ ok}{E \vdash C\texttt{<}\overline{T}\texttt{>} <: [\overline{T}/\overline{X}]I}$$

Expression Typing: $\boxed{E \vdash e : \tau}$

$$\frac{E \vdash e : I \qquad E \vdash I <: I'}{E \vdash e : I'} \qquad \frac{}{E \vdash i4 : \texttt{int32}}$$

$$\frac{}{E \vdash i8 : \texttt{int64}} \qquad \frac{}{\overline{X} \triangleright \overline{s}, \ldots x : \tau \ldots \vdash x : \tau}$$

$$\frac{E \vdash I\ ok \qquad E \vdash e : \texttt{Rep}(I) \qquad \mathit{fields}(I) = \overline{T}\ \overline{f} \qquad E \vdash \overline{e'} : \overline{T}}{E \vdash I(e, \overline{e'}) : I}$$

$$\frac{\mathit{mtype}(I.m) = \texttt{<}\overline{X} \triangleright \overline{s}\texttt{>}\overline{\tau} \to T \qquad E \vdash \overline{U} \triangleright \overline{s} \qquad E \vdash \overline{e} : [\overline{U}/\overline{X}]\overline{\tau}}{E \vdash \overline{e}\ \texttt{call}\ I\texttt{::}m\texttt{<}\overline{U}\texttt{>} : [\overline{U}/\overline{X}]T}$$

$$\frac{\mathit{mtype}(I.m) = \texttt{<}\overline{X} \triangleright \overline{s}\texttt{>}\overline{T} \to T' \qquad E \vdash \overline{U} \triangleright \overline{s} \qquad E \vdash e_0 : I \qquad E \vdash \overline{e} : [\overline{U}/\overline{X}]\overline{T}}{E \vdash e_0\ \overline{e}\ \texttt{callvirt}\ I\texttt{::}m\texttt{<}\overline{U}\texttt{>} : [\overline{U}/\overline{X}]T'}$$

$$\frac{E \vdash e : I \qquad \mathit{fields}(I) = \overline{T}\ \overline{f}}{E \vdash e\ \texttt{ldfld}\ I\texttt{::}f_i : T_i}$$

$$\frac{E \vdash e : I' \qquad E \vdash e' : \texttt{Rep}(I) \qquad E \vdash e'' : I}{E \vdash e\ \texttt{isinst}_I\ e'\ \texttt{or}\ e'' : I}$$

$$\frac{E \vdash T\ ok}{E \vdash R_T : \texttt{Rep}(T)} \qquad \frac{E \vdash M\ ok}{E \vdash R_M : \texttt{Rep}(M)}$$

$$\frac{E \vdash C\texttt{<}\overline{T}\texttt{>}\ ok \qquad E \vdash \overline{e} : \overline{\texttt{Rep}(T)}}{E \vdash \texttt{mkrep}_{C\texttt{<}\overline{T}\texttt{>}}\overline{e} : \texttt{Rep}(C\texttt{<}\overline{T}\texttt{>})}$$

$$\frac{E \vdash C\texttt{<}\overline{T}\texttt{>::}m\texttt{<}\overline{U}\texttt{>}\ ok \qquad E \vdash \overline{e} : \texttt{Rep}(T) \qquad E \vdash \overline{e'} : \overline{\texttt{Rep}(U)}}{E \vdash \texttt{mkrep}_{C\texttt{<}\overline{T}\texttt{>::}m\texttt{<}\overline{U}\texttt{>}}(\overline{e}, \overline{e'}) : \texttt{Rep}(C\texttt{<}\overline{T}\texttt{>::}m\texttt{<}\overline{U}\texttt{>})}$$

$$\frac{E \vdash e : I \qquad \mathit{dictty}(I) = \overline{\tau}}{E \vdash \texttt{objdict}_i e : \tau_i}$$

$$\frac{E \vdash e : \texttt{Rep}(M) \qquad \mathit{dictty}(M) = \overline{\tau}}{E \vdash \texttt{mdict}_i e : \tau_i}$$

Figure 8. Expression typing.

Reflected Subtyping: $\boxed{R_I \prec R_I}$

$$\frac{\{\} \vdash I <: I'}{R_I \prec R_{I'}}$$

Reduction: (congruence rules omitted) $\boxed{\delta \vdash e \mapsto e'}$

$$\frac{\mathit{mbody}(I.m\texttt{<}\overline{T}\texttt{>}) = \langle \overline{x}, e' \rangle}{\delta \vdash \overline{v}\ \texttt{call}\ I\texttt{::}m\texttt{<}\overline{T}\texttt{>} \mapsto [\overline{v}/\overline{x}]e'}$$

$$\frac{v_0 = I'(v', v'') \qquad \mathit{mbody}(I'.m\texttt{<}\overline{T}\texttt{>}) = \langle \overline{x}, e' \rangle}{\delta \vdash v_0\ \overline{v}\ \texttt{callvirt}\ I\texttt{::}m\texttt{<}\overline{T}\texttt{>} \mapsto [v_0/\texttt{this}, \overline{v}/\overline{x}]e'}$$

$$\frac{v = I'(v', \overline{v}) \qquad \mathit{fields}(I) = \overline{T}\ \overline{f}}{\delta \vdash v\ \texttt{ldfld}\ I\texttt{::}f_i \mapsto v_i}$$

$$\frac{v_1 = I(v, \overline{v'}) \qquad v \prec v_2}{\delta \vdash v_1\ \texttt{isinst}_T\ v_2\ \texttt{or}\ v_3 \mapsto v_1}$$

$$\frac{v_1 = I(v, \overline{v'}) \qquad v \not\prec v_2}{\delta \vdash v_1\ \texttt{isinst}_T\ v_2\ \texttt{or}\ v_3 \mapsto v_3}$$

$$\frac{v_i = R_{T_i}}{\delta \vdash \texttt{mkrep}_{C\texttt{<}\overline{T}\texttt{>}}\overline{v} \mapsto R_{C\texttt{<}\overline{T}\texttt{>}}}$$

$$\frac{v_i = R_{T_i} \qquad v'_i = R_{U_i}}{\delta \vdash \texttt{mkrep}_{C\texttt{<}\overline{T}\texttt{>::}m\texttt{<}\overline{U}\texttt{>}}(\overline{v}, \overline{v'}) \mapsto R_{C\texttt{<}\overline{T}\texttt{>::}m\texttt{<}\overline{U}\texttt{>}}}$$

$$\frac{v = C\texttt{<}\overline{T}\texttt{>}(R_{C\texttt{<}\overline{T}\texttt{>}}, \overline{w}) \qquad \delta(C\texttt{<}\overline{T}\texttt{>}) = \overline{v}}{\delta \vdash \texttt{objdict}_i v \mapsto v_i}$$

$$\frac{v = R_{C\texttt{<}\overline{T}\texttt{>::}m\texttt{<}\overline{U}\texttt{>}} \qquad \delta(C\texttt{<}\overline{T}\texttt{>::}m\texttt{<}\overline{U}\texttt{>}) = \overline{v}}{\delta \vdash \texttt{mdict}_i v \mapsto v_i}$$

Figure 9. Reduction rules of BILC.

The most interesting reduction rules are those for type test; they show that type-reps are inspected at run-time. We use a rule for "reflected subtyping" to handle these cases, lifting the static subtype relation over closed types to a relation over type-reps.

The reduction rule for object dictionary lookup ($\texttt{objdict}_i v$) inspects the type-rep stored in the object for fetching a dictionary out of δ. Although the class instantiation also appears as the type tag of the object, it is not accessible at run-time when all types are erased. The reduction rule for method dictionary lookup ($\texttt{mdict}_i v$), in contrast, directly inspects the type-rep argument.

An inspection of the reduction rules shows that types are irrelevant for the evaluation. A type-erasure semantics [5], in which all type-related operations and parameters are erased, is presented in the companion technical report [28].

As in BILG, we annotate judgments and helpers with the class table $\mathcal{D}$ when we wish to be more explicit. We say that $\mathcal{D}$ is a valid class table if $\vdash^{\mathcal{D}} cd\ ok$ for each class definition cd in $\mathcal{D}$.

Lemma 1 (BILC substitution) Suppose that $\mathcal{D}$ is a valid class table. If $\overline{X} \triangleright \overline{s}, \overline{x} : \overline{\tau'} \vdash^{\mathcal{D}} e : \tau$, $\vdash \overline{T} \triangleright \overline{s}$, and $\vdash^{\mathcal{D}} \overline{v} : [\overline{T}/\overline{X}]\overline{\tau'}$ then $\vdash^{\mathcal{D}} [\overline{v}/\overline{x}, \overline{T}/\overline{X}]e : [\overline{T}/\overline{X}]\tau$.

Proof sketch. By induction on the structure of the derivation $\overline{X} \triangleright \overline{s}, \overline{x} : \overline{\tau'} \vdash^{\mathcal{D}} e : \tau$. □

Theorem 2 (BILC progress) Suppose that $\mathcal{D}$ is a valid class table. If $\vdash^{\mathcal{D}} \delta$ and $\{\} \vdash^{\mathcal{D}} e : \tau$ then either e is a value or there exists e' such that $\delta \vdash^{\mathcal{D}} e \mapsto e'$.

Proof sketch. By induction on the structure of the typing derivation. □

Theorem 3 (BILC evaluation preserves typing) Suppose that $\mathcal{D}$ is a valid class table, $\overline{X} \triangleright \overline{s}, \overline{x} : \overline{\tau} \vdash^{\mathcal{D}} e : \tau$, $\vdash^{\mathcal{D}} v_i : [\overline{U}/\overline{X}]\tau_i$ holds for all $v_i \in \overline{v}$ and $\tau_i \in \overline{\tau}$, and $\vdash^{\mathcal{D}} \delta$. If $\delta \vdash^{\mathcal{D}} [\overline{v}/\overline{x}, \overline{U}/\overline{X}]e \mapsto e'$ then $\overline{X} \triangleright \overline{s}, \overline{x} : \overline{\tau} \vdash^{\mathcal{D}} e' : \tau$.

Proof sketch. By induction on the structure of the evaluation derivation. □

4 Translation

4.1 Overview

BILC as introduced in the previous section is expressive enough to demonstrate most of the key ideas used in the implementation of generic IL. Run-time type information is captured using type-reps. Objects contain the type-reps of their class instantiations; thus the class type-parameters can be accessed at run-time. To share code as much as possible and at the same time efficiently support exact run-time types, type-reps of open types are pre-computed and stored in dictionaries, which are passed in as an extra parameter for static methods, and accessed through the self pointer for non-generic virtual methods. For generic virtual methods, type-reps for open types can be constructed at run-time. All these features are studied in our formal translation.

Based on the type arguments, every generic class or method is translated into several specialized versions, with some of them shared by different instantiations. Following the current implementation, we use a global policy, generating unshared code for primitive instantiations and shared code for the rest. One can think of this policy as a *partition* $\{\texttt{ref}, \texttt{i4}, \texttt{i8}\}$ of all closed types, with every type satisfying exactly one constraint.

4.2 Example

We first give an example to illustrate the idea of static method translation. The source program that we are compiling, as shown in Figure 10, is a simple generic class $C\texttt{<}X\texttt{>}$ which defines a generic static method $m\texttt{<}Y\texttt{>}$. The body of the method refers to an open type $C_2\texttt{<}X\texttt{>}$ and an open method descriptor $C_3\texttt{<}X\texttt{>::}m_3\texttt{<}Y\texttt{>}$.

The translation specializes the class C and its method m into several versions in the target language, one for each constraint in the partition, based on their type arguments. Given the above partition $\{\texttt{ref}, \texttt{i4}, \texttt{i8}\}$, we use $C\texttt{@r}$ as the class name for the specialized version of C where the type argument satisfies `ref`, and $C\texttt{@i4}$ and $C\texttt{@i8}$ for those where the type argument satisfies `i4` and `i8` respectively. Similar mangling applies to the method name m.

```
class C<X> : C'<X> {
  X f;
  static int32 m<Y>(X x, Y y) {
    ...
    e1 newobj C1<>
    ...
    e2 isinst C2<X> or e2'
    ...
    e3 call int32 C3<X>::m3<Y>(int32)
    ...
  }
}
```

Figure 10. Example source program.

To access open descriptors inside static method bodies whose code is shared between different instantiations, we let static methods take an extra argument as the type-rep of its own instantiation. The callers of the method must provide the actual type-rep given the instantiation information.

The type-rep of a static method instantiation contains enough information about the type arguments, which is sufficient in constructing the type-reps of open descriptors. However, if the construction happens repetitively, extra overhead is incurred. We apply a "lifting" optimization where all the type-reps of open descriptors form a dictionary which is constructed only once on the caller's side.

Dictionary constructions are abstracted using the dictionary map (δ) in our formalization. For any particular BILG method, all the sites where open types will be needed can be determined statically, and furthermore these type expressions are fully known statically with respect to the type parameters in scope. Our translation puts the types of the dictionary elements in the class or method definition for static type-checking. Based on these types and the fact that every representation type is inhabited by exactly one type-rep values, the dictionary map simply maps any given class instantiations or method descriptors to their dictionaries.

The statically determined dictionary layout indicates that one can access the dictionary for open descriptors by specifying fixed offsets. This is a much cheaper operation than the run-time construction of type-reps.

Based on this scheme, a translation of the above class C is shown in Figure 11, where we use $|e_i|$ to stand for the translation of e_i. Class C is specialized into three different classes $C\texttt{@r}$, $C\texttt{@i4}$ and $C\texttt{@i8}$. As indicated by the `with` clauses of the class definitions, the dictionaries of these specialized classes are empty. This is because our translation makes use of dictionaries of classes for virtual methods only.

Take the definition of $C\texttt{@r}$ as an example, the super class C' is specialized based on the type argument. In correspondence with the method m in the source program, we now define three specialized versions. Method $m\texttt{@r}$ suffices in explaining the idea of static method translation. Assuming $C_2\texttt{<}X\texttt{>}$ and $C_3\texttt{<}X\texttt{>::}m_3\texttt{<}Y\texttt{>}$ are the only open descriptors in the source program, the dictionary type is as shown in the `with` clause. We let the method definition take an extra parameter x_d, whose type dictates that only the actual type-rep of the method can be passed in. Inside the method body, we use the dictionary lookup expression $\texttt{mdict}_i x_d$ to access for open type-reps. For the closed type $C_1\texttt{<>}$ in the source program, the corresponding type-rep is statically known as $R_{C_1\texttt{<>}}$.

Specializing at `ref`

```
class C@r<X ▷ ref> : C'@r<X> {
  X f;
  static int32 m@r<Y ▷ ref>
      (Rep(C@r<X>::m@r<Y>) x_d, X x, Y y) {
    ...
    C_1<>(R_C1<>, |e_1|)
    ...
    |e_2| isinst_C2@r<X> (mdict_1 x_d) or |e'_2|
    ...
    (mdict_2 x_d) |e_3|
        call C_3@r<X>::m_3@r<Y>
    ...
  } with (Rep(C_2@r<X>), Rep(C_3@r<X>::m_3@r<Y>))

  static int32 m@i4<>
      (Rep(C@r<X>::m@i4<>) x_d, X x, int32 y) {
    ...
    C_1<>(R_C1<>, |e_1|)
    ...
    |e_2| isinst_C2@r<X> (mdict_1 x_d) or |e'_2|
    ...
    (mdict_2 x_d) |e_3|
        call C_3@r<X>::m_3@i4<>
    ...
  } with (Rep(C_2@r<X>), Rep(C_3@r<X>::m_3@i4<>))

  ...code for m@i8 omitted...
} with ()
```

Specializing at `i4`

```
class C@i4<> : C'@i4<> {
  int32 f;
  static int32 m@r<Y ▷ ref>
      (Rep(C@i4<>::m@r<Y>) x_d, int32 x, Y y) {
    ...
    C_1<>(R_C1<>, |e_1|)
    ...
    |e_2| isinst_C2@i4<> R_C2@i4<> or |e'_2|
    ...
    (mdict_1 x_d) |e_3| call C_3@i4<>::m_3@r<Y>
    ...
  } with (Rep(C_3@i4<>::m_3@r<X>))

  ...code for m@i4 and m@i8 omitted...
} with ()
```

Specializing at `i8` **Omitted.**

Figure 11. Translation of the example source program.

The remainder of the code is generated following similar ideas. In relation with an actual implementation, the argument passed to static methods as x_d can be understood as a pointer to the method's dictionary. The dictionary lookup operation `mdict` consists of dereferencing the dictionary pointer and fetching an element based on a fixed offset. The potentially infinite nature of the dictionary map is handled by constructing dictionaries on-demand.

4.3 Formal Translation

The above example illustrates the translation of static methods, which makes use of a dictionary for each method instantiation. Although non-generic virtual methods could use the same scheme, its non-generic nature allows optimizations. We let the dictionary of a class instantiation be a combined dictionary for all its non-generic virtual methods. Inside a virtual method body, this dictionary can be accessed through the self pointer; thus there is no need to carry an extra argument.

Generic virtual methods are altogether more challenging. The dictionary map cannot generate an appropriate dictionary from the caller's side, because it is unknown statically which method body will be called. In our formalization, we fall back to construct type-reps of open descriptors at run-time. This provides a good comparison with, and demonstrates the advantage of, the dictionary-passing scheme used for static methods. We will discuss more efficient implementations for generic virtual methods in section 5.

For simplicity, we omit the special treatment of non-generic classes, for which optimizations are straightforward.

The formal translation roughly consists of expression translation, method translation, and class translation.

Expression translation Expression translation is shown in Figure 12. It is parameterized on a *descriptor lookup* function ψ and a *specialization environment* ρ. The function ψ returns type-reps for *all* descriptors ($\psi(D) : \texttt{Rep}(D)$). The environment ρ maps type variables to their constraints ($\rho ::= \overline{X} \mapsto \overline{s}$). These are both provided by the method translation.

Most parts of the expression translation simply propagate the translation to sub-components. The only interesting cases are object creation, run-time type test, and method invocation. Object creation and run-time type test obtain type-reps using ψ. Static calls obtain the type-reps of the method descriptors and use them as an extra argument. There is nothing special about the translation of non-generic virtual calls, because the implicit self pointer suffices in locating the dictionary. On contrast, generic virtual calls require passing in the type-reps of the type arguments to the generic method, so that type-reps of open types inside the target virtual method can be constructed.

We define a few helpers to abstract operations used in the translation. Some target class or method may expect less type arguments due to specialization. This is captured as *argument specialization*, which performs a target-to-target transformation on vector of types by throwing away anthing that is specialized for a singleton constraint. *Type specialization* tells us which constraint a type belongs to; it refers to ρ for the specialization choices of type variables.

Type translation is mostly straightforward. To translate an instantiated type, we first translate the type arguments recursively. A specialized version of the class is then chosen, based on the constraints

Exp Translation: $\boxed{|e|^{\Psi}_{\rho} = e'}$

$$
\begin{array}{rcl}
|\texttt{ldc.i4}\ i4|^{\Psi}_{\rho} & = & i4 \\
|\texttt{ldc.i8}\ i8|^{\Psi}_{\rho} & = & i8 \\
|\texttt{ldarg}\ x|^{\Psi}_{\rho} & = & x \\
|\overline{e}\ \texttt{newobj}\ I|^{\Psi}_{\rho} & = & |I|_{\rho}(e_R, |\overline{e}|^{\Psi}_{\rho}) \\
 & & \text{where } e_R = \Psi(|I|_{\rho}) \\
|\overline{e}\ \texttt{call}\ M|^{\Psi}_{\rho} & = & e_d\ |\overline{e}|^{\Psi}_{\rho}\ \texttt{call}\ |M|_{\rho} \\
 & & \text{and } e_d = \Psi(|M|_{\rho}) \\
|e_0\ \overline{e}\ \texttt{callvirt}\ M|^{\Psi}_{\rho} & = & |e_0|^{\Psi}_{\rho}\ |\overline{e}|^{\Psi}_{\rho}\ \texttt{callvirt}\ |M|_{\rho} \\
 & & \text{if } M \text{ non-generic} \\
|e_0\ \overline{e}\ \texttt{callvirt}\ M|^{\Psi}_{\rho} & = & |e_0|^{\Psi}_{\rho}\ \overline{e_R}\ |\overline{e}|^{\Psi}_{\rho}\ \texttt{callvirt}\ |M|_{\rho} \\
 & & \text{where } |M|_{\rho} = C\texttt{<}\overline{T}\texttt{>::}m\texttt{<}\overline{U}\texttt{>} \\
 & & \text{and } \overline{e_R} = \Psi(\overline{T}\,\overline{U}) \\
|e\ \texttt{ldfld}\ T\ I\texttt{::}f|^{\Psi}_{\rho} & = & |e|^{\Psi}_{\rho}\ \texttt{ldfld}\ |I|_{\rho}\texttt{::}f \\
|e\ \texttt{isinst}\ I\ \texttt{or}\ e_1|^{\Psi}_{\rho} & = & |e|^{\Psi}_{\rho}\ \texttt{isinst}_{|I|_{\rho}}\ e_R\ \texttt{or}\ |e_1|^{\Psi}_{\rho} \\
 & & \text{where } e_R = \Psi(|I|_{\rho})
\end{array}
$$

Arguments Specialization: $\boxed{\overline{T}\%\overline{s} = \overline{T'}}$

$$
\begin{array}{rcl}
[\,]\%[\,] & = & [\,] \\
(T,\overline{T})\%(s,\overline{s}) & = & \begin{cases} T,\overline{T}\%\overline{s} & \text{if } s = \texttt{ref} \\ \overline{T}\%\overline{s} & \text{otherwise} \end{cases}
\end{array}
$$

Type Specialization: $\boxed{spec_{\rho}(T) = s}$

$$
\begin{array}{rcl}
spec_{\rho}(\texttt{int32}) & = & \texttt{i4} \\
spec_{\rho}(\texttt{int64}) & = & \texttt{i8} \\
spec_{\rho}(X) & = & \rho(X) \\
spec_{\rho}(C\texttt{<}\overline{T}\texttt{>}) & = & \texttt{ref}
\end{array}
$$

Type Translation: $\boxed{|T|_{\rho} = T'}$

$$
\begin{array}{rcl}
|\texttt{int32}|_{\rho} & = & \texttt{int32} \\
|\texttt{int64}|_{\rho} & = & \texttt{int64} \\
|X|_{\rho} & = & \begin{cases} X & \text{if } \rho(X) = \texttt{ref} \\ \texttt{int32} & \text{if } \rho(X) = \texttt{i4} \\ \texttt{int64} & \text{if } \rho(X) = \texttt{i8} \end{cases} \\
\left|C\texttt{<}\overline{T}\texttt{>}\right|_{\rho} & = & C@\overline{s}\texttt{<}\left|\overline{T}\right|_{\rho}\%\overline{s}\texttt{>} \text{ where } \overline{s} = spec_{\rho}(\overline{T})
\end{array}
$$

Method Descriptor Translation: $\boxed{|M|_{\rho} = M'}$

$$
\begin{array}{rcl}
\left|U\ I\texttt{::}m\texttt{<}\overline{T}\texttt{>}(\overline{U})\right|_{\rho} & = & |I|_{\rho}\texttt{::}m@\overline{s}\texttt{<}\left|\overline{T}\right|_{\rho}\%\overline{s}\texttt{>} \\
 & & \text{where } \overline{s} = spec_{\rho}(\overline{T})
\end{array}
$$

Value Translation: $\boxed{|v|_{\rho} = v'}$

$$
\begin{array}{rcl}
|i4|_{\rho} & = & i4 \\
|i8|_{\rho} & = & i8 \\
\left|I(\overline{f} \mapsto \overline{v})\right|_{\rho} & = & |I|_{\rho}(R_{|I|_{\rho}}, |\overline{v}|_{\rho})
\end{array}
$$

Figure 12. Expression translation.

Object dictionary lookup: $\boxed{objdict^{\overline{\tau}}_{x:\tau}(D) = e}$

$$
objdict^{\overline{\tau}}_{x:\tau}(D) = \begin{cases} R_D & \text{if } D \text{ closed} \\ \texttt{objdict}_i x & \text{if } \tau_i = \texttt{Rep}(D) \end{cases}
$$

Method dictionary lookup: $\boxed{mdict^{\overline{\tau}}_{x:\tau}(D) = e}$

$$
mdict^{\overline{\tau}}_{x:\tau}(D) = \begin{cases} R_D & \text{if } D \text{ closed} \\ \texttt{mdict}_i x & \text{if } \tau_i = \texttt{Rep}(D) \end{cases}
$$

Run-time lookup: $\boxed{mkrep^{\overline{X}}_{\overline{x}:\overline{\tau}}(D) = e}$

$$
\begin{array}{rcl}
mkrep^{\overline{X}}_{\overline{x}:\overline{\tau}}(X_i) & = & x_i \\
mkrep^{\overline{X}}_{\overline{x}:\overline{\tau}}(D) & = & R_D, \text{ if } D \text{ closed} \\
mkrep^{\overline{X}}_{\overline{x}:\overline{\tau}}(C\texttt{<}\overline{T}\texttt{>}) & = & \texttt{mkrep}_{C\texttt{<}\overline{T}\texttt{>}}(\overline{e}) \\
 & & \text{where } e_i = mkrep^{\overline{X}}_{\overline{x}:\overline{\tau}}(T_i) \\
mkrep^{\overline{X}}_{\overline{x}:\overline{\tau}}(C\texttt{<}\overline{T}\texttt{>::}m\texttt{<}\overline{U}\texttt{>}) & = & \texttt{mkrep}_{C\texttt{<}\overline{T}\texttt{>::}m\texttt{<}\overline{U}\texttt{>}}(\overline{e}\,\overline{e'}) \\
 & & \text{where } e_i = mkrep^{\overline{X}}_{\overline{x}:\overline{\tau}}T_i \\
 & & e'_i = mkrep^{\overline{X}}_{\overline{x}:\overline{\tau}}U_i
\end{array}
$$

Figure 14. Descriptor lookup.

of these type arguments. Furthermore, argument specialization filters out types that are fully-specialized. Method descriptor translation is defined similarly.

Class translation As shown in Figure 13 (where the trivial translation of `object` is omitted), a BILG class is translated into a set of classes in BILC, based on the constraints of the type arguments $\overline{X}$. These constraints form a specialization environment ρ. The set of all possible specialization environments for $\overline{X}$ is abstracted as $\text{SEnv}(\overline{X})$. A target class is generated for each $\rho \in \text{SEnv}(\overline{X})$.

For a specialization environment ρ, the translation mangles the class name with the constraints of the type arguments. The arguments are specialized properly, and the super class instantiation and field types are translated using type translation.

Given a BILG class definition, the dictionary types ($\overline{\tau}$) of the translated BILC classes can be determined by walking through all non-generic virtual method bodies in the source and collecting all open descriptors. Straightforward recursion over the program structure suffices in defining this operation, whose lengthy definition is omitted due to space constraints.

Based on the dictionary type, the dictionary map δ can easily generate the actual dictionary for any given class instantiation. For similar space concerns, we also omit a formal definition of this map.

Once the dictionary layout of a class is determined, the descriptor lookup function ψ can be defined by applying the *object dictionary lookup* function (*objdict*) on the dictionary type and the self pointer. As shown in Figure 14, for a closed descriptor, *objdict* returns its type-rep directly; for an open descriptor, *objdict* returns an expression which queries the proper dictionary slot for its type-rep.

Given the descriptor lookup function ψ, the specialization environment ρ, and the class instantiation itself, every method in the source class is translated into a set of methods, and all these sets are combined to form the methods of the target class.

Method Translation: $\boxed{|md|^{\psi,I}_{\rho} = \overline{md}}$

$$\left|\texttt{virtual}\ U\ m\texttt{<>}(\overline{T}\ \overline{x})\ \{\ e\}\right|^{\psi,I}_{\rho} = \left\{\texttt{virtual}\ |U|_{\rho}\ m\texttt{<>}(\left|\overline{T}\right|_{\rho}\ \overline{x})\ \{\ |e|^{\psi}_{\rho}\}\right\}$$

$$\left|\texttt{static}\ U\ m\texttt{<}\overline{Y}\texttt{>}(\overline{T}\ \overline{x})\ \{\ e\}\right|^{\psi,I}_{\rho} = \left\{\texttt{static}\ |U|_{\rho\cup\rho'}\ m@\overline{s}\texttt{<}\overline{Y'} \triangleright \overline{\texttt{ref}}\texttt{>}(\tau_d\ x_d, \left|\overline{T}\right|_{\rho\cup\rho'}\ \overline{x})\ \{\ |e|^{\psi'}_{\rho\cup\rho'}\}\ \texttt{with}\ \overline{\tau}\right\}_{\rho'\in \mathrm{SEnv}(\overline{Y})}$$

where $\rho(\overline{Y}) = \overline{s}$ and $\overline{Y}\,\%\,\overline{s} = \overline{Y'}$ and $\tau_d = \texttt{Rep}(I\texttt{::}m@\overline{s}\texttt{<}\overline{Y'}\texttt{>})$
and $\overline{\tau}$ is the dictionary type and $\psi' = mdict^{\overline{\tau}}_{x_d:\tau_d}$

$$\left|\texttt{virtual}\ U\ m\texttt{<}\overline{Y}\texttt{>}(\overline{T}\ \overline{x})\ \{\ e\}\right|^{\psi,C\texttt{<}\overline{X}\texttt{>}}_{\rho} = \left\{\texttt{virtual}\ |U|_{\rho\cup\rho'}\ m@\overline{s}\texttt{<}\overline{Y'} \triangleright \overline{\texttt{ref}}\texttt{>}(\overline{\tau}\ \overline{z}, \left|\overline{T}\right|_{\rho\cup\rho'}\ \overline{x})\ \{\ |e|^{\psi'}_{\rho\cup\rho'}\}\right\}_{\rho'\in \mathrm{SEnv}(\overline{Y})}$$

where $\rho(\overline{Y}) = \overline{s}$ and $\overline{Y}\,\%\,\overline{s} = \overline{Y'}$ and $\overline{X}\,\overline{Y'} = Z_1, \ldots, Z_n$
and $\psi' = mkrep^{\overline{Z}}_{\overline{z}:\overline{\tau}}$ and $\tau_i = \texttt{Rep}(Z_i)$ and length of $\overline{Y}$ is greater than 0

Class Translation: $\boxed{|cd| = \overline{cd}}$

$$\left|\texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \{\overline{T}\ \overline{f}\texttt{;}\ \overline{md}\}\right| = \left\{\texttt{class}\ C@\overline{s}\texttt{<}\overline{X'} \triangleright \overline{\texttt{ref}}\texttt{>}\ \texttt{:}\ |I|_{\rho}\ \{\left|\overline{T}\right|_{\rho}\ \overline{f}\texttt{;}\ \left|\overline{md}\right|^{\psi,C@\overline{s}\texttt{<}\overline{X'}\texttt{>}}_{\rho}\}\ \texttt{with}\ \overline{\tau}\right\}_{\rho\in \mathrm{SEnv}(\overline{X})}$$

where $\rho(\overline{X}) = \overline{s}$ and $\overline{X}\,\%\,\overline{s} = \overline{X'}$
and $\overline{\tau}$ is the dictionary type and $\psi = objdict^{\overline{\tau}}_{\texttt{this}:C@\overline{s}\texttt{<}\overline{X'}\texttt{>}}$

Figure 13. Class and method translation.

Method translation Similar to classes, methods are translated into sets of specialized methods. For simplicity, we assume implicit convertion between our set and vector notations.

The translation of a non-generic virtual method is simply performed by translating the signature and method body, passing on the descriptor lookup function ψ and specialization environment ρ.

A static method is translated into multiple versions based on all possible specialization environments ρ' of the method type arguments. Aside from mangling method names, specializing type arguments, and deciding dictionary types, the interesting part of this translation is that an extra parameter is introduced. The type of this extra parameter dictates that the actual argument passed in can only be a type-rep of the method instantiation. For the method body to be able to look up open descriptors using the extra argument, the descriptor lookup function ψ' is defined using *method dictionary lookup* (*mdict*, as shown in Figure 14), which returns type-reps for closed descriptors, and dictionary lookup expressions for open descriptors based on the dictionary layout.

The remainder of static method translation involves translating the signature and method body, and passing on the appropriate descriptor lookup function ψ' and specialization environment $\rho \cup \rho'$ to expression translation.

In contrast, the translation of generic virtual methods takes the type-reps of the type arguments themselves as extra parameters. The descriptor lookup function ψ' is defined using "run-time lookup" (*mkrep*), which returns expressions that construct type-reps at run-time when necessary.

By comparing method dictionary lookup (*mdict*) and run-time lookup (*mkrep*), it is clear that the dictionary-passing scheme (which involves only de-referencing and numeric indexing) is more efficient than building type-reps at run-time (which involves the `mkrep` operation). In the actual implementation of generic IL, a scheme based on run-time lookup is typically slower by at least an order of magnitude than one based on dictionary-passing [14].

Soundness of translation We first define a translation of a type environment E given a dictionary environment ψ and a *matching* specialization environment ρ that covers all type variables in E. In the case where ψ is an object dictionary lookup function, the type translation of E is undefined if the binding for the self pointer argument $(x:\tau)$ does not occur in E. This is merely used to simply the soundness proof.

$$\left|\overline{X},\overline{x}:\overline{T}\right|_{\rho} = \overline{X}\,\%\,\rho(\overline{X}) \triangleright \overline{\texttt{ref}}, \overline{x}:\left|\overline{T}\right|_{\rho}$$

$$\left|\overline{X},\overline{x}:\overline{T}\right|^{objdict^{\overline{\tau}}_{x:\tau}}_{\rho} = \left|\overline{X},\overline{x}:\overline{T}\right|_{\rho} \quad (\text{if } x:\tau \in \left|\overline{X},\overline{x}:\overline{T}\right|_{\rho})$$

$$\left|\overline{X},\overline{x}:\overline{T}\right|^{mdict^{\overline{\tau}}_{x:\tau}}_{\rho} = \left|\overline{X},\overline{x}:\overline{T}\right|_{\rho}, x:\tau$$

$$\left|\overline{X},\overline{x}:\overline{T}\right|^{mkrep^{\overline{X}\,\%\,\rho(\overline{X})}_{\overline{z}:\overline{\tau}}}_{\rho} = \left|\overline{X},\overline{x}:\overline{T}\right|_{\rho}, \overline{z}:\overline{\tau}$$

Based on our formal translation, a valid class table $\mathcal{D}$ of BILG is translated into a valid class table $\mathcal{D}'$ of BILC. The following type-preservation theorem on the translation of expressions can be used to derive the preservation of well-formedness on that of methods and classes, and hence the above preservation of valid class tables. Note that although expression typing assumes a class table, it only uses it in syntactic ways. Thus our type-preservation theorem of expression translation does not reply on the preservation theorem of valid class tables.

Theorem 4 (Type preservation) If $\mathcal{D}$ is a valid class table and translates to $\mathcal{D}'$, $E \vdash^{\mathcal{D}} e : T$, ρ is a matching specialization

environment of E, and ψ satisfies that $|E|^{\psi}_{\rho} \vdash^{\mathcal{D}'} \psi(D) : \texttt{Rep}(D)$ holds for any $D \in dom(\psi)$, then $|E|^{\psi}_{\rho} \vdash^{\mathcal{D}'} |e|^{\psi}_{\rho} : |T|_{\rho}$.

Proof sketch. The proof is done straightforwardly by induction on the structure of the derivation $E \vdash^{\mathcal{D}} e : T$, after factoring out lemmas on field and method lookup macros and subtyping. The key point is that these lemmas only refer to the class table in syntactic ways. Expressions of BILG are all translated into their counterparts in BILC. Although some of them refer to BILC expression on type-reps and dictionaries, they only do so through the descriptor lookup function ψ, whose result types are the desired ones by assumption. □

Our translation is also semantics-preserving. This theorem is formalized by assuming a correct descriptor lookup function ψ which yields expressions that reduce to the correct type-rep.

Theorem 5 (Semantics preservation) Suppose $\mathcal{D}$ is a valid class table and translates to $\mathcal{D}'$, $\vdash^{\mathcal{D}'} \delta$. Suppose $\overline{X}, \bar{x} : \overline{T} \vdash^{\mathcal{D}} e : T'$, $\vdash^{\mathcal{D}} \bar{v} : [\overline{U}/\overline{X}]\overline{T}$, $\rho \in \text{SEnv}(\overline{X})$, $spec_{\rho}(\overline{U}) = \rho(\overline{X}) = \bar{s}$, and ψ satisfies that $\left|\overline{X}, \bar{x} : \overline{T}\right|^{\psi}_{\rho} \vdash^{\mathcal{D}'} \psi(D) : \texttt{Rep}(D)$ for any $D \in dom(\psi)$. Let $\overline{X'} \triangleright \overline{\texttt{ref}}, \bar{x} : \bar{\tau}, \overline{x'} : \overline{\tau'} = \left|\overline{X}, \bar{x} : \overline{T}\right|^{\psi}_{\rho}$ and $\overline{U'} = \left|\overline{U}\right|_{\rho} \% \bar{s}$. If $(\bar{x} = \bar{v}) \vdash^{\mathcal{D}} [\overline{U}/\overline{X}]e \Downarrow w$, and $\vdash^{\mathcal{D}'} \overline{v'} : [\overline{U'}/\overline{X'}]\overline{\tau'}$, then $\delta \vdash^{\mathcal{D}'} [\overline{v'}/\overline{x'}, |\bar{v}|_{\rho}/\bar{x}]\left|[\overline{U}/\overline{X}]e\right|^{\psi}_{\rho} \mapsto^{*} |w|_{\rho}$.

Proof sketch. The proof is done straightforwardly by induction on the structure of the derivation $(\bar{x} = \bar{v}) \vdash^{\mathcal{D}} e \Downarrow w$, after factoring out lemmas on field and method lookup macros. Translations of BILG expressions do not directly refer to δ, because they do not refer to BILC expressions on type-reps and dictionaries directly. They are abstracted away by the descriptor lookup function ψ, whose result types are the desired representation types by assumption. These result expressions reduce to the desired type-reps, because representation types are singleton types. □.

5 Discussion and Related Work

Type-reps BILC makes use of term-level type-reps for operations that require exact run-time types, avoiding a type-passing interpretation (that is, one in which type parameters are interpreted both statically *and* as values passed at runtime). In spirit our type-reps are similar to those of intensional type analysis (ITA) [5] in that every representation type (the `Rep` construct of BILC) is inhabited by exactly one type-rep value, hence type-reps faithfully reflect type information at run-time. However, the handling of type-reps in BILC is quite different from that of ITA. Most importantly, our type-reps are based on *names* of types, instead of *structures*. In particular, BILC does not provide a structural way of constructing and deconstructing type-reps. Instead, type-rep construction is abstracted using a primitive `mkrep`, and the type-rep values (R_D) are inspected by expressions that require types at run-time. For instance, the type test expression (`isinst`) inspects type-reps based on a reflected subtyping relation. (It is interesting to note that the efficient implementation of reflected subtyping tests are the subject of much recent research [23]).

We also extended the idea of name-based type-reps to method descriptors. This greatly simplifies the handling of dictionaries. In the actual implementation of IL generics, a dictionary may contain both type-reps and pointers to other dictionaries, exhibiting a graph structure. Our `Rep` and R constructs are simply based on names, relying on the dictionary map (δ) to provide a level of indirection to connect related dictionaries. This avoids the direct tackling of recursion and sharing in dictionaries. The dictionary map also helps to abstract away from the potential infinite nature of dictionaries due to polymorphic recursion. An implementation can provide a map which builds these dictionaries on demand. In contrast, a model that makes explicit all these features appears to require higher-kinded recursive types, whose formalization is conceivable, but certainly less accessible (see later).

Name-based type-reps may also be an appropriate model for other features that demand name equivalence, such as reflection and serialization. For example, the CLR class `System.Type` (equivalently, `java.lang.Class` in Java) could be modelled by the existential $\exists X.\texttt{Rep}(X)$.

Value classes Our formalization can be extended to support parameterized value classes (called *structs* in C♯), a distinctive feature of which is that runtime types are not carried by values, since subtyping on value types is not supported. Hence to support sharing of method code in value classes, *all* methods must be passed an extra method-rep argument providing access to a dictionary.

Constraints on type parameters can be extended to support sharing of generic classes across distinct value class instantiations, for example using the same code for `Set<Point>` and `Set<Size>` where `Point` and `Size` are both value classes with two `int` fields. The grammar of constraints would be extended with syntax $\{\bar{s}\}$ denoting a 'field layout'; a value class would satisfy such a constraint if its constituent fields satisfied the constraints $\bar{s}$ component-wise.

Such constraints lead to infinitary class specialization; the translation of classes and methods as shown in Figure 13 then truly abstracts away from the implementation, which clearly cannot be realized by static compilation!

Specialization policies We make the observation that the specialization policy need not be fixed in the implementation, and furthermore other interesting policies exist, such as generating specialized code for type arguments that are `string`. In fact, our formalization can be customized using different partitionings of the set of all closed types. For a partitioning to be valid regarding the translation, obvious requirements include that it must be complete (covering all closed types) and disjoint (no type belongs to more than one partitions). It must also provide constraint satisfaction, argument specialization, and type specialization judgments. Keen readers may have noticed that we explicitly specify constraints in BILC even though all type arguments have the same constraint `ref`. This allows us to accomodate easily other specialization policies.

A less obvious necessary condition is that it must be congruent with respect to the structure of the types. For example, for the actual implementation we considered generating specialized code for *all* value classes, thus avoiding any dictionary lookups at runtime, but this does breaks congruence: for example, `string` shares the same constraint as `object` but `Pair<string,string>` and `Pair<object,object>` would be distinct, if `Pair` is a two-parameter value class. Then a generic method `m<X>` that references `Pair<X,X>` cannot be specialized under the constraint $X \triangleright$ `ref` because no appropriate specialization of `Pair` exists.

Moving in the other direction, towards less specialization and more sharing, it is worth noting that types with different constraints can be shared if coercion operations such as boxing are employed.

Generic virtual methods Virtual methods which are themselves generic introduces a new level of expressivity in the system of polymorphism that supports a kind of "first-class" polymorphism [15]. The implementation of this in the presence of infinitary code-specialization requires JIT compilation, and when combined with dictionaries requires some new implementation techniques. In particular an appropriate dictionary cannot be generated from the caller's side. Our formal translation constructs type-reps of open descriptors inside generic virtual methods at run-time, to demonstrate the advantage of dictionary passing. Nonetheless, there are more efficient schemes for implementing generic virtual methods, *e.g.*, one may arrange to let the callee construct the dictionary and reuse it inside the method body. Although different method calls of the same instantiation would repeat the dictionary construction, the cost is still amortized inside a single method call.

Structural typing Our target language BILC essentially embeds in its type system the source types of BILG. This simplifies the formal type system and semantics, and abstracts away the complexities of inheritance and virtual method dispatch.

An alternative approach is to encode source types in a lower-level language possessing structural type equivalence; typically this language is based on a foundational calculus such as F_ω. Of course, modelling inheritance and virtual methods is not trivial, for example involving existentials and recursive types, features that would not be considered lightly by an implementer.

But dispensing with nominal typing brings benefits: more optimizations can be expressed, and choices of representation made more explicit. For example, given a generic class C`<X>` and an instantiation C`<string>` we might specialize the code to produce a class C`@string`, but also generate code for C`@ref<`$X \triangleright$`ref>` that can be used for any reference type instantiation. Polymorphic contexts would use the latter code, whilst the specialized code would be used when the instantiation is known to be `string`. This requires that C`@ref<string>` be type-equivalent to C`@string`.

Type-theoretic constructs such as $\times$ and μ make it possible to describe the types of dictionaries without the use of names for method dictionaries. For example:

- Suppose a generic method `m<X>` uses the open type `Vec<X>` and invokes two methods `n<X>` and `p<X>` which both in turn invoke `q<X>` which uses the type `Set<X>`. Then the type of the dictionary passed to `m<string>` can be expressed as `Rep(Vec<string>)` $\times$ (`Rep(Set<string>)` $\times$ `Rep(Set<string>)`).
- Cycles in dictionary types can be expressed using recursive types. For example, suppose `m<X>` uses type `Vec<X>` and invokes `n<X>` which in turn uses `Set<X>` and invokes `m<X>`. The type of the dictionary passed to `m<string>` is μD.`Rep(Vec<string>)` $\times$ (`Rep(Set<string>)` $\times D$).
- Polymorphic recursion requires the use of higher-kinded recursive types. For example, suppose `m<X>` uses `Vec<X>` and then invokes `m<Set<X>>`. The type of the dictionary passed to `m<string>` is ($\mu D.\lambda X$.`Rep(Vec<`X`>)` $\times$ D(`Set<`X`>`)) `string`.

Previous approaches to implementing polymorphism Traditionally, implementation techniques for parametric polymorphism have employed a uniform representation for values used in polymorphic contexts so that each polymorphic function is compiled to a single piece of native code. In general, this requires expensive boxing for primitive types such as `int` and `float`, and a great deal of research involves cunning ways of alleviating this cost by repositioning or removing altogether the box and unbox operations [16].

Furthermore, most research has concentrated on ML-like languages with purely static type systems in which source types do not affect evaluation. Interestingly, much recent work on the implementation of polymorphism has reintroduced types into evaluation ("intensional type analysis") so that polymorphic code can switch on the type at which it is used [9, 25]. The manipulation of types at runtime can be expensive, particularly the construction of types from type parameters that are not known statically. It is possible to avoid all runtime type construction by lifting type applications to top-level, which can be considered 'link-time' [24]. This technique is similar to our use of dictionaries which can be created just once.

Similar type dictionary techniques are employed by Viroli and Natali [27] in their implementation of generics for Java. In their source-to-source translation, a single non-generic class in the target is used for all instantiations of a generic class in the source (only reference instantiations are permitted). Dictionaries (which they call 'friend' types) are manipulated using reflection features of Java.

Code specialization (monomorphization) is rarely used as an implementation technique. The idea of specializing by representation (instead of source type) as used in the implementation of CLR generics has also been applied to whole program compilation for Standard ML [2].

6 Conclusion and Future Work

We have presented a formalization of the generics implementation for the .NET Common Language Runtime, combining code sharing, code specialization and efficient support for run-time types, using dictionaries. In particular, we presented a type-preserving and semantics-preserving translation from BILG, which is a miniature formalization of Generic IL, to BILC, which abstracts the extended Common Language Runtime for generics. This work not only serves as an application of advanced type theories to a widely used industrial product and helps understanding possible implementation techniques, but also yields interesting research results, such as name-based type-reps.

Potential future work includes extensions on other language features (*e.g.*, value types), further translation down to lower-level languages (*e.g.*, TAL [19]), a separate modelling of the generics implementation in a lower-level calculus with structural equivalence (*e.g.*, FLINT [26]), and formalizing Just-In-Time compilation to support laziness.

7 References

[1] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding parameterized types to Java. In *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, October 1997.

[2] P. N. Benton, A. J. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *3rd ACM SIGPLAN International Conference on Functional Programming*, September 1998.

[3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java

programming language. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, October 1998.

[4] R. Cartwright and G. L. Steele. Compatible genericity with run-time types for the Java programming language. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, October 1998.

[5] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *International Conference on Functional Programming*, pages 301–312. ACM Press, Sept. 1998.

[6] Ecma International. ECMA and ISO C# and Common Language Infrastructure standards. See `http://msdn.microsoft.com/net/ecma/`.

[7] A. Gordon and D. Syme. Typing a multi-language intermediate code. In *27th Annual ACM Symposium on Principles of Programming Languages*, January 2001.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (Second Edition)*. Addison-Wesley, 2000.

[9] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, Jan. 1995. ACM Press.

[10] A. Hejlsberg. The C♯ programming language. Invited talk at Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 2002.

[11] A. Hejlsberg and S. Wiltamuth. C# 2.0 language reference. See `http://msdn.microsoft.com/vcsharp/`.

[12] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1999.

[13] A. Jeffrey. Generic Java type inference is unsound. The Types Forum, December 2001.

[14] A. J. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation*. ACM, 2001.

[15] A. J. Kennedy and D. Syme. Transposing F to C#: Expressivity of parametric polymorphism in an object-oriented language. In *Concurrency and Computation: Practice and Experience*, to appear.

[16] X. Leroy. Unboxed objects and polymorphic typing. In *19th symposium Principles of Programming Languages*, pages 177–188. ACM Press, 1992.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (Second Edition)*. Addison-Wesley, 1999.

[18] Microsoft Corporation. The .NET Common Language Runtime. See `http://msdn.microsoft.com/net/`.

[19] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *25th ACM Symposium on Principles of Programming Languages*, pages 85–97. ACM Press, Jan. 1998.

[20] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for java. In *24th ACM Symposium on Principles of Programming Languages*, pages 132–145, January 1997.

[21] M. Odersky, E. Runne, and P. Wadler. Two ways to bake your Pizza – translating parameterised types into Java. Technical Report CIS-97-016, University of South Australia, 1997.

[22] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[23] K. Palacz and J. Vitek. Subtype tests in real time. In *European Conference on Object Oriented Programming*, 2003.

[24] B. Saha and Z. Shao. Optimal type lifting. In *Types in Compilation Workshop*, pages 156–177, Kyoto, Japan, March 1998.

[25] Z. Shao. Flexible representation analysis. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 85–98, Amsterdam, The Netherlands, June 1997.

[26] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[27] M. Viroli and A. Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, October 2000.

[28] D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .NET Common Language Runtime. Technical report, Microsoft Research, 2003. To appear.

[29] D. Yu, V. Trifonov, and Z. Shao. Type-preserving compilation of Featherweight IL. In *Proc. 2002 Workshop on Formal Techniques for Java-like Programs (FTfJP'02)*, June 2002.