

Multi-stage Programming for Mainstream Languages

Edwin Westbrook Mathias Ricken Jun Inoue Yilong Yao Tamer Abdelatif¹ 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 [20, 21, 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 [12, 1], while another one uses delimited control to express effects and to limit their scope [10]. 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). We provide proof sketches in this paper and make the full proofs available in the companion technical report [22].
- 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

¹ Ain Shams University.

```

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

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 [20, 3, 2, 12, 10]. 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. [12] support unrestricted imperative operations on open terms but choose not to provide α -equivalence for future-stage code. Their system delegates hygiene to a specialized binder λ^* , 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. [10], 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. [10] 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. [15], Kamin et al. [11], and Zook et al. [23] focus on implementation, applications, and on quantifying the performance benefits. These extensions were not formalized. Neverov and Roe [13] 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] 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. [7] 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. [12] 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.

```

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.

2.1 Staging Constructs

Mint extends Java 1.6 with the three standard MSP constructs: brackets, escape, and run [20, 21, 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>`.

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 [13]. 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 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.

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

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

```
Code<Void> fieldIter(FieldFun fun, Code<A> o,
    CodeClass<A> clazz) {
    Code<Void> c = <| { } |>;
    for (CodeField<A,?> f : clazz.getFields())
        c = <| { 'c;
            (fun.call(f.get(o),
                f.getType())); } |>;
    return c;
}
interface FieldFun {
    <T> // for any T
    Code<Void> call(Code<T> c, CodeClass<T> t);
}
```

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` creates a code object that recursively prints an object and all of its fields:

```
public static class PrintFieldFun
    implements FieldFun {
    public separable <T> Code<Void>
    call(Code<T> c, CodeClass<T> t) {
        return <| { System.out.println('c'); } |>;
    }
}
```

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

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.

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, 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 type of errors. An example of the first type can be seen in what follows:

```
<| { Code<Void> z = foo();
    '(17); } |>
```

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:

```
<| { Code<Void> z = foo();
    '(z); } |>
```

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:

```
<| { Code<Void> z = foo();
    '(<| z |>.run()); } |>
```

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:

```
<| { Integer y = foo();
    Integer z = '(x = <| y |>); } |>;
```
2. Throwing an exception that contains a code object, for example:

```
Code<Integer> meth(Code<Integer> c) {
    throw new CodeContainerException(c);
}
<| { Integer y; '(meth(<| y |>)); } |>
```
3. Cross-stage persistence (CSP) of a code object, an example of which is displayed in Figure ??.

The first two cases are traditional conditions for scope extrusion. 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`.

```

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
            ((ThunkCSPer.doCSP(new Thunk() {
                Code<Integer> call() {
                    return <| y |>;
                })
            })).call(1) |>
    }
}

```

Figure 3. Cross-stage Persistence of Code Objects

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

```

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

```

where `T` is the `Thunk` that returns `<| y |>`. In a substitution-based semantics, calling `T` with `1` would return `<| 1 |>`, 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 `<| y |>`; preventing this behavior would require the `run()` method to traverse the definition of the `call()` method of `T` and to replace `<| y |>` with `<| 1 |>`, 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 [18]. A substitution-based semantics does not allow us to capture it. The object hiding the free variable is usually a closure in λ calculus-based models, and when computation proceeds via substitution, the entire λ 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 `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`, as this type is not *final*. This is justified because an `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 `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 clauses directly address 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 new keyword `separable`. Note that a call to `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 `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 [?] 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> cv2 = <| { cmd2.run(); } |>;
}
```

4.2 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` [19], 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:

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

```
class App implements Exp { /* ... */
    separable Code<Integer> eval(Env e, FEnv f) {
        return <| (f.get(_s)).apply(
            (_body.eval(e,f))) |>;
    }
}
```

The `get` method is again used to look up the function named by `_s`. The return type of `get` is now `Code<Fun>`, 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.

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:

```
public static void
roll(int start, int stop, int step, Iter I) {
    for(int x = start; x < stop; x += step)
        I.iteration(x);
}
```

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:

```
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 `I.iteration` is not.

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

```
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:

```

public static separable
<A> Code<Void> sserialize(ClassCode<A> type,
                        final Code<A> obj) {
    if (type.getCodeClass()==Byte.class)
        return <| {
            writeByte('((Code<Byte>)obj));
        } |>;
    else if (type.getCodeClass()==Integer.class)
        return <| {
            writeInt('((Code<Integer>)obj));
        } |>;
    Code<Void> result = <| { } |>;
    for(final FieldCode<A,?> 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 in the Metaphor paper [13].

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 [17] (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 ??, 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

extensible class names	D
final class names	F
variables	x
field names	f
method names	m
heap locations	l
classes	$C ::= D \mid F$
separability marker	$S ::= \text{sep} \mid \text{insep}$
types	$\tau ::= C \mid \text{Code}(S, \tau)$
class declarations	$CL ::= \text{class } C \text{ extends } D$ $\quad \{ \langle \tau_i f_i \rangle_i^I; \langle M_j^0 \rangle_j^J \}$
method declarations	$M^n ::= S \tau m(\langle \tau_i x_i \rangle_i) \{ e^n \}$
class hierarchy	$P ::= \langle CL_i \rangle_i$
programs	$p ::= P, e^0$
expressions	$e^n ::= x \mid l \mid e^n.f \mid (e^n.f := e^n)$ $\quad \mid e^n.m(\langle e_i^n \rangle_i)$ $\quad \mid \text{let } x \leftarrow \text{new } C(\langle e_i^n \rangle_i) \text{ in } e^n$ $\quad \mid \text{new } D(\langle e_i^n \rangle_i) \{ \langle M_j^0 \rangle_j^J \}$ $\quad \mid \langle e^{n+1} \rangle \mid \langle e^{n-1} \rangle [n > 0]$ $\quad \mid e^n.\text{run}()$
values	$v^n ::= l \mid e^{n-1} [n > 0]$

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

Figure 4. Lightweight Mint syntax.

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:

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

Uses of x , including assignments to x , can then be replaced by uses of $x_{\text{cell}}.x$.

5.1 Syntax

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

Notation. We write $\langle A_i \rangle_{i=I}^J$ 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 $\langle a, b, c, \dots \rangle$ 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 $\langle A_i \rangle_i, A$ to mean $\langle A_i \rangle_i \circ \langle A \rangle$ if the intention is clear. $\langle e_i \rangle_{i=I}^J [i_0 \rightarrow x]$ is the same sequence as $\langle e_i \rangle_i$ except that e_{i_0} is replaced by x .

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. $\text{Code}(S, \tau)$ 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 ($\text{new } C(\dots)$) for instantiating ordinary (i.e., non-AIC) classes because one can write ($\text{let } x \leftarrow \text{new } C(\dots) \text{ in } x$) instead. Sequencing ($e_1; e_2$) is also omitted because this sequence can be written $\text{seq.call}(e_1, e_2)$, where 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 $\text{Code}(\text{insep}, \tau)$. This distinction is necessary in the case of a separable expression which itself contains a nested escape $\sim e$, since we must know for type preservation that $\sim e$ 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 4 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 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 $\mathcal{E}^{n,k}$ 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 $\mathcal{E}^{n,k}[r^k]$ where r^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 ($\mathcal{E}_e^{n,k}$) when plugged in, and one that yields a method declaration ($\mathcal{E}_M^{n,k}$). Both variants can be plugged with expressions only.

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

Figure 5 shows the small-step semantics for Lightweight Mint. This is given as the judgment $H_1, e_1 \xrightarrow{n} H_2, e_2$ 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 $\xrightarrow[k]{\text{prim}}$ at level k . Most of the primitive reduction steps are straightforward, including rules

operational terms	
heaps	$H : l \xrightarrow{\text{fin}} h$
runtime type tags	$T ::= C \mid \text{sub } D \{ \langle M_i^0 \rangle_i \} \mid \text{Code}$
heap elements	$h ::= (C, \langle l_i \rangle_i) \mid (\text{Code}, \langle e^0 \rangle) \mid (\text{sub } D \{ \langle M_i^0 \rangle_i \}, \langle l_j \rangle_j)$
pseudo-expressions	$\hat{e}^n ::= e^n \mid M^n$
pseudo-values	$\hat{v}^n ::= v^n \mid M^{n-1} [n > 0]$

evaluation contexts	
$\mathcal{E}^{n,k}$	$::= \mathcal{E}_e^{n,k} \mid \mathcal{E}_M^{n,k}$
$\mathcal{E}_M^{n,k}$	$::= S \tau m(\langle \tau_i x_i \rangle_i) \{ \mathcal{E}_e^{n,k} \} [n > 0]$
$\mathcal{E}_e^{n,k}$	$::= \bullet [n = k] \mid \mathcal{E}_e^{n,k}.f \mid (\mathcal{E}_e^{n,k}.f := e^n) \mid (v^n.f := \mathcal{E}_e^{n,k}) \mid \mathcal{E}_e^{n,k}.m(\langle e_i^n \rangle_i) \mid v^n.m(\langle v_i^n \rangle_i, \mathcal{E}_e^{n,k}, \langle e_j^n \rangle_j) \mid \text{let } x \leftarrow \text{new } C(\langle v_i^n \rangle_i, \mathcal{E}_e^{n,k}, \langle e_j^n \rangle_j) \text{ in } e^n \mid \text{let } x \leftarrow \text{new } C(\langle v_i^n \rangle_i) \text{ in } \mathcal{E}_e^{n,k} [n > 0] \mid \text{new } D(\langle v_i^n \rangle_i, \mathcal{E}_e^{n,k}, \langle e_j^n \rangle_j) \{ \langle M_a^n \rangle_a \} \mid \text{new } D(\langle v_i^n \rangle_i) \{ \langle M_j^{n-1} \rangle_j, \mathcal{E}_M^{n,k}, \langle M_a^n \rangle_a \} [n > 0] \mid \langle \mathcal{E}_e^{n+1,k} \rangle \mid \langle \mathcal{E}_e^{n-1,m} [n > 0] \rangle \mathcal{E}_e^{n,k}.\text{run}()$

fields(τ) or fields(T)	
fields(Object)	$= \text{fields}(\text{Code}) = \text{fields}(\text{Code}(S, \tau)) = \langle \rangle$
fields(sub $D \{ \langle M_i^0 \rangle_i \}$)	$= \text{fields}(D)$
fields(C)	$= \langle \tau_i f_i \rangle_i \circ \text{fields}(D')$

where $\text{class } C \text{ extends } D' \{ \langle \tau_i f_i \rangle_i; \dots \} \in P$

mname(M^n)	
mname($S \tau m(\dots) \{ \dots \}$)	$= m$

method(m, τ) or method(m, T)	
method($m, \text{sub } D \{ \langle M_i^0 \rangle_i \}$)	$= \begin{cases} M_i^0 & \text{if } \text{mname}(M_i^0) = m \\ \text{method}(m, D) & \text{otherwise} \end{cases}$
method(m, C)	$= \begin{cases} M_j^0 & \text{if } \text{mname}(M_j^0) = m \\ \text{method}(m, D') & \text{otherwise} \end{cases}$

assuming that $\text{class } C \text{ extends } D' \{ \dots; \langle M_j^0 \rangle_j \} \in P$.

mbody(M^0) or mbody(m, τ) or mbody(m, T)	
mbody($S \tau m(\langle \tau_i x_i \rangle_i) \{ e^0 \}$)	$= (\langle x_i \rangle_i, e^0)$
mbody(m, τ)	$= \text{mbody}(\text{method}(m, \tau))$
mbody(m, T)	$= \text{mbody}(\text{method}(m, T))$

Variables returned by mbody are always fresh.

Figure 5. Preliminary definitions for operational semantics.

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 let -form execution by substitution. The local binding L found in LJ [17] 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 ??, because substitution here does not substitute into data in the heap.

$$\boxed{H, e^n \xrightarrow[n]{\text{prim}} H, e^n}$$

$$\frac{l \notin \text{dom } H}{H, \text{new } D(\langle l_i \rangle_i) \{ \langle M_j^0 \rangle_j \} \xrightarrow[0]{\text{prim}} H[l \mapsto (\text{sub } D \{ \langle M_j^0 \rangle_j \}, \langle l_i \rangle_i)], l}$$

$$\frac{H(l) = (T, \langle l_i \rangle_i) \quad \text{fields}(T) = \langle f_i \rangle_i}{H, l, f_{i_0} \xrightarrow[0]{\text{prim}} H, l_{i_0}}$$

$$\frac{H(l) = (T, \langle l_i \rangle_i)}{H, (l, f_{i_0} := l') \xrightarrow[0]{\text{prim}} H[l \mapsto (T, \langle l_i \rangle_i [i_0 \rightarrow l']), l']}$$

$$\frac{H(l) = (T, \dots) \quad \text{mbody}(m, T) = (\langle x_i \rangle_i, e^0)}{H, l.m(\langle l_i \rangle_i) \xrightarrow[0]{\text{prim}} H, [\langle l_i \rangle_i / \langle x_i \rangle_i][l/\text{this}]e^0}$$

$$\frac{l \notin \text{dom } H}{H, \text{let } x \leftarrow \text{new } C(\langle l_i \rangle_i) \text{ in } e^0 \xrightarrow[0]{\text{prim}} H[l \mapsto (C, \langle l_i \rangle_i), [l/x]e^0}$$

$$\frac{H(l) = (\text{Code}, \langle e^0 \rangle)}{H, \text{! } l \xrightarrow[1]{\text{prim}} H, e^0} \quad \frac{H(l) = (\text{Code}, \langle e^0 \rangle)}{H, l.\text{run}() \xrightarrow[0]{\text{prim}} H, e^0}$$

$$\frac{l \notin \text{dom } H}{H, \langle e^0 \rangle \xrightarrow[0]{\text{prim}} H[l \mapsto (\text{Code}, \langle e^0 \rangle)], l}$$

$$\boxed{H, \hat{e}^n \xrightarrow[n]{\text{prim}} H, \hat{e}^n}$$

$$\frac{H_1, e_1^k \xrightarrow[k]{\text{prim}} H_2, e_2^k}{H_1, \mathcal{E}^{n,k}[e_1^k] \xrightarrow[n]{\text{prim}} H_2, \mathcal{E}^{n,k}[e_2^k]}$$

Figure 6. Small-step semantics for Lightweight Mint.

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 [18], 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 6 gives preliminary definitions for the type system. A variable typing (or type environment) comes in pairs, separated by a $|$. The predicate $\text{iscf}(\tau)$ means that τ is code-free. Note that $\text{iscf}()$ is defined co-inductively. The auxiliary functions $\text{ftypes}()$, $\text{ftype}_i()$, and $\text{mtype}()$ are similar to those defined for the operational semantics, but they extract type information.

Figure 7 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 must be well-formed; the expression e contained 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 let forms and

$$\boxed{\text{typing terms}}$$

$$\begin{array}{ll}
\text{variable typing} & \Gamma : x \xrightarrow{\text{fin}} \tau^n \\
\text{store typing} & \Sigma : l \xrightarrow{\text{fin}} \tau \\
\text{variable typing pair} & \bar{\Gamma} ::= (\Gamma | \Gamma) \\
\text{pseudo-types} & \hat{\tau} ::= \tau | \langle \tau_i \rangle_i \xrightarrow{S} \tau
\end{array}$$

$$\boxed{\text{iscf}(\tau)}$$

$$\frac{}{\neg \text{iscf}(\text{Code}(S, \tau))} \quad \frac{}{\neg \text{iscf}(D)} \quad \frac{\exists i. \neg \text{iscf}(\text{ftype}_i(F))}{\neg \text{iscf}(F)}$$

NB: Object is a D .

$$\boxed{\text{cf}(\Sigma)}$$

$$\text{cf}(\Sigma) = \Sigma | L \text{ where } L = \{l \in \text{dom}(\Sigma) : \text{iscf}(\Sigma(l))\}.$$

$$\boxed{\text{cf}(\Sigma)}$$

$$\text{locs}(\hat{e}) = l : l \text{ is a subterm of } \hat{e}$$

$$\boxed{\text{ftypes}(\tau)}$$

$$\text{ftypes}(\tau) = \langle \tau_i \rangle_i \text{ assuming } \text{fields}(\tau) = \langle \tau_i f_i \rangle_i$$

$$\boxed{\text{ftype}_i(\tau) \text{ or } \text{ftype}(f, \tau)}$$

$$\text{ftype}_i(\tau) = \text{ftype}(f_i, \tau) = \tau_i \text{ assuming } \tau_i f_i \in \text{fields}(\tau)$$

$$\boxed{\text{mtype}(m, \tau) \text{ or } \text{mtype}(M^0)}$$

$$\begin{array}{l}
\text{mtype}(S \tau m \langle \tau_i x_i \rangle_i \{e^0\}) = \langle \tau_i \rangle_i \xrightarrow{S} \tau \\
\text{mtype}(m, \tau) = \text{mtype}(\text{method}(m, \tau)) \\
\text{assuming } \text{class } C \text{ extends } D' \{ \dots \} \in P
\end{array}$$

Figure 7. Preliminary definitions for the type system.

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 $\text{fields}(\tau)$, $\text{mtype}(m, \tau)$ 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 7 concerns typing for pseudo-expressions. This is given by the judgment $\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n \hat{e}^n : \hat{\tau} | S$ which states that the pseudo-expression \hat{e}^n has type $\hat{\tau}$ at level n under the stack $\langle \Sigma_i \rangle_i$ of store typings and the pair $\bar{\Gamma}$ of contexts. If $S = \text{sep}$, this judgment further states that the pseudo-expression \hat{e}^n is weakly separable. The reason the variable typing $\bar{\Gamma}$ is partitioned into two parts is to check weak separability: the right part of $\bar{\Gamma}$ 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 $\bar{\Gamma}$ and no locations are repeated in $\langle \Sigma_i \rangle_i$.

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 con-

<div style="border: 1px solid black; padding: 2px; margin-bottom: 10px;">$\tau \prec \tau'$</div> $\frac{}{\tau \prec \text{Object}} \quad \frac{}{C \prec C} \quad \frac{\tau_1 \prec \tau_2 \quad \tau_2 \prec \tau_3}{\tau_1 \prec \tau_3}$ $\frac{\tau \prec \tau'}{\text{Code}\langle S, \tau \rangle \prec \text{Code}\langle S, \tau' \rangle} \quad \frac{}{\text{Code}\langle \text{sep}, \tau \rangle \prec \text{Code}\langle \text{insep}, \tau \rangle}$ $\frac{\text{class } C \text{ extends } D \{ \dots \} \in P}{C \prec D}$	<div style="border: 1px solid black; padding: 2px; margin-bottom: 10px;">$\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n \text{sub } D \{ \langle M_i^n \rangle \}$</div> $\frac{\langle \langle \Sigma_i \rangle_i; \Gamma_1 \Gamma_2, \text{this} : D^n \vdash^n M_j^n : \tau_j S_j \rangle_j \quad n > 0 \vee \text{dom}(\cup_i \Sigma_i) \supseteq \text{locs}(\text{sub } D \{ \langle M_j^n \rangle_j \})}{\langle \Sigma_i \rangle_i; \Gamma_1 \Gamma_2 \vdash^n \text{sub } D \{ \langle M_j^n \rangle_j \}}$ <p>where $\tau_j = \text{mtype}(\text{mname}(M_j^n), D)$.</p> <div style="border: 1px solid black; padding: 2px; margin-bottom: 10px;">$\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n M^n : \langle \tau_i \rangle_i \xrightarrow{S} \tau S$</div> $\frac{\langle \Sigma_i \rangle_i, \Sigma; \Gamma_1, \Gamma_2, \langle x_i : \tau_i^n \rangle_i \emptyset \vdash^n e^n : \tau S}{\langle \Sigma_i \rangle_i; \Gamma_1 \Gamma_2 \vdash^n S \tau m(\langle \tau_i x_i \rangle_i) \{ e^n \} : \langle \tau_i \rangle_i \xrightarrow{S} \tau S'}$
<div style="border: 1px solid black; padding: 2px; margin-bottom: 10px;">$\vdash p$</div> $\frac{\vdash P \quad \langle \rangle; \emptyset \emptyset \vdash^0 e^0 : \tau S \quad \text{locs}(e^0) = \emptyset}{\vdash P, e^0}$	<div style="border: 1px solid black; padding: 2px; margin-bottom: 10px;">$\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash H$</div> $\frac{\forall l \in \text{dom}(\cup_i \Sigma_i). \langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash H(l) : (\cup_i \Sigma_i)(l)}{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash H}$
<div style="border: 1px solid black; padding: 2px; margin-bottom: 10px;">$\vdash P$</div> $\frac{\langle CL_i \rangle_i \text{ acyclic} \quad \text{no field names clash} \quad \langle \vdash CL_i \rangle_i}{\vdash \langle CL_i \rangle_i}$	<div style="border: 1px solid black; padding: 2px; margin-bottom: 10px;">$\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash h : \tau$</div> $\frac{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash h : \tau \quad \tau \prec \tau' \quad \langle (\cup_i \Sigma_i)(l_j) \prec \text{ftype}_j(C) \rangle_j}{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash h : \tau' \quad \langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash (C, \langle l_j \rangle) : C}$ $\frac{\langle \Sigma_i \rangle_i; \bar{\Gamma}^{\geq 1} \vdash^0 \langle e^0 \rangle : \text{Code}\langle S, \tau \rangle S'}{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash (\text{Code}, \langle e^0 \rangle) : \text{Code}\langle S, \tau \rangle}$
<div style="border: 1px solid black; padding: 2px; margin-bottom: 10px;">$\vdash CL$</div> $\frac{\langle \text{locs}(M_i^0) = \emptyset \rangle_i \quad \langle \langle \rangle; \emptyset \text{this} : C^0 \vdash^0 M_i^0 \rangle_i \quad \langle \text{mtype}(\text{mname}(M_i), D) = \text{undef or mtype}(M_i^0) \rangle_i}{\vdash \text{class } C \text{ extends } D \{ \langle \tau_j f_j \rangle_j; \langle M_i^0 \rangle_i \}}$	<div style="border: 1px solid black; padding: 2px; margin-bottom: 10px;">$\langle \Sigma_i \rangle_i; \bar{\Gamma}^{\geq 1} \vdash^0 \text{sub } D \{ \langle M_j^0 \rangle_j \}$</div> $\frac{\langle \langle \Sigma_i \rangle_i; \bar{\Gamma}^{\geq 1} \vdash^0 \text{sub } D \{ \langle M_j^0 \rangle_j \} \quad \langle (\cup_i \Sigma_i)(l_k) \prec \text{ftype}_k(D) \rangle_k}{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash (\text{sub } D \{ \langle M_j^0 \rangle_j \}, \langle l_k \rangle_k) : D}$ <p>where $\bar{\Gamma}^{\geq 1}(x) = \tau^n \iff \bar{\Gamma}(x) = \tau^n \wedge n \geq 1$ (likewise for Γ).</p>
<div style="border: 1px solid black; padding: 2px; margin-bottom: 10px;">$\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n e^n : \tau S$</div> $\frac{\tau' \prec \tau \quad \langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n \hat{e}^n : \tau' S}{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n \hat{e}^n : \tau S} \quad \frac{\bar{\Gamma}(x) = \tau^n \quad \text{isfc}(\tau) \vee k = 0}{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^{n+k} x : \tau S} \quad \frac{(\cup_i \Sigma_i)(l) = \tau \quad \text{isfc}(\tau) \vee n = 0}{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n l : \tau S}$ $\frac{\langle \langle \Sigma_i \rangle_i; \Gamma_1 \Gamma_2 \vdash^n e_j^n : \text{ftype}_j(C) S \rangle_j \quad \langle \Sigma_i \rangle_i, \Sigma; \Gamma_1, \Gamma_2 x : C^n \vdash^n e^n : \tau S}{\langle \Sigma_i \rangle_i; \Gamma_1 \Gamma_2 \vdash^n (\text{let } x \leftarrow \text{new } C(\langle e_j^n \rangle_j) \text{ in } e^n) : \tau S} \quad \frac{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n e^n : \tau S}{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n e^n . f : \text{ftype}(f, \tau) S} \quad \frac{\langle \langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n e_j^n : \tau_j S \rangle_{j=1}^2 \quad \text{ftype}(f, \tau_1) = \tau_2}{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n (e_1 . f := e_2) : \tau_2 \text{insep}}$ $\frac{\bar{\Gamma}(x) = \tau_1^n \quad \langle \Sigma_i \rangle_i; \Gamma_1 \Gamma_2 \vdash^n e^n : \tau_2 \text{sep} \quad \text{ftype}(f, \tau_1) = \tau_2 \quad \text{isfc}(\tau_1) \vee x \in \text{dom } \Gamma_2}{\langle \Sigma_i \rangle_i; \Gamma_1 \Gamma_2 \vdash^n (x . f := e^n) : \tau_2 \text{sep}} \quad \frac{(\cup_i \Sigma_i)(l) = \tau_1 \quad \text{isfc}(\tau_1) \vee (n = 0 \wedge l \in \text{dom } \Sigma_l) \quad \text{ftype}(f, \tau_1) = \tau_2 \quad \langle \Sigma_i \rangle_i^l; \bar{\Gamma} \vdash^n e^n : \tau_2 \text{sep}}{\langle \Sigma_i \rangle_i^l; \bar{\Gamma} \vdash^n (l . f := e^n) : \tau_2 \text{sep}}$ $\frac{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n e^n : \tau S \quad \langle \langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n e_j^n : \tau_j S \rangle_j \quad \text{mtype}(m, \tau) = \langle \tau_j \rangle_j \xrightarrow{S} \tau}{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n e^n . m(\langle e_j^n \rangle_j) : \tau S} \quad \frac{\langle \langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n e_j^n : \text{ftype}_j(D) S \rangle_j \quad \langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n \text{sub } D \{ \langle M_k^n \rangle_k \}}{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n \text{new } D(\langle e_j^n \rangle_j) \{ \langle M_k^n \rangle_k \} : D S}$ $\frac{\langle \Sigma_i \rangle_i; \Gamma_1, \Gamma_2 \emptyset \vdash^{n+1} e : \tau S}{\langle \Sigma_i \rangle_i; \Gamma_1 \Gamma_2 \vdash^n \langle e \rangle : \text{Code}\langle S, \tau \rangle S'} \quad \frac{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n e^n : \text{Code}\langle S, \tau \rangle \text{sep}}{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^{n+1} \epsilon e : \tau S} \quad \frac{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n e : \text{Code}\langle S, \tau \rangle S'}{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n e . \text{run}() \text{insep}}$	

Figure 8. Type system for Lightweight Mint.

text 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 `let`-expressions by extending the current context with the `let`-bound variable, while the rule after types 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 $\langle \Sigma_i \rangle_i$, to allow for the possibility of heap locations containing code objects with the variable x free.

The next three rules type field assignments ($e_1.f := e_2$) by checking the type of e_1 is some τ_1 and then checking that the type τ_2 of e_2 is the appropriate field type of τ_1 . The first of these rules applies to arbitrary e_1 , and requires τ_2 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 $\bar{\Gamma}$, 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, where typing $\langle e \rangle$ requires typing e at the next level and adds the code type, typing $\text{'}e$ requires typing e at a code type on the previous level and removes the code type, and typing $e.\text{run}()$ 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 $\text{'}e$ are only weakly separable if e has type $\text{Code}(\text{sep}, \tau)$.

The remainder of Figure 7 has rules for the following judgments. The judgment $\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n \text{sub } D \{ \langle M_i^n \rangle \}$ states that an AIC that subclasses D with method definitions $\langle M_i^n \rangle$ is well-formed. This requires the methods $\langle M_i^n \rangle$ to have the appropriate types. It also requires, if $n = 0$, that all the locations in the AIC are contained in $\text{dom}(\cup_i \Sigma_i)$, effectively ensuring that no new frames can be added to the stack of store typings. The judgment $\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n M^n : \langle \tau_i \rangle_i \xrightarrow{S} \tau | S$ states that method M has input types $\langle \tau_i \rangle_i$, output type τ , 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 $n > 0$. 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 $\bar{\Gamma}$ to the end, indicating that weakly separable expressions in the method cannot freely access variables bound at or before the method M .

The judgment $\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash H$ states that the store H is well-formed under the given stack of store typings. This judgment includes the typing context $\bar{\Gamma}$ 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 $\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash h : \tau$ is then used to state that heap form h has type τ . 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 $\bar{\Gamma}$ 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 $\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash H$ holds then H restricted to $\text{dom}(\cup_i \Sigma_i)$ is closed under reachability, meaning that no location in this domain can reference a location outside of it.

5.4 Soundness

We now sketch the key parts of our proof of Type Soundness. Complete proofs can be found in the technical report [22]. Type soundness is proved by the usual Preservation and Progress lemmas. Progress is proved with the following lemma:

Lemma 1 (Unique Decomposition). *If $\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n \hat{e}^n : \tau | S$ and \hat{e}^n is not a pseudo-value then \hat{e}^n is uniquely decomposed as $\hat{e}^n = \mathcal{E}^{n,m}[\tau^m]$, where $=$ denotes syntactic equality modulo α 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 $\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n (H, \hat{e}^n) : \tau | S$ then specifies that the configuration (H, \hat{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 be well-formed with respect to the current environment $\langle \Sigma_i \rangle_i; \bar{\Gamma}$. For example, the rule for `let` forms becomes:

$$\frac{\langle \langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n (H, e_j^n) : \text{ftype}_j(C) | S \rangle_j \quad \langle \Sigma_i \rangle_i; \Sigma; \bar{\Gamma}, x : C^n \vdash^n (H, e^n) : \tau | S \quad \langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash H}{\langle \Sigma_i \rangle_i; \bar{\Gamma} \vdash^n (H, \text{let } x \leftarrow \text{new } C \langle (e_j^n) \rangle_j \text{ in } e^n) : \tau | S}$$

A second technical difficulty is that a reduction step inside a `let` form or AIC that pushes a new frame Σ onto $\langle \Sigma_i \rangle_i$ might modify a code-free location in $\text{dom}(\cup_i \Sigma_i)$ to reference a location in the new frame Σ . 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 Σ 's of $\langle \Sigma_i \rangle_i$ into one, giving a shorter store typing sequence.

Lemma 2 (Smashing). *If*

1. $\langle \Sigma_i \rangle_i^I; \Gamma_1 | \Gamma_2 \vdash H_1$
2. $H_1|_L = H_2|_L$ where $L = \text{dom}(\cup_i \Sigma_i) - \text{dom}(\text{cf}(\cup_i \Sigma_i))$
3. $\langle \Sigma_i \rangle_i^I; \Sigma; \Gamma'_1 | \Gamma'_2 \vdash H_2$
4. $\Gamma'_1 \cup \Gamma'_2 \supseteq \Gamma_1 \cup \Gamma_2$

then $\langle \Sigma_i \rangle_i^{I-1}, (\Sigma_I \cup \text{cf}(\Sigma)); \Gamma_1 | \Gamma_2 \vdash H_2$.

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 those technicalities are left to the technical report.

Lemma 3 (Preservation). *If $\langle \Sigma_i \rangle_i; \Sigma_R; \Gamma_1 | \Gamma_2 \vdash^n (H_1, \hat{e}_1^n) : \tau | S$ and $(H_1, \hat{e}_1^n) \xrightarrow{n} (H_2, \hat{e}_2^n)$, then $\exists \Sigma'_R$ such that*

1. $\Sigma'_R \supseteq \Sigma_R$
2. $\langle \Sigma_i \rangle_i; \Sigma'_R; \Gamma_1 | \Gamma_2 \vdash^n (H_2, \hat{e}_2^n) : \tau | S$
3. $H_1|_L = H_2|_L$ where $L = \text{dom}(\cup_i \Sigma_i) - \text{dom}(\text{cf}(\cup_i \Sigma_i))$

Proof is by induction on the typing judgment.

6. Implementation

To verify the expressivity of the design and obtain performance results, we created an implementation of Mint by modifying OpenJDK [14], 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 Mint compiler introduces an additional compilation phase, Staging Translation, after the compiler has performed flow analysis. This phase uses a pretty-printer, which is built into OpenJDK for printing source, in order to translate the parts of brackets that are not escapes or cross-stage persistent variables into string fragments. For example, `<| 2 * ('x + 1) |>` is translated into a data structure containing the string `"2 * ("`, the expression `x`, and the string `") + 1)".` Cross-stage persistent variables are stored as values in a table in the code object in which they occur. In addition, for each binding construct in brackets, such as method definitions in anonymous inner classes or `let` expressions, the translation ensures that a fresh name is created for this variable at runtime. This is required to avoid accidental variable capture [5].

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.
- `fib` recursively computes the generalized Fibonacci function, using `let`-insertion to avoid code duplication in the staged version.
- `mmult` performs an optimized matrix multiplication with special treatment for every 1 and 0 in the left matrix.
- `eval-fact` calculates factorials using the `lint` interpreter discussed in Section 4.2.
- `eval-fib` calculates the standard Fibonacci sequence using the `lint` interpreter.
- `unroll` performs the loop unrolling example of Section 4.3.
- `serialize` performs serialization using the serializer generator discussed in Section 4.4.

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 OSX Tiger. More details of these benchmarks are available in the companion technical report [22].

The results are given in Figure 8. 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 `serializer` 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.

Benchmark	speed-up	unstaged μs	staged μs	gen μs	compile μs
<code>power</code>	4.6x	0.079	0.017	1.3	33,000
<code>fib</code>	4.1x	0.070	0.017	8.2	35,000
<code>mmult</code>	1.5x	1.8	1.3	12.0	84,000
<code>eval-fact</code>	8.4x	0.83	0.1	1.7	37,000
<code>eval-fib</code>	10.0x	19.0	1.9	2.4	57,000
<code>unroll</code>	1.4x	0.140	0.097	2.9	47,000
<code>serialize</code>	18.0x	1.5	0.08	6.2	35,000

Figure 9. Benchmark results.

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 program annotations 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 [9, 10]. While the approach makes use of advanced 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 Aktemur extending this system, explicitly include the types of all free variables in the type of a code fragment [12, 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. A 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 [11] adds MSP to Java, while Meta-AspectJ [23] 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 [13] 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 [8] and MorphJ [7] 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 languages allow 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 [15], 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 [16], 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

- [1] Baris Aktumur. Type Checking Program Generators Using the Record Calculus, 2009. <http://loome.cs.uiuc.edu/pubs/transformationForTyping.pdf>.
- [2] Davide Ancona and Eugenio Moggi. A fresh calculus for name management. In *GPCE '04: Proceedings of the 3rd International Conference on Generative Programming and Component Engineering*, volume 3286, pages 206–224, 2004.
- [3] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. Closed Types as a Simple Approach to Safe Imperative Multi-stage Programming. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 25–36, 2000.
- [4] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. ML-like inference for classifiers. In *ESOP '04: Proceedings of the 13th European Symposium on Programming*, pages 79–93, 2004.
- [5] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In *GPCE '03: Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, pages 57–76, New York, NY, USA, 2003. Springer-Verlag New York, Inc.
- [6] Manuel Fähndrich, Michael Carbin, and James R. Larus. Reflective program generation with patterns. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 275–284, 2006.
- [7] Shan Shan Huang and Yannis Smaragdakis. Expressive and safe static reflection with MorphJ. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 79–89, 2008.
- [8] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically safe program generation with safegen, 2005.
- [9] Yuki Yoshi Kameyama, Oleg Kiselyov, and Chung chieh Shan. Closing the stage: from staged code to typed closures. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 147–157, 2008.
- [10] Yuki Yoshi Kameyama, Oleg Kiselyov, and Chung chieh Shan. Shifting the stage: Staging with delimited control. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 111–120, 2009.
- [11] Sam Kamin, Lars Clausen, and Ava Jarvis. Jumbo: Run-time code generation for Java and its applications. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 48–56, 2003.
- [12] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. *SIGPLAN Not.*, 41(1):257–268, 2006.
- [13] Gregory Neverov and Paul Roe. Metaphor: A Multi-stage, Object-Oriented Programming Language. In *GPCE '04: Proceedings of the 3rd International Conference on Generative Programming and Component Engineering*, pages 168–185, 2004.
- [14] OpenJDK Project. <http://openjdk.java.net>.
- [15] U.P. Schultz and J.L. Lawall C. Consel. Automatic Program Specialization for Java. *ACM Transactions on Programming Languages and Systems*, 25(4):452–499, 2003.
- [16] Peter Sestoft. Runtime code generation with JVM and CLR, 2002. <http://www.itu.dk/~sestoft/rtcg/rtcg.pdf>.
- [17] Rok Strniša, Peter Sewell, and Matthew Parkinson. The Java module system: Core design and semantic definition. In *OOPSLA '07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, pages 499–514, 2007.
- [18] Walid Taha. *Multistage programming: Its theory and applications*. PhD thesis, Oregon Graduate Institute, 1999. Supervisor-Tim Sheard.
- [19] Walid Taha. A gentle introduction to multi-stage programming. In *DSPG '03: Proceedings of the International Seminar on Domain-Specific Program Generation*, 2003.
- [20] Walid Taha, Zine el-abidine Benaïssa, and Tim Sheard. Multi-Stage Programming: Axiomatization and Type Safety (Extended Abstract). In *ICALP '98: 25th International Colloquium on Automata, Languages, and Programming*, pages 918–929, 1998.
- [21] Walid Taha and Michael Florentin Nielsen. Environment classifiers. *SIGPLAN Not.*, 38(1):26–37, 2003.
- [22] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. TR09-02: Multi-stage Programming for Mainstream Languages. Technical report, Rice University, 2009. http://compsci.rice.edu/TR/TR_Download.cfm?SDID=256.
- [23] David Zook, Shan Shan Huang, and Yannis Smaragdakis. Generating AspectJ Programs with Meta-AspectJ. In *GPCE '04: Proceedings of the 3rd International Conference on Generative Programming and Component Engineering*, pages 1–18, 2004.