Multi-stage Programming for Mainstream Languages

Edwin Westbrook  Mathias Ricken  Jun Inoue  Yilong Yao  Tamer Abdelatif\(^1\) Walid Taha

Rice University
\{emw4, mgricken, ji2, yy3\}@cs.rice.edu, eng.tamerabdo@gmail.com, taha@cs.rice.edu

Abstract
Multi-stage programming (MSP) constructs enable a disciplined approach to program generation. In the purely functional setting, it is possible to statically type-check MSP constructs to ensure that they can only generate well-typed programs. Despite numerous attempts, it has been difficult to extend this guarantee in the presence of key features of mainstream languages, especially imperative constructs.

This paper proposes a new method for achieving this guarantee and shows that it is powerful enough to express classic applications of MSP in Java. Our key insight is that safety can be regained by ensuring that the bodies of escapes are weakly separable from the rest of the code. This means that computational effects occurring inside an escape can only be visible outside the escape through types guaranteed to not contain code. Our method is simpler than prior proposals, and we expect that it can be intuitively understood by programmers. We formalize a calculus to demonstrate the soundness of the proposed approach. An implementation called Mint, which extends the Java OpenJDK compiler, is used to validate both the expressivity of the system and the performance gains attainable by using MSP in this setting.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms Languages

Keywords Multi-staged languages, Multi-stage programming, Type systems, Java

1. Introduction
Multi-stage programming (MSP) languages provide a hygienic quasi-quotation mechanism intended for program generation. Hygiene ensures that generated programs are free of accidental variable capture, a problem that makes using strings to generate programs in preprocessors like \(cpp\) notoriously hard to use. Research on functional languages has shown that it is possible to statically check MSP programs to ensure that they can only be used to generate well-typed programs [22, 23, 4]. Unfortunately, extending this static typing guarantee to mainstream languages has proved to be challenging. In particular, standard features of mainstream languages, such as imperative assignment, lead to scope extrusion, in which variables in code escape the scopes where they are defined.

Several approaches to solving this problem have been proposed. Two of these proposals use record polymorphism and index the types of code objects with their free variables [13, 1], while another one uses delimited control to express effects and to limit their scope [11]. These are powerful systems that give the expert MSP user fine-grained control over scoping in code. However, there is still a need for a type system that makes MSP accessible to general programmers and domain experts.

1.1 Contributions
To address this need, we propose a new approach to type-safe MSP. We argue that this approach is better suited for type-safe MSP in mainstream language than previous proposals. Our contributions include:

- A minimal language extension to support MSP in Java, combining the three standard MSP constructs with a reflection library of staged versions of the standard Java reflection classes (Section 2.2).

- The notion of weak separability, which limits the computational effects that can occur inside the bodies of escapes (Section 3). Weak separability is enforced by a small set of restrictions that ensure that any effects that can be observed outside escape expressions do not involve code objects. We expect that the restrictions will be easily and intuitively understood by mainstream programmers.

- Demonstration of the expressivity of a language with these restrictions through both standard pure examples and examples with imperative features (Section 4).

- A type system based on weak separability, an operational semantics that formalizes the runtime behavior of an object-oriented MSP language with effects, and a proof that running any well-typed program is guaranteed to be free of any runtime errors, including possible scope extrusion and generation (and execution) of ill-formed code (Section 5). Full proofs are available in Appendix A.

- An implementation of this proposal, published online at http://plresearch.org/JavaMint (Section 6). The implementation is based on the Java OpenJDK compiler from Sun Microsystems.

- Validation of the performance impact of MSP in Mint, showing that it is consistent with prior studies (Section 7).

1.2 Comparisons with Related Efforts
Several efforts have been made to accommodate effects in the context of multi-stage programming, as well as to accommodate object-oriented features. In what follows we summarize the most closely related efforts.

Early efforts to develop sound type systems for MSP languages with effects focused on introducing imperative features to functional MSP languages [22, 3, 2, 13, 11]. All of these support manipulation of open terms and guarantee well-formedness of the generated code, but they significantly differ in the approaches and extents to which they support effects. Calcagno et al. [3] allows imperative operations on codes but do not support imperative operations on open terms. Kim et al. [13] support unrestricted imperative op-
operations on open terms but choose not to provide α-equivalence for future-stage code. Their system delegates hygiene to a specialized binder \( \lambda \), whose operation can be explained only in terms of an implicit "gensym." They present an inferable polymorphic type system. Ancona and Moggi [2] incorporate imperative operations on open terms and provide hygiene. The imperative primitive in all of these, except for Kameyama et al. [11], are ML-style "boxed" references, which is not in line with Java semantics. Pervasive, unboxed references, an essential feature of Java, exacerbate the problem. (See Section 3 for a detailed discussion.) Kameyama et al. [11] use delimited control as their imperative primitive, which is more general than mutable stores. They maintain hygiene and support imperative operations on open terms, but they choose not to allow any side effect to occur inside a future-stage binder that is visible from the code outside.

Until recently, efforts to introduce MSP to the object-oriented setting focused on engineering aspects. The staged extensions of Java by Schultz et al. [16], Kamin et al. [12], and Zook et al. [24] focus on implementation, applications, and on quantifying the performance benefits. These extensions were not formalized. Neverov and Roe [14] formalize a core typed, Java-like calculus but leave the type soundness unproved. Their calculus also does not have side effects. Huang et al. [8] focus on reflection, and do not allow manipulation of arbitrary code values (in particular open terms). They prove soundness, but their system does not model side effects. Aktemur [1] and Kim et al. [13] rely on a form of record typing that makes the type and the type system complex.

As such, our approach is closest to that of Calcagno et al. [3], in which code values involved in effects are checked against certain closedness criteria. In contrast, we identify and solve the problem of finding an appropriate notion that can work with Java’s complex object model.

2. Programming in Mint

As noted earlier, Mint extends Java with three MSP constructs and a library of staged reflection primitives. The guiding principle in Mint's design is parsimony. In this section we introduce the design from the programmer's perspective.

2.1 Staging Constructs

Mint extends Java 1.6 with the three standard MSP constructs: brackets, escape, and run [22, 23, 4]. Brackets are written as `(| `) and delay the enclosed computation by returning it as a code object. For example, `(| 2 + 3 |)` is a value. Brackets can contain a block of statements if it surrounded by curly brackets:

```
{| { C.foo();
   C.bar(); } | >;
```

Code objects have type `Code<T>`, where `T` is the type of the expression contained. For example, `(| 2 |)` has type `Code<Integer>`. A bracketed block of statements always has type `Code<Void>`.

![Figure 1. The unstaged power function](image1)

```java
public static
Integer power(Integer x, Integer n) {
    if (n == 1)
        return x;
    else
        return x * power(x, n-1);
}
```

Figure 1. The unstaged power function

```
public static
Code<Integer> spower(Code<Integer> x, int n) {
    if (n == 1)
        return x;
    else
        return <| 'x * (spower(x, n-1)) | >;
}
```

```
public static abstract class PowerFun {
    public abstract int apply(int x);
}
```

```
Code<? extends PowerFun> CodePower17 =
<|
    new PowerFun() {
        public int apply(final int x) {
            return <| 'spower(<|x|>, 17));
        }
    } |
    PowerFun spower17 = CodePower17.run();
    int val = spower17.apply(2);
```

![Figure 2. The staged power function.](image2)

Code objects can be escaped or run. Escapes are written as `|` and allow code objects to be spliced into other brackets to create bigger code objects. For example,

```
Code<Integer> x = <| 2 + 3 | >;
Code<Integer> y = <| 1 + 'x | >;
stores <| 1 + (2 + 3) | > into y. Run is provided as a method run() that code objects support. For example, executing
```
```
    int z = y.run();
```
```
after the above example sets z to 6.
```
```
Basic MSP in Mint can be illustrated using the classic power function example. Figure 1 displays the unstaged power function in Java. Figure 2 displays a staged version. This staged method spower takes in an argument `x` that is a piece of code for an integer, along with an integer `n`, and returns a piece of code that multiplies `x` by itself `n` times.

To use spower we create code for an anonymous inner class PowerFun. The generated class, which is assigned to the variable CodePower17, has an apply method with a body that is generated by spower called with exponent 17. This creates code that multiplies the input by itself 17 times. The code CodePower17 is then compiled and run with the run() method, which produces a PowerFun and assigns it to spower17, and finally val is bound to the result of calling the apply method of spower17 on 2, computing 2\(^{17}\).

2.2 Staged Reflection Primitives

Neverov observed that staging and reflection in languages like C# and Java can be highly synergistic [14]. He also noticed that fully exploiting this synergy requires providing a special library of staged reflection primitives. Mint provides such a library. The primitives are based on those in the standard reflection primitives in the Java library, including the `Class<A>` and `Field` classes.¹

To represent these in Mint, the library adds two corresponding types, `ClassCode<A>` and `FieldCode<A,B>`. The `ClassCode<A>` type is indexed by the class itself, just like the type it is modelled after. For example, the corresponding class for `Integer` objects has type `ClassCode<Integer>`. Any `ClassCode<A>` object provides methods

¹The Mint reflection library does not support all reflection primitives. For example, the `Method` and `Constructor` require multiple arguments. This requires adding indexed types to Java, and is therefore outside the scope of this work.
for manipulating class corresponding to the methods of Class\(A\). For example, the cast method of ClassCode\(A\) takes any code object of type Code\(\langle\text{Object}\rangle\) and inserts an unsafe cast in the code object, yielding a code object of type Code\(\langle\text{A}\rangle\). Because the cast is inserted into the code, any exceptions raised by the cast will not happen until the code is run with the run() method. The class also provides methods for looking up a class by name and for retrieving the fields, methods, and constructors of a class.

The type FieldCode\(A,B\) represents a field in class \(A\) that has type \(B\). It provides a get method which takes a Code\(\langle A\rangle\) value and returns a value of type Code\(\langle B\rangle\). This method constructs field selection (intuitively, a \(\langle'f',\langle\text{I}\rangle\rangle\) code fragment) on that object. The type also provides a getType method to return a ClassCode\(\langle A\rangle\) object for the type \(B\).

The following example illustrates the use of these classes. The code defines a method fieldIter that uses the getField\(a\) method to iterate over all the statically known fields of an object of type \(A\):

\[\text{Code< Void> fieldIter(FieldFun fun, Code< A> o, CodeClass< A> clazz) \{\}}\]
\[\text{Code< Void> c = \langle\{\}\};}\]
\[\text{for (CodeField< A,T> f : clazz.getFields())} \]
\[\text{c = \langle\{'}c;\text{'(fun.call(f.get(o),}\}
\[\text{f.getTypeType()); \} \} >;}\]
\[\text{return c;}\]
\[\text{\}}\]
\[\text{interface FieldFun \{}\]
\[\text{\langle T\rangle // for any T}\]
\[\text{Code< Void> call(Code< T> c, CodeClass< T> t);}\]
\[\text{\}}\]

For each field in class \(A\), the method creates a projection of \(o\) for that field and passes it and the staged class object for the type of the field to fun.call(), where user-defined processing is performed. The code objects returned by the FieldFun.call method are accumulated in the code object \(c\). For example, calling fieldIter with the following FieldFun method recursively prints an object and all of its fields:

\[\text{public static class PrintFieldFun implements FieldFun \{}\]
\[\text{public separable \langle T\rangle Code< Void>}\]
\[\text{call(Code< T> c, CodeClass< T> t) \{}\]
\[\text{return \langle\{System.out.println('c'); \} \} >;}\]
\[\text{\}}\]

In Section 4 these two types will be used to implement a staged serializer in Mint.

3. The Scope Extrusion Problem

A key challenge in statically ensuring safety in imperative MSP languages is preventing scope extrusion. In MSP languages, free variables arise when escaped computations involve code fragments containing variables bound inside surrounding brackets. In other words, they arise in programs that build a code fragment containing a binding construct, and in the body of the binding construct there is an escaped computation that refers to a variable introduced by that binding construct. In the purely functional setting, this can never lead to scope extrusion. However, in the presence of effects, the escaped computation can leak the code value to, say, a global store. Generally, the store is taken to exist outside the scope of the term being evaluated, and therefore, there is no obvious way to associate a unique binder with a variable that occurs free in a code fragment in the store.

In this section, we review the basic types of errors that can arise in an MSP language, examine the scope extrusion problem in more detail, and explain why simple approaches are inadequate, and how the notion of separability can lead to a practical solution.

3.1 Basic Errors in Untyped MSP Programs

Three basic types of errors can arise in any language that supports staging constructs, namely, (1) running or escaping a non-code value, (2) using a variable before it is bound, (3) similar behavior that can result from using the run construct. Any sound type system must prevent these types of errors. An example of the first type can be seen in what follows:

\[\langle\{}\text{Code< Void> z = foo();}\]
\[\text{'(17); } \}\rangle\]

Because 17 is not a code value, this escape operation would fail.

An example of the second type can be illustrated with the following code:

\[\langle\{}\text{Code< Void> z = foo();}\]
\[\text{'(z); } \}\rangle\]

Incorrect use of the run construct can lead to essentially the same kind of error. For example, we can write code that effectively reduces to the same code we have above:

\[\langle\{}\text{Code< Void> z = foo();}\]
\[\text{'(\langle\{}z \rangle \text{}.run()); } \}\rangle\]

This is a classic example of how the run construct dynamically changes the level of a term.

3.2 When can Scope Extrusion Occur?

Scope extrusion occurs when any one of the following situations arises in the body of an escape:

1. Assigning a code object to a variable or field that is reachable outside the escape, for example:

\[\langle\{}\text{Integer y = foo();}\]
\[\text{Integer x = ' \langle\{}y \rangle }; \}\rangle\]

2. Throwing an exception that contains a code object, for example:

\[\langle\{}\text{Code< Integer> meth(Code< Integer> c) \{}\]
\[\text{throw new CodeContainerException(c);}\]
\[\text{\}}\]
\[\langle\{}\text{Integer y; ' \{meth(' \langle\{}y \rangle '); } \}\rangle\]

3. Cross-stage persistence (CSP) of a code object, an example of which is displayed in Figure 3.

The first two cases are traditional conditions for scope extrusion. The first example extrudes \(y\) from its scope by assigning \(\langle\{}y\rangle\) to the variable \(x\) bound outside of the scope of \(y\), while the second example throws an exception containing \(\langle\{}y\rangle\) outside the scope of \(y\).

The third case is more subtle. The call to doCSP in the example injects the anonymous inner subclass of Thunk into the returned code using CSP, yielding:

\[\langle\{}\text{new IntCodeFun()} \{\]
\[\text{Code< Integer> call(Integer y)\{}\]
\[\text{return T.call(\});}\]
\[\text{\}.call(1) \} \}

where \(T\) is the Thunk that returns \(\langle\{}y\rangle\). In a substitution-based semantics, calling \(T\) with \(1\) would return \(\langle\{}1\rangle\), and no scope extrusion would occur. However, at the more realistic level of an environment-based semantics that uses an environment to implement substitution efficiently, \(T\) would return the literal value \(\langle\{}y\rangle\); preventing this behavior would require the run() method
interface IntCodeFun {
  Code<Integer> call (Integer y); 
}
interface Thunk { Code<Integer> call (); }
classThunkCSPer {
  Code<Code<Integer>> doCSP (Thunk f) {
    return <| f. call () |>;
  }
  <| new IntCodeFun () {
    Code<Integer> call (Integer y) {
      return ' (ThunkCSPer. doCSP (new Thunk()) {
        Code<Integer> call () {
          return <| y | >;
        } |
      } |
    } |
  } |
}

Figure 3. Cross-stage Persistence of Code Objects

to traverse the definition of the call() method of \( \tau \) and to replace \(<| y | > \) with \(<| z | > \), which would be difficult in the JVM. This behavior, known as the “hidden free-variable problem,” arises commonly in environment-based implementations of multi-stage languages [19]. A substitution-based semantics does not allow us to capture it. The object hiding the free variable is usually a closure in \( \lambda \) calculus-based models, and when computation proceeds via substitution, the entire \( \lambda \) term (as well as open terms therein) is not moved off to an explicit heap, which makes scope extrusion impossible. To prevent this problem, the type system must place special restrictions on CSP with reference types. In a language with unboxed references, this must be extended to all (non-primitive) types, and its impact on the expressivity of the language is more pervasive.

3.3 Weak Separability

We can prevent the three situations mentioned in the previous section using the following notion:

**Definition 1.** A program fragment is separable if it is observable from the surrounding runtime environment only through its return value.

A separable program fragment appears purely functional; it does not have any side effects at all. Mint, however, does allow side effects as long as they only involve values that are code-free:

**Definition 2.** A type is code-free if all of its fields are code-free, and its class is final, meaning it is not allowed to be subclassed. A value is code-free if its type is code-free.

The requirement that a class is final ensures that a subclass with an additional field of type \( \text{Code}<\tau> \) cannot be substituted at runtime. Code-free types include number types such as \( \text{Integer} \) and \( \text{Double} \), the \( \text{String} \) class, arrays of code-free types, and all of Java’s reflection classes such as \( \text{Class} \) and \( \text{Field} \). It does not include \( \text{Object} \), as this type is not final. This is justified because an \( \text{Object} \) could be a code object at runtime.

We can now define the notion of weak separability that describes the code Mint allows inside escapes:

**Definition 3.** A program fragment is weakly separable if it is observable from the enclosing runtime environment only through its return value or through side effects involving only code-free values.

Requiring that code inside escapes is weakly separable is sufficient to prevent scope extrusion. This is proved in Section 5. The Mint type-checker enforces this by ensuring that the following hold of any term inside an escape:

1. Assignment is made only to variables bound within the term;
2. Exceptions are only thrown when the exception value is either an exception caught by a previous \texttt{catch} in the program fragment, or a constructor call \texttt{new C(e1, ..., en)} where the \( e_i \) are code-free;
3. Cross-stage persistence occurs only for \texttt{final} variables of code-free types;
4. Only methods and constructors whose bodies are weakly separable are called.

The first three clauses directly address the three cases of scope extrusion in the previous section. Note that the \texttt{final} restriction on CSP variables exists so that the value of the variable does not change over the lifetime of the code object; Java has a similar restriction for variables referenced inside anonymous inner classes.

The last clause ensures that all methods called from the body of a weakly separable program fragment also satisfy weak separability. To check this condition, methods that are going to be called from the body of an escape are explicitly annotated in Mint with the new keyword \texttt{separable}. Note that a call to \texttt{run()} is not weakly separable, so incorrect use of it as described in Section 3.1 is prevented as well.

Weak separability statically ensures that no code object created inside an escape can leak out of the escape. Thus, scope extrusion is not possible. The restrictions are easy to understand and follow, and the compiler can point out exactly where the errors are if the user violates them. We also believe the reasons behind them are simple to understand, given an explanation of scope extrusion. The remaining question is whether the system is too restrictive. This is answered in the following section.

4. Expressivity

Weak separability does not severely restrict expressiveness, and many useful MSP programs can be written in Mint. Intuitively, this is because code generators do not rely heavily on computational effects. Most classic applications of MSP, such as interpreters, use code generators that are purely functional. This does not mean that the generated code is functional, just that the generators are.

In addition, the \texttt{run()} method is only ever called at the top level in almost all applications of MSP, and cross-stage persistence is mostly used for primitive types.

To illustrate these points, the remainder of this section describes the implications of weak separability and examines a number of MSP examples in Mint, including: staging an interpreter, a classic MSP example; staging a for loop to do loop unrolling, demonstrating a generator for imperative code; and a staged serializer that uses Mint’s reflection capabilities. The performance of these examples is evaluated in Section 7.

4.1 Programming with Weak Separability

While classes used in CSP and in escapes need to be code-free, the restrictions that this places on programs can be avoided in most cases. In practice, there are two main difficulties. First, only weakly separable methods can be called from within escapes. This excludes most existing classes, such as those in the standard Java API, from being used in escapes. However, there is no restriction placed on the code generated by an escape, so the restriction is essentially on code generators themselves. We have yet to find a case when inseparable calls were necessary inside an escape.

The second difficulty is that subtype polymorphism cannot be used in CSP, because classes used in CSP need to be final. For
example, programs that use the Runnable interface to implement the command design pattern [7] cannot execute the commands abstractly if they use CSP, as in this example:

class MyCmd implements Runnable { ... }
public void someMethod() {
  Runnable cmd = new MyCmd();
  // error: Runnable not final
  Code<Void> cv = <| { cmd.run(); } | >;
}

We can regain the ability to perform dynamic dispatch by making subclasses be final and rewriting our program to use either static variables or final local variables, as follows:

final class MyCmd implements Runnable { ... }
public static Runnable cmd1 = new MyCmd();
public void someMethod() {
  final MyCmd cmd2 = new MyCmd();
  // ok: cmd1 is static, not CSP
  Code<Void> cv2 = <| { cmd1.run(); } | >;
  // ok: MyCmd is final
  Code<Void> cv3 = <| { cmd2.run(); } | >;
}

4.2 Staged Interpreter

Staged interpreters are a classic application of MSP [20, 21]. To demonstrate that staged interpreters can be written in Mint, we have implemented an interpreter for a small programming language called lint [20], which supports integer arithmetic, conditionals, and recursive function definitions of one argument.

The unstaged interpreter represents expressions with the Exp interface, and instantiates this interface with one class for each separable so that they can be called from inside an escape. Staging applications, function lookup is done using the FEnv.get(String s) method, returning an integer. In applications, function lookup is done using the FEnv.get(String s) method, returning a Fun object with an int apply(int v) method, which is then applied to the argument of the application.

interface Env { public int get(String s); } interface FEnv { public Fun get(String y); } interface Fun { public int apply(int param); }

Two empty environments, env0 and fenv0, unconditionally throw an exception in their get methods to signal a failed lookup. The environments are extended using the ext and fext methods. For instance, ext is

static Env ext(final Env env, final String s, final int v) {
  return new Env() {
    public int get(String y) {
      if (y.equals(s)) return v;
      else return env.get(y);
    }
  };
}

Recursive functions are implemented using anonymous inner classes to express closures. The code below creates a function environment fenv1 with the declaration of the identity function id(x) = x:

final Exp body = new Var("x");
FEnv fenv1 = fext(fenv0, "id", new Fun() {
  public int apply(final int param) {
    return (body.eval(ext(env0, "x", param),
                  fext(fenv0, "id", this)));
  }
});

The staged interpreter redefines the Env.eval method to return Code<Integer>, so that evaluating an expression yields code to compute its value. The variable environment returns Code<int>, and the function environment returns Code<? extends Fun>.

interface Exp {
  public escape_safe
  Code<Integer> eval(Env e, FEnv f);
} interface Env {
  public escape_safe
  Code<Integer> get(String y);
} interface FEnv {
  public escape_safe
  Code<? extends Fun> get(String y);
}

The return type of the FEnv.get method uses a wildcard with an upper bound of Fun. This is necessary since the type of the value produced by the code object is not exactly Fun, but rather a subtype of Fun.

The Exp.eval, Env.get, and FEnv.get methods are marked as separable so that they can be called from inside an escape. Staging the above AST classes yields the following:

interface Exp {
  public separable
  Code<Integer> eval(Env e, FEnv f);
} class Int implements Exp { /* ... */
  public separable

Evaluating a program is now a two-step process. The `eval` method now returns code for an integer, running that code returns the integer. Section 7 provides performance comparisons between staged and unstaged interpretation.

### 4.3 Loop Unrolling

As discussed above, weak separability does not restrict the computational effects in generated code; it does so only in the code generators themselves. As an example of this, we consider a code generator for loop unrolling, and how it can be used to unroll a loop with non-local side effects. We can write a generic loop in standard Java as follows:

```java
public static separable Code<Void> roll(int start, int stop, int step, SIter I) {
    Code<Void> c = <| {} | >;
    for(int x = start; x < stop; x += step){
        c = <| { 'c'; '(' I.iteration(x)); } | >;
    }
    return c;
}
```

This uses an interface called Iter to specify an arbitrary action for each iteration of the loop through the iteration method, which has return type void. To unroll this loop, we can stage the `roll` method as follows:

```java
public static separable Code<Void> unroll(int start, int stop, int step, SIter I) {
    Code<Void> c = <| {} | >;
    for(int x = start; x < stop; x += step){
        c = <| { 'c'; '(' I.iteration(x)); } | >;
    }
    return c;
}
```

This method uses an interface SIter to specify a code object for each iteration of the loop through the iteration method, which for SIter has return type Code<Void>. These code objects are accumulated into a code object `c` containing the sequence of statements for the whole loop. This code generator is written in an imperative style consistent with the prevailing Java culture. The body of this method is weakly separable because `c` is bound inside the method. The code object returned by `iteration` is not.

For example, the following class generates code that accumulates the indices used in the loop iteration into an object given by the `cell`:

```java
static class sIncrIter implements SIter {
    Code<IntCell> cell;
    public separable Code<Void> iteration(final int i) {
        return <| { ('cell').value += i; } | >;
    }
}
```

### 4.4 Serializer Generator

A serializer is a program that recursively converts an object and all of its fields to a string representation. Serializers are often slow, however, because they must use Java’s reflection primitives to determine the fields of an object at runtime. Here we show how to write a staged serializer, which generates a serializer for a given static type. This approach performs the necessary reflection when the serializer is generated, and then generates code to serialize all of a given object’s fields:

```java
public static separable <A> Code<Void> serialize(ClassCode<A> type, final Code<A> obj) {
    if (type.getCodeClass() == Byte.class)
        return <| {
            writeByte('('((Code<Byte>)obj));
        } | >;
    else return new Var("x").
    FEnv env = fext(fenv0, "id", <| new Fun {} | >)
    final Fun fthis = this;
    return <| {
        fbody.eval(extend(env0, "x", <| FParam | >),
                   fext(fenv0, "id", <| fthis | >));
    } | >;
}
```
The key lemma involved in this approach is the Smashing Lemma, ensuring that scope extrusion cannot occur through assignments. Only refer to locations in later frames if the latter are code-free, when new variables are bound (i.e., non-AIC) classes because one can write disjoint sets of names gives a simpler system. Earlier frames can be prevented by our system.

Given sequences $e_i$, their concatenation is written $e_1 e_2$. The syntax of LM is given in Figure 4. Expressions are stratified into levels. An expression is at level $n$ if, for every point in the expression, the nesting of escapes is at most $n$ levels deeper than brackets. Clearly, a level-$n$ expression is also a level-($n+1$) expression. This stratification induces a similar structure on method declarations. A complete program must not have any unmatched escapes, so the bodies of methods declared in the class hierarchy are required to be at level 0. Likewise, the initial expression in a program is required to be at level 0. Values are also stratified: a value at level 0 is just a heap location, and a value at level $>0$ is any lower-level expression.

5. Type Safety

We now turn to formalizing a subset of Mint, called Lightweight Mint (LM), and to proving type safety. Type safety implies that scope extrusion is not possible in Mint.

LM is based on Lightweight Java [18] (LJ), a subset of Java that includes imperative features. LM includes staging constructs (brackets, escapes, and run), assignments, and anonymous inner classes (AICs). These features—especially the staging constructs and AICs—make the operational semantics and type system large; staging constructs alone double the number of rules in the operational semantics, while AICs increase the complexity of the type system. All of these features, however, are necessary to capture the additional semantics, while AICs make the operational semantics and type system large; additional fields or methods can be emulated by declaring primitive fields. This completely disallows assignments in escapes, however, to local variables in LM. All assignments must instead be to object fields. This example was inspired by a similar example in the Metaphor paper [14].

Let $x = \text{new} \ C (...) \ in \ ...$

which always allocates a new instance of a class $C$ which is not an AIC. We then relax the restrictions on escapes to allow field assignments if the object containing the field was allocated by a let inside the escape. Local variable assignment can then be modeled by replacing any local variable binding $x$ of type $C$ for which there is an assignment by a let-binding of a new variable $x$ of type CCell, defined as follows:

```java
public class CCell { public C x; }
```

Uses of $x$, including assignments to $x$, can then be replaced by uses of $x$.x.

extensible class names $D$

final class names $F$

variables $x$

field names $f$

method names $m$

heap locations $l$

classes $C ::= D | F$

separability marker $S ::= \text{sep} | \text{insep}$

types $\tau ::= C \mid \text{Code}(S, \tau)$

class declarations $CL ::= \text{class } C \text{ extends } D$

method declarations $M^n ::= C \tau m(e^n)$

class hierarchy $P ::= (CL)_i$

programs $p ::= P; e^0$

expressions $e^n ::= x | l | e^n.f | (e^n.f := e^n)$

values $v^n ::= l | e^n-1[n > 0]$

NB: Production rules marked $[n > 0]$ can be used only if $n > 0$.

5.1 Syntax

In this section, we formalize the syntax of LM. We use the following sequence notation:

**Notation.** We write $\langle A_i \rangle_{i=1}^n$ for a sequence with index $i$ ranging over $I..J$, inclusive. $I$ may be omitted, and it defaults to 1. The superscript is omitted in addition if the index range is clear from context. In general, sequences indexed by different variables have different bounds. The sequence may be explicitly written out like $\langle a, b, c, \ldots \rangle$ with no subscript. The empty sequence is written $\langle \rangle$. Given sequences $s_1$ and $s_2$, their concatenation is written $s_1 \circ s_2$.

We may write $\langle A \rangle_i$, $A$ to mean $\langle A \rangle_i \circ (\langle A \rangle_i)$ if the intention is clear.

The syntax of LM is given in Figure 4. Expressions are stratified into levels. An expression is at level $n$ if, for every point in the expression, the nesting of escapes is at most $n$ levels deeper than brackets. Clearly, a level-$n$ expression is also a level-($n+1$) expression. This stratification induces a similar structure on method declarations. A complete program must not have any unmatched escapes, so the bodies of methods declared in the class hierarchy are required to be at level 0. Likewise, the initial expression in a program is required to be at level 0. Values are also stratified: a value at level 0 is just a heap location, and a value at level $>0$ is any lower-level expression.

We categorize classes as final ($F$) or extensible ($D$) depending upon their names. In the implementation, they are rather categorized according to the manner in which they are declared, but using disjoint sets of names gives a simpler system. Code$(S, \tau)$ falls under neither classification. We do not allow an AIC to have fields or methods that its parent does not, although we allow method overrides. Additional fields or methods can be emulated by declaring (statically) a new subclass with those fields and creating anonymous subclasses of those.

We do not include the syntax ($\text{new} \ C (...) \ for$) for instantiating ordinary (i.e., non-AIC) classes because one can write ($\text{let } x \leftarrow \text{new} \ C (...) \ in x$) instead. Sequencing ($e_1 ; e_2$) is also omitted be-
cause this sequence can be written \( \text{seq.call}(e_1, e_2) \), where \( \text{seq.call} \) is a method that ignores its first argument and returns its second.

As a technical point, the code type is indexed by a separability marker which indicates whether a code object is itself separable. Specifically, \text{Code}(\text{sep}, \tau)\) is the type of code objects containing separable code, which is a subtype of the standard code type, written \text{Code}(\text{sep}, \tau)\). This distinction is necessary in the case of a separable expression which itself contains a nested escape \( \ast e \), since we must know for type preservation that \( \ast e \) is guaranteed to reduce only to separable code. In this case, \( e \) must have type \text{Code}(\text{sep}, \tau)\).

All judgments and functions in the following discussions implicitly take a class hierarchy \( P \) as a parameter. We avoid writing it out explicitly because it is fixed for each program and there is no fear of confusion.

### 5.2 Operational Semantics

Figure 5 shows preliminary definitions that we need for the operational semantics. A heap is a finite mapping from locations to heap elements, where a heap element contains a runtime type tag with either the contents of the object or a code value if the tag is \text{Code}. We use the phrase pseudo-expressions to refer to syntactic elements that are either expressions or method declarations, and similarly we use pseudo-values to refer to values or method declarations.

An evaluation context \( E_{n,k} \) is indexed by two levels, the level \( n \) outside of the context and the level \( k \) inside. The intent is for any well-typed level-\( n \) expression to be decomposed uniquely as \( E_{n,k} \) where \( \tau^k \) is a reduce at level \( k \); unless the expression is a (level-\( n \)) value. There are two variants of evaluation contexts, one that yields an expression \( E_{n,k} \) when plugged in, and one that yields a method declaration \( E_{n,k} \). Both variants can be plugged with expressions only.

The function fields() extracts the fields of a type, while the method() function looks up a method. method() respects the method overriding rules. mbody() extracts the specified method’s formal arguments and body. Code types do not have methods (run() is formally not a method). mname() extracts the method name from a method declaration.

Figure 6 shows the small-step semantics for Lightweight Mint. This is given as the judgment \( H_{1,1} e_1 \xrightarrow{\text{pin}} H_{2,2} e_2 \) stating that heap \( H_1 \) and expression \( e_1 \) take a single step at level \( n \) to heap \( H_2 \) and expression \( e_2 \). This judgment is the closure under \( n \), \( k \)-evaluation contexts of the primitive one-step relation \( \xrightarrow{\text{pin}} \) at level \( k \). Most of the primitive reduction steps are straightforward, including rules for class instantiation, method invocation, and assignment. These reductions only occur at level \( 0 \), to prevent reductions from occurring inside code objects. Since local variables are immutable, we model method invocation and \( \lambda \text{-form} \) execution by substitution. The local binding \( L \) found in LJ [18] and similar formalisms is therefore unnecessary, and the small-step judgment is made between heap-term pairs rather than bindings-heap-term triples. Note that using substitution is not the same as a substitution-based semantics such as discussed in Section 3.2, because substitution here does not substitute into data in the heap.

There are also three staging-related reduction rules, for escape, run, and brackets. The rules for escape and run remove an expression from its brackets, with the only difference being that escape reduces only at level \( 1 \) (escape is illegal at level \( 0 \)) and run only reduces at level \( 0 \). These are standard in multi-stage languages [19], except that the code values are on the heap. The rule for brackets allocates a code object on the heap, CSP, which can be regarded as execution at arbitrarily high levels, is automatically taken care of by substitution and does not give rise to a redex.

### 5.3 Type System

Figure 7 gives preliminary definitions for the type system. A variable typing (or type environment) comes in pairs, separated by a \( \cdot \). The predicate iscf(\( \tau \)) means that \( \tau \) is code-free. Note that iscf(\( f \)) is defined co-inductively. The auxiliary functions ftypes(\( f \)), ftype(\( f \)), and mtype(\( f \)) are similar to those defined for the operational semantics, but they extract type information.

Figure 8 shows the type system. The top-level judgment \( \vdash p \) asserts that program \( p \) is well-formed. This ensures that \( p \) is a valid “initial state” of execution: the class hierarchy \( P \) contained in \( p \).
let \( x \leftarrow \text{new } C(l_0) \) in \( e^0 \) 

\[
\frac{H(l) = \langle \text{Code}, \{e^0\} \rangle; \ \ H(l) = \langle \text{Code}, \{e^0\} \rangle; \ \ H, l, \\text{run}() \xrightarrow{\text{prim}} H, e^0}{l \not\in \text{dom } H}
\]

must be well-formed; the expression \( e \) contains in \( p \) must be well-typed; and \( e \) must contain no store locations. This last check must be explicitly added here, because the typing rules for \( \text{let} \) forms and AICs allow frames to be pushed onto the stack of store typings. A class hierarchy \( P \) is well-formed, given by judgment \( \vdash P \), if \( P \) is acyclic, field names and types (including inherited ones) do not clash within each class, and each class is well-formed. We omit a formalization of the first two checks but will use them implicitly by assuming that auxiliary functions like fields(\( \tau \)), mtype(\( m, \tau \)) are always unambiguous and that the sequence returned by fields is finite and has no duplicates. Classes are well-formed if they contain no locations, their methods are well-typed, and any methods they share with their superclass have the same type as in the superclass.

The bottom half of Figure 8 concerns typing for pseudo-expressions. This is given by the judgment \( \langle \Sigma_1 \rangle; \Gamma \vdash^n \varepsilon^n : \tau | S \) which states that the pseudo-expression \( \varepsilon^n \) has type \( \tau \) at level \( n \) under the stack \( \langle \Sigma_1 \rangle \) of store typings and the pair \( \Gamma \) of contexts. If \( S = \text{sep} \), this judgment further states that the pseudo-expression \( \varepsilon^n \) is weakly separable. The reason the variable typing \( \Gamma \) is partitioned into two parts is to check weak separability: the right part of \( \Gamma \) contains the variables that were bound within the current method or enclosing escape. These are the variables whose fields can be assigned to without violating weak separability. We always assume that no variables are repeated in \( \Gamma \) and no locations are repeated in \( \langle \Sigma_1 \rangle \).

Most of the rules for typing pseudo-expressions are straightforward. The first rule generalizes subtypes to supertypes. The next two rules look up the types for variables and locations in the context and store typing, respectively, where CSP is only allowed (by allowing \( k \) or \( n \), respectively, to be non-zero) if the associated type is code-free. Further, in order for a variable to be typed as separable, it must occur in the second half of the context pair. The next rule types \( \text{let} \)-expressions by extending the current context with \( \text{let} \)-variables and no locations are repeated in \( \langle \Sigma_1 \rangle \).

The next three rules type field assignments \( x : \Sigma \rightarrow \tau \). If \( x \) is a variable in the right half of \( \Gamma \), then typing \( \Sigma \) to \( \tau \) at level \( n \) requires \( \tau \) to be code-free if the assignment is to be weakly separable. The second and third rules for assignments allow the assignment to be weakly separable if either \( e_1 \) is a variable in the right half of \( \Gamma \), or \( e_1 \) is a location in the last frame of the store typings and the whole assignment is typable at level 0, respectively.
Figure 8. Type system for Lightweight Mint.
The next rule, after those for assignment, types method calls by looking up the type of the given method, while the following rule types AICs by checking the class definition and the argument types. Finally, the last three rules type brackets, escape, and run, where typing \( \{ e \} \) requires typing \( e \) at the next level and adds the code type, typing \( e . c u n \) types \( e \) at a code type on the previous level and removes the code type, and typing \( e . c u n \) types \( e \) at a code type on the same level and removes the code type. Brackets can always be weakly separable, run is never weakly separable, and escapes \( e \) are only weakly separable if \( e \) has type \( \text{Code}(\text{sep}, \tau) \).

The remainder of Figure 8 has rules for the following judgments.

The judgment \( \langle \Sigma \rangle_i; \Gamma \vdash^n (H, e^n) : \tau|S \) states that an AIC that subclasses \( D \) with method definitions \( \langle M \rangle^n \) is well-formed. This requires the methods \( \langle M \rangle^n \) to have the appropriate types. It also requires, if \( n = 0 \), that all the locations in the AIC are contained in \( \text{dom}(\cup, \Sigma_i) \), effectively ensuring that no new frames can be added to the stack of store typings. The judgment \( \langle \Sigma \rangle_i; \Gamma \vdash^n \text{M}^n : \langle\tau\rangle_i \overset{\tau}{\rightarrow} \tau|S \) states that method \( M \) has input types \( \langle\tau\rangle_i \), output type \( \tau \), and further is weakly separable if \( S = \text{sep} \). Note that this rule is allowed to push a new frame onto the stack of store typings when the level \( n \) is positive. This is because there may be some locations in the store that contain code that include the free variables bound inside \( M \). Note also that passing inside a method resets the vertical bar \( | \) in \( \Gamma \) to the end, indicating that weakly separable expressions in the method cannot freely access variables bound at or before the method \( M \).

The judgment \( \langle \Sigma \rangle_i; \Gamma \vdash H \) states that the store \( H \) is well-formed under the given stack of store typings. This judgment includes the typing context \( \Gamma \) because the store may contain code with free variables. This judgment requires that, for all locations \( l \) in the stack of store typings, the heap for \( H(l) \) is well-typed. Note that there may be more locations in \( H \) than in the domain of \( \langle \Sigma \rangle_i \), allowing the possibility that other frames could be pushed onto this stack.

The judgment \( \langle \Sigma \rangle_i; \Gamma \vdash h : \tau \) is then used to state that heap form \( h \) has type \( \tau \). The rules for this judgment require that the expressions contained in the heap form \( h \) are well-typed. The typing context used to type these expressions is the restriction of \( \Gamma \) to the variables of level greater than 0. This is because heap forms are allowed to have code objects with free variables in them, but these free variables must be bound in other code objects, meaning they must have been bound at level greater than 0. Note that, as a side effect of these definitions, if \( \langle \Sigma \rangle_i; \Gamma \vdash H \) holds then \( H \) restricted to \( \text{dom}(\cup, \Sigma_i) \) is closed under reachability, meaning that no location in this domain can reference a location outside of it.

5.4 Soundness

Type soundness is proved by the usual Preservation and Progress lemmas. Progress is proved with the following lemma:

**Lemma 1 (Unique Decomposition).** If \( \langle \Sigma \rangle_i; \Gamma \vdash^n \alpha^n \tau_S : \tau|S \) and \( \alpha^n \) is not a pseudo-value then \( \alpha^n \) is uniquely decomposed as \( \alpha^n = \varepsilon^n, m^n[r^n] \), where \( \varepsilon \) denotes syntactic equality modulo \( \alpha \) conversion.

**Proof.** By straightforward induction on \( \alpha^n \).

Our statement of Unique Decomposition implies Progress because any well-typed expression is either a value or contains a redex that can be contracted by the operational rules. In addition, uniqueness also ensures that our semantics is deterministic.

The proof of Preservation is more complicated. One technical difficulty is that there is no restriction on the additional frames that may be introduced by the typing rule for methods; i.e., this rule could add locations to the store typing that are not in the current heap. To address this problem, we introduce typing for configurations, or pairs of heaps and pseudo-expressions. The judgment \( \langle \Sigma \rangle_i; \Gamma \vdash^n (H, e^n) : \tau|S \) then specifies that the configuration \( (H, e^n) \) is well-typed. The rules for this judgment are identical to those for pseudo-expression typing except that each rule also requires the heap \( H \) to be well-formed with respect to the current environment \( \langle \Sigma \rangle_i; \Gamma \).

For example, the rule for let forms becomes:

\[
\frac{\langle \Sigma \rangle_i; \Gamma \vdash^n (H, e^{n}) : \text{ftype}(\ell) |S \rangle \quad \langle \Sigma \rangle_i, \Sigma, \Gamma, x : C^n \vdash^n (H, e^n) : \tau |S \quad \langle \Sigma \rangle_i; \Gamma \vdash H \quad \text{let} \ x \leftarrow \text{new} \ C(e^n) \quad \text{in} \ e^n \ : \tau |S}
\]

A second technical difficulty is that a reduction step inside a let form or AIC that pushes a new frame onto \( \langle \Sigma \rangle_i \) might modify a code-free location in \( \text{dom}(\cup, \Sigma_i) \) to reference a location in the new frame \( \Sigma \). The resulting heap would thus not be well-formed under \( \langle \Sigma \rangle_i \), because this portion of the heap would not be closed under reachability. To deal with this problem requires the Smashing Lemma, which knocks the last two \( \Sigma \)'s of \( \langle \Sigma \rangle_i \) into one, giving a shorter store typing sequence.

**Lemma 2 (Smashing).** If

1. \( \langle \Sigma \rangle_i, \Sigma_1, \Gamma_1; \Gamma_2 \vdash H_1 \)
2. \( H_1|L = H_2|L \) where \( L = \text{dom}(\cup, \Sigma_i) - \text{dom}(\ell, \Sigma_i) \)
3. \( \langle \Sigma \rangle_i, \Sigma_1, \Gamma_1; \Gamma_2 \vdash H_2 \)
4. \( \Gamma_1 \cup \Gamma_2 \supseteq \Gamma_1 \cup \Gamma_2 \)

then \( \langle \Sigma \rangle_i|^{\ell-1}, (\Sigma_1 \cup \text{cf}(\Sigma)); \Gamma_1; \Gamma_2 \vdash H_2 \).

Note that the Smashing Lemma is at the heart of proving the absence of scope extrusion, as it states that any code locations that could potentially cause scope extrusion are not reachable outside their respective scopes.

We are now ready to prove Preservation. The statement below is an abridged version. For technical reasons, we need to add some more hypotheses and conclusions to make the proof work. The details of these technicalities are left to the Appendix.

**Lemma 3 (Preservation).** If \( \langle \Sigma \rangle_i, \Sigma_1, \Sigma_2; \Gamma_1; \Gamma_2 \vdash^n (H_1, e_1^n) : \tau |S \) and \( (H_2, e_2^n) \) then \( \Sigma \) such that

1. \( \Sigma_0 \supseteq \Sigma_R \)
2. \( \langle \Sigma \rangle_i, \Sigma_0; \Gamma_1; \Gamma_2 \vdash^n (H_2, e_2^n) : \tau |S \)
3. \( H_1|L = H_2|L \) where \( L = \text{dom}(\cup, \Sigma_i) - \text{dom}(\ell, \Sigma_i) \)

Proof is by induction on the typing judgment, and is given in the appendix.

6. Implementation

To verify the expressivity of the design and obtain performance results, we created an implementation of Mint by modifying OpenJDK [15], a Java Development Kit (JDK) based entirely on open source. Since we only modified the compiler and maintain full binary compatibility, the generated class files can be executed with any Java Runtime Environment, version 6 or higher. The only change required when running multi-stage programs is the placement of a small library on the boot classpath, making the compiler for future-stage code available.

The compiler included in OpenJDK contains a pretty printer geared towards converting abstract syntax trees (ASTs) to Java source that can be compiled again. By using the pretty printer, we are able to generate source for future stages with minimal changes to the compiler. The fact that we are generating human-readable source also has debugging benefits. In the future, the performance of compiling code objects may be increased by serializing and de-serializing the ASTs directly, thereby circumventing the compiler’s parser.

After the source input has been parsed and entered into symbol tables, the OpenJDK compiler without our modifications proceeds...
in five phases. The Mint compiler adds a sixth stage, called Staging Translation, yielding the following stages in order:

- **Attribution**: Names and expressions in the AST are resolved and types are assigned to the AST nodes. Most type errors are detected at this stage.
- **Flow Analysis**: Unreachable code and the use of uninitialized variables is detected.
- **Staging Translation**: Brackets are translated into ASTs that create code objects. This phase was introduced in the Mint compiler and does not exist in the original OpenJDK.
- **Type Translation**: Generic type information is erased.
- **Lowering**: “Syntactic sugar” like inner classes and foreach loops are replaced by simpler constructs.
- **Generation**: Bytecode is generated for the AST and class files are written.

The main modifications to the OpenJDK compiler, other than adding an additional compilation stage, were in Attribution. In Attribution, we perform the type-checking necessary for brackets, escape, and run, ensuring specifically that the body of each escape is weakly separable. Attribution also checks the separability of methods and constructors declared with the separable modifier and reports errors if unsafe operations are performed. Finally, Attribution also records the stage at which a variable is defined and the stage it is used: If the variable is used at a later stage than it is defined, it will be prepared for cross-stage persistence (CSP). If the variable is used in an earlier stage than the one it is defined in, an error is reported.

During Staging Translation, the new compiler stage in the Mint compiler, each bracket is replaced with a constructor call to MSPTreeCode<T>, a concrete implementation of Code<T>, which is given as an interface in Mint. The body of the bracket is passed to the constructor for MSPTreeCode as a simplified tree in which most of the AST has been converted into strings using the pretty printer; only escapes, CSP variables, and variable identifiers to be gensym-renamed are maintained as separate nodes.

Mint also extends Java to include a `let` construct to bind values in an expression, as opposed to a statement block. For instance, the expression `let int x=1, y=2*x; 3*y` is translated into a data structure containing:

- the string "let int "
- a gensym for `lv`
- the string "= 1; 2 * "
- the AST of CSP variable `csp`
- the string " + 3 * "
- the AST of escaped expression `c`
- the string " + "
- and a gensym for `lv`.

The node that represents an escape in a bracket body stores the AST of the expression that was escaped; a reference to a CSP variable contains the AST of the identifier. Furthermore, all variables introduced inside brackets are gensym-renamed [5]. For each such variable that needs to be renamed, a `let` expression binds a dynamically created, fresh name to a string variable (gensym$$1$$ in the example below), and the value of that string variable is used wherever the identifier to be renamed used to occur.

More concretely, the code generated for the last bracket in the example above is approximately as follows:

```java
Code<Integer> x = let
final String gensym$$1$$ = varGenSym();
new MSPTreeCode(new InteriorNode(
    new StrTree("let int ",
    new StrTree(gensym$$1$$),
    new StrTree("= 1; 2 * ",
    new CSPTree(csp),
    new StrTree(" + 3 * ",
    MSPTreeCode.escape(c),
    new StrTree(" + "),
    new StrTree(gensym$$1$$))));
```

This code first creates a gensym called gensym$$1$$ for the variable `lv`. It then creates an MSPTreeCode containing a tree of all the objects mentioned above: StrTree is used for nodes containing strings, the string given by gensym$$1$$; CSPTree is used for CSP variables; and MSPTreeCode.escape(c) is used to implement escapes, by copying the tree contained in the code object `c`.

When a code object is run, the proper values are filled in for escapes, CSP variables, and gensym-renamed identifiers. The entire tree is flattened and pasted into a template to create the Java source of a class implementing Code<T>, with the bracket’s body in its run method. The name for this class is also generated fresh.

Escapes and gensym-renamed identifiers are simple to process: The subtrees included by escapes are processed recursively, and renamed identifiers are treated like strings. CSP variables, on the other hand, are not in scope inside the new code object and need to be treated specially: the code object contains an Object array, called the CSP table, that is initialized with the values of the CSP variables in the constructor. References to CSP variables are replaced with array accesses.

The source that is compiled when the code object `x` in the example above is run looks like this:

```java
public class $$Code1$$ implements SafeCode<Integer> {
private Object[] ct;

public $$Code1$$(Object[] t) { ct = t; }

public Integer run() {
return (let int var$$$1$$ = 1;
    2 * ((Integer)ct[0]) +
    3 * (123) + var$$$1$$);
}
}
```

The fresh symbol `var$$$1$$` has been substituted for all occurrences of variable `lv`. The body of the escaped code object `c` is present without overhead, as desired. The reference to the CSP variable `csp` has been replaced by an access to the CSP table `ct`.

The source is passed as a string to the Mint compiler, where it is parsed, analyzed and translated as described above. The compiler then generates bytecode in memory. Since a single compilation unit in Java may be compiled into several class files, e.g. because of inner classes, the compiler returns a set of class name-bytecode pairs. Since anonymous inner classes are assigned names in the
compiler using an internal numbering scheme, the class names have to be returned along with the generated bytecode.

The bytecode for the generated classes is added to a hash table, with the class names used as keys. A custom class loader intercepts attempts by the Java virtual machine (JVM) to load a class and checks if the hash table has bytecode available for the requested class. If so, the bytecode generated by the Mint compiler is used; otherwise, the custom class loader uses Java’s default class loader, which attempts to load the class from a file.

A new instance of the generated class is created using reflection and the values of the CSP variables are passed to the constructor, filling the code object’s CSP table. Finally, the new instance’s run method is called to execute the code in the bracket.

It is important to use the custom class loader for all classes, not just those generated from brackets. If the same class is loaded by different class loaders, the JVM considers their instances incompatible and throws a ClassCastException if an object is assigned to a variable of the same class loaded by another class loader. This problem can be avoided by installing the custom class loader in a small launcher application before the program’s main method is executed. The launcher is included in the runtime library, together with with the Mint compiler and the Code<T> and SafeCode<T> interfaces.

### 7. Performance

In order to measure the performance impact of MSP in Mint, we have benchmarked a set of Mint examples. These include the following:

- `power` is the power example from Section 2.1, called with base 2 and exponent 17.
- `fib` recursively computes the 17th element of the generalized Fibonacci function starting from 2 and 3.
- `mmult` performs an optimized matrix multiplication, in which every 1 in the left matrix omits the floating-point multiplication at runtime and every 0 omits the multiplication and the addition. The benchmark is called with a four-dimensional rotation matrix as the left matrix and an arbitrary four-by-four matrix as the right.
- `eval-fact` calculates the factorial of 10 using the `lint` interpreter discussed in Section 4.2.
- `eval-fib` calculates the 10th number in the standard Fibonacci sequence using the `lint` interpreter.
- `unroll` performs the loop unrolling example of Section 4.3, using the accumulator discussed in that section over 10 iterations.
- `serialize` uses the serializer generator discussed in Section 4.4 to write the primitive fields contained in an object hierarchy two levels deep to an output stream.

Each operation in the benchmarking process (staged, gencode, compile, unstaged) is run for a number of repetitions so that the total time for that operation is 1-2 s. The average runtime of a single repetition is then calculated for each operation. Timings were recorded on an Apple MacBook with a 2.0 GHz Intel Core Duo processor, 2 MB of L2 cache, and 2 GB main memory, running Mac OS X Tiger.

The results are given in Figure 9. Performance improved in all cases. The speedups achieved range from 1.4 to 18.3, with speedup defined as unstaged time divided by staged time. The `mmult` and `unroll` benchmarks involved mostly tight for loops and could not be sped up substantially. On the other hand, the staged versions of `power` and `fib` reduced the call overhead involved in the recursive functions and executed almost five times faster than the unstaged code. Staging the `lint` interpreter improved the performance of the `eval-fact` and `eval-fib` benchmarks by about an order of magnitude. Finally, the `serialize` benchmark benefited the most from staging: the removal of call overhead and reflection reduced the execution time by a factor of 18.3. Note that the compiler overhead is currently significant. In the future, we hope to reduce this by circumventing the compiler’s parser.

#### Figure 9. Benchmark results.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>speed-up</th>
<th>unstaged $\mu$s</th>
<th>staged $\mu$s</th>
<th>gen $\mu$s</th>
<th>compile ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>power</td>
<td>4.6x</td>
<td>0.079</td>
<td>0.017</td>
<td>1.3</td>
<td>33</td>
</tr>
<tr>
<td>fib</td>
<td>4.1x</td>
<td>0.070</td>
<td>0.017</td>
<td>8.2</td>
<td>35</td>
</tr>
<tr>
<td>mmult</td>
<td>1.5x</td>
<td>1.8</td>
<td>1.3</td>
<td>12.0</td>
<td>84</td>
</tr>
<tr>
<td>eval-fact</td>
<td>8.4x</td>
<td>0.83</td>
<td>0.1</td>
<td>1.7</td>
<td>37</td>
</tr>
<tr>
<td>eval-fib</td>
<td>10.0x</td>
<td>19.0</td>
<td>1.9</td>
<td>2.4</td>
<td>57</td>
</tr>
<tr>
<td>unroll</td>
<td>1.4x</td>
<td>0.140</td>
<td>0.097</td>
<td>2.9</td>
<td>47</td>
</tr>
<tr>
<td>serialize</td>
<td>18.0x</td>
<td>1.5</td>
<td>0.08</td>
<td>6.2</td>
<td>35</td>
</tr>
</tbody>
</table>

#### 8. Related Work

Finding a practical static type system for safe imperative MSP has been a long-standing challenge. One of the earliest approaches to the problem introduced the notion of "closedness types" [3], which express that a code object has no free variables. Effects in this system are limited to closed code, so that no scope extrusion is possible. Two possible drawbacks of this approach are that (1) it requires additional annotation to mark closed code, and (2) there are cases for which this constraint can be too restrictive.

Recently, Kameyama, Kiselyov, and Shan introduced a new approach to dealing with imperative MSP using delimited control [10, 11]. While the approach makes use of control features (shift and reset), the essence of this approach is very similar to weak separability. Instead of limiting effects to be contained inside escapes, however, this work places an implicit reset just inside every variable-binding construct in code, so that no effects can move out of variable bindings. This is a strong version of the separability constraint used in Mint.

The system of Kim et al, and the more recent one by Akiemur extending this system, explicitly include the types of all free variables in the type of a code fragment [13, 1]. Because this would be too restrictive in a simply typed setting, record polymorphism (or rho polymorphism) is used in both proposals to make the types more flexible. There are two potential limitations to this approach. First, types can easily get quite big, and in languages where types must be explicitly stated (such as Java), this can be a burden on the programmer. Second, more technical point, is that the practice of multi-stage programming shows that it is often convenient to use many common type conversions (isomorphisms) to convert a value from being a code of a function to a function that maps a code argument to a code result, and to use other similar conversions, known in the partial evaluation community as two-level eta-expansions. Many such expansions cannot be written in this system.

There have been a number of systems that combine MSP with object-oriented languages. Jumbo [12] adds MSP to Java, while Meta-AspectJ [24] adds MSP to AspectJ. Both of these systems ensure that generated code is syntactically well-formed, but they do not ensure that generated code satisfies type-checking, which means that code generation could fail at runtime. The Metaphor system [14] combines MSP with reflection in C#, allowing code objects that compute fields and types as well as expressions. Metaphor includes a type system that ensures that generated code passes type-checking, and that also allows simple operations on types such as conditions. Metaphor does not handle the scope extrusion problem, however.
There are also a number of other approaches to code generation in object-oriented languages. Compile-time reflection [6] and the SafeGen [9] and MorphJ [8] systems are aimed at increasing the expressivity of object-oriented languages by adding a compile-time language of reflection and code generation. The compile-time language allows the programmer to generate parts of class definitions in a generic manner by iterating over the methods and fields of the class, making it easy to write automatic unit testing and logging, for example. Other approaches are aimed at increasing performance. The JSpec system [16], for instance, performs automatic program specialization, which examines user code and unfolds method calls and other overheads that can be determined statically. Runtime code generation [17], in contrast, is a low-level means for a program to generate code at runtime in terms of virtual machine bytecodes, which has been shown to allow considerable speedup.

9. Conclusion
This paper has proposed a practical approach to adding MSP to mainstream languages in a type-safe manner that prevents scope extrusion. The approach is simpler than prior proposals, and we expect that it will be easily and intuitively understood by programmers. The key insight is that safety can be ensured with weak separability, which places straightforward restrictions on the forms and types of computational effects that occur inside escape expressions, so that these effects cannot cause code to leak outside of escapes. The proposal has been validated both by proving that weak separability is enough to ensure safety and by demonstrating by example that many useful MSP applications can still be written that adhere to these restrictions.

A future direction for this work is to try to simplify the idea of weak separability to more closely match the intuition behind the concept. We believe there is some system similar to environment classifiers, in which quantifying on type variables can be used to implicitly capture the property that we wish to express. Instead of quantifying a type variable at the occurrence of run() as in environment classifiers, however, we believe that weak separability can be expressed by quantifying a type variable at the occurrence of an escape. This would simplify the type system and possibly add more expressive power to the language.

Acknowledgments
We thank Yannis Smaragdakis for his helpful comments.

References
A. Proofs

We give a detailed proof of Lightweight Mint’s type safety in this section. Due to space limitations, discussions of a number of subtle technical details of the type system have been omitted in the main text.

**Notation.** A variable typing pair $\Gamma$ is assumed to decompose as $\Gamma_1 | \Gamma_2$, and similarly $\Gamma' = \Gamma_1' | \Gamma_2'$.

**Definition 4.** A store typing sequence is well-formed iff all of its code-free bindings are in the first $\Sigma_1$, and the individual store typings have pairwise disjoint domains:

$$I \geq 2 \quad \frac{|\text{cf}(\Sigma_1) = 2 | \text{dom}(\Sigma_1) \cap \text{dom}(\Sigma_2) = \emptyset_{i \neq j}.}$$

**Definition 5.** Let the metavariable $\text{PT}$ range over proof trees of configuration typing judgments. $\text{PT}$ satisfies the **disjointness criterion** if any two store typing sequences $(\Sigma_i)_i$ and $(\Sigma_j)_j$ that appear in $\text{PT}$ have a common prefix of length at least 2 $(\exists K \geq 2, (\Sigma_i)_i, (\Sigma_j)_j, \iota = 1)$ and the rest have disjoint domains $(\text{dom}(\Sigma_i) \cap \text{dom}(\Sigma_j))_{i,j,K}$. $\text{PT}$ is well-formed iff it only uses well-formed store typing sequences and satisfies the disjointness criterion.

**Definition 6.** A location $l$ is said to appear in $\text{PT}$ iff $\text{PT}$ uses a store typing sequence that contains $l$ in the domain of the sequence’s union. An $l$ is said to be fresh for $\text{PT}$ iff it does not appear in $\text{PT}$. An $l$ is local to $\text{PT}$ iff for any $\text{PT}'$, if $\text{PT}$ and $\text{PT}'$ are disjoint subtrees of a well-formed tree, then $l$ is fresh for $\text{PT}'$. An $l$ is fresh or local in a configuration typing judgment if it is fresh or local, respectively, for some proof tree of the typing judgment.

Note that $l$ is local to $\text{PT}$ iff it appears in a $\Sigma$ that is introduced in $\text{PT}$ as a result of an extension of the store typing sequence that happens within $\text{PT}$. Note also that this well-formedness concern for derivation trees is only for configurations, and expression typing derivations are always well-formed.

We would like to assume that all proof trees and store typing sequences are well-formed. This does not reduce the expressivity of the type system because a user program must not contain locations, and therefore it can be typed by a derivation that only uses store typing sequences of the form $\emptyset, \emptyset, \ldots, \emptyset$. This claim is made precise by the following propositions.

**Proposition 4.** If an initial configuration is typed as

$$\langle \emptyset, \emptyset \rangle \vdash (0, e^0) : \tau | S$$

by a not necessarily well-formed derivation and $\text{locs}(e^0) = \emptyset$, then $\langle \emptyset, \emptyset \rangle \vdash (0, e^0) : \tau | S$ by a well-formed derivation.

**Proof.** The expression part contains no locations, so the store typing sequence is only used to type the heap which is empty and is therefore well-formed under, and only under, store typing sequences of the form $\emptyset, \emptyset, \ldots, \emptyset$. Thus, every configuration typing judgment in the derivation of $(*)$ is of the form $\langle \emptyset, \emptyset \rangle, \Gamma \vdash^n (0, e^0) : \tau | S$ and every heap well-formedness judgment is of the form $\langle \emptyset, \emptyset \rangle, \Gamma \vdash 0$. If we consistently replace $I$ by $I + 2$ in all such judgments, then we have a derivation tree for $\langle \emptyset, \emptyset \rangle, \emptyset \vdash^n (0, e^0) : \tau | S$. The typing derivation constructed here is clearly well-formed.

Note that Proposition 4 starts by assuming typability under $\emptyset$ because that is what we used for program typing in Figure 8.

**Proposition 5.** If $M^0$ appears in a well-typed class $C$, then $\langle \emptyset, \emptyset \rangle \vdash (0, e^0) : M^0 : \text{mtype}(M^0) | S$ by a derivation that does not involve ill-formed store typing sequences.

**Proof.** By hypothesis,

$$\langle \emptyset, \emptyset \rangle \vdash (0, e^0) : M^0 : \text{mtype}(M^0) | S.$$ 

By a similar reasoning as Proposition 4, we can prepend $\langle \emptyset, \emptyset \rangle$ to each store typing sequence in the typing derivation, which is necessarily of the form $\langle \emptyset, \emptyset, \ldots, \emptyset \rangle$.

NB: Proposition 5 replaces the ill-formed store typing sequence $\langle \emptyset \rangle$ in the typing for $M^0$ with the well-formed $\langle \emptyset, \emptyset \rangle$.

Hereafter, the assumption that all store typing sequences and typing derivations are well-formed is in effect. It is needed for typings of configurations under updated environments to propagate through congruence rules. This statement is formalized below as Lemma 6. It states that when a small step $H_1, e_1, H_2, e_2$ has been taken on a subterm $e_1$ of some bigger term, the heap attached to the typing of any other disjoint subterm can be safely replaced by $H_2$. The freshness assumption is crucial to this lemma.

Lemma 6 is the only part of the proof that needs the well-formedness assumption. We will update the statements of the Smashing and Preservation Lemmas to observe the well-formedness assumption, but the effect on the Smashing Lemma is mainly simplification rather than a change in its meaning, and the modification to the Preservation Lemma is only concerned with propagating the right well-formedness conditions.

**Lemma 6.** Suppose

$$\langle \Sigma_i \rangle, \Sigma, \Sigma \vdash^n (H_1, e^0) : \tau | S$$

and $\langle \Sigma_j \rangle, \Sigma, \Sigma \vdash^n (H_2, e^0) : \tau | S$. Then $\langle \Sigma_i \rangle, \Sigma, \Sigma \vdash^n (H_1, e^0) : \tau | S$.

**Proof.** Induction on $e^0$. If we can invoke IH on every immediate subterm, then the conclusion becomes obvious. The only obstacle to invoking IH is that if a subterm is typed under an extended environment, say $\langle \Sigma_j \rangle, \Sigma, \Sigma \vdash^n (H_1, e^0)$, then $\langle \Sigma_i \rangle, \Sigma, \Sigma \vdash^n (H_1, e^0)$ implies $\langle \Sigma_i \rangle, \Sigma, \Sigma \vdash^n (H_2, e^0)$.

For any $l \in \text{dom} \Sigma$, we have $H_1(l) = H_2(l)$ since locations that the heaps disagree on are fresh for $*$ unless it is in $\cup \Sigma_j'$. Then

$$\langle \Sigma_i \rangle, \Sigma, \Sigma \vdash^n (H_1, e^0) : \tau | S$$

by inversion

$$\langle \Sigma_j \rangle, \Sigma, \Sigma \vdash^n (H_2, e^0) : \tau | S.$$ 

For any $l \in \text{dom} \cup \Sigma_j'$, we have $\langle \Sigma_j \rangle, \Sigma, \Sigma \vdash^n (H_1, e^0) : \tau | S$, and $\Sigma, \Sigma \vdash^n (H_2, e^0) : \tau | S$. Then $\Sigma, \Sigma \vdash^n (H_1, e^0) : \tau | S$.

If $e^0 = (\emptyset, e^0)$, the variable typing’s partitioning bar ($\emptyset$) is moved but the store typing sequence is not extended. In this case, we just use $I_{\eta}$ weakening.

**Lemma 7 (\Sigma_i relevance).** If we have $\langle \Sigma_i \rangle, \Sigma, \Sigma \vdash^n (H_1, e^0) : \tau | S$ and $\Sigma, \Sigma \vdash^n (\text{locs}(e^0)) = (\cup \Sigma_j \text{locs}(e^0))$, then $\langle \Sigma_j \rangle, \Sigma, \Sigma \vdash^n (H_1, e^0) : \tau | S$.

**Proof.** Proof is by induction on $e^0$. When looking up a location the $\Sigma_i$’s are always unioned together, so it clearly only matters what the union of the sequence contains.

The only non-trivial inductive cases are the ones that extend the store typing sequence. Take $e^0 = S \tau \cdot m(\xi) (k) (e^0)$ for example, and let $L = \text{locs}(e^0)$. By inversion we have

$$\exists \Sigma, \langle \Sigma_i \rangle, \Sigma, \Sigma, \Sigma, \Sigma \vdash (k, x_0 : \tau_0) k : \emptyset \vdash^n e^0, \tau | S.$$
Then \( (\{G \cup \Sigma \} \cup \Sigma)[L] = (\{G \cup \Sigma \} \cup \Sigma)[L] = (\{G \cup \Sigma \} \cup \Sigma)[L] \) so we can use IH on \( e^n \) to obtain
\[
\langle \Sigma'[j] \rangle, \Sigma, \Gamma, \alpha, x_k \vdash \tau^n_k : \tau | S'.
\]
The conclusion immediately follows.

**Lemma 11.** If \( \langle \Sigma \rangle_i; \Gamma \vdash \bar{e} : \bar{\tau} | S \) and \( \langle \Sigma \rangle_i; \Gamma \vdash \bar{e} : \bar{\tau} | S \) then \( \langle \Sigma \rangle_i; \Gamma \vdash \bar{e} : \bar{\tau} | S \). Proof. Straightforward induction on \( \bar{e} \), noting that part-wise containment \( \Gamma' \subseteq \Gamma \) is preserved by manipulations of the form \( \Gamma_1 | \Gamma_2 \leftarrow \Gamma_1, \Gamma_2, \Gamma_3 |\Gamma | \Gamma_4 \) and \( \Gamma_5 \) and \( \Gamma_6 \) that are independent of \( \Gamma_1 \) and \( \Gamma_2 \).

**Lemma 12.** If \( \langle \Sigma \rangle_i; \Gamma \vdash \bar{e} : \bar{\tau} | S \) then \( \langle \Sigma \rangle_i; \Gamma \vdash \bar{e} : \bar{\tau} | S \). Proof. Induction on \( \bar{e} \). It is evident from the typing rules that we maintain the invariant that the term can always be seen as lower-level than the level of the typing judgment, and therefore that the level of the typing judgment is \( > 0 \). Hence when we encounter a rule that looks up the store typing (i.e. \( e^n \leftarrow t \) or \( (l \leftarrow e^n) \), it is the case that \( \text{isctf}(\langle u, \Sigma \rangle(l)) \). By the assumption that the derivation is well-formed, the store typing sequence is well-formed, hence \( l \in \Sigma \).

A common issue with multi-stage type systems is the fact that run changes the level of a term dynamic. The Demotion Lemma ensures that this change does not destroy well-typedness. Since the code to run is always fetched from the heap, we get well-typedness of the term from well-formedness of the heap.

**Lemma 13 (Demotion).** If \( \langle \Sigma \rangle_i; \langle \emptyset, \emptyset \rangle \vdash \text{Code}(S, \tau) \) and \( \langle \Sigma \rangle_i; \langle \emptyset, \emptyset \rangle \vdash \text{Code}(S, \tau) \) then \( \langle \Sigma \rangle_i; \langle \emptyset, \emptyset \rangle \vdash \text{Code}(S, \tau) \). Proof. Generalize to:
\[
\langle \Sigma \rangle_i; \Gamma \vdash \bar{e} : \bar{\tau} | S \Rightarrow \langle \Sigma \rangle_i; \Gamma \vdash \bar{e} : \bar{\tau} | S
\]
where \( \Gamma \vdash \bar{e} : \bar{\tau} | S \). Then \( \langle \Sigma \rangle_i; \Gamma \vdash \bar{e} : \bar{\tau} | S \). In both cases, we can replace \( \Gamma \) with \( \Gamma \) by \( \Gamma \) weakening. Notice that the partitioning bar is moved all the way to the right before the variable typing has a chance to be looked up, so that the assumption \( \Gamma_1 | \Gamma_2 \vdash \bar{e} : \bar{\tau} | S \) is turned into part-wise containment, matching the hypothesis of \( \Gamma \) weakening.

**Lemma 10 (\( \langle \Sigma \rangle_i \). ** If \( \langle \Sigma \rangle_i; \Gamma \vdash \bar{e} : \bar{\tau} | S \) then \( \langle \Sigma \rangle_i; \Gamma \vdash \bar{e} : \bar{\tau} | S \). Proof. Immediate consequence of \( \Gamma \) weakening.

**Lemma 11.** If \( \langle \Sigma \rangle_i; \Gamma \vdash \bar{e} : \bar{\tau} | S \) then \( \langle \Sigma \rangle_i; \Gamma \vdash \bar{e} : \bar{\tau} | S \). Proof. Induction on \( \bar{e} \). Note that in each case \( \langle \Sigma \rangle_i; \Gamma \vdash \bar{e} \) is assured by \( \Gamma \) weakening.

If \( \bar{e} = x \), inversion on the configuration typing gives \( \text{isctf}(\tau) \) and \( \bar{e} = x \), which is just the premise we need to justify \( \langle \Sigma \rangle_i; \Gamma \vdash \bar{e} : \bar{\tau} | S \). Hence we can use IH to get \( \langle \Sigma \rangle_i; \Gamma \vdash \bar{e} : \bar{\tau} | S \). Then \( \langle \Sigma \rangle_i; \Gamma \vdash \bar{e} : \bar{\tau} | S \). For \( n \neq 0 \), this is enough to get the conclusion. If \( n = 0 \), then we need \( \text{dom}(\langle \Sigma \rangle_i) \supseteq \text{dom}(\langle \Sigma \rangle_i) \) in addition; this is ensured by Lemma 11.

The remaining cases are straightforward.
Suppose $e^n = (\text{let } y \leftarrow \text{new } C((e'_{1})) \text{ in } e^n)$. If $x = y$, then the substitution is the identity on this term so the conclusion is immediate. Otherwise, we have $(\Sigma_i);\Gamma \vdash e^n : (H, \Gamma) \vdash (\Sigma_i)(l)$, by IH. By inversion,

$(\Sigma_i);\Sigma_i \Gamma_1 \Gamma_2 y : C^n \vdash (H, \Gamma) \vdash (\Sigma_i)(l).$

The store typing sequence is extended, so the l is no longer in the rightmost $\Sigma$ (if it was in $\Sigma_i), but the binding for x is no longer to the right of the partitioning bar either, so we can apply IH. This gives $(\Sigma_i);\Gamma_1 \Gamma_2 y : C^n \vdash (H, \Gamma) \vdash (\Sigma_i)(l).$ Thus $(\Sigma_i);\Gamma \vdash e^n : (H, \Gamma) \vdash (\Sigma_i)(l).$

If $e^n = M^n$, the argument is similar to the preceding case.

Finally, we are ready to prove Preservation. As noted in the main text, there are some invariants that are not captured in the statement of Lemma 3. We give a complete statement here.

**Lemma 17 (Preservation (extended version)).** If

(1) $(\Sigma_i);\Gamma \vdash H \quad (H, e^n) : \tau | S$

(2) $H_1, e^n \models H_2, e^n$

(3) $S = sp \lor (\tau^n \models \emptyset | \emptyset)$

(4) $\Gamma \vdash \Gamma \setminus \tau$

then $\exists (\Sigma_i)^'_{(i \neq (\not\in e^n)}$ such that

(i) $(\Sigma_i);\Gamma \vdash H \quad (H, e^n) : \tau | S$

(ii) $\Sigma_1 \models \Sigma_1 \land \Sigma_i \models \Sigma_i \land (\Sigma_i = \Sigma_i); \models \Sigma_1 \models \Sigma_i \models \Sigma_1 \models \Sigma_i$

(iii) $H_1 (l) \neq H_2 (\tau) \implies (l \not\in \text{ dom} (H_1)) \lor (l \not\in \text{ dom} (\Sigma_i \cup \Sigma_i)) \lor (l \models \text{ local to } (\not\in e^n))$

where $H_1 (l) \neq H_2 (l)$ includes the case where one is defined but the other not.

**Proof.** Induction on $e^n$. This is just a matter of checking

$(\Sigma_i); \tau \models (\Sigma_i)^'_{(i \neq (\not\in e^n)};\Gamma \vdash H (l) : (\Sigma_i \Sigma_i)(l)$

for every $l \in \text{ dom} (\Sigma_i \Sigma_i)$, by $\text{ dom} ((\Sigma_i \Sigma_i) \cup (\not\in e^n))$ by $\text{ H1} \vdash (\Sigma_i \Sigma_i) \vdash H$.

and (*) follows by $\gamma$-weakening.

We now turn to the Smashing Lemma. We refine its statement to take advantage of the well-formedness assumptions. The new lemma still captures the same idea but under a simpler setting: we need typing of code-containing locations only in the scope of any future-stage variables that they may refer to, and its proof relies on the fact that most of the heap has not changed. It is possible to prove it without the well-formedness assumptions, but it would only obfuscate the argument.

**Lemma 16 (Smashing Lemma (refined)).** If

(1) $(\Sigma_i);\Gamma \vdash H_1$

(2) $(\Sigma_i);\Gamma \vdash H_2$

(3) $\forall l \not\in \text{ dom} (\Sigma_i \Sigma_i), H_1 (l) \neq H_2 (l) \implies l \not\in \text{ dom} (\Sigma_i \Sigma_i)$

(IV) $\Sigma_i \subseteq \Sigma_i \land \Sigma_i \subseteq \Sigma_i \land \Sigma_i \land (\not\in e^n) \models \Sigma_i \models \Sigma_i$

where $H_1 (l) \neq H_2 (l)$ includes the case where one is defined while the other is not, then

$(\Sigma_i);\Gamma \vdash H_2.$

**Proof.** For every location $l \not\in \text{ dom} (\Sigma_i \Sigma_i)$ such that $H_1 (l) = H_2 (l)$, we have

$(\Sigma_i);\Gamma \vdash H_2 (l) : (\Sigma_i \Sigma_i)(l)$

by (I), and we can replace the store typing sequence with $(\Sigma_i \Sigma_i)$ due to $\Sigma_i$ weakening.

For any other $l \not\in \text{ dom} (\Sigma_i \Sigma_i)$, (III) tells us that $l \in \text{ dom} \Sigma_i$ so $(\Sigma_i \Sigma_i \Sigma_i)(l) = \Sigma_i (l)$ and $\text{ iscf (}\Sigma_i \Sigma_i \models (l)).$ It follows that $(\Sigma_i \models (l) = F$ and by (II), $H_2 (l) = (F, k, k)$, so $\forall k, \text{ iscf (}\Sigma_i \Sigma_i \models (l))$, hence $k \in \text{ dom} \Sigma_i.$ Therefore, $H_2 (l)$ is well-formed under any well-formed store typing sequence starting with $\Sigma_i \models (l),$ including $(\Sigma_i \Sigma_i \models (l)).$

Thus $(\Sigma_i \Sigma_i \models (l), \not\in e^n) \vdash H_2.$

We need a similar lemma that does not lower the level of the typing to handle escapes.

**Lemma 15 (Augmentation Lemma).** If $(\Sigma_i);\Gamma \vdash e^n \models \tau | S$ and $(\Sigma_i);\Gamma \vdash H \land \text{ dom} (\Sigma_i \Sigma_i) \models \text{ locs (}e^n) \text{ where } \Sigma_i$ relevance, $(\Sigma_i) \models (\emptyset, 0, 0, \ldots, 0)$ without loss of generality. Then, $\Sigma_i \Sigma_i = (\Sigma_i \Sigma_i) \cup (\not\in e^n)$, so

$(\Sigma_i);\Gamma \vdash H (l) : (\Sigma_i \Sigma_i)(l)$

for every $l \in \text{ dom} (\Sigma_i \Sigma_i)$ by $(\not\in e^n)$ by $\text{ H1} \vdash (\Sigma_i \Sigma_i) \vdash H$.

and (*) follows by $\gamma$-weakening.

We now turn to the Smashing Lemma. We refine its statement to take advantage of the well-formedness assumptions. The new lemma still captures the same idea but under a simpler setting: we need typing of code-containing locations only in the scope of any future-stage variables that they may refer to, and its proof relies on the fact that most of the heap has not changed. It is possible to prove it without the well-formedness assumptions, but it would only obfuscate the argument.

**Lemma 16 (Smashing Lemma (refined)).** If

(I) $(\Sigma_i);\Gamma \vdash H_1$

(II) $(\Sigma_i);\Gamma \vdash H_2$

(III) $\forall l \not\in \text{ dom} (\Sigma_i \Sigma_i), H_1 (l) \neq H_2 (l) \implies l \not\in \text{ dom} (\Sigma_i \Sigma_i)$

(IV) $\Sigma_i \subseteq \Sigma_i \land \Sigma_i \subseteq \Sigma_i \land \Sigma_i \land (\not\in e^n) \models \Sigma_i \models \Sigma_i$

where $H_1 (l) \neq H_2 (l)$ includes the case where one is defined while the other is not, then

$(\Sigma_i);\Gamma \vdash H_2.$

**Proof.** For every location $l \not\in \text{ dom} (\Sigma_i \Sigma_i)$ such that $H_1 (l) = H_2 (l)$, we have

$(\Sigma_i);\Gamma \vdash H_2 (l) : (\Sigma_i \Sigma_i)(l)$

by (I), and we can replace the store typing sequence with $(\Sigma_i \Sigma_i)$ due to $\Sigma_i$ weakening.

For any other $l \not\in \text{ dom} (\Sigma_i \Sigma_i)$, (III) tells us that $l \in \text{ dom} \Sigma_i$ so $(\Sigma_i \Sigma_i \Sigma_i)(l) = \Sigma_i (l)$ and $\text{ iscf (}\Sigma_i \Sigma_i \models (l)).$ It follows that $(\Sigma_i \models (l) = F$ and by (II), $H_2 (l) = (F, k, k)$, so $\forall k, \text{ iscf (}\Sigma_i \Sigma_i \models (l))$, hence $k \in \text{ dom} \Sigma_i.$ Therefore, $H_2 (l)$ is well-formed under any well-formed store typing sequence starting with $\Sigma_i \models (l),$ including $(\Sigma_i \Sigma_i \models (l)).$

Thus $(\Sigma_i \Sigma_i \models (l), \not\in e^n) \vdash H_2.$

Finally, we are ready to prove Preservation. As noted in the main text, there are some invariants that are not captured in the statement of Lemma 3. We give a complete statement here.
by (+1) and $\Sigma_i$ weakening. We also have $(\Sigma_i_j) ; \Gamma_1, \Gamma_2 \vdash x : C^n \vdash (C, (l_j)y_j) : C$ because $(\langle \downarrow, \Sigma_i_j(l_j) \rangle = (\downarrow, \Sigma_i_j(l_j)) \prec \text{ftype}_p(C))$, so

$$\tag{2} \langle \Sigma_i_j(l_j) ; \Gamma_1, \Gamma_2 ; x : C^n \vdash H_2.$$  

$l$ is fresh in $H_2$ so it is fresh in (+1). Thus by (+1), (2), we can use Lemma 6 to get

$$\langle \Sigma_i_j(l_j) ; \Gamma_1, \Gamma_2 ; x : C^n \vdash (H_2, e^0) : \vec{\cap} \vdash \Sigma_i.$$  

Then, noting that $l \in \text{dom}(\Sigma'_i l) \cup \Sigma_i l$, we have

$$\langle \Sigma'_i l ; \Gamma_1, \Gamma_2 ; l \vdash (H_2, [l/x]^0) : \vec{\cap} \vdash \Sigma_i,$$  

by the Substitution Lemma. By Lemma 18, we can move the partitioning bar to the left, thus

$$\langle \Sigma'_i(l_j) ; \Gamma \vdash l \vdash (H_2, l) : \text{Code}(l, \tau), \tau \rangle \vdash \text{Code}(S', \tau).$$  

By (I) and (IV), we have

$$\langle \Sigma'_i(l_j) ; \Gamma \vdash l \vdash (H_2, l) : \text{Code}(S', \tau) \rangle \vdash \Sigma_i,$$  

so the new heap element $\text{Code}(\langle e^0 \rangle) l$ is well-formed. All other locations are unmodified, so they are well-formed under $(\Sigma_i) l ; \Gamma$ by $\Sigma_i$ weakening. Thus $(\Sigma'_i) l ; \Gamma \vdash H_2$, therefore

$$\langle \Sigma'_i(l_j) ; \Gamma \vdash l \vdash (H_2, l) : \text{Code}(S', \tau) \rangle \vdash \Sigma_i.$$  

There is one case that modifies the heap without extending it:

- Suppose $\Sigma'_i(l_j) = (l, j_0) := (l')$. Then $\Sigma'_i(l_j) = (l \notin \text{dom}(H_1) \land \vec{\cap} \vdash \text{Code}(S'). \tau)$. Take $\Sigma_i = \Sigma$ and $\Sigma'_i = \Sigma_i \{ l \mapsto \text{Code}(S', \tau) \}$. By (I) and (IV), we have

$$\langle \Sigma'_i(l_j) ; \Gamma \vdash l \vdash (H_2, l) : \text{Code}(S', \tau) \rangle \vdash \Sigma_i,$$  

so all other primitive reductions, we have $H_1 = H_2$ so with $\Sigma_i = \Sigma$ and $\Sigma'_i = \Sigma_i (ii)$ and (iii) hold.

- Suppose $\Sigma'_i(l_j) = (l, j_0) \vdash (l \notin \text{dom}(H_1)) \land \vec{\cap} \vdash \text{Code}(S'). \tau)$. By inversion $\text{isc}(l, \Sigma_i(l)) \lor (l \in \text{dom}(\Sigma_i, \tau)$, and the first conjunct implies $l \in \Sigma_i$, so (iiii) holds. Take $\Sigma'_i = \Sigma_i$. The updated heap element $(l, (l, j_0) \vdash l \vdash (l')$ is well-formed because by inversion $(\Sigma'_i, l) \vdash \text{Code}(S', \tau)$, where $\vec{\cap} \vdash (l, \Sigma_i(l))$. The other locations are unmodified so they remain well-formed. Thus $H_2$ is well-formed, and we have

$$\langle \Sigma'_i(l_j) ; \Gamma \vdash l \vdash (H_2, l) : \text{Code}(S', \tau) \rangle \vdash \Sigma_i.$$  

For all other primitive reductions, we have $H_1 = H_2$ so with $\Sigma_i = \Sigma$ and $\Sigma'_i = \Sigma_i (ii)$ and (iii) hold.

- Suppose $\Sigma'_i(l_j) = (l, j_0) \vdash (l \notin \text{dom}(H_1)) \land \vec{\cap} \vdash \text{Code}(S'). \tau)$. By inversion $\text{isc}(l, \Sigma_i(l)) \lor (l \in \text{dom}(\Sigma_i, \tau)$, and the first conjunct implies $l \in \Sigma_i$, so (iiii) holds. Take $\Sigma'_i = \Sigma_i$. The updated heap element $(l, (l, j_0) \vdash l \vdash (l')$ is well-formed because by inversion $(\Sigma'_i, l) \vdash \text{Code}(S', \tau)$, where $\vec{\cap} \vdash (l, \Sigma_i(l))$. The other locations are unmodified so they remain well-formed. Thus $H_2$ is well-formed, and we have

$$\langle \Sigma'_i(l_j) ; \Gamma \vdash l \vdash (H_2, l) : \text{Code}(S', \tau) \rangle \vdash \Sigma_i.$$  

If $m$ does not match one of the $(\langle M, e \rangle)$, or if $T = C$, then the method implementation comes from the static class hierarchy, $P$. In that case, by Proposition 5

$$\langle \emptyset, \emptyset ; \vec{\cap} \vdash (e^0) \rangle \vdash \text{Code}(\langle e^0 \rangle) \vdash \vec{\cap} \vdash S.$$  

By $\Sigma_i$ relevance and $\vec{\cap}$ weakening,

$$\langle \Sigma'_i(l_j) ; \Gamma \vdash l \vdash (H_2, e^0) : \vec{\cap} \vdash \Sigma_i,$$  

Then repeating the argument using the Augmentation and Substitution Lemmas gives (i).

- Suppose $\Sigma'_i(l_j) = \{ l \land H_1(l) = \text{Code}(e^0) \}$. Then $n = 1$, and since $H_2$ is well-formed, we have $(\Sigma'_i(l_j) ; \vec{\cap} \vdash (H_2, e^0) : \vec{\cap} \vdash S)$ using Lemma 18. Then by Lemma 11 $\text{loc}(e^0) \subseteq \text{dom}(\Sigma_i)$ so by the Augmentation Lemma, $(\Sigma'_i(l_j) ; \vec{\cap} \vdash (H_2, e^0) : \vec{\cap} \vdash S)$.

- Suppose $\Sigma'_i(l_j) = e^0$. Then $n = 0$ and $S = \text{insep}$ and $H_1(l) = \text{Code}(\langle e^0 \rangle)$. By (III) and (IV), $\vec{\cap} \vdash \emptyset$. Then by well-formedness of $H_1,$

$$\langle \Sigma'_i(l_j) ; \emptyset \vdash (\text{Code}, \langle e^0 \rangle) : \text{Code}(S, \tau) \rangle$$  

$H_1 = H_2$ so by the Substitution Lemma $(\Sigma'_i(l_j) ; \vec{\cap} \vdash \Sigma_i,$ and the derivations of these are disjoint subtrees of (I).

We want to apply IH to (+4), but to do so we must check (III) and (IV). Because $n > 0$ the $x$ : $C^n$ is not a level-0 binding, hence (IV) holds for the subconfiguration. If $S = \text{insep}$ then $\vec{\cap} \vdash \emptyset$ so $(\Gamma_1, \Gamma_2 ; x : C^n)^{n+1} = \emptyset$ so (III) is satisfied. Therefore, by IH(i) (that is, conclusion (ii) of IH), $\exists \Sigma_i(l_j) \vdash \Sigma'_i(l_j) \vdash (H_2, e^0) : \vec{\cap} \vdash S.$

By IH(ii), $\forall l \notin \text{dom}(\Sigma_i)$, $H_1(l) \neq H_2(l)$ implies $l \in \text{dom}(\Sigma_i \cup \Sigma_i^{l+1})$. Thus by the refined Smashing Lemma, $\Sigma'_i(l_j) ; \vec{\cap} \vdash H_2.$

IH(iii) also states that any $l$ that $H_1$ and $H_2$ disagree on satisfy one of:

- $l \notin \text{dom}(H_1)$, in which case $l$ is fresh for (+3) because the domain of store typings in a configuration typing is bounded by the domain of the heap.

- $l$ is local to (+4), in which case it is fresh in (+3) because they are disjoint subtrees.

- $l$ is in $\text{dom}(\Sigma_i \cup \Sigma_i)$, If $l \in \text{dom}(\Sigma_i \cup \Sigma_i)$ then $l \in \text{dom}(\Sigma_i \cup \Sigma_i)$.

Else $l \in \Sigma$ implies that $l$ is fresh for (+3) because it is a subtree of (I) that is disjoint from (+4), and $S$ is introduced at the root of (+4).

Hence $l$ is fresh in (+3) or $l \in \text{dom}(\Sigma_i \cup \Sigma_i)$. Therefore, by Lemma 6,

$$\langle \Sigma'_i(l_j) ; \vec{\cap} \vdash (H_2, e^0) : \text{Code}(\langle e^0 \rangle) \rangle$$  

which is the last piece needed for (i).

- $\Sigma_i(l_j) \vdash \Sigma'_i(l_j)$ satisfies (ii) because $(\Sigma'_i(l_j))^{l+1}$ obeys IH(ii). It also satisfies (iii) because by IH(iii), $H_1(l) \neq H_2(l)$ ensures one of three conditions:

  - $l \notin \text{dom}(H_1)$,

  - $l \in \text{dom}(\Sigma_i \cup \Sigma_i)$, if $l \in \text{dom}(\Sigma_i \cup \Sigma_i)$ then $l \in \text{dom}(\Sigma_i \cup \Sigma_i)$.

  - If $l \in \Sigma$, then it is local to (I).
• $l$ is local to $(\ast 4)$. Then it is also local to the supertree, (I).
• If $E^n = E_{M}^{n,k}$, the argument is mostly a repetition of the previous case.
• The remaining cases are all straightforward. We simply use IH to obtain $(\Sigma_i)'_i$, and apply Lemma 6 to see that the subterms that did not participate in the small step remain well-typed if we augment them with $H_2$.

This concludes the proof.

Lemma 18. If $(\Sigma_i)_{i}; \Gamma_1, \Gamma_2 | \emptyset \vdash E^n (: b_\tau | S)$ holds, then $(\Sigma_i)_{i}; \Gamma_1 | \Gamma_2 \vdash E^n (: b_\tau | S)$.

Proof. Straightforward induction on $E^n$. The partitioning bar is irrelevant for checking heap well-formedness (as seen by $\Gamma_H$ weakening), and the only typing rule that uses the bar, the one for $(x.f := e^n)$, only becomes more permissive when the bar is moved to the left. Typing rules that move the bar always move it all the way to the right (and perhaps adds new bindings on the right end) so the invariant is maintined that the typing judgment in the hypothesis has the bar farther to the left than the judgment in the conclusion.