Multi-stage Programming for Mainstream Languages

Edwin Westbrook  Mathias Ricken  Jun Inoue  Yilong Yao  Tamer Abdelatif†  Walid Taha
Rice University
{emw4,mgricken,ji2,yy3}@cs.rice.edu, eng.tamerabdo@gmail.com, taha@cs.rice.edu

Abstract
Multi-stage programming (MSP) provides a disciplined approach to run-time code generation. In the purely functional setting, it has been shown how MSP can be used to reduce the overhead of abstractions, allowing clean, maintainable code without paying performance penalties. Unfortunately, MSP is difficult to combine with imperative features, which are prevalent in mainstream languages. The central difficulty is scope extrusion, wherein free variables can inadvertently be moved outside the scopes of their binders. This paper proposes a new approach to combining MSP with imperative features that occupies a “sweet spot” in the design space in terms of expressiveness of useful MSP programs and being intuitive and easy for programmers to understand. The key insight is that escapes must be weakly separable from the rest of the code, meaning that the only computational effects occurring inside an escape are those that are guaranteed not to contain code. To demonstrate the feasibility of this approach, we formalize a type system based on Lightweight Java which we prove sound, and we also provide an implementation, called Mint, 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
Abstraction mechanisms, such as reflection and design patterns, are useful for writing clean, maintainable code. Often, however, such mechanisms come with a steep performance overhead, making them less useful in real systems. One approach to this problem is multi-stage programming (MSP), a language feature that provides a disciplined form of runtime code generation. Just by inserting staging annotations, a form of quasi-quotation, the programmer can change programs that use expensive abstractions into program generators which generate programs without the abstractions. This ameliorates the runtime cost of abstractions, because overhead is only paid when the generators are executed, not when the programs they generate are executed.

A key component of MSP is type safety, which ensures statically that all programs generated at runtime will be well-formed. Unfortunately, although it is known how to ensure type safety for purely functional languages [4, 19, 20], it is unclear how this guarantee can be extended to mainstream languages such as Java. In particular, standard features of mainstream languages, such as imperative assignment, lead to scope extrusion, in which variables in code fragments may move out of the scopes where they are defined. Several approaches to this problem have been proposed [1, 8, 9, 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 that we argue occupies a “sweet spot” in the design space in terms of expressiveness of useful MSP programs and being intuitive and easy for programmers to understand. Our contributions include:

• An MSP extension of Java called Mint (Section 2), followed by an analysis of the problem of scope extrusion in naive approaches to MSP in mainstream languages (Section 3).

• The notion of weak separability, which prevents scope extrusion in Mint (Section 4). Specifically, weak separability ensures 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.

• A demonstration (by example) of the expressivity of Mint (Section 5).

• A discussion of the Mint reflection library, which allows the overhead of reflection to be removed (Section 6).

• A formalization of Mint based on Lightweight Java, and a proof that this calculus is type-safe (Section 7). This means that 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. We provide proof sketches in this paper and make the full proofs available in the companion technical report [21].

• An implementation of Mint, published online at http://plresearch.org/JavaMint, along with validation of the performance impact of MSP in Mint (Section 8). The implementation is based on the Java OpenJDK compiler from Sun Microsystems [13].
1.2 Comparisons with Related Efforts

Several efforts have been made to construct type-safe languages for MSP with effects [2, 3, 9, 11, 19]. 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. Aktemur [1] and Kim et al. [11] support unrestricted imperative operations on code, but they significantly differ in the approaches and extents to which they support effects and guarantee well-formedness of the generated code [2, 3, 9, 11, 19]. All of these support manipulations of open terms, but use a more restricted version of the weak separability condition introduced here that does not allow any side effect to occur inside a future-stage binder that is visible from the code outside. The imperative primitive in all of these, except for Kameyama et al. [8, 9], are ML-style “boxed” references, which is not in line with Java semantics. Pervasive, unboxed references, an essential feature of Java, exacerbate the problem. Kameyama et al. [8, 9] use delimited control as their imperative primitive, which is more general than mutable stores.

Until recently, efforts to introduce MSP to the object-oriented setting focused on engineering aspects. The staged extensions of Java by Sestoft [15], Schultz et al. [14], Kamin et al. [10], and Zook et al. [22] focus on implementation, applications, and on quantifying the performance benefits. These extensions were not formalized. Neverov and Roe [12] formalize a core typed, Java-like calculus but leave the type soundness and well-formedness unproven. Their calculus also does not have side effects. Huang et al. [7] state that their system guarantees well-formedness and well-typedness of generated code, but they do not prove such a result or formalize their system. In later work, Huang et al. [6] 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. Faehndrich et al. [5] propose a similar system, which allows the user to perform limited manipulations of code values, using reflection, in a type-safe manner.

The weak separability approach presented here is closest to that of Kameyama et al. [8, 9] in that both are based on limiting the effects allowed in escapes. Our work, however, allows effects that occur in escapes to be visible outside the escapes, as long as those effects do not involve code objects. This is better suited to Java programming, which in general makes heavy use of effects. In addition, Kameyama et al. express their limitation of effects using the delimited control operator reset, which is not in mainstream languages like Java. Our work gives an intuitive and straightforward means to express weak separability in Java.

2. Multi-Stage Programming in Mint

Mint extends Java 1.6 with the three standard MSP constructs: brackets, escape, and run (see e.g. [18]). 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:

\[
\begin{align*}
<! & \{ \\
& \text{C.foo();} \\
& \text{C.bar();} \\
& \} > \\
\end{align*}
\]

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

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,

\[
\begin{align*}
\text{Code< Integer> } x &= <! | 2 + 3 | >; \\
\text{Code< Integer> } y &= <! | 1 + 'x' | >; \\
\end{align*}
\]

stores \(<! | 1 + (2 + 3) | >\) into \(y\). Run is provided as a method \texttt{run()} that code objects support. For example, executing

\[
\begin{align*}
\text{int } z &= y.\text{run();} \\
\end{align*}
\]

after the above example sets \(z\) to 6.

Mint also allows cross-stage persistence (CSP), wherein a variable bound outside brackets can be used inside the brackets, as in

\[
\begin{align*}
\text{int } x &= 1; \\
\text{Code< Integer> } y &= <! | x * 1 | >; \\
\text{Weak separability places certain restrictions on CSP}; \text{ see Section 4.}
\end{align*}
\]

Basic MSP in Mint can be illustrated using the classic power function example. Figure 1 displays the unstaged power function in Java. Figure 1 displays a staged version. This staged method \texttt{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.

3. The Scope Extrusion Problem

One of the most important properties of MSP languages is the guarantee that program generators will always produce well-formed code. It is known how to achieve this guarantee in the purely functional setting [4, 19, 20]. This is more challenging in the presence of imperative features, however, because of the possibility of scope extrusion, where a code object containing a variable is used outside the scope of the binder for that variable. If such a code object were allowed to be compiled and run, a runtime error would result, because the result of compiling and running code with free variables is undefined.

Scope extrusion can be caused by the following situations:

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

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

![Figure 1. Staging the Power Function in Mint](attachment:image-url)
interface IntCodeFun {
    Code<Integer> call (Integer y);
}
interface Thunk { Code<Integer> call (); }

class ThunkCSPer {
    Code<Code<Integer>> doCSP(Thunk f) {
        return <| f.call() | >;
    }
}

{| new IntCodeFun () {
    Code<Integer> call (Integer y) {
        return '(! f.call () | >;
    }
}|>

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

Code<Integer> meth (Code<Integer> c) {
    throw new CodeContainerException(c);
}

{| new IntCodeFun () {
    Code<Integer> call (Integer y) {
        meth (Code< Integer > x);
    }
}|>

3. Cross-stage persistence (CSP) of a code object, an example of which is displayed in Figure 2.

The first two cases are straightforward; the first example extrudes y from its scope by assigning <| y | > to the variable x bound outside of the scope of y, while the second example throws an exception containing <| y | > outside the scope of y. The third example, however, is more subtle. This example creates an anonymous inner subclass of Thunk, whose call() method returns a code object containing the variable y. This Thunk object is then passed to doCSP in the escape, yielding the code object

{| new IntCodeFun () {
    Code<Integer> call (Integer y) {
        return T.call ();
    }
}|> (T = <| 1 | >);

where T is the subclass of Thunk. In a substitution-based semantics, running this code object would substitute 1 for y in T, thus producing a new copy of T whose call method returns <| 1 | >, and no scope extrusion would occur. Such a semantics, however, would be impractical, because it would involve traversing the (possibly compiled) method definitions of T, and it would also be confusing, because the call method of T itself would not be called when this example was run. Instead, Mint uses an environment-based semantics, in which case running the above code would simply return the value of T.call(), which would produce the code object <| y | >. This behavior, known as the "hidden free-variable problem," arises commonly in environment-based semantics of multi-stage languages [17].

Figure 2. Cross-stage Persistence of Code Objects

4. Weak Separability

The three situations mentioned in the previous section can be prevented using the following informal definition:

Informal Definition 1. A program fragment is weakly separable if it is observable from the enclosing runtime environment only through its return value and through side effects involving only code-free values.

Intuitively, requiring escapes to be weakly separable will prevent scope extrusion, because no code object can be moved outside of an escape. A similar but stronger restriction was used in the systems of Kameyama et al. [8, 9] to ensure that no effects occurring inside escapes could be visible outside the escapes. (Technically, that work placed this restriction on future-stage binders.) We call that condition separability.

We leave the notion of weak separability only informally defined here both because formalizing it would require complex semantic definitions and because it is in general undecidable. Instead, we give here a conservative approximation of weak separability that is used by Mint. This approximation is decidable, and we show below that still leaves an expressive language. Unless otherwise specified, the phrase “weakly separable” in the remainder of this document refers to this approximation.

To define the approximation, we first define code-free:

Definition 1. A type is code-free if it is not a subtype of Code<T>, the types of all of its fields are code-free, and its class is final. A value is code-free if its type is code-free.

The requirement that a class is final means that it is not allowed to be subclassed. This ensures that a subclass with an additional field of type Code<T> cannot be substituted at runtime. Code-free types include number types such as Integer and Double, the String class, arrays of code-free types, and all of Java’s reflection classes such as Class and Field. It does not include Object, for example, as this type is not final, and an Object could be a code object at runtime.

We can now define the approximation of weak separability used in Mint for escapes:

Definition 2. A Mint term is weakly separable iff:
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 catch in the program fragment, or a constructor call new C(e1, ..., en) where the ei are code-free;
3. Cross-stage persistence occurs only for final variables of code-free types;
4. Only methods and constructors whose bodies are weakly separable are called.

The first three of these clauses directly preclude the three cases of scope extrusion in the previous section. Note that the 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 keyword separable.

5. 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 even 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 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; and staging a for loop to do loop unrolling, demonstrating a generator for imperative code. Section 6 gives a third example, which is a staged serializer that uses Mint's reflection capabilities. The performance of all these examples is evaluated in Section 8.

5.1 Staged Interpreter

Staged interpreters are a classic application of MSP. To demonstrate that staged interpreters can be written in Mint, we have implemented an interpreter for a small programming language called lint [18], 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 kind of AST node in the language. This interface specifies the single method eval for evaluating the given expression, which takes two environments, one for looking up variables and the other for looking up defined functions. For example, applications of defined functions are implemented as follows:

```java
private String _e;
private Exp _body;
public App(String s, Exp e) {
    _s = s; _e = e;
}
public int eval(Env env, FEnv f) {
    return f.get(_s).apply(_e.eval(env,f));
}
```

where f.get(_s) looks up the defined function named by the string _s as an object of the class Fun. The Fun class has an apply method for applying a function to an argument, and this is used here to apply the function to the argument _body. If _s does not name a valid, defined function in f, then get throws an exception.

The staged interpreter redefines the eval method to return Code<Integer>, so that evaluating an expression yields code to compute its value. This method is marked as separable so that it can be called from inside an escape. For example, staging the App class above yields the following:

```java
class App implements Exp {
    private String _e;
    private Exp _body;
    public App(String s, Exp e) {
        _s = s; _e = e;
    }
    public int eval(Env env, FEnv f) {
        return f.get(_s).apply(_e.eval(env,f));
    }
}
```

The get method is again used to look up the function named by _s. The return type of get is now Code<Exp>, meaning that get returns code for the defined function. This code is spliced into the returned code, and its result is applied to the evaluation of the argument using the apply method. The argument for apply is obtained by splicing in the code object returned by _body.eval. If _s does not name a valid, defined function, then the get method throws an exception. Note that this is the only computational effect in the whole staged interpreter that happens inside a code generator. It is weakly separable, however, because the thrown exception need only contain the string argument _s that was not found in the environment; the exception therefore is code-free.

5.2 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 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 uses an interface called SIter 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>
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 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 I.iteration is not.

For example, the following class generates code that accumulates the indices used in the loop iteration into cell:

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

6. Staged Reflection Primitives

Neverov observed that staging and reflection in languages like C# and Java can be highly synergistic [12]. 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.

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 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<Object> and inserts an unsafe cast in the code object, yielding a code object of type Code<A>. 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.

1 The 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.
The type FieldCode\langle A, B \rangle 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 'a', f \rangle code fragment) on that object. The type also provides a getType method to return a ClassCode\langle B \rangle object for the type B.

The following example illustrates the use of these classes. The code defines a serializer, which 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\langle Void \rangle sserialize(ClassCode\langle A \rangle type, final Code\langle A \rangle obj) {
    if (type.getCodeClass() == Byte.class)
        return <| {
            writeByte('(('Code\langle Byte\rangle.obj));
        } |>
        else if (type.getCodeClass() == Integer.class)
            return <| {
                writeByte('(('Code\langle Integer\rangle.obj));
            } |>
            Code\langle Void \rangle result = <| { } |>
            for (final FieldCode\langle A, ? \rangle fc:
                type.getFields()) {
                result = <| {
                    'result;
                    '('sserializeField(fc, obj));
                } |>
            }
            return result;
}
```

The code to write primitive fields is generated directly. Non-primitive fields are visited recursively. The code is then spliced together and returned. This example was inspired by a similar example given by Neverov and Roe [12].

7. 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 [16] (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 safety issues that arise in Mint. Specifically, assignments are required to cause many forms of scope extrusion, and AICs are required to create the scopes (i.e., the additional variable bindings) that can be extruded. AICs also lead to more complex possibilities for scope extrusion as shown in Section 3, which we wish to show are prevented by our system.

A significant development of the type system is the use of a sequence of store typings rather than a single store typing. This sequence is a stack that grows from left to right, where a new “frame” is pushed onto the stack when we enter a new scope (i.e., when new variables are bound) in a code object. Earlier frames can only refer to locations in later frames if the latter are code-free, ensuring that scope extrusion cannot occur through assignments. The key lemma involved in this approach is the Smashing Lemma, which allows a stack of \( n + 1 \) frames to be smashed into a valid stack of \( n \) frames by “smashing” the code-free locations in the last frame into the penultimate frame.

To simplify the formalism somewhat, we disallow assignments to local variables in LM. All assignments must instead be to object fields. This completely disallows assignments in escapes, however, in which assignments are only allowed to local variables. To rectify this problem, we add a restricted form of let, written as

Let x = 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.cell 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.cell.x.

7.1 Syntax

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

**Notation.** We write \((a, b, c, \ldots)\) for sequences, with the shorthand \((A_i)_{i=0}^n\) for \((A_0, A_1, \ldots, A_n)\). I may be omitted, and it defaults to 0. J may also be omitted when clear from context. The empty sequence is written \(\langle \rangle\). Concatenation of sequences \(s_1 \cup s_2\) is written \(s_1 \circ s_2\), with the shorthand \((A_i)_i\) for \((A_0) \circ (A)\). We also use \((e_i)_{i=0}^n\) to denote \((e_0, e_1, \ldots, e_n)\), which are simplified.

The syntax of LM is given in Figure 3. 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 \( \text{new}\ C(\ldots) \) for instantiating ordinary (i.e., non-AIC) classes because one can write \( \text{let}\ x \leftarrow \text{new}\ C(\ldots)\ \text{in}\ x \) instead. Sequencing \( (e_1; e_2) \) is also omitted because 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.

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 \( \text{Code}(\text{insep}, \tau) \). This distinction is necessary in the case of a separable expression which contains a nested escape “*e*, since we must know for type preservation that “*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.

### 7.2 Operational Semantics

Figure 4 shows preliminary definitions that we need for the operational semantics. A heap is a finite mapping from locations to heap elements. A heap term looks up a method.

An evaluation context \( \mathcal{E}^{n,k} \) is indexed by two levels, the level \( n \) of the context and the level \( k \) inside. The intent is for any well-typed level-\( n \) expression to be decomposed uniquely as \( \mathcal{E}^{n,k} \) where \( k \) is a redex at level \( k \), unless the expression is a (level-\( n \) value. There are two variants of evaluation contexts, one that yields an expression \( \mathcal{E}^{n,k} \) when plugged in, and one that yields a method declaration \( \mathcal{E}^{n,k}_M \). 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. The 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 declaration.

Figure 5 shows the small-step semantics for Lightweight Mint. This is given as the judgment \( H_1, e_1 \Downarrow H_2, e_2 \) 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 \( \Downarrow \) 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 \( \text{let}\)-form execution by substitution.

The local binding \( L \) found in LJ [16] and similar formalisms is therefore unnecessary, and the small-step judgment is made between heap terms 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, 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 [17],

---

**Figure 4.** Preliminary definitions for operational semantics.
The predicate able typing (or type environment) comes in pairs, separated by a |

Figure 6 gives preliminary definitions for the type system. A vari-

7.3 Type System

These are the variables whose fields can be assigned to without vi-

The reason the variable typing

The bottom half of Figure 7 defines typing for pseudo-expressions

The bottom half of Figure 7 defines typing for pseudo-expressions

The bottom half of Figure 7 defines typing for pseudo-expressions

by assuming that auxiliary functions like fields(τ), mtype(m, τ) 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 top-level judgment ⊢ cf(Σ) : Σ | L where L = { l ∈ dom(Σ) : iscf(l(Σ(l)))}. locs(ε) = { l : l is a subterm of ε}

fypes(τ) = (τ, 1), assuming fields(τ) = (τ, f)

type(ε, f, τ) = ftype(f, ε, τ) = (τ, f, τ)

assuming class C extends Dτ1 . . . τn ∈ P

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 uses let-expressions by extending the current context with the let-bound variable, while the rule after type field lookups by typing the object and then looking up the relevant field type. Note that, in typing the body of a let form, a new frame Σ can be added to the current stack Σ, to allow for the possibility of heap locations containing code objects with the variable x free.

The next three rules type field assignments (ε1, f := ε2) by checking the type of ε1 is some τ, and then checking that the type τ of ε2 is the appropriate field type of τ. The first of these rules applies to arbitrary ε1, and requires τ2 to be code-free if the assignment is to be weakly separable. The second and third rules

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

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.
\[\text{Type system for Lightweight Mint.}\]
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.

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, and where typing \( \{e\} \) requires typing \( e \) at the next level and adds the code type, typing \( \langle e \rangle \) requires typing \( e \) at a code type on the previous level and removes the code type, and typing \( e \).run \( \{ \) 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 \( \langle e \rangle \) are only weakly separable if \( e \) has type \( \text{Code( semp, } \tau) \).

The remainder of Figure 7 has rules for the following judgments. The judgment \( \langle \Sigma_i \rangle_i ; \Gamma \vdash^\alpha M : \tau \) states that an AIC that subclasses \( D \) with method definitions \( \{M_i^n\} \) is well-formed. This requires the methods \( \{M_i^n\} \) 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 \( \langle \Sigma_i \rangle_i ; \Gamma \vdash^\alpha M^n : \langle \tau \rangle_i, \Sigma_i \rightarrow_{\alpha} \tau[S] \) states that method \( M \) has input types \( \langle \tau \rangle_i \), output type \( \tau \), and further is weakly separable if \( S = \text{semp} \). Note that this rule is allowed to push a new frame onto the stack of store typings when the level \( n > 0 \). 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 \( \langle \Sigma_i \rangle_i ; \Gamma \vdash^\alpha H \) 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 \( \langle \Sigma_i \rangle_i \), allowing the possibility that other frames could be pushed onto this stack. The judgment \( \langle \Sigma_i \rangle_i ; \Gamma \vdash h : \tau \) 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 not 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 \( \langle \Sigma_i \rangle_i ; \Gamma \vdash H \) 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.

### 7.4 Soundness

We now sketch the key parts of our proof of Type Soundness. Complete proofs can be found in the technical report [21]. Type soundness is proved by the usual Preservation and Progress lemmas. Progress is proved with the following lemma:

**Lemma 1 (Unique Decomposition).** If \( \langle \Sigma_i \rangle_i ; \Gamma \vdash^\alpha e^n : \tau[S] \) and \( \bar{e}^n \) is not a pseudo-value then \( \bar{e}^\alpha \) is uniquely decomposed as \( \bar{e}^\alpha = \bar{e}^{a,n}[\bar{e}^m] \), where \( \bar{e}^m = \text{syntactic equality modulo } \alpha \) conversion.

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 \( \langle \Sigma_i \rangle_i ; \Gamma \vdash^\alpha (H, e^n) : \tau[S] \) 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 \( \langle \Sigma_i \rangle_i \).

For example, the rule for \( \text{let} \) forms becomes:

\[
\langle \Sigma_i \rangle_i ; \Gamma \vdash^\alpha (H, e^n) : \text{ftype}(\langle C \rangle[S])
\]

A second technical difficulty is that a reduction step inside a \text{let} form or AIC that pushes a new frame \( \Sigma \) onto \( \langle \Sigma_i \rangle_i \), 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 \( \langle \Sigma_i \rangle_i \), because this portion of the heap would not be closed under reachability. To deal with this problem requires the Smashing Lemma, which smashes the last two \( \Sigma \)'s of \( \langle \Sigma_i \rangle_i \) into one, giving a shorter store typing sequence.

**Lemma 2 (Smashing).** If

1. \( \langle \Sigma_i \rangle_i , \Gamma_1 \Gamma_2 \vdash H_1 \)
2. \( H_1 \mid L = H_2 \mid L \) where \( L = \text{dom}(\cup \Sigma_i) - \text{dom}(\text{cf}(\cup \Sigma_i)) \)
3. \( \langle \Sigma_i \rangle_i \Gamma_1 \Gamma_2 \vdash H_2 \)

then \( \langle \Sigma_i \rangle_i \Gamma_1 \vdash H_2 \).

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 those technicalities are left to the technical report.

**Lemma 3 (Preservation).** If \( \langle \Sigma_i \rangle_i , \Sigma_R ; \Gamma_1 \Gamma_2 \vdash^\alpha (H_1, e^n_2) : \tau[S] \) and \( (H_1, e^n_2) \vdash^\alpha (H_2, e^n_2) : \tau[S] \), then \( \Sigma_R \) such that

1. \( \Sigma_R \supseteq \Sigma_R \)
2. \( \langle \Sigma_i \rangle_i , \Sigma_R ; \Gamma_1 \Gamma_2 \vdash^\alpha (H_2, e^n_2) : \tau[S] \)
3. \( H_1 \mid L = H_2 \mid L \) where \( L = \text{dom}(\cup \Sigma_i) - \text{dom}(\text{cf}(\cup \Sigma_i)) \)

Proof is by induction on the typing judgment.

### 8. Implementation and Performance

To verify the expressivity of the design and obtain performance results, we created an implementation of Mint by modifying OpenJDK [13], 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.

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

- \( \text{power} \) is the power example from Section 2.
- \( \text{fib} \) recursively computes the generalized Fibonacci function, using let-insertion to avoid code duplication in the staged version.
<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>
</tr>
</thead>
<tbody>
<tr>
<td>power</td>
<td>9.0x</td>
<td>0.061</td>
<td>0.0067</td>
<td>1.7</td>
</tr>
<tr>
<td>fib</td>
<td>8.8x</td>
<td>0.058</td>
<td>0.0066</td>
<td>9.0</td>
</tr>
<tr>
<td>mmult</td>
<td>1.3x</td>
<td>1.3</td>
<td>1.0</td>
<td>12.0</td>
</tr>
<tr>
<td>eval-fact</td>
<td>10.0x</td>
<td>0.83</td>
<td>0.081</td>
<td>1.9</td>
</tr>
<tr>
<td>eval-fib</td>
<td>10.0x</td>
<td>18.0</td>
<td>1.8</td>
<td>2.6</td>
</tr>
<tr>
<td>unroll</td>
<td>1.6x</td>
<td>0.16</td>
<td>0.097</td>
<td>3.3</td>
</tr>
<tr>
<td>serialize</td>
<td>19.0x</td>
<td>1.5</td>
<td>0.080</td>
<td>6.5</td>
</tr>
</tbody>
</table>

Figure 8. Benchmark results.

- `mmult` performs sparse matrix multiplication, which is optimized for every 1 and 0 in the left matrix.
- `eval-fact` calculates factorials using the `lint` interpreter discussed in Section 5.1.
- `eval-fib` calculates the standard Fibonacci sequence using the `lint` interpreter.
- `unroll` performs the loop unrolling example of Section 5.2.
- `serialize` performs serialization using the `serializer` generator discussed in Section 6.

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. More details of these benchmarks are available in the companion technical report [21].

The results are given in Figure 8. Performance improved in all cases. The speedups achieved range from 1.3 to 19.0, 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 approximately nine 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 19.3.

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


