Type-Preserving Compilation of Featherweight IL (Extended Abstract)

Dachuan Yu Valery Trifonov Zhong Shao
Department of Computer Science, Yale University
New Haven, CT 06520-8285, U.S.A.
{yu,trifonov,shao}@cs.yale.edu

Abstract

We present a type-preserving compilation of Featherweight IL. Featherweight IL is a significant subset of MS IL which models new features including value classes and their interaction with reference classes. Our translation makes use of a high-level intermediate language called Functional Featherweight IL. The target language LFLINT is a low-level language which is close to machine level implementations. During the compilation, we preserve and further identify the basic block structures of the program, and perform CPS and closure conversions. We use memory based fixpoint to handle mutually recursive classes at compile time. Standard linking techniques can be applied for separate compilation. A type-preservation theorem for the formal translation is presented. In the long run, our work aims at supporting certifying compilation of high-level class-based languages.

1 Introduction

The growing use of the Internet provides new possibilities as well as new challenges. Distributed applications, web-based services, and mobile code infrastructures pose various safety and security requirements. The interoperation of programs written in different languages and running on different platforms further complicates the situation. For example, it is usually desirable to run even untrusted code efficiently; strong guarantees are crucial for safety critical systems; and programs for special applications (*e.g.,* embedded systems) often face special requirements (*e.g.,* resource usage). The common practice in approaching these problems is to engage a safety and security mechanism based on type-safe execution [27, 29, 11, 19, 12, 4, 37], and more generally, to reason about program properties using types [13, 2, 32].

The JVM platform of Sun Microsystems and the .NET framework of Microsoft make use of verifiable bytecode languages, namely JVML [21] and MS IL [23, 33], to approach these problems. In these frameworks, software components distributed in the object-oriented intermediate languages are executed by the Java Virtual Machine or the Common Language Runtime. The bytecode contains type signatures and other symbolic information for verification purposes. The soundness of the underlying type system guarantees certain properties such as program safety. While JVML is more biased towards Java programs [8, 28, 3, 35], MS IL aims to support a wide variety of source languages and high-level constructs, so as to enable safe and proper interoperation and integration of software components and provide the foundation for developers to build various types of applications.

JVML and MS IL take important steps to overcome the challenges, but these intermediate languages are still very high-level. They require much further compilation and optimization to run efficiently on real hardware. Since compilers are huge pieces of software, it is at least suspectable that the safety and other guarantees provided by the verifiable high-level intermediate languages might be compromised. The potential unsoundness in the huge Trusted Computing Base (TCB) may undermine the safety of the entire system.

On the other hand, there has also been much recent work focusing on using type systems and logics to reason about program properties for low-level code [26, 25, 24, 1, 10]. This work aims at practical systems for executing (untrusted) code both safely and efficiently. Furthermore, type information can be useful for low-level optimizations [34, 6] and accurate garbage collection [5, 36].

The idea of type-preserving compilation is to try to eliminate the gap between high-level verifiable programs and low-level safe and efficient code. Type information is propagated through compilation and optimization passes. Verification or type checking is achievable on both the source and the target sides of the compilation. By doing this, a significant part of the compiler is removed from the TCB.

Our previous work [14, 18] developed encodings for many Java features in our intermediate language FLINT, based on F_ω. These encodings were implemented in a JVML compiler [15] which uses a high-level language, λJVM [17], as an intermediate representation between JVML and FLINT. λJVM has the same primitive instructions and types as JVML, but is easier to verify and more amenable to optimizations. Since MS IL serves better as a common high-level intermediate language than JVML, we extend the type-preserving compilation technique of our previous work to a significant subset of MS IL, called FIL. The compilation of FIL programs proceeds through a high-level intermediate language Functional FIL (FFIL – the λJVM counterpart) and targets a low-level intermediate language LFLINT. The novel aspects of this work are as follows.

- The target of the compilation is a language LFLINT, which is at a lower level than FLINT. While its type system is similar to that of FLINT, the code structure of LFLINT differs, in particular basic blocks are explicit in the syntax. This allows us to take advantage of the control flow and basic block structure already

present in FFIL code. LFLINT describes programs in continuation-passing, closure-passing style; by only refining the basic block graph of FFIL programs during the conversion to CPS, and by exploiting the fixed nesting structure of method code in closure conversion, we avoid performing the extra work involved in compiling through a language with F_ω-like control.

- MS IL provides support for a wide variety of programming languages through a number of features and constructs, some of which are not present in JVML. Our compilation scheme covers value classes, invoke-instance semantics, boxing and unboxing of objects between value types and the corresponding reference types, and managed pointers. Our techniques also extend to other features, including delegates [38].

- The target language LFLINT is at a sufficiently low level of abstraction, where we can describe mutually-recursive dependencies between code fragments directly as circular references in the code heap. This eliminates the necessity to perform linking of a set of compiled mutually recursive classes by a fixpoint operator as in our previous work. Separate compilation is beyond the scope of this paper; however, the notion of location used in our scheme can be extended to include "external references," which will enable the application of standard linking technnology for similar object file formats [20].

- We use a high-level functional IL (called FFIL) when compiling FIL. It makes data flow explicit, verification simple, and is well-suited for translation into lower-level representations like LFLINT. We present the type systems for both FFIL and LFLINT in the companion technical report [38]. Based on them, we formalize the type-preservation theorem of our translation.

Due to space constraints, we demonstrate the high-level idea with selected translation rules. The complete formal translation can be found in the companion technical report [38].

2 Featherweight IL

Gordon and Syme [7] introduced Baby IL (BIL) which models a significant subset of MS IL. To the authors' knowledge, it is so far the largest subset of MS IL for which a soundness theorem is proven. Our source language Featherweight IL (FIL) is closely related to BIL, so its type system is based on well-understood ideas. However, we have made some adaptations. The main difference is in the semantics of the `unbox` instruction. According to MS IL documents [23], unboxing "is not required to" make copies of objects. However, in C# unboxing does create an extra copy of the object: "following a boxing or unboxing operation, changes made to the unboxed struct are not reflected in the boxed struct" [22]. The `unbox` instruction of FIL simulates the latter effect.

The syntax of FIL is shown in Figure 1. Semantically, it is a stack-based language: Instructions (expressions) pop operands off the stack, and push results onto the stack. We omit the semantics of this language, since they are very similar to those defined by Gordon and Syme.

The types (T) include primitive (`Void` and `Int`), pointer ($T\&$), reference (`class` C), and value types (`value class` VC). The semantics of an object of a reference type is similar to that in Java and other object-oriented languages. The semantics of an object of a value type is similar to that of a C `struct`. Every value class has a corresponding implicit reference class. Objects of value types can be "boxed" into objects of reference types; boxed objects can be "unboxed" back into objects of value types.

$$
\begin{array}{lll}
(\textit{Type}) & T & ::= \texttt{Void} \mid \texttt{Int} \mid \texttt{class}\ C \\
 & & \quad \mid \texttt{value class}\ VC \mid T\& \\
(\textit{Class}) & C & ::= RC \mid VC \\
(\textit{RDec}) & RD & ::= \texttt{ref class}\ RC \lhd RC\ \{(T\ f)^*\ K\ M^*\} \\
(\textit{VDec}) & VD & ::= \texttt{val class}\ VC \lhd RC\ \{(T\ f)^*\ K\ M^*\} \\
(\textit{Ctor}) & K & ::= C{::}\texttt{Ctor}\ (T\ f)^* \\
(\textit{Meth}) & M & ::= T\ m\ (T\ f)^*\ \{e\} \\
(\textit{MDes}) & MD & ::= T\ C{::}m\ (T)^* \\
(\textit{Expr}) & e & ::= \texttt{ldc}\ i \mid \texttt{cond}\ e\ e\ e \mid \texttt{while}\ e\ e \mid \texttt{seq}\ e\ e \\
 & & \quad \mid \texttt{ldind}\ e \mid \texttt{stind}\ e\ e \mid \texttt{ldarga}\ j \mid \texttt{starg}\ e\ j \\
 & & \quad \mid \texttt{newobj}\ K\ e^* \mid \texttt{box}\ e\ VC \mid \texttt{unbox}\ e\ VC \\
 & & \quad \mid \texttt{ldflda}\ e\ T\ C :: f \mid \texttt{stfld}\ e\ e\ T\ C :: f \\
 & & \quad \mid \texttt{callvirt}\ e\ MD\ e^* \mid \texttt{callinst}\ e\ MD\ e^*
\end{array}
$$

Figure 1: Syntax of FIL.

A class declaration (RD or VD) contains the name of the new class and its super class, a sequence of field declarations, a constructor, and a sequence of method declarations. Value classes cannot be extended; they are "sealed". For simplicity we assume no field hiding, *i.e.,* all fields in a class are distinct from those in its superclass. We also assume that every class contains exactly one constructor, which simply assigns its arguments to the fields of the object being created. A method declaration consists of the return type, name, arguments, and body of the method. A method descriptor provides the signature of a method (*i.e.,* argument types and result), and also specifies the name of the defining class. This name is necessary in FIL (as well as in MS IL) due to the use of value classes: Since a value-class object contains only the values of the fields, there is no information associated with it about the methods of the class, hence a method invocation on a value class object must specify its defining class. In general these type annotations also serve as a "minimum type interface" for separate compilation.

Simple expressions include loading integer constants, branch, loop and sequencing. Pointer operations (`ldind`, `stind`) essentially model the managed pointers of MS IL. Arguments in the current stack frame can be loaded and stored using the corresponding instructions (`ldarga`, `starg`). Note that `ldarga` loads the *address* of the specified argument.

The remaining expressions manipulate objects. The instruction `newobj` is used for creating instances of both reference classes and value classes; the constructor associated with it contains information about the class. This instruction creates as well as initializes the object. There are boxing and unboxing instructions (`box`, `unbox`) to coerce objects between value classes and their corresponding implicit reference classes. These instructions change representations of objects and "make copies" of objects at run time. Instructions `ldflda` and `stfld` manipulates object fields; the type and the defining class of the fields are explicit in the instruc-

$$
\begin{array}{lll}
(\textit{Type}) & T & ::= \texttt{Void} \mid \texttt{Int} \mid (T_1 \ldots T_n) \to T \\
& & \quad \mid \texttt{class}\ C \mid \texttt{value class}\ VC \mid T\& \\
(\textit{Class}) & C & ::= RC \mid VC \\
(\textit{RDec}) & RD & ::= \texttt{ref class}\ RC \lhd RC\ \{(T\ f)^*\ K\ M^*\} \\
(\textit{VDec}) & VD & ::= \texttt{val class}\ VC \lhd RC\ \{(T\ f)^*\ K\ M^*\} \\
(\textit{Ctor}) & K & ::= C{::}\texttt{Ctor}\ (T\ f)^* \\
(\textit{Meth}) & M & ::= T\ m\ (T\ f)^*\ \{\mathbf{t}\} \\
(\textit{MDes}) & MD & ::= T\ C{::}m\ (T)^* \\
(\textit{Fun}) & fn & ::= \lambda(\mathbf{x}_1 : T_1 \ldots \mathbf{x}_n : T_n).\mathbf{t} \\
(\textit{Labl}) & L & ::= \ell \mid L.f \\
(\textit{Value}) & \mathbf{v} & ::= \mathbf{x} \mid () \mid i \mid L \\
(\textit{Term}) & \mathbf{t} & ::= \texttt{letrec}\ (\mathbf{x} : T = fn)^*.\mathbf{t} \mid \texttt{let}\ \mathbf{x} = \mathbf{p}; \mathbf{t} \\
& & \quad \mid \mathbf{p}; \mathbf{t} \mid \texttt{if}\ \mathbf{v}\ \texttt{then}\ \mathbf{t}\ \texttt{else}\ \mathbf{t} \mid \texttt{return} \\
& & \quad \mid \texttt{return}\ \mathbf{v} \mid \mathbf{v}\ (\mathbf{v}_1 \ldots \mathbf{v}_n) \\
(\textit{Prim}) & \mathbf{p} & ::= \texttt{cell}\ \mathbf{v} \mid !\ \mathbf{v} \mid \mathbf{v} := \mathbf{v} \mid \texttt{newobj}\ C\ \mathbf{v}^* \\
& & \quad \mid \texttt{callvirt}\ \mathbf{v}\ MD\ \mathbf{v}^* \mid \texttt{box}\ \mathbf{v}\ VC \\
& & \quad \mid \texttt{callinst}\ \mathbf{v}\ MD\ \mathbf{v}^* \mid \texttt{unbox}\ \mathbf{v}\ VC \\
& & \quad \mid \texttt{ldflda}\ \mathbf{v}\ T\ C :: f \mid \texttt{stfld}\ \mathbf{v}\ \mathbf{v}\ T\ C :: f
\end{array}
$$

Figure 2: Syntax of FFIL.

tions. Note that `ldflda` loads the *address* of a specified field.

Methods of reference classes and value classes are invoked using different instructions to achieve different semantics. The *invoke-virtual* semantics (using `callvirt`) is the traditional virtual method invocation semantics. The *invoke-instance* semantics (using `callinst`) is similar to the *invoke-special* semantics in Java: The method being invoked is resolved statically using the method descriptor specified in the instruction. It also takes a *self* pointer as one of the arguments. In MS IL, either semantics can be applied to both reference-class and value-class objects. This is also achievable in FIL and our compilation, because the method descriptor contains the name of the class and is sufficient to identify the nature of the object. However, here we use discriminating semantics for simplicity.

Lastly, an program in FIL consists of a set of (mutually recursive) class declarations and a main expression.

3 Functional Featherweight IL

Our first step in translating FIL is to a high-level intermediate language called Functional Featherweight IL (FFIL). FFIL is specially designed for FIL. It has the same primitive instructions and types. The difference is that FFIL replaces the implicit operand stack and untyped local variables with explicit data flow and fully-typed single-assignment bindings. In FFIL, functions are lightweight and can be implemented as jumps (tail calls). They are used in recursive function bindings (`letrec`). The translation from FIL to FFIL largely follows the ideas described in previous work [17]. The first step is to find the basic blocks in a method body. Next, data flow analysis must infer types for the stack and local variables at each program point. Lastly, we use symbolic execution to translate each block to a function.

There is a non-trivial difference due to the fact that `ldarga` loads the address of the argument. Thus arguments on the stack, whose address may be taken, have to be represented using reference cells. The simplest way is to use reference cells for all the arguments, which is clearly inefficient. Wrapping function arguments into mutable records may help, but for simplicity, we are using reference cells to simulate mutable records in LFLINT. Describing the analysis to determine which arguments need to be mutable is beyond the goals of this paper. Similar issues arise in the compilation of exceptions in Java programs [9, 17].

The syntax of FFIL is shown in Figure 2. Special reserved variables (*e.g.,* the self pointer `this`) are not shown. The types are similar to those of FIL, except that an arrow type for functions is added. Reference types (`class` C) are inhabited by reference class objects and boxed value class objects; value types (`value class` VC) are inhabited by value class objects. Class declarations, constructors, methods and method descriptors remain the same as FIL. The only major difference from FIL is that expressions are separated into three categories: values, terms, and primitives.

FFIL values include variables, unit, integers, and labels for modeling the dynamic semantics of reference cells. FFIL terms include binding forms which bind mutually recursive functions and results of primitive operation. If the result of a primitive is unused, the sequencing form may be used. Following BIL, conditional terms test integers against zero instead of using booleans. Loop expressions of FIL are compiled into basic blocks. Finally, the base cases of terms are the return and the function call.

FFIL primitives cover those FIL instructions which are not for control flow or stack manipulation. There are three instructions managing pointers. The instruction `cell` $\mathbf{v}$ creates a reference cell which holds the value $\mathbf{v}$. In FIL, pointers are introduced by instructions which load the address of arguments or fields. The FFIL instructions ! $\mathbf{v}$ and $\mathbf{v} := \mathbf{v}$ correspond to FIL instructions `ldind` and `stind`. Based on the assumption that a class defines exactly one constructor, we replace the constructor with the class name in the object creation instruction to get `newobj` $C\ \mathbf{v}^*$. The remaining object manipulating primitives, including method invocation, field operation, boxing and unboxing, are essentially the same as in FIL. Note that the stack manipulating primitives (`ldarga` and `starg`) are compiled away because arguments on the stack are turned into function arguments.

An FFIL program $(CT, \mathbf{t})$ consists of a fixed class table CT, mapping class names to declarations, and a main term $\mathbf{t}$. For ease of presentation we extend CT to include mappings for the implicit reference classes, whose names have the subscript $\Box$ after the names of the corresponding value classes. An implicit reference class $VC_\Box$ declares the same superclass, fields and constructor as the corresponding value class VC. However, the methods are different. A method M of class VC expects a self pointer to a value type object, while the corresponding method $M_\Box$ of class $VC_\Box$ expects a self pointer to a boxed object. Thus to reuse the method body of M, an unboxing operation is performed on the self pointer (`this`) at the beginning of the method body of $M_\Box$.

$$\frac{CT(VC) = \texttt{val class}\ VC \lhd RC\ \{(T\ f)^*\ K\ M^*\}}{CT(VC_\Box) = \texttt{ref class}\ VC_\Box \lhd RC\ \{(T\ f)^*\ K\ {M_\Box}^*\}}$$

$$
\begin{array}{rrl}
\text{Kinds} & \kappa & ::= \mathsf{Type} \mid \mathsf{R}^L \mid \kappa \Rightarrow \kappa' \mid \{(l :: \kappa)^*\} \\
\text{Types} & \tau & ::= \alpha \mid \lambda\alpha::\kappa.\,\tau \mid \tau\ \tau' \mid \{(l = \tau)^*\} \mid \tau \cdot l \\
 & & \quad \mid \mathbf{int} \mid \mathbf{ref} \mid \neg \mid \mathsf{Abs}^L \mid l : \tau\,;\tau' \mid \{\tau\} \\
 & & \quad \mid \forall\alpha::\kappa.\,\tau \mid \exists\alpha::\kappa.\,\tau \mid \mu\alpha::\kappa.\,\tau \\
\text{Selectors} & s & ::= \circ \mid s \cdot l \\
\text{Values} & v & ::= i \mid \ell \mid \{(l = v)^*\} \mid v\ [\tau] \\
 & & \quad \mid \langle \alpha::\kappa = \tau,\, v : \tau' \rangle \\
 & & \quad \mid \mathbf{fold}\ v\ \mathbf{as}\ \mu\alpha::\kappa.\,\tau\ \mathbf{at}\ \lambda\gamma::\kappa.\,s[\gamma] \\
\text{Primitives} & p & ::= x \mid v \mid p.l \mid \mathbf{cell}\ p \mid !\,p \\
 & & \quad \mid \mathbf{unfold}\ p\ \mathbf{as}\ \mu\alpha::\kappa.\,\tau\ \mathbf{at}\ \lambda\gamma::\kappa.\,s[\gamma] \\
\text{Operations} & q & ::= p \mid p := p' \\
\text{Computations} & e & ::= \mathbf{jmp}\ p\ p' \mid \mathbf{let}\ x : \tau = q\ \mathbf{in}\ e \\
 & & \quad \mid \mathbf{ifz}\ p\ \mathbf{then}\ e\ \mathbf{else}\ e' \\
 & & \quad \mid \mathbf{open}\ p\ \mathbf{as}\ \langle \alpha::\kappa,\, x : \tau \rangle\ \mathbf{in}\ e \\
\text{Heap blocks} & b & ::= v \mid \boldsymbol{\lambda} x : \tau.\,e \mid \boldsymbol{\Lambda}\alpha::\kappa.\,b \\
\text{Heap} & H & ::= \{(\ell : \tau \mapsto b)^*\}
\end{array}
$$

Figure 3: Syntax of LFLINT.

4 LFLINT

The syntax and derived forms of our low level common intermediate language are shown in Figure 3 and Figure 4. Except for the control flow constructs designed for continuation-passing, closure-passing representation, the constructs of this language are similar to those we have used in Java [15] and SML/NJ [31] compilers: existential and recursive types, row polymorphism, ordered records, reference cells, etc. The metavariable ℓ ranges over heap locations; the dynamic semantics fetches the corresponding values stored in the heap when appropriate.

The basic block structure is maintained by separating expressions into values, primitives, operations, computations, and heap blocks. Values v have no computational effects; type applications are considered values because they have no effect at run time under type-erasure semantics [25]. Primitives p may have side effects (*e.g.,* creation of reference cells), but they cannot interfere with each other. Operations q may have interfering side effects, hence their order is important. That is why a computation e, corresponding to a basic block, performs sequences of operations and control branches, ending with a jump (**jmp**) to a continuation. Heap blocks b are either data (values) or code (single-argument computations, written as lambda abstractions); they are stored at locations ℓ in the heap H. Finally, a program (H, e) consists of a heap and a root computation.

Most of the language constructs and the type language of LFLINT are explained in great detail in previous work [16]. We briefly explain some of the constructs here. $\neg\alpha$ is the type of a continuation which takes an argument of type α. Following Rémy [30] we introduce a kind of rows R^L, where L is the set of labels banned from the row. Abs^L is the empty row missing L, and $l : \tau\,;\tau'$ constructs a row from the element l of type τ and the row τ'. The record constructor $\{\cdot\}$ lifts a complete row type (with no labels missing) to kind Type. We refer interested readers to the companion technical report for the semantics of the computation language.

$$
\begin{array}{rl}
\mathbf{void} & \equiv \{\mathsf{Abs}^{\emptyset}\} \\
l_1 : \tau_1, \ldots, l_n : \tau_n & \equiv l_1 : \tau_1\,; \ldots l_n : \tau_n\,; \mathsf{Abs}^{\{l_1 \ldots l_n\}} \\
\tau_1 \times \cdots \times \tau_n & \equiv \{1 : \tau_1, \ldots, n : \tau_n\} \\
\mathbf{cont}\ \tau & \equiv \exists\alpha::\mathsf{Type}.\,\neg(\tau \times \alpha) \times \alpha \\
\langle v_1, \ldots, v_n \rangle & \equiv \{1 = v_1, \ldots, n = v_n\} \\
\boldsymbol{\lambda}\{x_1 : \tau_1, \ldots, x_n : \tau_n\}.\,e & \equiv \boldsymbol{\lambda} x : (\tau_1 \times \cdots \times \tau_n). \\
 & \quad \mathbf{let}\ x_1 = x.1 \\
 & \quad\ \mathbf{in} \ldots \mathbf{let}\ x_n = x.n\ \mathbf{in}\ e \\
 & \quad \text{where } x \notin \{x_1, \ldots, x_n\}
\end{array}
$$

Figure 4: Derived syntactic forms of LFLINT.

5 Translation

In the beginning of the compilation, we create a unique location ℓ for every class, and map class names to these locations in a mapping `Cmap`. This `Cmap` is propagated through the compilation so that classes can refer to each other using locations. Each FFIL class (including both reference class and value class) is compiled into a closed LFLINT record which refers to class information of other classes through locations, and provides its own class information. A reference class (including the implicit corresponding reference class of a value class) is compiled into a record of two elements; one for the method table, the other for the constructor. Besides these two, a value class is compiled into a record with extra components for boxing and unboxing operations.

The compilation produces a program consisting of a main computation and an initial heap in which the computation can be carried on. Based on the mapping `Cmap`, the initial heap maps locations to the corresponding LFLINT records.

5.1 Object encoding

In previous work [16] we defined the following encoding for Java objects. An object is essentially a record which contains both a vtable pointer and the fields. This record is folded with a recursive type for typing the self argument of its methods. It is further wrapped within an existential to provide subtyping (an object of a given class may be used as an object of a superclass), hiding some methods and fields. The following example shows an incomplete type for objects of a given class `C`. It indicates that class `C` contains methods $\mathsf{getx}, \mathsf{setx}$ and fields x, y. However, further methods and fields may be hidden in the tail, if the dynamic class of the object is a proper subclass of `C`.

$$
\begin{array}{l}
\exists\mathsf{tail}.\,\mu\mathsf{self}.\,\{\mathsf{vtab} : \{\mathsf{getx} : \mathsf{self} \rightarrow \mathbf{int}\,; \\
\qquad\qquad\qquad\qquad \mathsf{setx} : (\mathsf{self} \times \mathbf{int}) \rightarrow \mathbf{void}\,; \\
\qquad\qquad\qquad\qquad \mathsf{tail} \cdot \mathsf{m}\ \mathsf{self}\}\,; \\
\qquad\qquad\quad \mathsf{x} : \mathbf{int}\,;\ \mathsf{y} : \mathbf{int}\,;\ \mathsf{tail} \cdot \mathsf{f}\}
\end{array}
$$

This is not the whole story yet, because object types may refer to each other recursively due to recursive references among classes. Suppose the object types of all classes are put together in a type tuple *World*; then the type of an object of any class would be dependent on this *World*, which means *World* is recursive. Thus there would be another recursive binding outside the above incomplete type example.

Detailed explanations of this object encoding can be found in the previous work [16]. For the rest of this paper, it should suffice to know that based on this encoding, objects

are often built using a "fold-pack-fold" idiom. On the other hand, objects are often "de-constructed" using an "unfold-open-unfold" idiom before their members are accessed. The folding, unfolding, packing and opening operations ensure type-correctness, but they have no effect or overhead at run time in type-erasure semantics.

In the FFIL compilation of this paper, we use the same object encoding for reference type objects and boxed value type objects. However, the encoding of value type objects is much simpler. Based on the intention of value types, these objects simply consist of the values of the fields. Thus we encode such an object as a record which contains all its fields. In comparison with our encoding for reference type objects, the recursive type for self arguments and the vtable is clearly unnecessary for value types, since value type objects do not contain the method table. The existential package is not needed for value types either: On one hand, value classes are sealed and do not have subclasses; on the other hand, an object of a value type cannot be upcast to a superclass (it will have to be boxed before being upcast).

However, a value class VC may still have fields of other class types, which in turn may have fields of type VC. Hence value class types are translated as recursive records, and value type objects must be folded and unfolded properly.

Before we move on to explain the translations, we should note that it would be more efficient to represent objects by mutable records. However, in this paper mutable records are simulated using records of reference cells for simplicity.

Now we need to define a function mapping types from FFIL to LFLINT. Informally, if an FFIL expression exp has type T, then its translation in LFLINT should have type $\textsc{typ}(\mathsf{w}, T)$, in a context where w is a type record which contains all object types (*i.e.,* the *World*).

$$
\begin{array}{ll}
\textsc{typ}(\mathsf{w}, \texttt{Void}) & \equiv \textbf{void} \\
\textsc{typ}(\mathsf{w}, \texttt{Int}) & \equiv \textbf{int} \\
\textsc{typ}(\mathsf{w}, \texttt{class RC}) & \equiv \mathsf{w}{\cdot}\texttt{RC} \\
\textsc{typ}(\mathsf{w}, \texttt{class VC}) & \equiv \mathsf{w}{\cdot}\texttt{VC}_{\square} \\
\textsc{typ}(\mathsf{w}, \texttt{value class VC}) & \equiv \mathsf{w}{\cdot}\texttt{VC} \\
\textsc{typ}(\mathsf{w}, T\&) & \equiv \textbf{ref}\,\textsc{typ}(\mathsf{w}, T) \\
\textsc{typ}(\mathsf{w}, (T_1 \ldots T_n) \to T) & \equiv \\
\quad \neg(\textsc{typ}(\mathsf{w}, T_1) \times \cdots \times \textsc{typ}(\mathsf{w}, T_n) \times \textbf{cont}\,(\textsc{typ}(\mathsf{w}, T)))
\end{array}
$$

Due to space constraints, we omitted from this extended abstract some macros for defining object types.

5.2 Expression translation

The translation of FFIL expressions is separated into three categories: the translation of values, functions, and terms. Primitives are inlined into the let bindings for the translation. Selected translation rules are shown in Figure 5.

The value translation $\textsc{val}[\Gamma; \mathbf{v}]$ maps the FFIL value $\mathbf{v}$ to a LFLINT value, given a FFIL variable environment Γ. The rules for this translation, as well as the function translation FUN, are straightforward and omitted.

Term translations are formulated using the judgement $\textsc{exp}[\Gamma; \texttt{cty}; \texttt{Cmap}; H; \mathbf{t}] = (H', e)$. Γ is the FFIL type environment. The cast specification $\texttt{cty}$ is of the form $T_1 \Rightarrow T_2$, where T_1 is the type of the term $\mathbf{t}$ being translated, and T_2 is the type expected by the context of $\mathbf{t}$. Most translation rules just propagate this $\texttt{cty}$ argument; it is used non-trivially only when a value is returned (rule RETV). The environment $\texttt{Cmap}$ maps class names to unique locations in the heap; some of the translation rules rely on this mapping to generate references to other classes. The parameter H is a heap, *i.e.,* a mapping from labels to heap blocks. The final argument $\mathbf{t}$ is the term to be translated, The result of the translation is a pair of a new heap H', extending H with the newly generated heap blocks of code, and a root com-

$$\frac{\textsc{val}[\Gamma; \mathbf{v}] = v}{\begin{array}{l}\textsc{exp}[\Gamma; T_1 \Rightarrow T_2; \texttt{Cmap}; H; \texttt{return v}] = \\ \quad (H, \textsc{upcast}[T_1; T_2; v; x; \mathsf{comp}]) \\ \quad \text{where } \mathsf{comp} = \\ \quad\quad \textbf{open}\ \mathsf{retcont} \\ \quad\quad\quad \textbf{as}\ \langle \alpha{::}\mathsf{Type}, y : (\neg(\textsc{typ}(\mathsf{w}, T_2) \times \alpha)) \times \alpha \rangle \\ \quad\quad\quad \textbf{in jmp}\ (y.1)\ \langle x, y.2 \rangle\end{array}} \quad \text{(RETV)}$$

$$\frac{\begin{array}{l}\textsc{val}[\Gamma; \mathbf{v}] = v \\ \textsc{exp}[\Gamma, \mathbf{x} : T; \texttt{cty}; \texttt{Cmap}; H; \mathbf{t}] = (H', e)\end{array}}{\begin{array}{l}\textsc{exp}[\Gamma; \texttt{cty}; \texttt{Cmap}; H; \texttt{let x} = \texttt{ldflda v } T\ \texttt{RC :: f}; \mathbf{t}] = \\ \quad (H', \\ \quad\ \textbf{open}\ (\textbf{unfold}\ v\ \textbf{as}\ \mathit{World}\ \textbf{at}\ \lambda \mathsf{w}{::}\mathit{kcn}.\ \mathsf{w}{\cdot}\texttt{RC}) \\ \quad\quad \textbf{as}\ \langle \mathsf{tail}{::}\mathit{ktail}[\texttt{RC}], y : \mathit{SelfTy}[\texttt{RC}]\ \mathit{World}\ \mathsf{tail} \rangle \\ \quad\quad \textbf{in let}\ x : \textsc{typ}(\mathit{World}, T) = (\textbf{unfold}\ y).\mathsf{f}\ \textbf{in}\ e)\end{array}} \quad \text{(RLDFLDA)}$$

$$\frac{\begin{array}{l}\Gamma' = \Gamma, \mathbf{x} : T \qquad \textsc{typ}(\mathit{World}, T) = \tau \\ \Gamma' \vdash \mathbf{t} : T' \qquad \textsc{typ}(\mathit{World}, T') = \tau' \\ \textsc{exp}[\Gamma'; \texttt{cty}; \texttt{Cmap}; H; \mathbf{t}] = (H', e) \\ FV(e) - \{x\} = \{x_1, \ldots, x_m\} \\ \tau_{env} = \Gamma(\mathbf{x}_1) \times \cdots \times \Gamma(\mathbf{x}_n) \qquad \ell_K \notin \mathrm{dom}(H')\end{array}}{\begin{array}{l}\textsc{wrap}[\Gamma; \texttt{cty}; \texttt{Cmap}; H; (\texttt{let x} : T = \bullet; \mathbf{t}); \mathsf{comp}] = \\ \quad (H'[\ell_K : \neg(\tau \times \tau_{env}) \mapsto \boldsymbol{\lambda}\{x : \tau, x' : \tau_{env}\}. \\ \quad\quad\quad\quad \textbf{let}\ (x_j = x'.j)^{j \in \{1 \ldots m\}}\ \textbf{in}\ e], \\ \quad \textbf{let}\ k = \langle \alpha{::}\mathsf{Type} = \tau_{env}, \\ \quad\quad\quad \langle \ell_K, \langle x_1, \ldots, x_m \rangle \rangle : (\neg(\tau \times \alpha)) \times \alpha \rangle \\ \quad \textbf{in}\ \mathsf{comp})\end{array}} \quad \text{(WRAP)}$$

$$\frac{\begin{array}{l}\texttt{MD} = T\ \texttt{RC}'\ \mathsf{m}\ \{T_1 \ldots T_n\} \\ \left.\begin{array}{ll} \circ; \Gamma \vdash \mathbf{v} : \texttt{class RC} & \textsc{val}[\Gamma; \mathbf{v}] = v \\ \circ; \Gamma \vdash \mathbf{v}_i : T_i' & \textsc{val}[\Gamma; \mathbf{v}_i] = v_i \\ T_i' \subseteq T_i & \end{array}\right\}_{i \in \{1 \ldots n\}}\end{array}}{\begin{array}{l}\textsc{exp}[\Gamma; \texttt{cty}; \texttt{Cmap}; H; \\ \quad \texttt{let x} : T = \texttt{callvirt v MD } \mathbf{v}_1 \ldots \mathbf{v}_n; \mathbf{t}] = \\ \quad \textsc{wrap}[\Gamma; \texttt{cty}; \texttt{Cmap}; H; (\texttt{let x} : T = \bullet; \mathbf{t}); \mathsf{comp}_0] \\ \quad \text{where} \\ \quad\ \forall i \in \{1 \ldots n\}.\ \mathsf{comp}_{i-1} = \textsc{upcast}[T_i'; T_i; v_i; x_i; \mathsf{comp}_i], \\ \quad\ \mathsf{comp}_n = \textbf{open}\ (\textbf{unfold}\ v\ \textbf{as}\ \mathit{World}\ \textbf{at}\ \lambda \mathsf{w}{::}\mathit{kcn}.\ \mathsf{w}{\cdot}\texttt{RC}) \\ \quad\quad\quad\quad \textbf{as}\ \langle \mathsf{tail}{::}\mathit{ktail}[\texttt{RC}], y : \mathit{SelfTy}[\texttt{RC}]\ \mathit{World}\ \mathsf{tail} \rangle \\ \quad\quad\quad\quad \textbf{in jmp}\ ((\textbf{unfold}\ y).\texttt{vtab.m}) \\ \quad\quad\quad\quad\quad\quad \langle y, x_1, \ldots, x_n, k \rangle\end{array}} \quad \text{(CALLVIRT)}$$

$$\frac{\textsc{val}[\Gamma; \mathbf{v}] = v \qquad \ell_{\texttt{VC}} = \texttt{Cmap(VC)}}{\begin{array}{l}\textsc{exp}[\Gamma; \texttt{cty}; \texttt{Cmap}; H; \texttt{let x} : T = \texttt{box v VC}; \mathbf{t}] = \\ \quad \textsc{wrap}[\Gamma; \texttt{cty}; \texttt{Cmap}; H; (\texttt{let x} : T = \bullet; \mathbf{t}); \mathsf{comp}] \\ \quad \text{where } \mathsf{comp} = \textbf{jmp}\ (\ell_{\texttt{VC}}.\mathsf{box})\ \langle v, k \rangle\end{array}} \quad \text{(BOX)}$$

Figure 5: Selected translation rules.

putation e, whose evaluation in the context of H simulates the evaluation of $\mathtt{t}$.

The translations of terms except let bindings are relatively straightforward. The only case deserving special care is the translation of return instructions. FFIL uses `return v` to return from a method body. In the translation, before returning $\mathbf{v}$ we have to cast it up to the type expected by the context. Rule RETV uses an UPCAST macro to perform this. Essentially, UPCAST$[T_1; T_2; v; x; \mathsf{comp}]$ converts the value v of type TYP($World$, T_1) to a value of type TYP($World$, T_2), where T_1 is a subtype of T_2. Then it binds the result value to the specified variable x and continues with the rest of the computation specified by comp. After this is done in rule RETV, the computation continues by fetching the continuation closure from the special variable retcont, which is bound in the prologue of the method translation (to be explained in section 5.3).

The primitives of FFIL can be separated into two categories, based on whether or not they involve implicit control flow transfer. Simple primitives include those manipulating reference cells and object fields. As an example, rule RLD-FLDA uses the "unfold-open-unfold" idiom (mentioned in section 5.1) to get access to the field.

The translation of the remaining primitives introduce new blocks into the heap (H). We extract the commonalities into a macro as shown in rule WRAP. Informally speaking, WRAP$[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; (\mathtt{let\ x} : T = \bullet; \mathbf{t}); \mathsf{comp}]$ creates a new code block ℓ_K, which performs the computation of the body $\mathbf{t}$, and encloses the computation comp within a continuation binding of k, with the intention that comp will use k to jump to ℓ_K when it completes.

The remaining two selected rules use WRAP for the translation. Now that all the new blocks and control transfer are abstracted away, all we need to understand here is the translation of the primitives, *i.e.,* what is the last argument comp provided to the WRAP macro. The translation of virtual method invocation (CALLVIRT) uses the "unfold-open-unfold" idiom to manipulate the self pointer, and selects the corresponding method from the vtable. Before jumping to that method, the arguments have to be cast to the expected types. We chain up some UPCAST macros (which involve computations) to achieve that. The upcasts are type operations which have no run-time effect. The translation of the boxing primitive delegates the task to the boxing function found in the record generated for the corresponding class. We will explain the boxing function and the generation of the class location mapping `Cmap` in the coming sections. For now, it suffices to say that $\ell_{\mathtt{C}}.\mathsf{box}$ yields the location of code which converts an object of a value type to an object of the corresponding reference type.

We have sketched the expression translation. To state the type preservation theorem, we define the translation $\mathcal{E}nv[\Gamma]$ of a FFIL typing environment Γ to a LFLINT typing environment, by mapping each type T in the range of Γ to TYP($World$, T). By inspection of the definition of the type translation function TYP it is easy to show that $\mathcal{E}nv[\Gamma]$ is a well-formed environment, assuming that all class names occurring in the range of Γ are also in cn.

Theorem 1 (Type preservation) If

$\ell_{\mathtt{C}}$ are distinct locations, where $\mathtt{C}$ ranges over cn;
$\mathtt{Cmap} = (\mathtt{C} : \mathit{ClassF}[\mathtt{C}] \mapsto \ell_{\mathtt{C}})^{\mathtt{C} \in cn}$;
$\Sigma_0 = (\ell_{\mathtt{C}} : \mathit{ClassF}[\mathtt{C}])^{\mathtt{C} \in cn}$;
$T \subseteq T'$;
$\vdash_h H : \Sigma$;
$\circ; \Gamma \vdash_c \mathbf{t} : T$;
EXP$[\Gamma; T \Rightarrow T'; \mathtt{Cmap}; H; \mathbf{t}] = (H', e)$; and
$\vdash_h H' : \Sigma'$,
then $\Sigma' \subseteq_s \Sigma$ and
$\Phi; (\Sigma_0, \Sigma'); (\mathcal{E}nv[\Gamma], \mathsf{retcont} : \mathrm{TYP}(\mathit{World}, T)) \vdash_c e$.

5.3 Class encoding

Each reference class declaration is compiled into a record with two elements, which are the locations of the extensible class dictionary and the class constructor. The extensible dictionary is constructed by building a method table, which is a record, polymorphic over the tail of the self type, containing type applications of the locations of the class methods to the tail. Thus, given the empty tail or the specific tail of a subclass as a type argument, the dictionary acts as a specialized method table for the class itself or the corresponding subclass, respectively. The constructor in FFIL simply takes as arguments the initial valus of all the fields in the correct order. In our translation, we build a record of the vtable and all the fields, properly packed and folded.

As with expression translation, some rules need to refer to the result of translating other classes. We achieve this using a special set of locations which act as "external references." Whenever we need to refer to a class $\mathtt{C}$, we use a location $\ell_{\mathtt{C}}$; the mapping from class names to these locations is provided to the class translation as the map `Cmap`, and propagated to the expression translation. The translation of the program (section 5.4) performs the "linking" of classes by placing the class record for $\mathtt{C}$ at location $\ell_{\mathtt{C}}$.

During the compilation of a reference class declaration, new heap blocks are introduced. The class translation function takes a set of heap blocks as an argument (H), and extends it with its own blocks. An additional map Mmap in class translation provides directly the locations of the dictionary and methods of a class, which we use to compile an inherited method as simply the location of the corresponding method in the superclass, without any indirection.

The translation of implicit reference classes is no different than that of other reference classes. However, value classes are translated into records with four components: the locations of the sealed dictionary, the class constructor, and the boxing and unboxing code. Selected parts of the value class translation are shown in Figure 6. With the definitions of some macros for dictionary, constructor, and class types omitted, the translation rules may look mysterious. However, ignoring the types in these rules, it should suffice to demonstrate the basic ideas.

The value class translation (VCDEC) resorts to a dictionary translation $\mathtt{DICT}_v$ for constructing the dictionary. However, unlike for reference classes, the resulting dictionary is not polymorphic, because value classes are sealed. The location of the polymorphic dictionary of the corresponding boxed type class is obtained from Mmap. This dictionary is then instantiated (via type application) and passed to the boxing translation `BOX` as the vtable pointer. There are two other helper translations $\mathtt{NEW}_v$ and `UNBOX` which are used to produce the constructor and the unboxing operation. The values obtained from these compilations are placed at fresh

Class declaration translation:

$$\frac{\begin{array}{l}\texttt{DICT}_v[\texttt{VC};\texttt{Cmap};H] = (H', v_{dict})\\ \pi_1(\texttt{Mmap}(\texttt{VC}_\square)) = \ell_{rdict}\\ \{\ell_{dict}, \ell_{new}, \ell_{box}, \ell_{unbox}\} \cap \mathrm{dom}(H') = \emptyset\\ H'' = H'[\\ \quad \ell_{dict}\colon \mathit{Dict}[\texttt{VC}]\ \texttt{w}\ (\mathit{SelfTy}[\texttt{VC}]\ \texttt{w}) \mapsto v_{dict},\\ \quad \ell_{new} : (\mathit{Ctor}[\texttt{VC}]\ \texttt{w}) \mapsto \texttt{NEW}_v[\texttt{VC}],\\ \quad \ell_{box} : (\mathit{Box}[\texttt{VC}]\ \texttt{w}) \mapsto \texttt{BOX}[\texttt{VC}; \ell_{rdict}\ [\mathit{Empty}[\texttt{VC}_\square]]],\\ \quad \ell_{unbox} : (\mathit{Unbox}[\texttt{VC}]\ \texttt{w}) \mapsto \texttt{UNBOX}[\texttt{VC}]\\ \quad \text{where } \texttt{w} = \mathit{World}\end{array}}{\begin{array}{l}\texttt{CDEC}_v[\texttt{Cmap};\texttt{Mmap};H;\texttt{VC}] = (H'',\\ \quad \{\texttt{dict} = \ell_{dict}, \texttt{new} = \ell_{new}, \texttt{box} = \ell_{box}, \texttt{unbox} = \ell_{unbox}\})\end{array}} \text{(VCDEC)}$$

Dictionary construction:

$$\frac{\begin{array}{l}\mathit{methvec}(\texttt{VC}) = [(\texttt{m}_1, \mathit{Sig}_1) \ldots (\texttt{m}_n, \mathit{Sig}_m)]\\ \forall i \in \{1 \ldots n\} \quad \texttt{METH}_v[\texttt{VC};\texttt{m}_i;H_{i-1};\texttt{Cmap}] = (H_i, v_i)\\ \{\ell_1, \ldots, \ell_n\} \cap \mathrm{dom}(H_n) = \emptyset\\ \tau_j = \mathit{Ty}[(\mathit{SelfTy}[\texttt{VC}]\ \mathit{World}); \mathit{World}; \mathit{Sig}_j]\\ H' = H_n[(\ell_j : \tau_j \mapsto v_j)^{j \in \{1 \ldots n\}}]\end{array}}{\texttt{DICT}_v[H_0;\texttt{Cmap};\texttt{VC}] = (H', \{\texttt{m}_1 = \ell_1, \ldots, \texttt{m}_n = \ell_n\})} \text{(VDICT)}$$

Method code:

$$\frac{\begin{array}{l}\mathit{CT}(\texttt{VC}) = \texttt{val class VC} \lhd \texttt{RC}\ \{\ldots \texttt{K}\ \texttt{M}_1 \ldots \texttt{M}_n\}\\ \exists j : \texttt{M}_j = T\ \texttt{m}\ (T_1\ \texttt{x}_1 \ldots T_m\ \texttt{x}_m)\ \{\texttt{t}\}\\ \Gamma = \texttt{x}_1 : T_1, \ldots, \texttt{x}_m : T_m, \texttt{this} : (\texttt{value class VC})\\ \Gamma \vdash \texttt{t} : T' \qquad T' \subseteq T\\ \textsc{exp}[\Gamma; T' \Rightarrow T; \texttt{Cmap}; H; \texttt{t}] = (H', e)\end{array}}{\begin{array}{l}\texttt{METH}_v[\texttt{VC};\texttt{m};H;\texttt{Cmap}] = (H',\\ \quad \lambda\{\texttt{this} : \mathit{SelfTy}[\texttt{VC}]\ \mathit{World},\\ \qquad x_1 : \textsc{typ}(\mathit{World}, T_1), \ldots, x_m : \textsc{typ}(\mathit{World}, T_m),\\ \qquad \texttt{retcont} : \textbf{cont}\ (\textsc{typ}(\mathit{World}, \tau))\}.\ e\end{array}} \text{(VMETH)}$$

Boxing:

$$\frac{\mathit{fields}(\texttt{VC}) = [(\texttt{f}_1, T_1) \ldots (\texttt{f}_m, T_m)]}{\begin{array}{l}\texttt{BOX}[\texttt{VC};\texttt{vtab}] =\\ \quad \lambda\{\texttt{vobj} : \textbf{ref}\ (\mathit{ObjTy}[\texttt{VC}]\ \mathit{World}),\ k : \textbf{cont}\ \mathit{World} \cdot \texttt{VC}_\square\}.\\ \quad \textbf{let}\ x_0 = \textbf{unfold}\ (!\ \texttt{vobj})\\ \quad \textbf{in let}\ x_1 = \textbf{cell}\ (!\ (x_0.\texttt{f}_1))\ \textbf{in}\ \ldots\\ \qquad \textbf{let}\ x_n = \textbf{cell}\ (!\ (x_0.\texttt{f}_n))\\ \qquad \textbf{in let}\ x = \textbf{fold}\ \{\texttt{vtab} = \texttt{vtab}, \texttt{f}_1 = x_1, \ldots, \texttt{f}_n = x_n\}\\ \qquad\qquad \textbf{as}\ \mathit{SelfTy}[\texttt{VC}_\square]\ \mathit{World}\ \mathit{Empty}[\texttt{VC}_\square]\\ \qquad\quad \textbf{in open}\ k\ \textbf{as}\ \langle \alpha :: \mathsf{Type}, y : \neg(\mathit{World} \cdot \texttt{VC}_\square \times \alpha) \times \alpha \rangle\\ \qquad\qquad \textbf{in jmp}\ (y.1)\ \langle \textsc{pack}[\texttt{VC}_\square; \mathit{Empty}[\texttt{VC}_\square]; x], y.2 \rangle\end{array}} \text{(CBOX)}$$

Figure 6: Translations of value class definitions.

locations in the heap. Finally, a record of these locations is formed as the class record.

The dictionary construction (VDICT) chains the compilation of each of the methods. The method translation (VMETH) abstracts the computation e, obtained by translating the method body, over the self pointer, the method arguments specified by the signature, and the continuation closure retcont (the special variable that the translation of a return instruction refers to). Also recall that the expression translation casts the result to the expected type.

The code generated for boxing (CBOX) coerces objects from value types to reference types by copying the object's

$$\frac{\begin{array}{l}\texttt{Cmap} = (\texttt{C} : \mathit{ClassF}[\texttt{C}] \mapsto \ell_{\texttt{C}})^{\texttt{C} \in cn} \quad \text{where } \ell_{\texttt{C}} \text{ are all distinct}\\ H_0 = () \qquad \texttt{Mmap}_0 = () \qquad \circ;\circ \vdash \texttt{t} : T\\ \forall\, \texttt{RC}_i \in rcn \qquad i \in \{1 \ldots n\} \qquad \texttt{RC}_i \text{ are ordered}\\ \quad \texttt{CDEC}[\texttt{Cmap};\texttt{Mmap}_{i-1};H_{i-1};\texttt{RC}_i] = (\texttt{Mmap}_i, H_i', v_i)\\ \quad H_i = H_i'[\ell_{\texttt{RC}_i} : \mathit{ClassF}[\texttt{RC}_i] \mapsto v_i]\\ \forall\, \texttt{VC}_j \in vcn \qquad j \in \{1 \ldots m\}\\ \quad \texttt{CDEC}_v[\texttt{Cmap};\texttt{Mmap}_n;H_{n+j-1};\texttt{VC}_j] = (H_{n+j}', v_j)\\ \quad H_{n+j} = H_{n+j}'[\ell_{\texttt{VC}_j} : \mathit{ClassF}[\texttt{VC}_j] \mapsto v_j]\end{array}}{\texttt{PROG}[cn;\texttt{t}] = \textsc{exp}[\circ; T \Rightarrow T; \texttt{Cmap}; H_{n+m}; \texttt{t}]}$$

Figure 7: Program translation.

fields and adding a vtable entry. The PACK macro, used also in the upcast and creation of new objects, not only packs the **fold**ed object record x but also folds it to $\textsc{typ}(\mathit{World}, \texttt{VC}_\square)$.

5.4 Program translation

Figure 7 gives the translation of FFIL programs. First, we create the mapping Cmap, assigning unique locations to all class names in cn. Then we translate all the reference classes. As usual, this also includes the implicit classes $\texttt{VC}_\square$. These classes have to be ordered according to the class hierarchy so that no class is compiled before its direct superclass. The compilations of these classes are chained, accumulating results via the method map and code heap arguments (Mmap and H). The next step is to compile all the value classes. Again, these compilations are chained to produce the final set of basic blocks H_{n+m}. Lastly, we translate the main body of the program, using the trivial cast type as the cty to indicate that the final result need not be cast.

Separate compilation is sacrificed when we use this memory-based fixpoint scheme to translate class declarations. However, it is well known how to write a linker for this kind of programs. Each class declaration would be compiled separately using external references to refer to other classes. These external references are resolved by the linker and replaced with the actual locations at link time. That is to say, standard linking techniques [20] can be applied here to achieve separate compilation.

Theorem 2 (Type-preserving program compilation) If $\circ;\circ \vdash_c \texttt{t} : T$, $\texttt{PROG}[cn;\texttt{t}] = (H, e)$, and $\vdash_h H : \Sigma$, then $\circ;\Sigma;(\texttt{retcont} : \textsc{typ}(\mathit{World}, T)) \vdash_c e$.

6 Conclusion

We have presented a type-preserving compilation of Featherweight IL. Most interesting new features of MS IL, including value classes, invoke-instance semantics, boxing and unboxing operations, and managed pointers, are supported. In comparison with our previous work that targeted FLINT, using LFLINT as our target language enables us to take a more direct compilation path. Basic block structures, identified in the intermediate compilation step, are preserved and further divided in the final form, which is very close to the machine level. We use heap locations to compile efficiently mutually recursive references between classes, and inherited methods. Standard linking techniques can be applied to obtain separate compilation. We intend to apply this formal translation to an MS IL compiler in the near future.

References

[1] A. W. Appel. Foundational proof-carrying code. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[2] H. P. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Sci. Pub. B.V., 1999.

[3] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Proc. 1998 ACM SIGPLAN Int'l Conf. on Functional Prog.*, pages 129–140, 1998.

[4] K. Crary and S. Weirich. Resource bound certification. In *Proc. 27th ACM Symp. on Principles of Prog. Lang.*, pages 184–198. ACM Press, 2000.

[5] A. Diwan. Compiler support for garbage collection in a statically typed language. In *Proc. ACM SIGPLAN '92 Conf. on Prog. Lang. Design and Implementation*, New York, June 1992. ACM Press.

[6] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 106–117, Montreal, Canada, June 1998.

[7] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *Proc. 28th ACM Symp. on Principles of Prog. Lang.*, pages 248–260, London, United Kingdom, January 2001.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[9] M. Gupta, J. Choi, and M. Hind. Optimizing Java programs in the presence of exceptions. In *Proc. 14th European Conf. on Object-Oriented Program*, June 2000.

[10] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. Seventeenth Annual IEEE Symposium on Logic In Computer Science (LICS'02)*, July 2002.

[11] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, New York, NY, USA, January 1998. ACM Press.

[12] M. Hennessy and J. Riely. Type-safe execution of mobile agents in anonymous networks. In *Secure Internet Programming: Proceedings of 4th Workshop on Mobile Object Systems*, pages 95–115. Springer-Verlag, 1998.

[13] W. A. Howard. The formulae-as-types notion of constructions. In *To H.B.Curry: Essays on Computational Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[14] C. League, Z. Shao, and V. Trifonov. Representing Java classes in a typed intermediate language. In *Proc. 1999 ACM SIGPLAN Int'l Conf. on Functional Prog.*, pages 183–196. ACM Press, Sept. 1999.

[15] C. League, Z. Shao, and V. Trifonov. Precision in practice: A type-preserving Java compiler. Technical Report YALEU/DCS/TR-1223, March 2002.

[16] C. League, Z. Shao, and V. Trifonov. Type-preserving compilation of Featherweight Java. *ACM Trans. on Programming Languages and Systems*, (to appear) 2002.

[17] C. League, V. Trifonov, and Z. Shao. Functional Java bytecode. In *Proc. 2001 Workshop on Intermediate Representation Engineering for the Java Virtual Machine at the 5th World Multi-conference on Systemics, Cybernetics, and Informatics*, July 2001.

[18] C. League, V. Trifonov, and Z. Shao. Type-preserving compilation of Featherweight Java. In *Foundations of Object-Oriented Languages (FOOL8)*, London, January 2001.

[19] X. Leroy and F. Rouaix. Security properties of typed applets. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 391–403, New York, NY, January 1998. ACM Press.

[20] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers, 2000.

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

[22] Microsoft Corp., *et al.* C# language specification. Drafts of the ECMA TC39/TG3 standardization process. `http://msdn.microsoft.com/net/ecma/`, 2001.

[23] Microsoft Corp., *et al.* Common language infrastructure. Drafts of the ECMA TC39/TG3 standardization process. `http://msdn.microsoft.com/net/ecma/`, 2001.

[24] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: a realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, May 1999.

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

[26] G. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Prog. Lang.*, pages 106–119, New York, Jan. 1997. ACM Press.

[27] G. Necula and P. Lee. Safe kernel extensions without runtime checking. In *Proc. 2nd USENIX Symp. on Operating System Design and Impl.*, pages 229–243, 1996.

[28] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symp. on Principles of Prog. Lang.*, pages 146–159, 1997. `http://cm.bell-labs.com/cm/cs/who/wadler/pizza/Docs/`.

[29] J. Palsberg and P. Ørbæk. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, November 1997.

[30] D. Rémy. Syntactic theories and the algebra of record terms. Technical Report 1869, INRIA, 1993.

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

[32] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Proc. 29th ACM Symp. on Principles of Prog. Lang.*, pages 217–232, Portland, OR, Jan. 2002. ACM Press.

[33] D. Syme. ILX: extending the .NET Common IL for functional language interoperability. In *Proc. BABEL Workshop on Multi-Language Infrastructure and Interoperability*. ACM, 2001.

[34] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. 1996 ACM Conf. on Prog. Lang. Design and Impl.*, pages 181–192. ACM Press, 1996.

[35] R. Tolksdorf. Programming languages for the Java Virtual Machine. `http://flp.cs.tu-berlin.de/~tolk/vmlanguages.html`.

[36] A. Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, pages 1–11. ACM Press, June 1994.

[37] D. Walker. A type system for expressive security policies. In *Proc. 27th ACM Symp. on Principles of Prog. Lang.*, pages 254–267, 2000.

[38] D. Yu, V. Trifonov, and Z. Shao. Type-preserving compilation of Featherweight IL. Technical Report YALEU/DCS/TR-1228, Dept. of Computer Science, Yale Univeristy, New Haven, CT, May 2002. `http://flint.cs.yale.edu/`.