Yale University
Department of Computer Science

Type-Preserving Compilation of Featherweight IL

Dachuan Yu Valery Trifonov Zhong Shao
Yale University

YALEU/DCS/TR-1228
May 22, 2002

This research is based on work supported in part by DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-9901011, and NSF ITR grant CCR-0081590. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

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.

Type-Preserving Compilation of Featherweight IL

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

Technical Report YALEU/DCS/TR-1228
May 22, 2002

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 [28, 30, 12, 20, 13, 5, 38], and more generally, to reason about program properties using types [14, 3, 33].

The JVM platform of Sun Microsystems and the .NET framework of Microsoft make use of verifiable bytecode languages, namely JVML [22] and MS IL [24, 34], 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 [9, 29, 4, 36], 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 [27, 26, 25, 2, 11]. This work aims at practical systems for executing (untrusted) code both safely and efficiently. Furthermore, type information can be useful for low-level optimizations [35, 7] and accurate garbage collection [6, 37].

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 [15, 19] developed encodings for many Java features in our intermediate language FLINT, based on F_ω. These encodings were implemented in a JVML compiler [16] which uses a high-level language, λJVM [18], 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,

This research is based on work supported in part by DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-9901011, and NSF ITR grant CCR-0081590. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

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. We also briefly explain how our techniques extend to support other features, including delegates.

- 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 [21].

- 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 appendix. Based on them, we formalize the type-preservation theorem of our translation.

Modeling every aspect of a type-preserving compiler is a challenging and length task. Even for Featherweight IL (FIL), the formal translation rules can be overwhelming, with CPS and closure conversion involved and the basic block structures maintained. Using memory based fixpoint as an effective implementation of the program linking process further complicates the translation. On one hand, we hope to provide a proper level of abstraction using macros for comprehensibility. On the other hand, we omitted some feature, namely dynamic casting, from previous work. The support for casting poses no new challenges in our case. The previous solution can be adopted directly.

The remainder of this paper is structured as follows. Section 2 introduces our source language and explains its new features. Section 3 describes the Functional FIL (FFIL) language; we only briefly describe the translation from FIL to FFIL, referring interested readers to our previous work [18] on more details. Section 4 gives our target language. In section 5, we dive into the details of every aspect of the compilation, and formalize the type-preservation theorem. Some extensions are described in section 6. The semantics of FFIL and LFLINT can be found in the appendix.

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

Figure 1: Syntax of FIL.

2 Featherweight IL

Gordon and Syme [8] 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 [24], 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" [23]. 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.

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 instructions. 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. As we will briefly explain in section 6, 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. There are no static methods in this languages, but supporting them is straightforward.

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 [18]. 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 [10, 18].

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

Figure 2: Syntax of FFIL.

Our previous work used *set types* to unify object types. Based on the assumption that the class hierarchy is known statically, we replace a set type with the least common ancestor of its element types in FFIL. Note that this assumption does not prevent separate compilation of code.

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` `v` cre-

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

Figure 3: Syntax of LFLINT.

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

Figure 4: Derived syntactic forms of LFLINT.

ates 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 $M_\Box$.

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

$$
\frac{M = T\ m\ (T\ f)^*\ \{\mathbf{t}\} \qquad M \text{ is a method of class } VC}{M_\Box = T\ m\ (T\ f)^*\ \{\texttt{let this} = \texttt{unbox this}\ VC; \mathbf{t}\}}
$$

Please refer to appendix A for the typing rules and an operational semantics of FFIL.

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 [16] and SML/NJ [32] 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 [26]. 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 [17]. 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 [31] 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 use iso-recursive types at higher kinds. The syntax for folding and unfolding them deserves some explanation. Suppose we wish to encode the following mutually recursive type abbreviations:

$$
\begin{array}{l}
\mathbf{type}\ \mathsf{even} = \mathbf{maybe}\ \{\mathsf{hd} : \mathbf{int}, \mathsf{tl} : \mathsf{odd}\} \\
\mathbf{type}\ \mathsf{odd} = \{\mathsf{hd} : \mathbf{int}, \mathsf{tl} : \mathsf{even}\}
\end{array}
$$

The solution is expressed as the fixpoint over a tuple:

$$
\begin{array}{l}
t = \mu\alpha{::}\{\mathsf{even} :: \mathsf{Type}, \mathsf{odd} :: \mathsf{Type}\}. \\
\quad \{\mathsf{even} = \mathbf{maybe}\ \{\mathsf{hd} : \mathbf{int}, \mathsf{tl} : \alpha{\cdot}\mathsf{odd}\}, \\
\quad\ \mathsf{odd} = \{\mathsf{hd} : \mathbf{int}, \mathsf{tl} : \alpha{\cdot}\mathsf{even}\}\}
\end{array}
$$

Now, the two recursive types are expressed as $t{\cdot}\mathsf{even}$ and $t{\cdot}\mathsf{odd}$. There are, however, no type equivalence rules for reducing $t{\cdot}\mathsf{odd}$; a term having this type must first be coerced to a type in which t is *unfolded*. We allow unfolding of recursive types within a tuple by specifying a *selector* after

the **at** keyword. Selectors are syntactically restricted to a (possibly empty) sequence of labeled selections from a tuple. The syntax $\lambda\gamma{::}\kappa.\, s[\gamma]$ allows identity ($\lambda\gamma{::}\kappa.\,\gamma$), one selection ($\lambda\gamma{::}\kappa.\,\gamma{\cdot}l_1$), two selections ($\lambda\gamma{::}\kappa.\,\gamma{\cdot}l_1{\cdot}l_2$), and so on. The formation rules (rule 93 and 100 in appendix B.1) further restrict the selectors to have a result of kind Type. Thus, if e has type $t{\cdot}\mathsf{odd}$, then the expression

$$\textbf{unfold}\ e\ \textbf{as}\ t\ \textbf{at}\ \lambda\gamma{::}\{\mathsf{even} :: \mathsf{Type},\ \mathsf{odd} :: \mathsf{Type}\}.\,\gamma{\cdot}\mathsf{odd}$$

has type $\{\mathsf{hd} : \textbf{int}, \mathsf{tl} : t{\cdot}\mathsf{even}\}$. For recursive types of kind Type, the only allowed selector is identity, so we omit it. We sometimes also omit the **as** annotation where it can be readily inferred.

The typing judgments are decidable, and the type system is sound with respect to a structured operational semantics. This target language also enjoys a type erasure property: type manipulations (*e.g.*, type abstraction, fold, pack, and open) can be erased before runtime without affecting the result. We refer interested readers to appendix B for more details.

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 [17] 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 getx, 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`.

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

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.

$$\mathit{fieldvec}(\mathtt{Obj}) = [(\mathtt{vtab}, vt)]$$

$$\frac{CT(\mathtt{C}) = \mathtt{ref/val\ class\ C} \lhd \mathtt{RC}\ \{T_1\ \mathsf{f}_1; \ldots T_m\ \mathsf{f}_m;\ \mathsf{K}\ \mathsf{M}^*\}}{\mathit{fieldvec}(\mathtt{C}) = \mathit{fieldvec}(\mathtt{RC}) +\!\!+ [(\mathsf{f}_1, T_1)\ldots(\mathsf{f}_m, T_m)]}$$

$$\frac{\mathit{fieldvec}(\mathtt{C}) = [(\mathtt{vtab}, vt)\ (\mathsf{f}_1, T_1)\ldots(\mathsf{f}_m, T_m)]}{\mathit{fieldvec}_v(\mathtt{C}) = [(\mathsf{f}_1, T_1)\ldots(\mathsf{f}_m, T_m)]}$$

$$\mathit{methvec}(\mathtt{Obj}) = []$$

$$\frac{CT(\mathtt{C}) = \mathtt{ref/val\ class\ C} \lhd \mathtt{RC}\ \{\ldots\ \mathsf{K}\ \mathsf{M}_1\ldots\mathsf{M}_m\}}{\mathit{methvec}(\mathtt{C}) = \mathit{methvec}(\mathtt{RC}) +\!\!+ \mathit{addmeth}(\mathtt{RC}, [\mathsf{M}_1\ldots\mathsf{M}_m])}$$

$$\frac{(\mathsf{m}, \ldots) \in \mathit{methvec}(\mathtt{RC})}{\begin{array}{l}\mathit{addmeth}(\mathtt{RC}, [T\ m\ (T_1\ f_1\ldots T_n\ f_n)\ \{\ldots\}\mathsf{M}_2\ldots\mathsf{M}_m]) =\\ \quad \mathit{addmeth}(\mathtt{RC}, [\mathsf{M}_2\ldots\mathsf{M}_m])\end{array}}$$

$$\frac{(\mathsf{m}, \ldots) \notin \mathit{methvec}(\mathtt{RC})}{\begin{array}{l}\mathit{addmeth}(\mathtt{RC}, [T\ m\ (T_1\ f_1\ldots T_n\ f_n)\ \{\ldots\}\mathsf{M}_2\ldots\mathsf{M}_m]) =\\ \quad [(\mathsf{m}, T_1\ldots T_n\to T)] +\!\!+ \mathit{addmeth}(\mathtt{RC}, [\mathsf{M}_2\ldots\mathsf{M}_m])\end{array}}$$

$$\mathit{addmeth}(\mathtt{RC}, []) = []$$

$$\frac{\begin{array}{l}CT(\mathtt{C}) = \mathtt{ref/val\ class\ C} \lhd \mathtt{RC}\ \{\ldots\ \mathsf{K}\ \mathsf{M}_1\ldots\mathsf{M}_m\}\\ \exists j : \mathsf{M}_j = T\ \mathsf{m}\ (T_1\ \mathbf{x}_1\ldots T_m\ \mathbf{x}_m)\ \{\mathbf{t}\}\end{array}}{\mathit{defcls}(\mathtt{C}, \mathsf{m}) = \mathtt{C}}$$

$$\frac{\begin{array}{l}CT(\mathtt{C}) = \mathtt{ref/val\ class\ C} \lhd \mathtt{RC}\ \{\ldots\ \mathsf{K}\ \mathsf{M}_1\ldots\mathsf{M}_m\}\\ \mathsf{m}\ \text{not defined in}\ \mathsf{M}_1\ldots\mathsf{M}_n\end{array}}{\mathit{defcls}(\mathtt{C}, \mathsf{m}) = \mathit{defcls}(\mathtt{RC}, \mathsf{m})}$$

Figure 5: Field and method layouts for object type, defining class for methods.

Detailed explanations of this object encoding can be found in the previous work [17]. 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 formal definitions, 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 define some functions or macros which govern the layout of fields and methods in object types in Figure 5. Square brackets $[\cdot]$ denote sequences. The sequence $s_1 +\!\!+ s_2$ is the concatenation of sequences s_1 and s_2. The domain of a sequence of pairs $\mathrm{dom}(s)$ is a set consisting of the first elements of each pair in s. The functions *fieldvec* and $\mathit{fieldvec}_v$ maps a class name C to a sequence of tuples of the form (f, T), indicating a field of type T named f. The difference between these two functions lies in the first element of the result sequence. *fieldvec* generates a special entry at the beginning of a sequence as a placeholder for the vtable; while $\mathit{fieldvec}_v$ does not generate this entry. The definitions are easy to understand, keeping in mind that the fields of a class are assumed to be distinct from those of its super classes.

The function *methvec* maps a class name C to a sequence of tuples of the form $(\mathsf{m}, \mathit{Sig})$, indicating a method named m with signature Sig. It makes use of the helper function *addmeth* which makes sure that overriding methods do not create new slots.

The last function *defcls* in Figure 5 returns the defining class of a method m. If a method m of a class C is inherited from a superclass RC, then *defcls* returns RC as the defining class.

Let *rcn* denote the set of reference class names in the program of interest, including Obj and implicit ones. Let *vcn* denote the set of value class names in the program of interest. Let *cn* denote the set of all class names ($\mathit{cn} = \mathit{rcn} \cup \mathit{vcn}$). By convention, RC and $\mathtt{VC}_\square$ belongs to *rcn*, VC belongs to *vcn*, and C belongs to *cn*. We abbreviate the kind of a tuple of all object types as *kcn*. The tuple of row kinds for reference class RC is abbreviated $\mathit{ktail}[\mathtt{RC}]$. It models the kind of a type tuple which contains entries for methods and fields defined between class RC and a subclass. We also abbreviate a "value" version of the tuple of row kinds of class C (either value class or reference class) as $\mathit{ktail}_v[\mathtt{C}]$ where there is no entry for the vtable in the fields. Since value classes are sealed, kind $\mathit{ktail}_v[\mathtt{VC}]$ is only inhabited by a "empty tail type" (it is later defined as $\mathit{Empty}[\mathtt{VC}]$) for any value class VC.

$$
\begin{array}{ll}
\mathit{kcn} & \equiv \{(\mathsf{C}{::}\mathsf{Type})^{\mathsf{C} \in \mathit{cn}}\} \\
\mathit{ktail}[\mathtt{RC}] & \equiv \{\mathsf{m}{::}\mathsf{Type} \Rightarrow \mathsf{R}^{\mathrm{dom}(\mathit{methvec}(\mathtt{RC}))},\ \mathsf{f}{::}\mathsf{R}^{\mathrm{dom}(\mathit{fieldvec}(\mathtt{RC}))}\} \\
\mathit{ktail}_v[\mathtt{C}] & \equiv \{\mathsf{m}{::}\mathsf{Type} \Rightarrow \mathsf{R}^{\mathrm{dom}(\mathit{methvec}(\mathtt{C}))}, \mathsf{f}{::}\mathsf{R}^{\mathrm{dom}(\mathit{fieldvec}_v(\mathtt{C}))}\}
\end{array}
$$

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* that we described at the beginning of this section).

$$
\begin{array}{ll}
\textsc{typ}(\mathsf{w}, \mathtt{Void}) & \equiv \mathbf{void} \\
\textsc{typ}(\mathsf{w}, \mathtt{Int}) & \equiv \mathbf{int} \\
\textsc{typ}(\mathsf{w}, \mathtt{class\ RC}) & \equiv \mathsf{w}{\cdot}\mathtt{RC} \\
\textsc{typ}(\mathsf{w}, \mathtt{class\ VC}) & \equiv \mathsf{w}{\cdot}\mathtt{VC}_\square \\
\textsc{typ}(\mathsf{w}, \mathtt{value\ class\ VC}) & \equiv \mathsf{w}{\cdot}\mathtt{VC} \\
\textsc{typ}(\mathsf{w}, T\&) & \equiv \mathbf{ref}\ \textsc{typ}(\mathsf{w}, T) \\
\textsc{typ}(\mathsf{w}, (T_1 \ldots T_n) \to T) & \equiv \\
\multicolumn{2}{r}{\neg(\textsc{typ}(\mathsf{w}, T_1) \times \cdots \times \textsc{typ}(\mathsf{w}, T_n) \times \mathbf{cont}\,(\textsc{typ}(\mathsf{w}, T)))}
\end{array}
$$

In figure 6 we define *Rows*, a type operator that produces rows containing the fields and methods introduced *between* two classes in a subclass relationship. Intuitively, $\mathit{Rows}[\mathtt{RC}, \mathtt{RC}']$ includes fields and methods in class RC but *not* in its ancestor class RC′. The other operator Rows_v is used to model similar ideas for value types. These two operators expect two arguments: w, the tuple containing object types for all classes; and tail, a tuple of rows containing members of subclasses. The difference between these two operators lies in the kind of the second argument tail.

Explaining any one of these two macros should suffice. $\mathit{Rows}[\mathtt{RC}, \mathtt{RC}']$ is defined by three cases. First, if RC and RC′ are the same class, then the result is just the tail—those members in subclasses of RC. Second, if RC is Obj (the root of the class hierarchy) and RC′ is the special symbol $\top$ then the result is the members declared in RC together with the tail. In this particular case, Obj declares no members, thus it's only the tail. Treating $\top$ as the trivial super class of Obj permits more uniform specifications in general when Obj contains members of its own. Finally, in the inductive case (where RC is a subclass of RC′) we look to RC's direct super class to take out the *new* members of RC, and append them to the members obtained by recursively using the same *Rows* macro. Conveniently, the tail argument is used to append new members onto the record.

The new fields in RC are precisely those listed in the declaration of RC; we fetch their types using the type translation macro and append tail·f. The new methods in RC are found using *addmeth*, and their type signatures are translated accordingly into types of code which takes records as arguments in continuation passing style. The first element of the record argument is the self pointer, the last argument is the continuation closure, and all the arguments declared in the method signature come in between.

Several variants of the object type for any given reference type are defined using the *Rows* operator. $\mathit{Empty}[\mathtt{RC}]$ denotes the tuple of empty field and method rows of kind $\mathit{ktail}[\mathtt{RC}]$. $\mathit{ObjRcd}[\mathtt{RC}]$ assembles the rows into records, leaving the subclass rows and self type open. $\mathit{SelfTy}[\mathtt{RC}]$ closes self using recursive types, and $\mathit{ObjTy}[\mathtt{RC}]$ hides the subclass rows with an existential. All of these variants remain abstracted over *World* which contains the types of other objects. Similar type macros are defined for objects of value types. Due to the simple encoding of these objects, their types are much simpler. Finally, we define the *World* as a recursive record of all object types.

5.2 Expression translation

Before we look at the expression translation, we introduce some term macros in Figure 7. The PACK macro is used for packaging objects of reference type. It packs and folds

$$Rows[\mathtt{RC}, \mathtt{RC}] = \lambda \mathsf{w}{::}kcn.\ \lambda \mathsf{tail}{::}ktail[\mathtt{RC}].\ \mathsf{tail}$$

$$Rows[\mathtt{Obj}, \top] = \lambda \mathsf{w}{::}kcn.\ \lambda \mathsf{tail}{::}ktail[\mathtt{Obj}].\ \mathsf{tail}$$

$$\frac{\begin{array}{l} CT(\mathtt{RC}) = \mathtt{ref\ class\ RC} \lhd \mathtt{RC}_1\ \{T_1\ \mathsf{f}_1 \ldots T_n\ \mathsf{f}_n\ \mathsf{K}\ \mathsf{M}_1 \ldots \mathsf{M}_m\} \\ addmeth(\mathtt{RC}_1, \mathsf{M}_1 \ldots \mathsf{M}_m) = [(\mathsf{m}_1, Sig_1) \ldots (\mathsf{m}_m, Sig_m)] \\ Rows[\mathtt{RC}_1, \mathtt{RC}'] = \tau \end{array}}{\begin{array}{l} Rows[\mathtt{RC}, \mathtt{RC}'] = \lambda \mathsf{w}{::}kcn.\ \lambda \mathsf{tail}{::}ktail[\mathtt{RC}]. \\ \quad \tau\ \mathsf{w}\ \{\mathsf{m} = \lambda \mathsf{self}{::}\mathsf{Type}.\ (\mathsf{m}_1 : Ty[\mathsf{self}; \mathsf{w}; Sig_1]\ ; \ldots \\ \qquad\qquad \mathsf{m}_m : Ty[\mathsf{self}; \mathsf{w}; Sig_m]\ ; \mathsf{tail}{\cdot}\mathsf{m}) \\ \qquad \mathsf{f} = (\mathsf{f}_1 : (\mathbf{ref}\ \mathrm{TYP}(\mathsf{w}, T_1))\ ; \ldots \\ \qquad\quad \mathsf{f}_n : (\mathbf{ref}\ \mathrm{TYP}(\mathsf{w}, T_n))\ ; \mathsf{tail}{\cdot}\mathsf{f})\} \end{array}}$$

$$Rows_v[\mathtt{C}, \mathtt{C}] = \lambda \mathsf{w}{::}kcn.\ \lambda \mathsf{tail}{::}ktail_v[\mathtt{C}].\ \mathsf{tail}$$

$$Rows_v[\mathtt{Obj}, \top] = \lambda \mathsf{w}{::}kcn.\ \lambda \mathsf{tail}{::}ktail_v[\mathtt{Obj}].\ \mathsf{tail}$$

$$\frac{\begin{array}{l} CT(\mathtt{C}) = \mathtt{ref/val\ class\ C} \lhd \mathtt{RC}\ \{T_1\ \mathsf{f}_1 \ldots T_n\ \mathsf{f}_n\ \mathsf{K}\ \mathsf{M}_1 \ldots \mathsf{M}_m\} \\ addmeth(\mathtt{RC}, \mathsf{M}_1 \ldots \mathsf{M}_m) = [(\mathsf{m}_1, Sig_1) \ldots (\mathsf{m}_m, Sig_m)] \\ Rows_v[\mathtt{RC}, \mathtt{RC}'] = \tau \end{array}}{\begin{array}{l} Rows_v[\mathtt{C}, \mathtt{RC}'] = \lambda \mathsf{w}{::}kcn.\ \lambda \mathsf{tail}{::}ktail_v[\mathtt{C}]. \\ \quad \tau\ \mathsf{w}\ \{\mathsf{m} = \lambda \mathsf{self}{::}\mathsf{Type}.\ (\mathsf{m}_1 : Ty[\mathsf{self}; \mathsf{w}; Sig_1]\ ; \ldots \\ \qquad\qquad \mathsf{m}_m : Ty[\mathsf{self}; \mathsf{w}; Sig_m]\ ; \mathsf{tail}{\cdot}\mathsf{m}) \\ \qquad \mathsf{f} = (\mathsf{f}_1 : (\mathbf{ref}\ \mathrm{TYP}(\mathsf{w}, T_1))\ ; \ldots \\ \qquad\quad \mathsf{f}_n : (\mathbf{ref}\ \mathrm{TYP}(\mathsf{w}, T_n))\ ; \mathsf{tail}{\cdot}\mathsf{f})\} \end{array}}$$

$$\begin{array}{l} Ty[\mathsf{self}; \mathsf{w}; T_1 \ldots T_n \rightarrow T] = \\ \neg(\mathsf{self} \times \mathrm{TYP}(\mathsf{w}, T_1) \times \cdots \times \mathrm{TYP}(\mathsf{w}, T_n) \times \mathbf{cont}\ \mathrm{TYP}(\mathsf{w}, T)) \end{array}$$

$$\begin{array}{lll} Empty[\mathtt{RC}] & \equiv \{\mathsf{m} = \lambda \mathsf{self}{::}\mathsf{Type}.\ \mathsf{Abs}^{\mathrm{dom}(methvec(\mathtt{RC}))}, & \\ & \quad \mathsf{f} = \mathsf{Abs}^{\mathrm{dom}(fieldvec(\mathtt{RC}))}\} & \\ ObjRcd[\mathtt{RC}] & \equiv \lambda \mathsf{w}{::}kcn.\ \lambda \mathsf{tail}{::}ktail[\mathtt{RC}].\ \lambda \mathsf{self}{::}\mathsf{Type}. & \\ & \quad \{\mathtt{vtab} : \{(Rows[\mathtt{RC}, \top]\ \mathsf{w}\ \mathsf{tail}).\mathsf{m}\ \mathsf{self}\}\ ; & \\ & \quad (Rows[\mathtt{RC}, \top]\ \mathsf{w}\ \mathsf{tail}).\mathsf{f}\} & \\ SelfTy[\mathtt{RC}] & \equiv \lambda \mathsf{w}{::}kcn.\ \lambda \mathsf{tail}{::}ktail[\mathtt{RC}].\ \mu \mathsf{self}{::}\mathsf{Type}. & \\ & \quad ObjRcd[\mathtt{RC}]\ \mathsf{w}\ \mathsf{tail}\ \mathsf{self} & \\ ObjTy[\mathtt{RC}] & \equiv \lambda \mathsf{w}{::}kcn.\ \exists \mathsf{tail}{::}ktail[\mathtt{RC}].\ SelfTy[\mathtt{RC}]\ \mathsf{w}\ \mathsf{tail} & \\ & & \\ Empty[\mathtt{VC}] & \equiv \{\mathsf{m} = \lambda \mathsf{self}{::}\mathsf{Type}.\ \mathsf{Abs}^{\mathrm{dom}(methvec(\mathtt{VC}))}, & \\ & \quad \mathsf{f} = \mathsf{Abs}^{\mathrm{dom}(fieldvec_v(\mathtt{VC}))}\} & \\ ObjRcd[\mathtt{VC}] & \equiv \lambda \mathsf{w}{::}kcn. & \\ & \quad \{(Rows_v[\mathtt{VC}, \mathtt{Obj}]\ \mathsf{w}\ Empty[\mathtt{VC}]).\mathsf{f}\} & \\ SelfTy[\mathtt{VC}] & \equiv ObjRcd[\mathtt{VC}] & \\ ObjTy[\mathtt{VC}] & \equiv SelfTy[\mathtt{VC}] & \\ & & \\ World & \equiv \mu \mathsf{w}{::}kcn.\ \{ & \\ & \quad (\mathtt{C} = ObjTy[\mathtt{C}]\ \mathsf{w})^{\mathtt{C} \in cn} & \\ & \} & \end{array}$$

Figure 6: Macros for object types.

an open-self value into a complete object type in mutual fixpoint with all others. Supposing that tail is some row tuple of kind $ktail[\mathtt{RC}]$ and v has type $(SelfTy[\mathtt{RC}]\ \mathsf{w}\ \mathsf{tail})$, the result value $\mathrm{PACK}[\mathtt{RC}; \mathsf{tail}; v]$ has type $\mathsf{w}{\cdot}\mathtt{RC}$. The $\mathrm{UPCASTRC}[\mathtt{RC}; \mathtt{RC}'; v; x; \mathsf{comp}]$ macro converts the value v of type $\mathsf{w}{\cdot}\mathtt{RC}$ to a value of type $\mathsf{w}{\cdot}\mathtt{RC}'$, where $\mathtt{RC}'$ is a superclass of $\mathtt{RC}$. It unfolds and opens the value and repackages it. After that, it binds the result value to the specified variable x and continues with the rest of the computation specified by comp. UPCAST is a generalized upcasting macro defined using UPCASTRC. It works on all types in FFIL.

$$\mathrm{PACK}[\mathtt{RC}; \mathsf{tail}; v] = \mathbf{fold}\ \langle \mathsf{tail}' {::} ktail[\mathtt{RC}] = \mathsf{tail},\ v : SelfTy[\mathtt{RC}]\ \ World\ \mathsf{tail}' \rangle\ \mathbf{as}\ World\ \mathbf{at}\ \lambda \alpha {::} kcn.\ \alpha{\cdot}\mathtt{RC} \quad \text{(PACK)}$$

$$\begin{array}{l} \mathrm{UPCASTRC}[\mathtt{RC}; \mathtt{RC}'; v; x; \mathsf{comp}] = \\ \quad \mathbf{open}\ (\mathbf{unfold}\ v\ \mathbf{as}\ World\ \mathbf{at}\ \lambda \mathsf{w}{::}kcn.\ \mathsf{w}{\cdot}\mathtt{RC}) \\ \qquad \mathbf{as}\ \langle \mathsf{tail}{::}ktail[\mathtt{RC}],\ y : SelfTy[\mathtt{RC}]\ \ World\ \mathsf{tail} \rangle \\ \qquad \mathbf{in\ let}\ x = \mathrm{PACK}[\mathtt{RC}'; Rows[\mathtt{RC}, \mathtt{RC}']\ \ World\ \mathsf{tail}; y] \\ \qquad\quad \mathbf{in}\ \mathsf{comp} \end{array} \quad \text{(UPC)}$$

$$\mathrm{UPCAST}[\mathbf{class}\ \mathtt{RC}; \mathbf{class}\ \mathtt{RC}'; v; x; \mathsf{comp}] = \mathrm{UPCASTRC}[\mathtt{RC}; \mathtt{RC}'; v; x; \mathsf{comp}] \quad \text{(RC)}$$

$$\mathrm{UPCAST}[\mathbf{class}\ \mathtt{VC}; \mathbf{class}\ \mathtt{RC}'; v; x; \mathsf{comp}] = \mathrm{UPCASTRC}[\mathtt{VC}_\Box; \mathtt{RC}'; v; x; \mathsf{comp}] \quad \text{(VC)}$$

$$\mathrm{UPCAST}[T; T; v; x; \mathsf{comp}] = \mathbf{let}\ x = v\ \mathbf{in}\ \mathsf{comp} \quad \text{(OTHER)}$$

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

Figure 7: Macros: pack, upcast and wrapper.

The WRAP macro is a bit tricky. It helps the translation of those primitive bindings which contain implicit control flow transfer. These computations need to be translated into multiple basic blocks. WRAP extracts the commonalities among these translations. Regardless of what the primitive is (c.f. the hole in the let binding), these bindings create an extra block labeled ℓ_K in the set of heap blocks. Suppose the primitive is translated into a computation block comp, then this extra block labeled ℓ_K serves as the continuation of comp. This continuation is built by setting up the variable environment properly, and using the expression translation function recursively to translate the subterm $\mathbf{t}$. At the end of the computation comp, control flow is transferred to ℓ_K. The closure bound to k is going to be used at the end of comp to connect the control flow. It is worth emphasizing that this WRAP is a *macro* to be inlined in the translation of let bindings of certain primitives. It is *not* a function because we are relying on the fact that comp is going to use the variable k properly.

Now we are ready to examine the translation of FFIL expressions. It is separated into three categories: the translation of values, lambda abstractions (functions), and terms. Primitives are inlined into the let bindings for the translation.

Figure 8 contains the translation of values, lambda ab-

$$\textsc{val}[\Gamma; \mathbf{x}] = x \qquad \text{(VAR)}$$

$$\textsc{val}[\Gamma; ()] = \langle\rangle \qquad \text{(UNIT)}$$

$$\textsc{val}[\Gamma; \hat{i}] = i \qquad \text{(INT)}$$

$$\frac{\begin{array}{l}\textsc{typ}(\mathit{World}, T_i) = \tau_i \\ \Gamma' = \Gamma, \mathbf{x}_1 : T_1 \ldots, \mathbf{x}_n : T_n \\ \textsc{exp}[\Gamma'; \mathtt{cty}; \mathtt{Cmap}; H; \mathbf{t}] = (H', e)\end{array}}{\begin{array}{l}\textsc{fun}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \lambda(\mathbf{x}_1 : T_1 \ldots \mathbf{x}_n : T_n).\mathbf{t}] = \\ \quad (H', \boldsymbol{\lambda}\{x_1 : \tau_1, \ \ldots, \ x_n : \tau_n\}.\, e)\end{array}} \qquad \text{(FUN)}$$

$$\frac{\begin{array}{l}\{(\ell_i)^{i\in\{1\ldots n\}}\} \cap \mathrm{dom}(H) = \emptyset \qquad \ell_i \text{ distinct} \\ H_0 = H \qquad \Gamma' = \Gamma, \mathbf{x}_1 : T_1 \ldots, \mathbf{x}_n : T_n \\ \forall i \in \{1 \ldots n\} \\ \quad \textsc{fun}[\Gamma'; \mathtt{cty}; \mathtt{Cmap}; H_{i-1}; fn_i] = (H_i, v_i) \\ \quad v_i' = v_i[\ell_1/x_1 \ldots \ell_n/x_n] \qquad \textsc{typ}(\mathit{World}, T_i) = \tau_i \\ \textsc{exp}[\Gamma'; \mathtt{cty}; \mathtt{Cmap}; H_n[(\ell_i : \tau_i \mapsto v_i')^{i\in\{1\ldots n\}}]; \mathbf{t}] \\ \quad = (H', e)\end{array}}{\begin{array}{l}\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \mathtt{letrec}\ (\mathbf{x}_i : T_i = fn_i)^{i\in\{1\ldots n\}}.\mathbf{t}] \\ \quad = (H', e[(\ell_i/x_i)^{i\in\{1\ldots n\}}])\end{array}} \qquad \text{(LETREC)}$$

$$\frac{\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \mathtt{let}\ _ = \mathbf{p}; \mathbf{t}] = (H', e)}{\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; (\mathbf{p}; \mathbf{t})] = (H', e)} \qquad \text{(SEQ)}$$

$$\frac{\begin{array}{l}\textsc{val}[\Gamma; \mathbf{v}] = v \\ \textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \mathbf{t}_1] = (H_1, e_1) \\ \textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H_1; \mathbf{t}_2] = (H_2, e_2)\end{array}}{\begin{array}{l}\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \mathtt{if}\ \mathbf{v}\ \mathtt{then}\ \mathbf{t}_1\ \mathtt{else}\ \mathbf{t}_2] = \\ \quad (H_2, \mathbf{ifz}\ v\ \mathbf{then}\ e_2\ \mathbf{else}\ e_1)\end{array}} \qquad \text{(IF)}$$

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

$$\frac{\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \mathtt{return}\ ()] = (H', e)}{\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \mathtt{return}] = (H', e)} \qquad \text{(RET)}$$

$$\frac{\begin{array}{c}\textsc{val}[\Gamma; \mathbf{v}] = v \\ \textsc{val}[\Gamma; \mathbf{v}_i] = v_i \quad \forall i \in \{1 \ldots n\}\end{array}}{\begin{array}{l}\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \mathbf{v}\ (\mathbf{v}_1 \ldots \mathbf{v}_n)] = \\ \quad (H, \mathbf{jmp}\ v\ \langle v_1, \ \ldots, \ v_n\rangle)\end{array}} \qquad \text{(APP)}$$

Figure 8: Syntax-directed translation of FFIL expressions.

stractions, and all terms except let bindings. Value translation VAL is easy. It translates FFIL values into LFLINT values. The special value L is only used in the dynamic semantics. They do not appear in either source programs or our translation.

Function translation FUN and term translation EXP are mutually recursive. There is only one rule for function translation. It translates the subterm using EXP and builds up the corresponding LFLINT abstraction.

Term translations are formulated using the judgement $\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \mathbf{t}] = (H', e)$. Γ is the FFIL type environment. The cast specification `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 `cty` argument; it is used nontrivially only when a value is returned (rule RETV). The environment `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 computation e, whose evaluation in the context of H simulates the evaluation of $\mathbf{t}$.

Recursive function bindings in FFIL are handled with rule LETREC. It applies the lambda abstraction translation on each of the bindings. These translations are chained so that the heap H is updated properly. Lastly it creates new heap blocks in H for the recursive functions, and translates the subterm $\mathbf{t}$.

Sequencing is a special case of let bindings (`let`). The translations for conditional and application are not surprising. The rule for return instructions deserves some explanation. FFIL uses `return v` to return from a method body. In the translation, before returning the value $\mathbf{v}$, we have to cast it up to the type expected by the context. Rule RETV uses the UPCAST macro to perform this. After this is done, 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).

Figure 9 contains the translation of simple primitive operations inlined in let bindings. The primitives are simple in the sense that they do not contain implicit control flow transfer.

Primitives manipulating reference cells are straightforward. Getting and putting fields on objects of reference types (RLDFLDA and RSTFLD) use the "unfold-open-unfold" idiom we talked about in section 5.1 to manipulate objects before selecting the fields. Getting and putting fields on objects of value types (VLDFLDA and VSTFLD) are not surprising, either.

Figure 10 contains the translation of the remaining primitive operations inlined in let bindings. These primitives involve further control flow transfer. The compilation cases shown here introduce new blocks into the heap (H). The commonalities here are abstracted into the macro WRAP. Recall that $\textsc{wrap}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; (\mathtt{let}\ \mathbf{x}{:}T = \bullet; \mathbf{t}); \mathsf{comp}]$ creates a new code block ℓ_K which performs the computation of the let binding body $\mathbf{t}$, and it surrounds the computation `comp` with a continuation binding of k with the intention that `comp` uses k to jump to the new code block ℓ_K when it completes.

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.

Object creations for reference classes and value classes are handled with the same rule (NEW). It eventually refers to the class's constructor exported by the run-time class using the location provided in the `Cmap` record. Before that happens, the arguments provided to the constructor has to be cast up to the expected types. We chain up the UPCAST macro (which involves computations) to achieve that. Recall

$$\frac{\textsc{val}[\Gamma;\mathbf{v}] = v \quad \textsc{exp}[\Gamma, \mathbf{x}:T; \mathtt{cty}; \mathtt{Cmap}; H; \mathbf{t}] = (H', e)}{\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \mathtt{let}\ \mathbf{x}:T = \mathtt{cell}\ \mathbf{v}; \mathbf{t}] = (H', \mathbf{let}\ x : \textsc{typ}(\mathit{World}, T) = \mathbf{cell}\ v\ \mathbf{in}\ e)} \quad \text{(ADDR)}$$

$$\frac{\textsc{val}[\Gamma;\mathbf{v}] = v \quad \textsc{exp}[\Gamma, \mathbf{x}:T; \mathtt{cty}; \mathtt{Cmap}; H; \mathbf{t}] = (H', e)}{\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \mathtt{let}\ \mathbf{x}:T =! \mathbf{v}; \mathbf{t}] = (H', \mathbf{let}\ x : \textsc{typ}(\mathit{World}, T) = ! v\ \mathbf{in}\ e)} \quad \text{(DEREF)}$$

$$\frac{\textsc{val}[\Gamma;\mathbf{v}_1] = v_1 \quad \textsc{val}[\Gamma;\mathbf{v}_2] = v_2 \quad \textsc{exp}[\Gamma, \mathbf{x}:T; \mathtt{cty}; \mathtt{Cmap}; H; \mathbf{t}] = (H', e)}{\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \mathtt{let}\ \mathbf{x}:T = (\mathbf{v}_1 := \mathbf{v}_2); \mathbf{t}] = (H', \mathbf{let}\ x : \textsc{typ}(\mathit{World}, T) = (v_1 := v_2)\ \mathbf{in}\ e)} \quad \text{(ASSIGN)}$$

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

$$\frac{\textsc{val}[\Gamma;\mathbf{v}] = v \quad \textsc{exp}[\Gamma, \mathbf{x}:T; \mathtt{cty}; \mathtt{Cmap}; H; \mathbf{t}] = (H', e)}{\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \mathtt{let}\ \mathbf{x} = \mathtt{ldflda}\ \mathbf{v}\ T\ \mathtt{VC} :: \mathsf{f}; \mathbf{t}] = (H', \mathbf{let}\ x = (\mathbf{unfold}\ (!\, v)).\mathsf{f}\ \mathbf{in}\ e)} \quad \text{(VLDFLDA)}$$

$$\frac{\textsc{val}[\Gamma;\mathbf{v}_1] = v_1 \quad \textsc{val}[\Gamma;\mathbf{v}_2] = v_2 \quad \textsc{exp}[\Gamma, \mathbf{x}:\mathtt{Void}; \mathtt{cty}; \mathtt{Cmap}; H; \mathbf{t}] = (H', e)}{\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \mathtt{let}\ \mathbf{x} = \mathtt{stfld}\ \mathbf{v}_1\ \mathbf{v}_2\ T\ \mathtt{RC} :: \mathsf{f}; \mathbf{t}] = (H', \mathbf{open}\ (\mathbf{unfold}\ v_1\ \mathbf{as}\ \mathit{World}\ \mathbf{at}\ \lambda \mathsf{w}{::}\mathit{kcn}.\, \mathsf{w}{\cdot}\mathtt{RC})\ \mathbf{as}\ \langle \mathsf{tail}{::}\mathit{ktail}[\mathtt{RC}], y : \mathit{SelfTy}[\mathtt{RC}]\ \mathit{World}\ \mathsf{tail}\rangle\ \mathbf{in}\ \mathbf{let}\ x = (((\mathbf{unfold}\ y).\mathsf{f}) := v_2)\ \mathbf{in}\ e)} \quad \text{(RSTFLD)}$$

$$\frac{\textsc{val}[\Gamma;\mathbf{v}_1] = v_1 \quad \textsc{val}[\Gamma;\mathbf{v}_2] = v_2 \quad \textsc{exp}[\Gamma, \mathbf{x}:\mathtt{Void}; \mathtt{cty}; \mathtt{Cmap}; H; \mathbf{t}] = (H', e)}{\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \mathtt{let}\ \mathbf{x} = \mathtt{stfld}\ \mathbf{v}_1\ \mathbf{v}_2\ T\ \mathtt{VC} :: \mathsf{f}; \mathbf{t}] = (H', \mathbf{let}\ x = (((\mathbf{unfold}\ (!\, v_1)).\mathsf{f}) := v_2)\ \mathbf{in}\ e)} \quad \text{(VSTFLD)}$$

Figure 9: Syntax-directed translation of FFIL primops 1.

that $\textsc{upcast}[T_i'; T_i; v_i; x_i; \mathsf{comp}_i]$ casts the value v_i according to the types, binds the result to x_i, and performs the rest of the computation comp_i. These upcasting computations are type operations which have no run-time effect. We will explain the class constructor and the generation of the class location mapping `Cmap` in the coming sections. For now it suffices to say that the $\ell_{\mathtt{C}}.\mathsf{new}$ yields the location of code which creates an object of class `C`.

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. Similarly as in the constructor translation, we use a chain of UPCAST macros to upcast the arguments. The semantics of instance method invocation is different. The result code of the compilation (CALLINST) retrieves the method from the method table of the statically determined value class. The translation of the boxing

$$\frac{\begin{array}{l}\mathit{fields}(\mathtt{C}) = [(\mathsf{f}_1, T_1) \ldots (\mathsf{f}_n, T_n)] \qquad \ell_{\mathtt{C}} = \mathtt{Cmap}(\mathtt{C}) \\ \forall i \in \{1 \ldots n\} \\ \quad \circ; \Gamma \vdash \mathbf{v}_i : T'_i \qquad \textsc{val}[\Gamma; \mathbf{v}_i] = v_i \qquad T'_i \subseteq T_i\end{array}}{\begin{array}{l}\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \mathtt{let}\ \mathbf{x}:T = \mathtt{newobj}\ \mathtt{C}\ \mathbf{v}_1 \ldots \mathbf{v}_n; \mathbf{t}] \\ = \textsc{wrap}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; (\mathtt{let}\ \mathbf{x}:T = \bullet; \mathbf{t}); \mathsf{comp}_0] \\ \quad \text{where} \\ \quad \forall i \in \{1 \ldots n\}.\ \mathsf{comp}_{i-1} = \textsc{upcast}[T'_i; T_i; v_i; x_i; \mathsf{comp}_i] \\ \quad \mathsf{comp}_n = \mathbf{jmp}\ (\ell_{\mathtt{C}}.\mathsf{new})\ \langle x_1, \ldots, x_n, k\rangle\end{array}} \quad \text{(NEW)}$$

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

$$\frac{\begin{array}{ll}\mathtt{MD} = T\ \mathtt{VC}\ \mathsf{m}\ \{T_1 \ldots T_n\} & \ell_{\mathtt{VC}} = \mathtt{Cmap}(\mathtt{VC}) \\ \circ; \Gamma \vdash \mathbf{v} : (\mathtt{value\ class\ VC}) \& & \textsc{val}[\Gamma; \mathbf{v}] = v \\ \circ; \Gamma \vdash \mathbf{v}_i : T'_i & \textsc{val}[\Gamma; \mathbf{v}_i] = v_i \\ T'_i \subseteq T_i & \end{array}\ \Big\}_{i \in \{1 \ldots n\}}}{\begin{array}{l}\textsc{exp}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; \\ \quad \mathtt{let}\ \mathbf{x}:T = \mathtt{callinst}\ \mathbf{v}\ \mathtt{MD}\ \mathbf{v}_1 \ldots \mathbf{v}_n; \mathbf{t}] = \\ \textsc{wrap}[\Gamma; \mathtt{cty}; \mathtt{Cmap}; H; (\mathtt{let}\ \mathbf{x}:T = \bullet; \mathbf{t}); \mathsf{comp}_0] \\ \text{where } \forall i \in \{1 \ldots n\} \\ \quad \mathsf{comp}_{i-1} = \textsc{upcast}[T'_i; T_i; v_i; x_i; \mathsf{comp}_i] \\ \quad \mathsf{comp}_n = \\ \quad\ \mathbf{jmp}\ (\ell_{\mathtt{VC}}.\mathtt{dict.m})\ \langle v, x_1, \ldots, x_n, k\rangle\end{array}} \quad \text{(CALLINST)}$$

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

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

Figure 10: Syntax-directed translation of FFIL primops 2.

and unboxing primitives delegate tasks to the boxing and unboxing functions exported by the run-time class record. As we will explain later, a value class record is going to contain entries for the dictionary as well as the boxing and unboxing functions.

Note that in some source level languages (e.g. C#), values of primitive types like `Int` can be boxed as an object. FIL does not support this. However, it is easy to add built-in value classes to handle this feature.

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

$\text{TYP}(\mathit{World}, T)$.

$$\begin{aligned}\mathcal{E}nv[\Gamma, \mathbf{x} : T] &= \mathcal{E}nv[\Gamma], x : \text{TYP}(\mathit{World}, T)\\ \mathcal{E}nv[\circ] &= \circ\end{aligned}$$

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

Proof sketch: by simultaneous induction on the structure of $\mathbf{t}$ and fn.

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 $\mathtt{Cmap}$, 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.

$$\begin{aligned}
\mathit{Dict}[\mathtt{RC}] &\equiv \lambda \mathsf{w}{::}\mathit{kcn}.\, \lambda \mathsf{self}{::}\mathsf{Type}.\\
&\quad \{(\mathit{Rows}[\mathtt{RC}, \top]\ \mathsf{w}\ \mathit{Empty}[\mathtt{RC}]) \cdot \mathsf{m}\ \mathsf{self}\}\\
\mathit{Ctor}[\mathtt{C}] &\equiv \lambda \mathsf{w}{::}\mathit{kcn}.\\
&\quad \neg(\text{TYP}(\mathsf{w}, T_1) \times \cdots \times \text{TYP}(\mathsf{w}, T_n) \times \mathbf{cont}\ \mathsf{w} \cdot \mathtt{C})\\
&\quad \text{where } \mathit{fieldvec}_v(\mathtt{C}) = [(\mathsf{f}_1, T_1) \ldots (\mathsf{f}_n, T_n)]\\
\mathit{Class}[\mathtt{RC}] &\equiv \lambda \mathsf{w}{::}\mathit{kcn}.\, \{\\
&\quad \mathtt{dict} : \forall \mathsf{tail}.\, \mathit{Dict}[\mathtt{RC}]\ \mathsf{w}\ (\mathit{SelfTy}[\mathtt{RC}]\ \mathsf{w}\ \mathsf{tail}),\\
&\quad \mathsf{new} : \mathit{Ctor}[\mathtt{RC}]\ \mathsf{w}\}\\
\mathit{ClassF}[\mathtt{RC}] &\equiv \mathit{Class}[\mathtt{RC}]\ \mathit{World}
\end{aligned}$$

$$\begin{aligned}
\mathit{Dict}[\mathtt{VC}] &\equiv \lambda \mathsf{w}{::}\mathit{kcn}.\, \lambda \mathsf{self}{::}\mathsf{Type}.\\
&\quad \{(\mathit{Rows}_v[\mathtt{VC}, \top]\ \mathsf{w}\ \mathit{Empty}[\mathtt{VC}]) \cdot \mathsf{m}\ \mathsf{self}\}\\
\mathit{Box}[\mathtt{VC}] &\equiv \lambda \mathsf{w}{::}\mathit{kcn}.\, \neg(\mathbf{ref}\,(\mathsf{w}.\mathtt{VC}) \times \mathbf{cont}\,\mathsf{w} \cdot \mathtt{VC}_\square)\\
\mathit{Unbox}[\mathtt{VC}] &\equiv \lambda \mathsf{w}{::}\mathit{kcn}.\, \neg(\mathsf{w}.\mathtt{VC}_\square \times \mathbf{cont}\,(\mathbf{ref}\,(\mathsf{w} \cdot \mathtt{VC})))\\
\mathit{Class}[\mathtt{VC}] &\equiv \lambda \mathsf{w}{::}\mathit{kcn}.\, \{\\
&\quad \mathtt{dict} : \mathit{Dict}[\mathtt{VC}]\ \mathsf{w}\ (\mathit{SelfTy}[\mathtt{VC}]\ \mathsf{w}),\\
&\quad \mathsf{new} : \mathit{Ctor}[\mathtt{VC}]\ \mathsf{w},\\
&\quad \mathsf{box} : \mathit{Box}[\mathtt{VC}]\ \mathsf{w},\\
&\quad \mathsf{unbox} : \mathit{Unbox}[\mathtt{VC}]\ \mathsf{w}\}\\
\mathit{ClassF}[\mathtt{VC}] &\equiv \mathit{Class}[\mathtt{VC}]\ \mathit{World}
\end{aligned}$$

Figure 11: Macros for dictionary, constructor, and class types.

We now discuss formally the translation of class declarations. In Figure 11, we define several macros for describing dictionary and class types. These macros are overloaded to work on both reference classes and value classes. The top four definitions are for reference classes. The bottom five definitions are for value classes. The constructor type of value classes shares the same macro as that of reference classes.

The translation of reference class declaration is shown in Figure 12. The translation CDEC resorts to the dictionary translation DICT for constructing the dictionary. The result dictionary is polymorphic over the tail of self type so that it can be inherited by subclasses. A fresh location ℓ_{dict} is created for storing this dictionary in the heap. Besides, DICT updates the heap and provides a mapping MM from method names to the locations of the corresponding method code in the updated heap. Then, Mmap is updated by mapping the class name RC to a pair consisting of the dictionary location ℓ_{dict} and the method mapping MM. Next, the dictionary location is passed to the constructor translation NEW. Finally, the translation updates the heap and builds a record which contains the corresponding locations of the dictionary and the constructor.

We explain the constructor translation first. It generates code which instantiates the dictionary with the empty tail type as the vtable pointer, pairs it with all the field arguments, and produces an object based on our encoding. The folding operation and the packaging macro forms the "fold-pack-fold" idiom we talked about in section 5.1. Eventually, it opens the continuation closure and jumps to the continuation with the newly created object and the environment provided in the closure as arguments.

The dictionary translation for class Obj essentially yields an empty record. For other classes, it resorts to method translation METH to translate each of its methods. These translations are chained together so that the heap H is updated accordingly. Finally, it puts the result values in fresh

Class declaration translation:

$$\frac{\begin{array}{l}\texttt{DICT}[\texttt{RC};\texttt{Cmap};\mathsf{Mmap};H]=(\mathsf{MM},H',v_{dict})\\ \{\ell_{dict},\ell_{new}\}\cap\mathrm{dom}(H')=\emptyset\\ \tau_{dict}=\forall\mathsf{tail}.\mathit{Dict}[\texttt{RC}]\ \mathsf{w}\ (\mathit{SelfTy}[\texttt{RC}]\ \mathsf{w}\ \mathsf{tail})\\ \mathsf{Mmap}'=\mathsf{Mmap}[\texttt{RC}\mapsto(\ell_{dict}:\tau_{dict},\mathsf{MM})]\\ H''=H'[\ell_{dict}:\tau_{dict}\mapsto v_{dict},\\ \qquad\ell_{new}:(\mathit{Ctor}[\texttt{RC}]\ \mathsf{w})\mapsto\texttt{NEW}[\texttt{RC};\ell_{dict}]]\\ \quad\text{where }\mathsf{w}=\mathit{World}\end{array}}{\begin{array}{l}\texttt{CDEC}[\texttt{Cmap};\mathsf{Mmap};H;\texttt{RC}]=(\mathsf{Mmap}',H'',\\ \quad\{\texttt{dict}=\ell_{dict},\mathsf{new}=\ell_{new}\}\end{array}}\ (\text{RCDEC})$$

Dictionary construction:

$$\texttt{DICT}[\texttt{Obj};\texttt{Cmap};\mathsf{Mmap};H]=((),H,\ \Lambda\mathsf{tail}{::}\mathit{ktail}[\texttt{Obj}].\{\})\ (\text{RDICTBASE})$$

$$\frac{\begin{array}{l}\mathit{methvec}(\texttt{RC})=[(\mathsf{m}_1,\mathit{Sig}_1)\ldots(\mathsf{m}_n,\mathit{Sig}_m)]\\ \texttt{METH}[\texttt{RC};\mathsf{m}_i;H_{i-1};\texttt{Cmap};\mathsf{Mmap}]=(H_i,v_i)\quad\forall i\in\{1\ldots n\}\\ \{\ell_1,\ldots,\ell_n\}\cap\mathrm{dom}(H_n)=\emptyset\\ \tau_j=\Lambda\mathsf{tail}{::}\mathit{ktail}[\texttt{RC}].\mathit{Ty}[(\mathit{SelfTy}[\texttt{RC}]\ \mathit{World}\ \mathsf{tail});\mathit{World};\mathit{Sig}_j]\\ H'=H_n[(\ell_j:\tau_j\mapsto v_j)^{j\in\{1\ldots n\}}]\\ \mathsf{MM}=(\mathsf{m}_j:\tau_j\mapsto\ell_j)^{j\in\{1\ldots n\}}\end{array}}{\begin{array}{l}\texttt{DICT}[H_0;\texttt{Cmap};\mathsf{Mmap};\texttt{RC}]=(\mathsf{MM},H',\\ \quad\Lambda\mathsf{tail}{::}\mathit{ktail}[\texttt{RC}].\{l_1=\ell_1\ [\mathsf{tail}],\ldots,l_n=\ell_n\ [\mathsf{tail}]\}\end{array}}\ (\text{RDICTSUB})$$

Constructor code:

$$\frac{\mathit{fields}(\texttt{RC})=[(\mathsf{f}_1,T_1)\ldots(\mathsf{f}_n,T_n)]}{\begin{array}{l}\texttt{NEW}[\texttt{RC};\ell_{dict}]=\\ \ \lambda\{\mathsf{f}_1:\mathrm{TYP}(\mathit{World},T_1),\ldots,\mathsf{f}_n:\mathrm{TYP}(\mathit{World},T_n),\\ \quad k:\mathsf{cont}\ \mathit{World}{\cdot}\texttt{RC}\}.\\ \ \mathsf{let}\ y_1=\mathsf{cell}\ \mathsf{f}_1\\ \ \mathsf{in}\ldots\mathsf{let}\ y_n=\mathsf{cell}\ \mathsf{f}_n\\ \qquad\mathsf{in let}\ x=\mathsf{fold}\ \{\texttt{vtab}=\ell_{dict}\ [\mathit{Empty}[\texttt{RC}]],\\ \qquad\qquad\mathsf{f}_1=y_1,\ldots,\mathsf{f}_n=y_n\}\\ \qquad\qquad\mathsf{as}\ \mathit{SelfTy}[\texttt{RC}]\ \mathit{World}\ \mathit{Empty}[\texttt{RC}]\\ \qquad\mathsf{in open}\ k\ \mathsf{as}\ \langle\alpha{::}\mathsf{Type},y:\neg(\mathit{World}{\cdot}\texttt{RC}\times\alpha)\times\alpha\rangle\\ \qquad\quad\mathsf{in jmp}\ (y.1)\ \langle\mathrm{PACK}[\texttt{RC};\mathit{Empty}[\texttt{RC}];x],\ y.2\rangle\end{array}}\ (\text{RCTOR})$$

Method code:

$$\frac{\begin{array}{l}CT(\texttt{RC})=\texttt{ref class RC}\lhd\texttt{RC}'\ \{\ldots\mathsf{K}\ \mathsf{M}_1\ldots\mathsf{M}_n\}\\ \mathsf{m}\text{ not defined in }\mathsf{M}_1\ldots\mathsf{M}_n\\ \mathit{defcls}(\texttt{RC},\mathsf{m})=\texttt{RC}_1\qquad(\pi_2(\mathsf{Mmap}(\texttt{RC}_1)))(\mathsf{m})=\ell_m\end{array}}{\begin{array}{l}\texttt{METH}[\texttt{RC};\mathsf{m};H;\texttt{Cmap};\mathsf{Mmap}]=(H,\\ \quad\Lambda\mathsf{tail}{::}\mathit{ktail}[\texttt{RC}].\ell_m\ [\mathit{Rows}[\texttt{RC},\texttt{RC}_1]\ \mathit{World}\ \mathsf{tail}]\end{array}}\ (\text{RMETHSUP})$$

$$\frac{\begin{array}{l}CT(\texttt{RC})=\texttt{ref class RC}\lhd\texttt{RC}'\ \{\ldots\mathsf{K}\ \mathsf{M}_1\ldots\mathsf{M}_n\}\\ \exists j:\mathsf{M}_j=T\ \mathsf{m}\ (T_1\ \mathbf{x}_1\ldots T_m\ \mathbf{x}_m)\ \{\mathbf{t}\}\\ \Gamma=\mathbf{x}_1:T_1,\ldots,\mathbf{x}_m:T_m,\mathbf{this}:(\texttt{class RC})\\ \Gamma\vdash\mathbf{t}:T'\qquad T'\subseteq T\\ \mathrm{EXP}[\Gamma;T'\Rightarrow T;\texttt{Cmap};H;\mathbf{t}]=(H',e)\end{array}}{\begin{array}{l}\texttt{METH}[\texttt{RC};\mathsf{m};H;\texttt{Cmap};\mathsf{Mmap}]=(H',\\ \ \Lambda\mathsf{tail}{::}\mathit{ktail}[\texttt{RC}].\\ \ \lambda\{\mathsf{self}:\mathit{SelfTy}[\texttt{RC}]\ \mathit{World}\ \mathsf{tail},\\ \quad x_1:\mathrm{TYP}(\mathit{World},T_1),\ldots,x_m:\mathrm{TYP}(\mathit{World},T_m),\\ \quad\mathsf{retcont}:\mathsf{cont}\ (\mathrm{TYP}(\mathit{World},\tau))\}.\\ \ \mathsf{let this}:(\mathit{World}{\cdot}\texttt{RC})=\mathrm{PACK}[\texttt{RC};\mathit{Empty}[\texttt{RC}];\mathsf{self}]\\ \ \mathsf{in}\ e\end{array}}\ (\text{RMETHADD})$$

Figure 12: Translation of reference class declarations.

locations of the heap, creates the mapping MM with the locations of these methods, and builds the polymorphic dictionary.

The method translation METH generates methods which is also polymorphic over the tail of the self type. By doing this, these methods can be reused by subclasses. METH supports two cases: (1) it looks up the method from a super class and reuse the result of the compilation, if the method is not defined in the current class; (2) it 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 translation of value class declaration is shown in Figure 13. It also resorts to a dictionary translation $\texttt{DICT}_v$ 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 $\texttt{NEW}_v$ and UNBOX which are used to produce the constructor and the unboxing operation. The values obtained from these compilations are placed at fresh locations in the heap. Finally, a record of these locations is formed as the class record.

The constructor translation would not be surprising.

The dictionary construction chains the compilation of each of the methods. This time, we do not need to create the method mapping MM. Neither do we need to retrieve entries from Mmap.

The method translation has only one case, because no method of a super class can be invoked on an object of value type. The methods inherited by a value class from a reference class cannot be invoked on value class objects before they are boxed.

Boxing and unboxing translation generates code which coerces objects between value types and reference types. They change the representation and in the meanwhile make copies of objects. Essentially, box adds a vtable entry into the object, while unbox removes that entry from the object.

Theorem 2 (Well-typed class declaration) If
$\texttt{Cmap}=(\texttt{C}:\mathit{ClassF}[\texttt{C}]\mapsto\ell_{\texttt{C}})^{\texttt{C}\in cn}$, and $\ell_{\texttt{C}}$ are all distinct,
$\Sigma_0=(\ell_{\texttt{C}}:\mathit{ClassF}[\texttt{C}])^{\texttt{C}\in cn}$, $\vdash_h H:\Sigma$,
$\forall$ superclass $\texttt{RC}'$ of RC (or VC) and $\mathsf{m}\in\mathrm{dom}(\mathit{methvec}(\texttt{RC}'))$,
$\mathsf{Mmap}(\texttt{RC}')=(\ell:\tau,\mathsf{MM})$ and $(\ell:\tau\mapsto\ldots)\in H$,
and $(\mathsf{m}:\tau_{\mathsf{m}}\mapsto\ell_{\mathsf{m}})\in\mathsf{MM}$ and $(\ell_{\mathsf{m}}:\tau_{\mathsf{m}}\mapsto\ldots)\in H$,
$\texttt{CDEC}[\texttt{Cmap};\mathsf{Mmap};H;\texttt{RC}]=(\mathsf{Mmap}',H',v_{RC})$
(or $\texttt{CDEC}_v[\texttt{Cmap};\mathsf{Mmap};H;\texttt{VC}]=(H',v_{VC})$),
$\vdash_h H':\Sigma'$,
then $\Sigma'\subseteq_s\Sigma$, and $\Phi;\Sigma_0,\Sigma';\Delta\vdash_v v_{RC}:\mathit{ClassF}[\texttt{RC}]$
(or $\Phi;\Sigma_0,\Sigma';\Delta\vdash_v v_{VC}:\mathit{ClassF}[\texttt{VC}]$).

Proof sketch: by inspection.

Class declaration translation:

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

Constructor code:

$$\frac{\mathit{fields}(\texttt{VC}) = [(\mathsf{f}_1, T_1) \ldots (\mathsf{f}_n, T_n)]}{\begin{array}{l}\texttt{NEW}_v[\texttt{VC}] =\\ \quad \boldsymbol{\lambda}\{\mathsf{f}_1 : \textsc{typ}(\mathit{World}, T_1), \ldots, \mathsf{f}_n : \textsc{typ}(\mathit{World}, T_n),\\ \quad\ k : \mathbf{cont}\ \mathit{World}{\cdot}\texttt{VC}\}.\\ \quad \mathbf{let}\ y_1 = \mathbf{cell}\ \mathsf{f}_1\\ \quad\ \mathbf{in} \ldots \mathbf{let}\ y_n = \mathbf{cell}\ \mathsf{f}_n\\ \qquad\qquad \mathbf{in\ open}\ k\ \mathbf{as}\ \langle\alpha{::}\mathsf{Type}, y : \neg(\mathit{World}{\cdot}\texttt{VC} \times \alpha) \times \alpha\rangle\\ \qquad\qquad\quad \mathbf{in\ jmp}\ (y.1)\ \langle \mathbf{fold}\ \{\mathsf{f}_1 = y_1, \ldots, \mathsf{f}_n = y_n\},\ y.2\rangle\end{array}}\ (\text{VCTOR})$$

Boxing:

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

Dictionary construction:

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

Method code:

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

Unboxing:

$$\frac{\mathit{fields}(\texttt{VC}) = [(\mathsf{f}_1, T_1) \ldots (\mathsf{f}_m, T_m)]}{\begin{array}{l}\texttt{UNBOX}[\texttt{VC}] =\\ \quad \boldsymbol{\lambda}\{\mathsf{robj} : \mathit{ObjTy}[\texttt{VC}_\square]\ \mathit{World},\ k : \mathbf{cont}\,(\mathbf{ref}\ \mathit{World}{\cdot}\texttt{VC})\}.\\ \quad \mathbf{open}\ (\mathbf{unfold}\ \mathsf{robj}\ \mathbf{as}\ \mathit{World}\ \mathbf{at}\ \lambda\mathsf{w}{::}\mathit{kcn}.\ \mathsf{w}{\cdot}\texttt{VC}_\square)\\ \qquad \mathbf{as}\ \langle \mathsf{tail}{::}\mathit{ktail}[\texttt{VC}_\square], x : \mathit{SelfTy}[\texttt{VC}_\square]\ \mathit{World}\ \mathsf{tail}\rangle\\ \qquad \mathbf{in\ let}\ x_0 = \mathbf{unfold}\ x\\ \qquad\quad \mathbf{in\ let}\ x_1 = \mathbf{cell}\ !\,(x_0.\mathsf{f}_1)\\ \qquad\quad \mathbf{in}\ \ldots \mathbf{let}\ x_n = \mathbf{cell}\ !\,(x_0.\mathsf{f}_n)\\ \qquad\qquad\quad \mathbf{in\ let}\ x' = \mathbf{cell}\ (\mathbf{fold}\ \{\mathsf{f}_1 = x_1, \ldots, \mathsf{f}_n = x_n\})\\ \qquad\qquad\qquad \mathbf{in\ open}\ k\\ \qquad\qquad\qquad\quad \mathbf{as}\ \langle\alpha{::}\mathsf{Type},\\ \qquad\qquad\qquad\qquad y : \neg(\mathbf{ref}\ \mathit{World}{\cdot}\texttt{VC} \times \alpha) \times \alpha\rangle\\ \qquad\qquad\qquad\quad \mathbf{in\ jmp}\ (y.1)\ \langle x',\ y.2\rangle\end{array}}\ (\text{CUNBOX})$$

Figure 13: Translations of value class definitions.

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

Figure 14: Program translation.

5.4 Program translation

Figure 14 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 [21] can be applied here to achieve separate compilation.

Theorem 3 (Type-preserving program compilation) If $\circ; \circ \vdash_c \mathbf{t} : T$, $\texttt{PROG}[cn; \mathbf{t}] = (H, e)$, and $\vdash_h H : \Sigma$, then $\circ; \Sigma; (\mathsf{retcont} : \textsc{typ}(\mathit{World}, T)) \vdash_c e$.

Proof sketch: by inspection.

6 Extensions

Our source language FIL is closely related to BIL so that the semantics is based on well-understood ideas. Meanwhile, this type-preserving compilation technique extendeds to support other features of MS IL. As shown by previous work, many features which also exist in Java can be supported, including multiple constructors, null references, exceptions, and dynamic casts.

MS IL supports *invoke-virtual* semantics and *invoke-instance* semantics on both objects of reference types and objects of value types. In BIL and FIL, methods defined in reference classes (virtual methods) can only be invoked using the invoke-virtual semantics; and methods defined in value classes (instance methods) can only be invoked using the invoke-instance semantics. This simplification is made mainly for ease of understanding. Some simple extensions suffice to support this MS IL feature. On one hand, since value classes cannot be extended, a virtual call on an instance method can be statically resolved, knowing that the dynamic type of the object is fixed. This is similar to the case of *final* classes in Java. On the other hand, an instance call on a virtual method would be resolved statically to refer to the method defined in the static class of the object according to the semantics. This is actually the *invoke-special* semantics in Java.

A new feature introduced in the Common Language Infrastructure [24] and supported by MS IL is *delegates*. Delegates in high-level languages (*e.g.,* C# [23, 1]) enable scenarios that some languages – C++, Pascal, Modula, and others – have addressed with function pointers. But unlike function pointers, delegates are object-oriented, type-safe, and secure. Nevertheless, these type-safety and security come from the fact that the implementation is provided by the Virtual Execution System [24], as opposed to user code.

Although delegates appear to be simply another kind of user defined classes, they are tightly controlled. When a C# delegate declaration is compiled into MS IL code, an implicit delegate class is constructed. A special constructor and a special `Invoke` method (they are so called *runtime managed*) are created for this class. The bodies of the constructor and the method `Invoke` are created by the Virtual Execution System. The instantiation of a delegate is compiled to execute the special constructor with a pointer to the target method as one of the arguments. The calling of a delegate is compiled to invoke the special `Invoke` method.

The safety of delegates replies on the correctness of the part of the system implementation which compiles delegates from high-level languages (*e.g.,* C#) into MS IL code. The pointer to the target method is manipulated using `ldftn`, `ldvirtftn` and `calli` instructions of MS IL. Meanwhile, these instructions operate on unmanaged pointers (type `native int`) as well.

To support delegates, we can introduce similar instructions `ldftn`, `ldvirtftn` and `calli`, yet the power of these instructions has to be limited for type-safety. Instrucstions `ldftn` and `ldvirtftn` take a method descriptor as the only argument, and produce a function pointer. Instruction `calli` takes a target object, a function pointer and some arguments to the target function as the arguments, and produces a function call. The type system has to make sure that the signature within the function pointer matches the actual target object and the arguments. The translation into LFLINT is straightforward based on our encodings.

7 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. Our strategy naturally extends to support other MS IL features including delegates. 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] B. Albahari, P. Drayton, and B. Merrill. *C# Essentials.* O'Reilly and Associates, 2001.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

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

[20] 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.

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

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

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

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

[25] 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.

[26] 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.

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

[28] 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.

[29] 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/`.

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

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

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

[33] 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.

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

[35] 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.

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

[37] 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.

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

A FFIL Type System

This section covers the type system of FFIL, including:

- Macros defining field lookup, method lookup, and "pointer free".
- Subtyping, class typing, method typing, environment formations and expression typing.
- An operational semantics in continuation passing style.
- Type soundness.

The class table is extended with implicit reference classes. We use similar "pointer free" mechanism as described by Gordon and Syme [8] to prevent dangling pointers.

A.1 Macros

Field lookup $\boxed{\mathit{fields}(C) = \ldots}$

$$\mathit{fields}(\texttt{Obj}) = [] \tag{1}$$

$$\frac{CT(C) = \texttt{ref/val class}\ C \lhd RC\ \{T_1\ f_1 \ldots T_n\ f_n\ K\ \ldots\}}{\mathit{fields}(C) = \mathit{fields}(RC) +\!\!+ [(T_1, f_1) \ldots (T_n, f_n)]} \tag{2}$$

Method lookup $\boxed{\begin{array}{l}\mathit{mtype}(m, C) = \ldots \\ \mathit{mbody}(m, C) = \ldots\end{array}}$

$$\frac{\begin{array}{l}CT(C) = \texttt{ref/val class}\ C \lhd RC\ \{\ldots\ K\ M_1\ \ldots\ M_n\} \\ \exists j : M_j = T\ m\ (T_1\ \mathbf{x}_1\ \ldots\ T_m\ \mathbf{x}_m)\ \{\mathbf{t}\}\end{array}}{\begin{array}{l}\mathit{mtype}(m, C) = T_1 \ldots T_m \to T \\ \mathit{mbody}(m, C) = (\mathbf{x}_1 \ldots \mathbf{x}_m, \mathbf{t})\end{array}} \tag{3}$$

$$\frac{CT(C) = \texttt{ref/val class}\ C\ \triangleleft\ RC\ \{\ldots\ K\ M_1\ \ldots\ M_n\} \quad m \text{ not defined in } M_1 \ldots M_n}{\begin{array}{l} mtype(m, C) = mtype(m, RC) \\ mbody(m, C) = mbody(m, RC) \end{array}} \tag{4}$$

Pointer free $\boxed{pointerFree(T)}$

$$pointerFree(T) \Leftrightarrow \neg(T = T'\& \text{ for some } T') \tag{5}$$

A.2 Static Semantics

Subtyping $\boxed{\tau \subseteq \tau'}$

$$T \subseteq T \tag{6}$$

$$\frac{CT(RC) = \texttt{ref class}\ RC\ \triangleleft\ RC'\ \{\ldots\} \quad \texttt{class}\ RC' \subseteq T}{\texttt{class}\ RC \subseteq T} \tag{7}$$

$$\frac{CT(VC) = \texttt{val class}\ VC\ \triangleleft\ RC'\ \{\ldots\} \quad \texttt{class}\ RC' \subseteq T}{\texttt{class}\ VC \subseteq T} \tag{8}$$

Class typing $\boxed{CD\ \textbf{ok}}$

$$\frac{\begin{array}{l} fields(RC) = [(T'_1, g_1) \ldots (T'_n, g_n)] \\ K = C{::}\texttt{Ctor}\ T'_1\ g_1 \ldots T'_n\ g_n\ T_1\ f_1 \ldots T_m\ f_m \\ pointerFree(T'_i) \quad \forall i \in \{1 \ldots n\} \\ pointerFree(T_i) \quad \forall i \in \{1 \ldots m\} \\ M_i \text{ ok in } C \quad \forall i \in \{1 \ldots k\} \end{array}}{\texttt{ref/val class}\ C\ \triangleleft\ RC\ \{T_1\ f_1 \ldots T_m\ f_m\ K\ M_1 \ldots M_k\} \text{ ok}} \tag{9}$$

Method typing $\boxed{M\ \textbf{ok in}\ C}$

$$\frac{\begin{array}{l} CT(RC) = \texttt{ref class}\ RC\ \triangleleft\ RC'\ \{\ldots\} \\ \circ; \mathbf{x}_1 : T_1, \ldots, \mathbf{x}_n : T_n, \mathbf{this} : \texttt{class}\ RC \vdash_c \mathbf{t} : T' \\ T' \subseteq T \qquad pointerFree(T) \end{array}}{T\ m\ (T_1\ \mathbf{x}_1 \ldots T_n\ \mathbf{x}_n)\ \{\mathbf{t}\} \text{ ok in } RC} \tag{10}$$

$$\frac{\begin{array}{l} CT(VC) = \texttt{val class}\ VC\ \triangleleft\ RC'\ \{\ldots\} \\ \circ; \mathbf{x}_1 : T_1, \ldots, \mathbf{x}_n : T_n, \mathbf{this} : (\texttt{value class}\ VC)\& \vdash_c \mathbf{t} : T' \\ T' \subseteq T \qquad pointerFree(T) \end{array}}{T\ m\ (T_1\ \mathbf{x}_1 \ldots T_n\ \mathbf{x}_n)\ \{\mathbf{t}\} \text{ ok in } VC} \tag{11}$$

Type environment formation $\boxed{\vdash \Gamma\ type\ env}$

$$\vdash \circ\ type\ env \tag{12}$$

$$\frac{\vdash \Gamma\ type\ env}{\vdash \Gamma, \mathbf{x} : T\ type\ env} \tag{13}$$

Store type formation $\boxed{\vdash \Sigma\ store\ type}$

$$\vdash \circ\ store\ type \tag{14}$$

$$\frac{\vdash \Sigma\ store\ type}{\vdash \Sigma, L : T\ store\ type} \tag{15}$$

Store type subtyping $\boxed{\Sigma' \subseteq_s \Sigma}$

$$\frac{\Sigma'(L) = \Sigma(L) \quad \forall L \in \mathrm{dom}(\Sigma)}{\Sigma' \subseteq_s \Sigma} \tag{16}$$

Expression typing $\boxed{\Sigma; \Gamma \vdash exp : T}$

$$\frac{\vdash \Gamma\ type\ env}{\Sigma; \Gamma \vdash_v \mathbf{x} : \Gamma(\mathbf{x})} \tag{17}$$

$$\Sigma; \Gamma \vdash_v () : \texttt{Void} \tag{18}$$

$$\Sigma; \Gamma \vdash_v i : \texttt{Int} \tag{19}$$

$$\frac{\vdash \Sigma\ store\ type \qquad \Sigma(L) = T}{\Sigma; \Gamma \vdash_v L : T} \tag{20}$$

$$\frac{\Sigma; \Gamma, \mathbf{x}_1 : T_1, \ldots, \mathbf{x}_n : T_n \vdash_c \mathbf{t} : T}{\Sigma; \Gamma \vdash_f \lambda(\mathbf{x}_1 : T_1 \ldots \mathbf{x}_n : T_n).\mathbf{t} : (T_1 \ldots T_n) \to T} \tag{21}$$

$$\frac{\begin{array}{l} \Sigma; \Gamma, \mathbf{x}_1 : T_1, \ldots, \mathbf{x}_n : T_n \vdash_f w_i : T_i \quad \forall i \in \{1 \ldots n\} \\ \Sigma; \Gamma, \mathbf{x}_1 : T_1, \ldots, \mathbf{x}_n : T_n \vdash_c \mathbf{t} : T \end{array}}{\Sigma; \Gamma \vdash_c \texttt{letrec}\ (\mathbf{x}_1 = w_1 \ldots \mathbf{x}_n = w_n).\mathbf{t} : T} \tag{22}$$

$$\frac{\Sigma; \Gamma \vdash_p \mathbf{p} : T' \qquad \Sigma; \Gamma, \mathbf{x} : T' \vdash_c \mathbf{t} : T}{\Sigma; \Gamma \vdash_c \texttt{let}\ \mathbf{x} = \mathbf{p}; \mathbf{t} : T} \tag{23}$$

$$\frac{\Sigma; \Gamma \vdash_p \mathbf{p} : \texttt{Void} \qquad \Sigma; \Gamma \vdash_c \mathbf{t} : T}{\Sigma; \Gamma \vdash_c \mathbf{p}; \mathbf{t} : T} \tag{24}$$

$$\frac{\Sigma; \Gamma \vdash_v \mathbf{v} : \texttt{Int} \qquad \Sigma; \Gamma \vdash_c \mathbf{t}_1 : T \qquad \Sigma; \Gamma \vdash_c \mathbf{t}_2 : T}{\Sigma; \Gamma \vdash_c \texttt{if}\ \mathbf{v}\ \texttt{then}\ \mathbf{t}_1\ \texttt{else}\ \mathbf{t}_2 : T} \tag{25}$$

$$\Sigma; \Gamma \vdash_c \texttt{return} : \texttt{Void} \tag{26}$$

$$\frac{\Sigma; \Gamma \vdash_v \mathbf{v} : T}{\Sigma; \Gamma \vdash_c \texttt{return}\ \mathbf{v} : T} \tag{27}$$

$$\frac{\begin{array}{l} \Sigma; \Gamma \vdash_v \mathbf{v} : (T_1 \ldots T_n) \to T \\ \Sigma; \Gamma \vdash_v \mathbf{v}_i : T_i \quad \forall i \in \{1 \ldots n\} \end{array}}{\Sigma; \Gamma \vdash_c \mathbf{v}\ (\mathbf{v}_1 \ldots \mathbf{v}_n) : T} \tag{28}$$

$$\frac{\Sigma; \Gamma \vdash_v \mathbf{v} : T \qquad pointerFree(T)}{\Sigma; \Gamma \vdash_p \texttt{cell}\ \mathbf{v} : T\&} \tag{29}$$

$$\frac{\Sigma; \Gamma \vdash_v \mathbf{v} : T\&}{\Sigma; \Gamma \vdash_p !\ \mathbf{v} : T} \tag{30}$$

$$\frac{\Sigma; \Gamma \vdash_v \mathbf{v}_1 : T\& \qquad \Sigma; \Gamma \vdash_v \mathbf{v}_2 : T \qquad pointerFree(T)}{\Sigma; \Gamma \vdash_p \mathbf{v}_1 := \mathbf{v}_2 : \texttt{Void}} \tag{31}$$

$$\frac{\begin{array}{l} fields(RC) = [(T_1, f_1) \ldots (T_n, f_n)] \\ \Sigma; \Gamma \vdash_v \mathbf{v}_i : T'_i \quad T'_i \subseteq T_i \quad \forall i \in \{1 \ldots n\} \end{array}}{\Sigma; \Gamma \vdash_p \texttt{newobj}\ RC\ (\mathbf{v}_1 \ldots \mathbf{v}_n) : \texttt{class}\ RC} \tag{32}$$

$$\frac{\begin{array}{l} fields(VC) = [(T_1, f_1) \ldots (T_n, f_n)] \\ \Sigma; \Gamma \vdash_v \mathbf{v}_i : T'_i \quad T'_i \subseteq T_i \quad \forall i \in \{1 \ldots n\} \end{array}}{\Sigma; \Gamma \vdash_p \texttt{newobj}\ VC\ (\mathbf{v}_1 \ldots \mathbf{v}_n) : \texttt{value class}\ VC} \tag{33}$$

$$\frac{\Sigma;\Gamma \vdash_v \mathbf{v} : \texttt{class}\ RC \qquad \texttt{class}\ RC \subseteq \texttt{class}\ RC' \quad mtype(m, RC) = T_1 \ldots T_n \to T \quad \Sigma;\Gamma \vdash_v \mathbf{v}_i : T_i' \quad T_i' \subseteq T_i \quad \forall i \in \{1 \ldots n\}}{\Sigma;\Gamma \vdash_p \texttt{callvirt}\ \mathbf{v}\ T\ RC'::m\ (T_1 \ldots T_n)\ (\mathbf{v}_1 \ldots \mathbf{v}_n) : T} \quad (34)$$

$$\frac{\Sigma;\Gamma \vdash_v \mathbf{v} : (\texttt{value class}\ VC)\& \quad mtype(m, VC) = T_1 \ldots T_n \to T \quad \Sigma;\Gamma \vdash_v \mathbf{v}_i : T_i' \quad T_i' \subseteq T_i \quad \forall i \in \{1 \ldots n\}}{\Sigma;\Gamma \vdash_p \texttt{callinst}\ \mathbf{v}\ T\ VC::m\ (T_1 \ldots T_n)\ (\mathbf{v}_1 \ldots \mathbf{v}_n) : T} \quad (35)$$

$$\frac{\Sigma;\Gamma \vdash_v \mathbf{v} : \texttt{class}\ RC \quad \mathit{fields}(RC) = [(T_1, f_1) \ldots (T_n, f_n)] \qquad i \in \{1 \ldots n\}}{\Sigma;\Gamma \vdash_p \texttt{ldflda}\ \mathbf{v}\ T_i\ RC :: f_i : T_i\&} \quad (36)$$

$$\frac{\Sigma;\Gamma \vdash_v \mathbf{v} : (\texttt{value class}\ VC)\& \quad \mathit{fields}(VC) = [(T_1, f_1) \ldots (T_n, f_n)] \qquad i \in \{1 \ldots n\}}{\Sigma;\Gamma \vdash_p \texttt{ldflda}\ \mathbf{v}\ T_i\ VC :: f_i : T_i\&} \quad (37)$$

$$\frac{\Sigma;\Gamma \vdash_v \mathbf{v}_1 : \texttt{class}\ RC \qquad \Sigma;\Gamma \vdash_v \mathbf{v}_2 : T_i \quad \mathit{fields}(RC) = [(T_1, f_1) \ldots (T_n, f_n)] \qquad i \in \{1 \ldots n\}}{\Sigma;\Gamma \vdash_p \texttt{stfld}\ \mathbf{v}_1\ \mathbf{v}_2\ T_i\ RC :: f_i : \texttt{Void}} \quad (38)$$

$$\frac{\Sigma;\Gamma \vdash_v \mathbf{v}_1 : (\texttt{value class}\ VC)\& \qquad \Sigma;\Gamma \vdash_v \mathbf{v}_2 : T_i \quad \mathit{fields}(VC) = [(T_1, f_1) \ldots (T_n, f_n)] \qquad i \in \{1 \ldots n\}}{\Sigma;\Gamma \vdash_p \texttt{stfld}\ \mathbf{v}_1\ \mathbf{v}_2\ T_i\ VC :: f_i : \texttt{Void}} \quad (39)$$

$$\frac{\Sigma;\Gamma \vdash_v \mathbf{v} : (\texttt{value class}\ VC)\&}{\Sigma;\Gamma \vdash_p \texttt{box}\ \mathbf{v}\ VC : \texttt{class}\ VC} \quad (40)$$

$$\frac{\Sigma;\Gamma \vdash_v \mathbf{v} : \texttt{class}\ VC}{\Sigma;\Gamma \vdash_p \texttt{unbox}\ \mathbf{v}\ VC : (\texttt{value class}\ VC)\&} \quad (41)$$

A.3 Dynamic Semantics

An operational semantics with stores in continuation passing style. Only let and seq involve continuation changes. Pointer and object manipulations involve the store.

Variable environment & machine value

$$\begin{array}{ll} \text{Variable Env} & E ::= \bullet \mid E,\mathbf{x} = V \\ \text{Machine Val} & V ::= () \mid i \mid L \mid \texttt{clos}\langle \mathbf{x}_1 \ldots \mathbf{x}_n, \mathbf{t}, E\rangle \\ & \quad \mid \texttt{rec}\langle \mathbf{x}_1 = w_1 \ldots \mathbf{x}_n = w_n, \mathbf{x}_i, E\rangle \end{array}$$

$$\boxed{\Sigma \vdash_{env} E : \Gamma}$$

$$\Sigma \vdash_{env} \bullet : \circ \quad (42)$$

$$\frac{\Sigma \vdash_{env} E : \Gamma \qquad \Sigma \vdash_m V : T}{\Sigma \vdash_{env} (E,\mathbf{x} = V) : (\Gamma, \mathbf{x} : T)} \quad (43)$$

$$\boxed{\Sigma \vdash_m V : T}$$

$$\Sigma \vdash_m () : \texttt{Void} \quad (44)$$

$$\Sigma \vdash_m i : \texttt{Int} \quad (45)$$

$$\frac{\vdash \Sigma\ \mathit{store\ type} \qquad \Sigma(L) = T}{\Sigma \vdash_m L : T} \quad (46)$$

$$\frac{\Sigma \vdash_{env} E : \Gamma \qquad \Sigma;\Gamma, \mathbf{x}_1 : T_1, \ldots, \mathbf{x}_n : T_n \vdash \mathbf{t} : T}{\Sigma \vdash_m \texttt{clos}\langle \mathbf{x}_1 \ldots \mathbf{x}_n, \mathbf{t}, E\rangle : (T_1 \ldots T_n) \to T} \quad (47)$$

$$\frac{\Sigma \vdash_{env} E : \Gamma \quad \Sigma;\Gamma, \mathbf{x}_1 : T_1, \ldots, \mathbf{x}_n : T_n \vdash w_i : T_i \quad \forall i \in \{1 \ldots n\}}{\Sigma \vdash_m \texttt{rec}\langle \mathbf{x}_1 = w_1 \ldots \mathbf{x}_n = w_n, \mathbf{x}_i, E\rangle : T_i} \quad (48)$$

Continuation stack

$$\begin{array}{ll} \text{Cont Stack} & K ::= \texttt{stop} \mid <\texttt{let}\ \mathbf{x} = \bullet; \mathbf{t} \mid E \mid K> \\ & \quad \mid <\bullet;\ \mathbf{t} \mid E \mid K> \end{array}$$

$$\boxed{\Sigma \vdash_k K : T \Rightarrow T'}$$

$$\Sigma \vdash_k \texttt{stop} : T \Rightarrow T \quad (49)$$

$$\frac{\Sigma \vdash_{env} E : \Gamma \qquad \Sigma;\Gamma, \mathbf{x} : T_1 \vdash \mathbf{t} : T_2 \quad \Sigma \vdash_k K : T_2 \Rightarrow T_3}{\Sigma \vdash_k <\texttt{let}\ \mathbf{x} = \bullet; \mathbf{t} \mid E \mid K> : T_1 \Rightarrow T_3} \quad (50)$$

$$\frac{\Sigma \vdash_{env} E : \Gamma \qquad \Sigma;\Gamma \vdash \mathbf{t} : T_2 \quad \Sigma \vdash_k K : T_2 \Rightarrow T_3}{\Sigma \vdash_k <\bullet;\ \mathbf{t} \mid E \mid K> : T_1 \Rightarrow T_3} \quad (51)$$

Store

$$\text{Store} \quad \Delta ::= \bullet \mid \Delta, L = C(f_1 \mapsto \mathbf{v}_1 \ldots f_n \mapsto \mathbf{v}_n)$$

$$\boxed{\Gamma \vdash_l \Delta : \Sigma}$$

$$\Gamma \vdash_l \bullet : \circ \quad (52)$$

$$\frac{\Gamma \vdash_l \Delta : \Sigma \qquad \Sigma;\Gamma \vdash_v \mathbf{v} : T}{\Gamma \vdash_l (\Delta, L = \mathbf{v}) : (\Sigma, L : T\&)} \quad (53)$$

$$\frac{\Gamma \vdash_l \Delta : \Sigma \qquad \mathit{fields}(RC) = [(T_1, f_1) \ldots (T_n, f_n)] \quad \Sigma;\Gamma \vdash_v \mathbf{v}_i : T_i \quad \forall i \in \{1 \ldots n\}}{\Gamma \vdash_l (\Delta, L = RC(f_1 \mapsto \mathbf{v}_1 \ldots f_n \mapsto \mathbf{v}_n)) : (\Sigma, L : \texttt{class}\ RC)} \quad (54)$$

$$\frac{\Gamma \vdash_l \Delta : \Sigma \qquad \mathit{fields}(VC) = [(T_1, f_1) \ldots (T_n, f_n)] \quad \Sigma;\Gamma \vdash_v \mathbf{v}_i : T_i \quad \forall i \in \{1 \ldots n\}}{\Gamma \vdash_l (\Delta, L = VC(f_1 \mapsto \mathbf{v}_1 \ldots f_n \mapsto \mathbf{v}_n)) : (\Sigma, L : \texttt{value class}\ VC)} \quad (55)$$

Store macros

$$\mathit{lookup}(\Delta, \ell) = \Delta(\ell) \quad (56)$$

$$\frac{\mathit{lookup}(\Delta, L) = C(f_1 \mapsto \mathbf{v}_1 \ldots f_n \mapsto \mathbf{v}_n)}{\mathit{lookup}(\Delta, L.f_i) = \mathbf{v}_i} \quad (57)$$

$$\frac{\Delta(\ell) = \mathbf{v}'}{\mathit{update}(\Delta, \ell, \mathbf{v}) = \Delta[\ell \mapsto \mathbf{v}]} \quad (58)$$

$$\frac{\Delta(L) = C(\ldots f_i \mapsto \mathbf{v}_i \ldots)}{\mathit{update}(\Delta, L.f_i, \mathbf{v}) = \Delta[L \mapsto C(\ldots f_i \mapsto \mathbf{v} \ldots)]} \quad (59)$$

State

State $S ::= \langle exp,\ \Delta,\ E,\ K\rangle$

$$\boxed{\vdash_s \langle exp,\ \Delta,\ E,\ K\rangle : T}$$

$$\frac{\Gamma \vdash_l \Delta : \Sigma \qquad \Sigma \vdash_{env} E : \Gamma \qquad \Sigma;\Gamma \vdash exp : T' \qquad \Sigma \vdash_k K : T' \Rightarrow T}{\vdash_s \langle exp,\ \Delta,\ E,\ K\rangle : T} \tag{60}$$

Converting syntactic values to machine values

$$\begin{array}{ll}\gamma(\mathbf{x}, E) & = E(\mathbf{x}) \\ \gamma((), E) & = () \\ \gamma(i, E) & = i \\ \gamma(L, E) & = L \\ \gamma(\lambda(\mathbf{x}_1 : T_1 \ldots \mathbf{x}_n : T_n).\mathbf{t}, E) & = \mathtt{clos}\langle \mathbf{x}_1 \ldots \mathbf{x}_n, \mathbf{t}, E\rangle\end{array}$$

Reduction $\boxed{S \hookrightarrow S'}$

$$\begin{array}{l}\langle \mathtt{letrec}\ (\mathbf{x}_1 = w_1 \ldots \mathbf{x}_n = w_n).\mathbf{t},\ \Delta,\ E,\ K\rangle \\ \hookrightarrow \langle \mathbf{t},\ \Delta,\ (E,\mathbf{x}_1 = \mathtt{rec}\langle R, \mathbf{x}_1, E\rangle, \ldots, \mathbf{x}_n = \mathtt{rec}\langle R, \mathbf{x}_n, E\rangle),\ K\rangle \\ \quad \text{where } R = (\mathbf{x}_i = w_i)^{i \in \{1\ldots n\}}\end{array} \tag{61}$$

$$\begin{array}{l}\langle \mathtt{let}\ \mathbf{x} = \mathbf{p}; \mathbf{t},\ \Delta,\ E,\ K\rangle \\ \hookrightarrow \langle \mathbf{p},\ \Delta,\ E,\ <\mathtt{let}\ \mathbf{x} = \bullet; \mathbf{t} \mid E \mid K>\rangle\end{array} \tag{62}$$

$$\langle \mathbf{p}; \mathbf{t},\ \Delta,\ E,\ K\rangle \hookrightarrow \langle \mathbf{p},\ \Delta,\ E,\ <\bullet;\ \mathbf{t} \mid E \mid K>\rangle \tag{63}$$

$$\frac{\gamma(\mathbf{v}, E) \neq 0}{\langle \mathtt{if}\ \mathbf{v}\ \mathtt{then}\ \mathbf{t}_1\ \mathtt{else}\ \mathbf{t}_2,\ \Delta,\ E,\ K\rangle \hookrightarrow \langle \mathbf{t}_1,\ \Delta,\ E,\ K\rangle} \tag{64}$$

$$\frac{\gamma(\mathbf{v}, E) = 0}{\langle \mathtt{if}\ \mathbf{v}\ \mathtt{then}\ \mathbf{t}_1\ \mathtt{else}\ \mathbf{t}_2,\ \Delta,\ E,\ K\rangle \hookrightarrow \langle \mathbf{t}_2,\ \Delta,\ E,\ K\rangle} \tag{65}$$

$$\langle \mathtt{return},\ \Delta,\ E,\ K\rangle \hookrightarrow \langle (),\ \Delta,\ E,\ K\rangle \tag{66}$$

$$\langle \mathtt{return}\ \mathbf{v},\ \Delta,\ E,\ K\rangle \hookrightarrow \langle \mathbf{v},\ \Delta,\ E,\ K\rangle \tag{67}$$

$$\frac{\gamma(\mathbf{v}, E) = \mathtt{clos}\langle \mathbf{x}_1 \ldots \mathbf{x}_n, \mathbf{t}, E'\rangle \qquad \gamma(\mathbf{v}_j, E) = V_j \quad \forall j \in \{1\ldots n\}}{\langle \mathbf{v}\ (\mathbf{v}_1 \ldots \mathbf{v}_n),\ \Delta,\ E,\ K\rangle \hookrightarrow \langle \mathbf{t},\ \Delta,\ E'[\mathbf{x}_i \mapsto V_i],\ K\rangle \quad \text{where } i = 1\ldots n} \tag{68}$$

$$\frac{\begin{array}{l}\gamma(\mathbf{v}, E) = \mathtt{rec}\langle (y_i = w_i)^{i \in \{1\ldots m\}}, y_k, E'\rangle \\ \gamma(\mathbf{v}_j, E) = V_j \quad \forall j \in \{1\ldots n\} \\ \gamma(w_k, E_r) = \mathtt{clos}\langle \mathbf{x}_1 \ldots \mathbf{x}_n, \mathbf{t}, E'_r\rangle \\ \text{where } E_r = (E', y_1 = \mathtt{rec}\langle R, y_1, E'\rangle, \ldots, y_n = \mathtt{rec}\langle R, y_n, E'\rangle) \\ \text{and } R = (y_i = w_i)^{i \in \{1\ldots m\}}\end{array}}{\begin{array}{l}\langle \mathbf{v}\ (\mathbf{v}_1 \ldots \mathbf{v}_n),\ \Delta,\ E,\ K\rangle \hookrightarrow \langle \mathbf{t},\ \Delta,\ E'_r[\mathbf{x}_j \mapsto V_j],\ K\rangle \\ \quad \text{where } j = 1\ldots n\end{array}} \tag{69}$$

$$\frac{L \notin \mathrm{dom}(\Delta)}{\langle \mathtt{cell}\ \mathbf{v},\ \Delta,\ E,\ K\rangle \hookrightarrow \langle L,\ (\Delta, L = \mathbf{v}),\ E,\ K\rangle} \tag{70}$$

$$\frac{\gamma(\mathbf{v}, E) = L \qquad \mathit{lookup}(\Delta, L) = \mathbf{v}'}{\langle !\ \mathbf{v},\ \Delta,\ E,\ K\rangle \hookrightarrow \langle \mathbf{v}',\ \Delta,\ E,\ K\rangle} \tag{71}$$

$$\frac{\gamma(\mathbf{v}, E) = L \qquad \mathit{update}(\Delta, L, \mathbf{v}') = \Delta'}{\langle \mathbf{v} := \mathbf{v}',\ \Delta,\ E,\ K\rangle \hookrightarrow \langle (),\ \Delta',\ E,\ K\rangle} \tag{72}$$

$$\frac{L \notin \mathrm{dom}(\Delta)}{\begin{array}{l}\langle \mathtt{newobj}\ C\ \mathbf{v}_1 \ldots \mathbf{v}_n,\ \Delta,\ E,\ K\rangle \\ \hookrightarrow \langle L,\ (\Delta, L = C(f_1 \mapsto \mathbf{v}_1 \ldots f_n \mapsto \mathbf{v}_n)),\ E,\ K\rangle\end{array}} \tag{73}$$

$$\frac{\begin{array}{l}\gamma(\mathbf{v}, E) = L \qquad \gamma(\mathbf{v}_i, E) = V_i \quad \forall i \in \{1\ldots n\} \\ \Delta(L) = RC(\ldots) \qquad \mathit{mbody}(m, RC) = (\mathbf{x}_1 \ldots \mathbf{x}_n, \mathbf{t})\end{array}}{\begin{array}{l}\langle \mathtt{callvirt}\ \mathbf{v}\ T\ RC'::m\ (T_1 \ldots T_n)\ \mathbf{v}_1 \ldots \mathbf{v}_n,\ \Delta,\ E,\ K\rangle \\ \hookrightarrow \langle \mathbf{t},\ \Delta,\ E[\mathtt{this}:=L, \mathbf{x}_1:=V_1 \ldots \mathbf{x}_n:=V_n],\ K\rangle\end{array}} \tag{74}$$

$$\frac{\begin{array}{l}\gamma(\mathbf{v}, E) = L \qquad \gamma(\mathbf{v}_i, E) = V_i \quad \forall i \in \{1\ldots n\} \\ \mathit{mbody}(m, VC) = (\mathbf{x}_1 \ldots \mathbf{x}_n, \mathbf{t})\end{array}}{\begin{array}{l}\langle \mathtt{callinst}\ \mathbf{v}\ T\ VC::m\ (T_1 \ldots T_n)\ \mathbf{v}_1 \ldots \mathbf{v}_n,\ \Delta,\ E,\ K\rangle \\ \hookrightarrow \langle \mathbf{t},\ \Delta,\ E[\mathtt{this}:=L, \mathbf{x}_1:=V_1 \ldots \mathbf{x}_n:=V_n],\ K\rangle\end{array}} \tag{75}$$

$$\frac{\gamma(\mathbf{v}, E) = L \qquad \Delta(L) = RC(\ldots f_i \mapsto \mathbf{v}_i \ldots) \qquad \mathit{fields}(RC) = [(T_1, f_1) \ldots (T_i, f_i) \ldots (T_n, f_n)]}{\langle \mathtt{ldflda}\ \mathbf{v}\ T_i\ RC :: f_i,\ \Delta,\ E,\ K\rangle \hookrightarrow \langle L.f_i,\ \Delta,\ E,\ K\rangle} \tag{76}$$

$$\frac{\begin{array}{l}\gamma(\mathbf{v}, E) = L \qquad \Delta(L) = L_1 \\ \Delta(L_1) = VC(\ldots f_i \mapsto \mathbf{v}_i \ldots) \\ \mathit{fields}(VC) = [(T_1, f_1) \ldots (T_i, f_i) \ldots (T_n, f_n)]\end{array}}{\langle \mathtt{ldflda}\ \mathbf{v}\ T_i\ VC :: f_i,\ \Delta,\ E,\ K\rangle \hookrightarrow \langle L_1.f_i,\ \Delta,\ E,\ K\rangle} \tag{77}$$

$$\frac{\begin{array}{l}\gamma(\mathbf{v}, E) = L \qquad \Delta(L) = RC(\ldots f_i \mapsto \mathbf{v}_i \ldots) \\ \mathit{fields}(RC) = [(T_1, f_1) \ldots (T_i, f_i) \ldots (T_n, f_n)]\end{array}}{\begin{array}{l}\langle \mathtt{stfld}\ \mathbf{v}\ \mathbf{v}'\ T_i\ RC :: f_i,\ \Delta,\ E,\ K\rangle \\ \hookrightarrow \langle (),\ \Delta[L \mapsto RC(\ldots f_i \mapsto \mathbf{v}' \ldots)],\ E,\ K\rangle\end{array}} \tag{78}$$

$$\frac{\begin{array}{l}\gamma(\mathbf{v}, E) = L \qquad \Delta(L) = L_1 \\ \Delta(L_1) = VC(\ldots f_i \mapsto \mathbf{v}_i \ldots) \\ \mathit{fields}(VC) = [(T_1, f_1) \ldots (T_i, f_i) \ldots (T_n, f_n)]\end{array}}{\begin{array}{l}\langle \mathtt{stfld}\ \mathbf{v}\ \mathbf{v}'\ T_i\ VC :: f_i,\ \Delta,\ E,\ K\rangle \\ \hookrightarrow \langle (),\ \Delta[L_1 \mapsto VC(\ldots f_i \mapsto \mathbf{v}' \ldots)],\ E,\ K\rangle\end{array}} \tag{79}$$

$$\frac{\begin{array}{l}\gamma(\mathbf{v}, E) = L \qquad \Delta(L) = L_1 \\ \Delta(L_1) = VC(f_1 \mapsto \mathbf{v}_1 \ldots f_n \mapsto \mathbf{v}_n) \qquad L' \notin \mathrm{dom}(\Delta)\end{array}}{\begin{array}{l}\langle \mathtt{box}\ \mathbf{v}\ VC,\ \Delta,\ E,\ K\rangle \\ \hookrightarrow \langle L',\ (\Delta, L' = VC_{\square}(f_1 \mapsto \mathbf{v}_1 \ldots f_n \mapsto \mathbf{v}_n)),\ E,\ K\rangle\end{array}} \tag{80}$$

$$\frac{\begin{array}{l}\gamma(\mathbf{v}, E) = L \qquad \Delta(L) = VC_{\square}(f_1 \mapsto \mathbf{v}_1 \ldots f_n \mapsto \mathbf{v}_n) \\ L' \notin \mathrm{dom}(\Delta) \qquad L_1 \notin \mathrm{dom}(\Delta) \\ \Delta' = (\Delta, L_1 = VC(f_1 \mapsto \mathbf{v}_1 \ldots f_n \mapsto \mathbf{v}_n), L' = L_1)\end{array}}{\langle \mathtt{unbox}\ \mathbf{v}\ VC,\ \Delta,\ E,\ K\rangle \hookrightarrow \langle L',\ \Delta',\ E,\ K\rangle} \tag{81}$$

$$\frac{\gamma(\mathbf{v}, E) = V}{\begin{array}{l}\langle \mathbf{v},\ \Delta,\ E,\ <\mathtt{let}\ \mathbf{x} = \bullet; \mathbf{t} \mid E \mid K>\rangle \\ \hookrightarrow \langle \mathbf{t},\ \Delta,\ E[\mathbf{x} \mapsto V],\ K\rangle\end{array}} \tag{82}$$

$$\frac{\gamma(\mathbf{v}, E) = V}{\langle \mathbf{v},\ \Delta,\ E,\ <\bullet;\ \mathbf{t} \mid E \mid K>\rangle \hookrightarrow \langle \mathbf{t},\ \Delta,\ E,\ K\rangle} \tag{83}$$

A.4 Type Soundness

Theorem 4 (FFIL Soundness) If $\vdash_s \langle exp,\ \Delta,\ E,\ K\rangle : T$, then either (1) $exp = \mathbf{v}$, and $K = \mathtt{stop}$ or (2) $\exists \langle exp',\ \Delta',\ E',\ K'\rangle$, such that $\langle exp,\ \Delta,\ E,\ K\rangle \hookrightarrow \langle exp',\ \Delta',\ E',\ K'\rangle$, and $\vdash_s \langle exp',\ \Delta',\ E',\ K'\rangle : T$.

Proof sketch: By simultaneous induction on the structure of the derivation of $\vdash_s \langle exp,\ \Delta,\ E,\ K\rangle : T$.

B LFLINT Type System

B.1 Static Semantics

This section covers store environment formation, store environment subtyping, typing rules for LFLINT terms, and heap and storage typing. The store environment is overloaded for both locations of the heap (which contains immutable code) and locations of the storage (which contains mutable data). The heap typing form can be used when the storage is empty. Please refer to the previous work [17] for other judgements and rules, including formations of kind, kind environment, type, type equivalence and type environment, etc.

Store environment formation $\boxed{\Phi \vdash \Sigma \ \mathit{store\ type}}$

$$\Phi \vdash \circ \ \mathit{store\ type} \tag{84}$$

$$\frac{\Phi \vdash \Sigma \ \mathit{store\ type} \qquad \Phi \vdash \tau :: \mathsf{Type}}{\Phi \vdash \Sigma, \ell : \tau \ \mathit{store\ type}} \tag{85}$$

Store environment subtyping $\boxed{\Sigma' \subseteq_s \Sigma}$

$$\frac{\Sigma'(\ell) = \Sigma(\ell) \quad \forall \ell \in \mathrm{dom}(\Sigma)}{\Sigma' \subseteq_s \Sigma} \tag{86}$$

Term typing $\boxed{\begin{array}{l}\Phi;\Sigma;\Delta \vdash_v v : \tau \\ \Phi;\Sigma;\Delta \vdash_p p : \tau \\ \Phi;\Sigma;\Delta \vdash_q q : \tau \\ \Phi;\Sigma;\Delta \vdash_c e \\ \Phi;\Sigma;\Delta \vdash_b b : \tau\end{array}}$

$$\frac{\Phi;\Sigma;\Delta \vdash_v v : \tau \qquad \Phi \vdash \tau = \tau' :: \mathsf{Type}}{\Phi;\Sigma;\Delta \vdash_v v : \tau'} \tag{87}$$

$$\Phi;\Sigma;\Delta \vdash_v i : \mathbf{int} \tag{88}$$

$$\frac{\Phi \vdash \Sigma \ \mathit{store\ type} \qquad \ell \in \mathrm{dom}(\Sigma)}{\Phi;\Sigma;\Delta \vdash_v \ell : \Sigma(\ell)} \tag{89}$$

$$\frac{\begin{array}{l} l_i = l_j \Rightarrow i = j \quad (\forall i,j \in \{1 \ldots n\}) \\ \Phi;\Sigma;\Delta \vdash_v v_i : \tau_i \quad (\forall i \in \{1 \ldots n\})\end{array}}{\Phi;\Sigma;\Delta \vdash_v \{l_1 = v_1 \ldots l_n = v_n\} : \{l_1 : \tau_1 \ldots l_n : \tau_n\}} \tag{90}$$

$$\frac{\Phi;\Sigma;\Delta \vdash_v v : \forall \alpha::\kappa.\, \tau \qquad \Phi \vdash \tau' :: \kappa}{\Phi;\Sigma;\Delta \vdash_v v\ [\tau'] : \tau[\alpha := \tau']} \tag{91}$$

$$\frac{\begin{array}{l}\Phi, \alpha :: \kappa \vdash \tau :: \mathsf{Type} \qquad \Phi \vdash \tau' :: \kappa \\ \Phi;\Sigma;\Delta \vdash_v v : \tau[\alpha := \tau']\end{array}}{\Phi;\Sigma;\Delta \vdash_v \langle \alpha::\kappa = \tau', v : \tau \rangle : \exists \alpha::\kappa.\, \tau} \tag{92}$$

$$\frac{\begin{array}{l}\Phi, \alpha :: \kappa \vdash \tau :: \kappa \qquad \Phi \vdash \tau' :: \kappa \Rightarrow \mathsf{Type} \\ \Phi;\Sigma;\Delta \vdash_v v : \tau'\ (\tau[\alpha := \mu\alpha::\kappa.\, \tau])\end{array}}{\Phi;\Sigma;\Delta \vdash_v \mathbf{fold}\ v\ \mathbf{as}\ \mu\alpha::\kappa.\, \tau\ \mathbf{at}\ \tau' : \tau'\ (\mu\alpha::\kappa.\, \tau)} \tag{93}$$

$$\frac{\Phi;\Sigma;\Delta \vdash_p p : \tau \qquad \Phi \vdash \tau = \tau' :: \mathsf{Type}}{\Phi;\Sigma;\Delta \vdash_p p : \tau'} \tag{94}$$

$$\frac{\Phi;\Sigma;\Delta \vdash_v v : \tau}{\Phi;\Sigma;\Delta \vdash_p v : \tau} \tag{95}$$

$$\frac{\Phi \vdash \Delta \ \mathit{type\ env} \qquad x \in \mathrm{dom}(\Delta)}{\Phi;\Sigma;\Delta \vdash_p x : \Delta(x)} \tag{96}$$

$$\frac{\Phi;\Sigma;\Delta \vdash_p p : \{l_1 : \tau_1\ ; \ldots l_n : \tau_n\ ; \tau\}}{\Phi;\Sigma;\Delta \vdash_p p.l_n : \tau_n} \tag{97}$$

$$\frac{\Phi;\Sigma;\Delta \vdash_p p : \tau}{\Phi;\Sigma;\Delta \vdash_p \mathbf{cell}\ p : \mathbf{ref}\ \tau} \tag{98}$$

$$\frac{\Phi;\Sigma;\Delta \vdash_p p : \mathbf{ref}\ \tau}{\Phi;\Sigma;\Delta \vdash_p !\, p : \tau} \tag{99}$$

$$\frac{\begin{array}{l}\Phi, \alpha :: \kappa \vdash \tau :: \kappa \qquad \Phi \vdash \tau' :: \kappa \Rightarrow \mathsf{Type} \\ \Phi;\Sigma;\Delta \vdash_p p : \tau'\ (\mu\alpha::\kappa.\, \tau)\end{array}}{\begin{array}{l}\Phi;\Sigma;\Delta \vdash_p \mathbf{unfold}\ p\ \mathbf{as}\ \mu\alpha::\kappa.\, \tau\ \mathbf{at}\ \tau' : \\ \qquad \tau'\ (\tau[\alpha := \mu\alpha::\kappa.\, \tau])\end{array}} \tag{100}$$

$$\frac{\Phi;\Sigma;\Delta \vdash_p p : \tau}{\Phi;\Sigma;\Delta \vdash_q p : \tau} \tag{101}$$

$$\frac{\Phi;\Sigma;\Delta \vdash_p p : \mathbf{ref}\ \tau \qquad \Phi;\Sigma;\Delta \vdash_p p' : \tau}{\Phi;\Sigma;\Delta \vdash_q p := p' : \mathbf{void}} \tag{102}$$

$$\frac{\Phi;\Sigma;\Delta \vdash_p p_1 : \neg\tau \qquad \Phi;\Sigma;\Delta \vdash_p p_2 : \tau}{\Phi;\Sigma;\Delta \vdash_c \mathbf{jmp}\ p_1\ p_2} \tag{103}$$

$$\frac{\Phi;\Sigma;\Delta \vdash_q q : \tau \qquad \Phi;\Sigma;\Delta, x : \tau \vdash_c e}{\Phi;\Sigma;\Delta \vdash_c \mathbf{let}\ x : \tau = q\ \mathbf{in}\ e} \tag{104}$$

$$\frac{\Phi;\Sigma;\Delta \vdash_p p : \mathbf{int} \qquad \Phi;\Sigma;\Delta \vdash_c e \qquad \Phi;\Sigma;\Delta \vdash_c e'}{\Phi;\Sigma;\Delta \vdash_c \mathbf{ifz}\ p\ \mathbf{then}\ e\ \mathbf{else}\ e'} \tag{105}$$

$$\frac{\begin{array}{l}\Phi;\Sigma;\Delta \vdash_p p : \exists \alpha::\kappa.\, \tau \\ \Phi, \alpha :: \kappa;\Sigma;\Delta, x : \tau \vdash_c e\end{array}}{\Phi;\Sigma;\Delta \vdash_c \mathbf{open}\ p\ \mathbf{as}\ \langle \alpha::\kappa, x : \tau \rangle\ \mathbf{in}\ e} \tag{106}$$

$$\frac{\Phi;\Sigma;\Delta \vdash_b b : \tau \qquad \Phi \vdash \tau = \tau' :: \mathsf{Type}}{\Phi;\Sigma;\Delta \vdash_b b : \tau'} \tag{107}$$

$$\frac{\Phi;\Sigma;\Delta \vdash_v v : \tau}{\Phi;\Sigma;\Delta \vdash_b v : \tau} \tag{108}$$

$$\frac{\Phi;\Sigma;x : \tau \vdash_c e}{\Phi;\Sigma;\circ \vdash_b \boldsymbol{\lambda} x : \tau.\, e : \neg\tau} \tag{109}$$

$$\frac{\Phi, \alpha :: \kappa;\Sigma;\circ \vdash_b b : \tau}{\Phi;\Sigma;\circ \vdash_b \boldsymbol{\Lambda}\alpha::\kappa.\, b : \forall \alpha::\kappa.\, \tau} \tag{110}$$

Storage

Storage $\quad S ::= \{(\ell \mapsto v)^*\}$

Heap and storage typing $\boxed{\vdash_s \langle H, S \rangle : \Sigma}$

$$\frac{\left.\begin{array}{ll}\circ;\Sigma;\circ \vdash_v S(\ell) : \tau & \text{if } \Sigma(\ell) = \mathbf{ref}\ \tau \\ \circ;\Sigma;\circ \vdash_v H(\ell) : \tau & \text{if } \Sigma(\ell) = \tau \neq \mathbf{ref}\ \tau'\end{array}\right\}_{\ell \in \mathrm{dom}(\Sigma)} \quad H(\ell) \neq \ell' \quad \forall \ell, \ell' \in \mathrm{dom}(H)}{\vdash_s \langle H, S \rangle : \Sigma} \tag{111}$$

Heap typing $\boxed{\vdash_h H : \Sigma}$

$$\frac{\vdash_s \langle H, \{\} \rangle : \Sigma}{\vdash_h H : \Sigma} \tag{112}$$

B.2 Dynamic Semantics

An operational semantics: primitive reduction and congruence.

Tracing type applications $\boxed{trace(b) = b'}$

$$trace(i) = i \tag{113}$$

$$trace(\ell) = trace(H(\ell)) \tag{114}$$

$$trace(\{(l = v)^*\}) = \{(l = v)^*\} \tag{115}$$

$$\frac{trace(v) = \mathbf{\Lambda}\alpha::\kappa.\, b}{trace(v\ [\tau]) = b[\alpha := \tau]} \tag{116}$$

$$trace(\langle \alpha::\kappa = \tau, v : \tau' \rangle) = \langle \alpha::\kappa = \tau, v : \tau' \rangle \tag{117}$$

$$trace(\textsf{fold}\ v\ \textsf{as}\ \mu\alpha::\kappa.\,\tau\ \textsf{at}\ \lambda\gamma::\kappa.\, s[\gamma]) = \textsf{fold}\ v\ \textsf{as}\ \mu\alpha::\kappa.\,\tau\ \textsf{at}\ \lambda\gamma::\kappa.\, s[\gamma] \tag{118}$$

$$trace(\mathbf{\lambda} x : \tau.\, e) = \mathbf{\lambda} x : \tau.\, e \tag{119}$$

$$trace(\mathbf{\Lambda}\alpha::\kappa.\, b) = \mathbf{\Lambda}\alpha::\kappa.\, b \tag{120}$$

Reduction $\boxed{S \mid p \hookrightarrow_H S' \mid p'}$

$$\frac{H(\ell) = \mathbf{\Lambda}\alpha::\kappa.\, b}{S \mid \ell\ [\tau] \hookrightarrow_H S \mid b[\alpha := \tau]} \tag{121}$$

$$\frac{S \mid p \hookrightarrow_H S' \mid p'}{S \mid p.l \hookrightarrow_H S' \mid p'.l} \tag{122}$$

$$\frac{v \neq \{l_1 = v_1, \ldots l_n = v_n\} \qquad trace(v) = v'}{S \mid v.l \hookrightarrow_H S \mid v'.l} \tag{123}$$

$$\frac{l = l_i \text{ where } i \in \{1 \ldots n\}}{S \mid \{l_1 = v_1, \ldots l_n = v_n\}.l \hookrightarrow_H S \mid v_i} \tag{124}$$

$$\frac{S \mid p \hookrightarrow_H S' \mid p'}{S \mid \textsf{cell}\ p \hookrightarrow_H S' \mid \textsf{cell}\ p'} \tag{125}$$

$$\frac{\ell \notin \mathrm{dom}(S) \cup \mathrm{dom}(H)}{S \mid \textsf{cell}\ v \hookrightarrow_H (S, \ell = v) \mid \ell} \tag{126}$$

$$\frac{S \mid p \hookrightarrow_H S' \mid p'}{S \mid !\, p \hookrightarrow_H S' \mid !\, p'} \tag{127}$$

$$\frac{S(\ell) = v}{S \mid !\, \ell \hookrightarrow_H S \mid v} \tag{128}$$

$$\frac{S \mid p \hookrightarrow_H S' \mid p'}{S \mid \textsf{unfold}\ p\ \textsf{as}\ \mu\alpha::\kappa.\,\tau\ \textsf{at}\ \tau' \hookrightarrow_H S' \mid \textsf{unfold}\ p'\ \textsf{as}\ \mu\alpha::\kappa.\,\tau\ \textsf{at}\ \tau'} \tag{129}$$

$$\frac{v \neq (\textsf{fold}\ v_1\ \textsf{as}\ \mu\alpha::\kappa.\,\tau\ \textsf{at}\ \tau') \qquad trace(v) = v'}{S \mid \textsf{unfold}\ v\ \textsf{as}\ \mu\alpha::\kappa.\,\tau\ \textsf{at}\ \tau' \hookrightarrow_H S \mid \textsf{unfold}\ v'\ \textsf{as}\ \mu\alpha::\kappa.\,\tau\ \textsf{at}\ \tau'} \tag{130}$$

$$\frac{}{S \mid \textsf{unfold}\ (\textsf{fold}\ v\ \textsf{as}\ \mu\alpha::\kappa.\,\tau\ \textsf{at}\ \tau')\ \textsf{as}\ \mu\alpha::\kappa.\,\tau\ \textsf{at}\ \tau' \hookrightarrow_H S \mid v} \tag{131}$$

$\boxed{S \mid q \hookrightarrow_H S' \mid q'}$

$$\frac{S \mid p \hookrightarrow_H S' \mid p'}{S \mid p := p_2 \hookrightarrow_H S' \mid p' := p_2} \tag{132}$$

$$\frac{S \mid p \hookrightarrow_H S' \mid p'}{S \mid \ell := p \hookrightarrow_H S' \mid \ell := p'} \tag{133}$$

$$\frac{\ell \in \mathrm{dom}(S)}{S \mid \ell := v \hookrightarrow_H S[\ell \mapsto v] \mid \langle\rangle} \tag{134}$$

$\boxed{S \mid e \hookrightarrow_H S' \mid e'}$

$$\frac{S \mid p \hookrightarrow_H S' \mid p'}{S \mid \textsf{jmp}\ p\ p_2 \hookrightarrow_H S' \mid \textsf{jmp}\ p'\ p_2} \tag{135}$$

$$\frac{S \mid p \hookrightarrow_H S' \mid p'}{S \mid \textsf{jmp}\ v\ p \hookrightarrow_H S' \mid \textsf{jmp}\ v\ p'} \tag{136}$$

$$\frac{trace(v) = v'}{S \mid \textsf{jmp}\ v\ v_1 \hookrightarrow_H S \mid \textsf{jmp}\ v'\ v_1} \tag{137}$$

$$\frac{trace(v) = \mathbf{\lambda} x : \tau.\, e}{S \mid \textsf{jmp}\ v\ v_1 \hookrightarrow_H S \mid e[x := v_1]} \tag{138}$$

$$\frac{S \mid q \hookrightarrow_H S' \mid q'}{S \mid \textsf{let}\ x : \tau = q\ \textsf{in}\ e \hookrightarrow_H S' \mid \textsf{let}\ x : \tau = q'\ \textsf{in}\ e} \tag{139}$$

$$\frac{}{S \mid \textsf{let}\ x : \tau = v\ \textsf{in}\ e \hookrightarrow_H S \mid e[x := v]} \tag{140}$$

$$\frac{S \mid p \hookrightarrow_H S' \mid p'}{S \mid \textsf{ifz}\ p\ \textsf{then}\ e\ \textsf{else}\ e' \hookrightarrow_H S' \mid \textsf{ifz}\ p'\ \textsf{then}\ e\ \textsf{else}\ e'} \tag{141}$$

$$\frac{v \neq i \qquad trace(v) = v'}{S \mid \textsf{ifz}\ v\ \textsf{then}\ e\ \textsf{else}\ e' \hookrightarrow_H S \mid \textsf{ifz}\ v'\ \textsf{then}\ e\ \textsf{else}\ e'} \tag{142}$$

$$\frac{}{S \mid \textsf{ifz}\ 0\ \textsf{then}\ e\ \textsf{else}\ e' \hookrightarrow_H S \mid e} \tag{143}$$

$$\frac{i \neq 0}{S \mid \textsf{ifz}\ i\ \textsf{then}\ e\ \textsf{else}\ e' \hookrightarrow_H S \mid e'} \tag{144}$$

$$\frac{S \mid p \hookrightarrow_H S' \mid p'}{S \mid \textsf{open}\ p\ \textsf{as}\ \langle \alpha::\kappa, x : \tau \rangle\ \textsf{in}\ e_1 \hookrightarrow_H S' \mid \textsf{open}\ p'\ \textsf{as}\ \langle \alpha::\kappa, x : \tau \rangle\ \textsf{in}\ e_1} \tag{145}$$

$$\frac{v \neq \langle \alpha::\kappa = \tau', v_1 : \tau \rangle \qquad trace(v) = v'}{S \mid \textsf{open}\ v\ \textsf{as}\ \langle \alpha::\kappa, x : \tau \rangle\ \textsf{in}\ e \hookrightarrow_H S \mid \textsf{open}\ v'\ \textsf{as}\ \langle \alpha::\kappa, x : \tau \rangle\ \textsf{in}\ e} \tag{146}$$

$$\frac{}{S \mid \textsf{open}\ \langle \alpha::\kappa = \tau', v : \tau \rangle\ \textsf{as}\ \langle \alpha::\kappa, x : \tau \rangle\ \textsf{in}\ e \hookrightarrow_H S \mid e[x := v][\alpha := \tau']} \tag{147}$$

B.3 Type Soundness

Theorem 5 (LFLINT Soundness) If $\Phi; \Sigma; \Delta \vdash_c e$ and $\vdash_s \langle H, S \rangle : \Sigma$, then $\exists (\Sigma', S', e')$ such that $S \mid e \hookrightarrow_H S' \mid e'$ and $\Phi; \Sigma'; \Delta \vdash_c e'$ and $\Sigma' \subseteq_s \Sigma$ and $\vdash_s \langle H, S' \rangle : \Sigma'$.

Proof sketch: By simultaneous induction on the structure of the derivations of $\Phi; \Sigma; \Delta \vdash_c e$ and $\vdash_s \langle H, S \rangle : \Sigma$.