Multi-stage Programming for Mainstream Languages

Edwin Westbrook Mathias Ricken Jun Inoue Yilong Yao Tamer Abdelatif[1] Walid Taha

Rice University

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

Abstract

Multi-stage programming (MSP) 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 how well useful MSP applications can be expressed and how easy it is for programmers to understand. The key insight is that escapes (or "anti-quotes") must be *weakly separable* from the rest of the code, i.e. the computational effects occurring inside an escape that are visible outside the escape are guaranteed to not 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 type system and the effect of staging on the performance of Java programs.

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. Our goal is to allow software developers to use such abstractions, but have them be executed in an efficient manner. 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 developer can change programs that use expensive abstractions into program *generators*, which generate programs without the abstractions. This reduces the runtime cost of abstractions, because the overhead is only paid when the generators are executed, not at the time the programs they generate are run.

A key issue for MSP is type safety, which ensures statically that all programs generated at runtime will be well-formed. Although it had been known how to ensure type safety for MSP in the purely functional setting [4, 28, 29], it previously remained an open challenge how to extend this guarantee to mainstream languages such as Java. In particular, standard features of mainstream languages, such as imperative assignment, can 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, 11, 12, 14] 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. Mint provides a type system that is well-suited for the prevalent culture of programming in languages like Java. Specifically, Mint allows imperative generators and does not require a functional programming style, a style that often is inconvenient in Java. The type system remains simple while being more expressive than any previous type system that statically ensures type safety of generated code.

[1] Ain Shams University.

Contributions

To make MSP accessible to programmers in mainstream languages, we propose a new approach to type-safe MSP that we argue occupies a "sweet spot" in the design space in terms of how well useful MSP applications can be expressed and how easy it is for programmers to understand. Our contributions include:

- After a brief introduction to staging in Java (Section 2), we analyze and explain why scope extrusion can arise with naive approaches to MSP in Java or in similar languages (Section 3).
- We introduce the notion of *weak separability* as a solution to the scope extrusion problem (Section 4). Weak separability ensures that any effects that can be observed outside escaped expressions will not involve code objects. We present a type system that realizes this idea, and show that it is sufficient to prevent scope extrusion.
- We demonstrate the expressivity of the type system using a number of small examples illustrative of important classes of programs for which MSP can be useful (Section 5). The examples show the use of MSP for building interpreters, numerical code, and reflective programs. The examples also emphasize that the type system allows expressing imperative generators that, for example, throw exceptions or store code in locations that cannot escape the scope of a surrounding dynamic binder.
- We formalize the semantics and prove the type safety of a core calculus illustrating the key features of the proposed type system. (Section 6). Proving type safety establishes that a well-typed program is guaranteed to be free of any runtime errors, including possible scope extrusion and generation (and execution) of ill-formed code. The novelty of the type system is in the use of a stack of store typings, instead of a single store typing, to mirror the dynamic binding structure of the term. This captures the fact that heap locations allocated inside dynamic binders may not be visible outside those binders. Full proofs are available in Appendix A.

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

(a) The Unstaged Power Function

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

(b) The Staged Power Function

Figure 1. Staging the Power Function in Mint

- We have implemented the type system in an MSP extension of Java OpenJDK called Mint [18] (Section 7). The implementation was used to type check all examples presented in the paper.
- We use the implementation to confirm that MSP in Java — a language typically implemented using JIT compilation — can lead to performance speedups similar to those seen by MSP extensions of other languages (Section 8).

2. Multi-Stage Programming in Mint

Mint extends Java 6 with the three standard MSP constructs: brackets, escape, and run (see e.g. [26]). 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 the block is surrounded by curly braces:

```
<| { C.foo();
     C.bar(); } |> // has type Code<Void>
```

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.

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

```
int x = 1;
Code<Integer> c = <| x + 1 |>;
```

Weak separability, the requirement Mint uses to ensure safety, places certain restrictions on CSP; see Section 4.

Basic MSP in Mint can be illustrated using the classic power example. Figure 1(a) displays the unstaged power function in Java. Figure 1(b) displays a staged version. The staged method `spower` takes in an argument `x` that is a piece of code for an integer, along with an integer `n`, and returns 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 in the purely functional setting [4, 28, 29]. In the presence of imperative features, however, guaranteeing this is more challenging 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 be emitted, 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:

   ```
   Code<Integer> x;
   <| { Integer y = foo();
        ‘(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 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

```
<| { Integer y = foo();
     Code<Integer> d = T.call(); } |>
```

where `T` is the anonymous inner subclass of `Thunk`. In a substitution-based semantics that substitutes values directly for variables, no scope extrusion would occur, because running this code object would substitute the return value of `foo()` for `y` in `T`, thus producing a new copy of `T` whose `call` method returns `<| r |>`, where `r` is the return value of `foo()`. Such a semantics, however, would be impractical for a language like Java, 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. In an environment-based semantics, which moves the value to some location in a separate environment and substitutes an index of that location (usually the variable itself) for the variable, running the above code would simply return the value of `T.call()`, which would produce the code object `<| y |>`, with `y` being out of its scope. The cross-stage persistence of the `Thunk` is essentially the same as storing the value of `T` in a global reference cell or hash table, and as soon as that is done, the `Thunk` has been moved outside the scope of `y`. Our formalization of Mint properly models this indirection, although we model the environment as part of the heap rather than as a separate entity (see Section 6).

4. Weak Separability

The three situations mentioned in the previous section necessarily all involve code objects; effects that do not involve code cannot cause scope extrusion. We therefore first define the term 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, all of its methods' return types are code-free, and its class is *final*. A *value* is **code-free** if its type is code-free.

If a value `x` is code-free, then no code object is reachable from `x`, and scope extrusion cannot occur due to effects involving `x`.

The requirement that a class is final means that the class is not allowed to be subclassed. This restriction ensures that a subclass with an additional field of type `Code<T>` cannot be substituted at runtime. Making Java classes final is unusual, but only because there are almost no benefits to declaring a class final. In our experience, most Java classes written by application developers can be final.

Commonly used 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.

Scope extrusion can be prevented by requiring that escapes are *weakly separable*, which we define informally:

Informal Definition 1. A term is **weakly separable** if both side effects observable outside the term and cross-stage persistence involve 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. [11, 12] to ensure that no effects occurring inside escapes could be visible outside the escapes.[1] We call the condition introduced by Kameyama et al. **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 undecidable in general. Instead, we provide a conservative approximation of weak separability that is used by Mint. This approximation is decidable, and we show below that it still leaves an expressive language. Unless otherwise specified, the phrase "weakly separable" in the remainder of this document refers to this approximation:

Definition 2. A Mint term e is **weakly separable** iff:

1. Assignment is made only to variables bound within e, or to fields or variables of code-free types;
2. Exceptions are only thrown by a `throw new C(e1, ..., en);` construct where the `ei` are code-free, or the exception is caught by an enclosing `try-catch` construct before it can leave e;
3. Cross-stage persistence occurs only for `final` variables of code-free types;
4. Only weakly separable methods and constructors are called.

We are using the word *term* to describe an expression, statement, or method body. The notion of *pseudo-expression* in Section 6.2 is a direct formalization of this kind of term.

The first three of the clauses in the definition above directly preclude the three cases of scope extrusion in the previous section. The restriction on throwing exceptions is syntactic and allows for a static check of code-freedom. 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 term 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`.

[1] Technically, Kameyama et al. placed this restriction on future-stage binders.

```
interface Thunk { Code<Integer> call(); }

Code<Code<Integer>> doCSP(Thunk t) {
 return <| t.call() |>;
}

<| {
 Integer y = foo();
 Code<Integer> d = `(doCSP(new Thunk() {
                       Code<Integer> call() {
                         return <| y |>;
                       }}));
} |>.run();
```

Figure 2. Cross-stage Persistence of Code Objects

To demonstrate how weak separability works in practice, the following code gives examples showing how each of the four clauses of weak separability can fail. Since all of these examples occur in the method `foo`, which is marked as separable, each of these examples represents a type error in the program. The first example, which assigns to `x.c`, violates clause 1 of Definition 2 since `x.c` has the non-code-free type `Code<C>`. The second example violates clause 2 since it passes a code object to the constructor of an exception which is being thrown. The third example violates clause 3 because it uses cross-stage persistence on the value `x` of type `C`; the type `C` is not code-free because it contains the field `c` of type `Code<C>`. Finally, the fourth example violates clause 4 because it calls the method `bar()`, which is not marked with the `separable` keyword.

```
class C {
  Code<Integer> c;

  void bar() { ... }

  separable C foo(final C x) {
    // x.c is not code-free (1)
    x.c = <|1|>;

    // CodeContainerException takes a
    // non-code-free argument (2)
    throw new CodeContainerException(<|x|>);

    // not a 'throw new C(e1, ..., en);'
    // construct (2)
    CodeFreeException e =
      new CodeFreeException();
    throw e; // error

    // CSP of x but C is not code-free (3)
    Code<C> y = <|x|>;

    // bar() is not marked separable (4)
    bar();
  }
}
```

5. Expressivity

Weak separability is a highly expressive notion that excludes only a few coding patterns, such as generators that store open code in a global data structure. Such situations can be addressed by introducing a local data structure and ensuring that it is not used outside the dynamic binder. The separability restriction is only problematic when the coding pattern *requires* scope extrusion.

In an imperative language like Java, being able to write code generators with side effects is a desirable property: An imperative programming style fits better into the language's culture, while functional programming in Java can often be inconvenient and verbose.

Weak separability does not severely restrict expressiveness because there is a tendency for the computational effects used in code generators to be separable: It is rare that generators are required to export code objects through side effects; all other side effects not involving code are weakly separable and therefore allowed. Furthermore, the `run()` method is only called outside of any brackets in almost all applications of MSP, and cross-stage persistence is mostly used for primitive types. Generated code is not restricted by weak separability at all.

To illustrate these points, the remainder of this section describes the implications of weak separability and examines a number of MSP examples in Mint, namely: staging an interpreter, a classic MSP example; staging array views [22] to remove abstraction overhead; and loop unrolling. The staged interpreter shows that throwing a code-free exception in a code generator is allowed. Both staging array views and loop unrolling demonstrate generators for imperative code. Section 5.4 gives another example, 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 [26, 27]. To demonstrate that staged interpreters can be written in Mint, we have implemented an interpreter for a small programming language called `lint` [26], which supports integer arithmetic, conditionals, and recursive function definitions of one variable.

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. The environments are modeled as functions, implemented using anonymous inner classes. The empty environments unconditionally throw an exception.

Integers, addition, variables and application of defined functions, for example, are implemented as follows:

```
interface Exp {
  public int eval(Env e, FEnv f);
}
class Int implements Exp {
  private int _v;
  public Int(int value) { _v = v; }
  public int eval(Env e, FEnv f) {
      return _v;
} }
class Add implements Exp {
  private Exp _l, _r;
  public Add(Exp l, Exp r) { _l = l; _r = r; }
  public int eval(Env e, FEnv f) {
    return _l.eval(e, f) + _r.eval(e, f);
} }
class Var implements Exp {
  private String _s;
  public Var(String s) { _s = s; }
  public int eval(Env e, FEnv f) {
    return e.get(_s);
} }
class App implements Exp {
  private String _s;
  private Exp _a; // argument
  public App(String s, Exp a) {
   _s = s; _a = a;
  }
  public int eval(Env e, FEnv f) {
    return f.get(_s).apply(_a.eval(e, f));
} }
```

Variable lookup is performed in a variable environment by calling the `Env.get(String s)` method, returning an integer. In applications, function lookup is done using the `FEnv.get(String s)` method, returning a `Fun` object with an `int apply(int v)` method, which is then applied to the argument of the application.

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

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

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

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

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

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

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

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

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

```
interface Exp {
  public separable
  Code<Integer> eval(Env e, FEnv f);
}
class Int implements Exp { /* ... */
  public separable
  Code<Integer> eval(Env e, FEnv f) {
    final int v = _v; return <| v |>;
} }
class Var implements Exp { /* ... */
  public separable
  Code<Integer> eval(Env e, FEnv f) {
    return e.get(_s);
} }
class Add implements Exp { /* ... */
  public separable
  Code<Integer> eval(Env e, FEnv f) {
    return <| ‘(_l.eval(e,f)) +
              ‘(_r.eval(e,f)) |>;
} }
class App implements Exp { /* ... */
  public separable
  Code<Integer> eval(Env e, FEnv f) {
    return <| ‘(f.get(_s)).apply(
              ‘(_a.eval(e,f))) |>;
} }
```

The `Int` and `Var` classes are straight-forward. Operations like `Add` recursively evaluate their subtrees and splice together the returned code objects. Function application in `App` again uses the `get` method 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 `_a.eval` escapes both the function and the code value from evaluating the argument, and then returns code to apply the function to the argument.

If `_s` does not name a valid, defined function, then the `get` method throws an exception. This is the only computational effect in the whole staged interpreter that happens inside a code generator. It is weakly separable because the thrown exception need only contain the string argument `_s` that was not found in the environment; the exception therefore is code-free.

The functions to extend the environments associate names with code objects now. During lookup, we have to use the `==` operator instead of the `String.equals` method, because the latter has not been declared separable. This is not a serious impediment, though, since Java strings are immutable and can be interned. Staging `Env.ext` yields:

```
static separable Env ext(final Env env,
  final String x, final Code<Integer> v) {
  return new Env() {
    public separable
    Code<Integer> get(String y) {
    // error: if (x.equals(y))
    if (x==y) return v;
    else return env.get(y);
} }; }
```

In the code that creates recursive functions, we again use anonymous inner classes for closures. Since the function environment associates names with code for functions, we have to put a reference to the function we are creating in brackets. However, a reference to `this` is not allowed, so we create a `final` local variable `fthis` and return code for it.

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

Evaluating a program is now a two-step process. The `eval` method now returns code for an integer, running that code returns the integer. Section 8 provides performance comparisons between staged and unstaged interpretation.

5.2 Array Views

As discussed, weak separability does not restrict the computational effects in generated code; it only restricts effects in the code generators themselves. As an example of this, we use staging to remove overhead in array views, which are useful for parallel programming.

It can be challenging for a compiler to parallelize Java code that uses multi-dimensional arrays, implemented in Java as nested one-dimensional arrays. This arrangement prevents the compiler from assuming that `A[i][j]` and `A[i+1][j]` refer to different locations. In one-dimensional arrays, each index refers to a different location, and this knowledge allows for simpler parallelization.

To address the problems with multi-dimensional arrays in Java, the Habanero project [8] provides *array views*, which map subsections of multi-dimensional arrays to one-dimensional arrays without requiring the programmer to perform the arithmetic manually [22].

Much simplified, an array view stores a reference to a one-dimensional base array and provides `get` and `set` methods for several numbers of dimensions. The example below shows `get` and `set` for a two-dimensional `double` array:

```
class DoubleArrayView {
 double[] base;
 public double get(int i, int j) {
  return base[offset + (j-j0) + jSize*(i-i0)];
 }
 public void set(double v, int i, int j) {
  base[offset + (j-j0) + jSize*(i-i0)] = v;
}}
```

The methods calculate the index in the base array for the coordinates (here: `i`, `j`) using the minimum values for those dimensions (here: `i0`, `j0`), the size of the dimensions (here: `iSize`, `jSize`) and the start index of the view (`offset`). This calculation is performed for every array access.

The overhead of this calculation can be removed by staging the array view. The base array itself and the parameters `i0`, `j0`, `iSize`, `jSize`, and `offset` describing the array view are replaced by code values that can be spliced together as needed. The `get` method returns `Code<Double>`, code to retrieve the value in the array. The `set` method returns `Code<Void>` and is a generator for code performing the array assignment.

```
class SDoubleArrayView {
 Code<double[]> base;
 public separable
 Code<Double> get(final int i, final int j) {
  return <| ‘(base)[‘(offset) + (j-‘(j0)) +
                    ‘(jSize)*(i-‘(i0))] |>; }
 public separable
 Code<Void> set(final Code<Double> v,
                final int i, final int j) {
  return <| {
    ‘(base)[‘(offset) + (j-‘(j0)) +
            ‘(jSize)*(i-‘(i0))] = ‘(v); } |>; }
}
```

Staging allows us to work with the array view at a high level without paying for the overhead at runtime. For example, a matrix transpose can be written as

```
public Code<Void> stranspose(int m, int n,
 final SDoubleArrayView input,
 final SDoubleArrayView output) {
 Code<Void> stats = <| { } |>;
 for(int i = 0; i < m; i++)
   for(int j = 0; j < m; j++)
     stats = <| {
       ‘stats;
       ‘(output.set(input.get(i,j),j,i)); } |>;
 return stats;
}
Code<Void> c = stranspose(4, 4, a, b);
```

This method generates code consisting of direct array accesses:

```
b[0+(0-0)+4*(0-0)] = a[0+(0-0)+4*(0-0)];
b[0+(0-0)+4*(1-0)] = a[0+(1-0)+4*(0-0)]; // ...
```

An optimizing compiler will replace the computations with constants, completely removing the cost of the abstraction in the generated code.

Note that the size of the dimensions does not have to be known statically, but at generation time; only the number of dimensions of the array has to be known statically. Therefore, the size of the array can vary, and this optimization is applicable in many situations.

This example performs loop unrolling as well. The code objects that perform the array assignments, returned by the `set` method, are accumulated into another code object `stats`. At the end of the two nested loops, `stats` will contain the sequence of statements for the entire matrix transpose operation. The code generator is written in an imperative style consistent with the prevalent Java culture. The body of the method is weakly separable because `stats` is bound inside the method; the code inside `stats` returned by the `stranspose` method is not weakly separable, but that is not required of generated code.

5.3 Loop Unrolling

The example above demonstrates how to unroll a specific loop. This can be done for a generic loop, which can be expressed 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 `cell`:

```
static class sIncrIter implements SIter {
  Code<IntCell> cell;

  public separable Code<Void>
  iteration(final int i) {
    return <| { (‘cell).value += i; } |>;
  }
}
```

Staged array views and loop unrolling serve as compelling examples how MSP can reduce abstraction overhead in an imperative setting.

5.4 Staged Reflection Primitives

Neverov observed that staging and reflection in languages like C# and Java can be highly synergistic [15]. 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 the standard reflection primitives in the Java library, including the `Class<A>` and `Field` classes. [2]

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 `Class<A>` it is modeled after. For example, the corresponding class for `Integer` objects has type `ClassCode<Integer>`. Any `ClassCode<A>` object provides methods for manipulating the 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 a 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 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<B>`. This method constructs field selection (intuitively, a `<| (‘a).f |>` code fragment) on that object. The type also provides a `getFieldClassCode` method to return a `ClassCode<B>` object for the type `B`.

The fields of a class are returned using the `getFields()` method in `ClassCode<A>`. The return type of `getFields()` is `FieldCode<A,?>[]`, where the `?` represents an existential type in Java's generics. The method therefore returns an array of fields contained in class `A`, all of which have "some" type.

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 produces code to serialize all of a given object's fields without reflection:

```
public static separable <A> Code<Void>
sserialize(ClassCode<A> t, final Code<A> o) {
  if (t.getCodeClass()==Byte.class)
    return <| {
```

[2] The Mint reflection library does not support all reflection primitives. For example, `Method` and `Constructor` have multiple parameters. This would require adding indexed types to Java, and is therefore outside the scope of this work.

```
      writeByte(‘((Code<Byte>)o)); } |>;
  else if (t.getCodeClass()==Integer.class)
    return <| {
      writeInt(‘((Code<Integer>)o)); } |>;
  Code<Void> result = <| { } |>;
  for(FieldCode<A,?> fc: t.getFields()) {
    result = <| { ‘result;
      ‘(sserializeField(fc, o)); } |>; }
  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 [15].

6. 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 [25] (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, for example, in Figure 2 of Section 3, which uses an AIC in combination with CSP to perform scope extrusion. We wish to show that such possibilities are prevented by our system.

A noteworthy feature of the type system is the use of a *stack* of store typings instead of a single store typing. The standard way to express type preservation of programs with mutable references is to have a store typing Σ that assigns types to locations that are allocated during evaluation. Preservation then states that, if e_1, typable with store typing Σ, takes a computation step to e_2 (possibly allocating new store locations), then e_2 is typable with a new, possibly extended, store typing Σ'. Instead, we use a stack of store typings, represented as a sequence. Each time evaluation enters the scope of a variable binding, say of `x`, we push a new Σ to the right of the sequence that may type heap locations holding code values containing `x` free. During execution, type assignments to non-code-free heap locations l allocated within this scope are always added to this Σ. The Σ is popped from the right upon leaving the scope of `x`, so references to l are only typable in the scope of `x`.

When we pop a Σ from the stack of Σ's, we always salvage code-free locations and tack them on to the Σ's that remain, so that those locations can be used elsewhere. The Smashing Lemma guarantees that this contraction, or smashing, of the store typing stack is safe. For this strategy to work, the Σ's must obey a condition somewhat similar to the Barendregt variable convention: whenever Σ is appended to the store typing stack, the locations in Σ must be fresh with respect to other Σ's in the same typing derivation; i.e., the domains of all distinct Σ's appearing in a single derivation tree must be disjoint. Otherwise, a location l that types $\langle\!|\texttt{x}|\!\rangle$ under one Σ can be used to type an incompatible object in a disjoint subtree of the derivation tree. A more rigorous definition of this constraint can be found in the appendix.

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	$::= \texttt{sep} \mid \texttt{insep}$
types	τ	$::= C \mid \texttt{Code}\langle S, \tau\rangle$
class declarations	CL	$::= \texttt{class}\ C\ \texttt{extends}\ D$ $\texttt{\{}\langle\tau_i\ f_i\rangle_i^I; \langle M_j^0\rangle_j^J\texttt{\}}$
method declarations	M^n	$::= S\ \tau\ m\texttt{(}\langle\tau_i\ x_i\rangle_i\texttt{)\{}e^n\texttt{\}}$
class hierarchy	P	$::= \langle CL_i\rangle_i$
programs	p	$::= P, e^0$
expressions	e^n	$::= x \mid l \mid e^n\texttt{.}f \mid (e^n\texttt{.}f := e^n)$ $\mid\ e^n\texttt{.}m\texttt{(}\langle e_i^n\rangle_i\texttt{)}$ $\mid\ \texttt{let}\ x \Leftarrow \texttt{new}\ C\texttt{(}\langle e_i^n\rangle_i\texttt{)}\ \texttt{in}\ e^n$ $\mid\ \texttt{new}\ D\texttt{(}\langle e_i^n\rangle_i^I\texttt{)\{}\langle M_j^n\rangle_j^J\texttt{\}}$ $\mid\ \langle\!|e^{n+1}|\!\rangle \mid\ ‘e^{n-1}[n>0]$ $\mid\ e^n\texttt{.run()}$
values	v^n	$::= l \mid e^{n-1}[n>0]$

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

Figure 3. Lightweight Mint syntax.

To simplify the formalism, LM disallows assignments to local variables; all assignments must be to object fields. This restriction by itself would completely rule out assignments in escapes, however. 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` that 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_cell.x`. Thus, we model the environment as part of the heap, meaning that the values of variables are always heap locations. This is equivalent to an environment-based semantics, which models variables as locations in a special environment object.

6.1 Syntax

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

Notation. We write $\langle a, b, c, \ldots\rangle$ for sequences, with the shorthand $\langle A_i\rangle_{i=I}^J$ for $\langle A_I, A_{I+1}, \ldots, A_J\rangle$. I may be omitted, and it defaults to 1. J may also be omitted when clear from context. The empty sequence is written $\langle\rangle$. Concatenation of sequences s_1 and s_2 is written $s_1 \circ s_2$, with the shorthand $\langle A_i\rangle_i, A$ for $\langle A_i\rangle_i \circ \langle A\rangle$. We also use $\langle e_i\rangle_{i=I}^J[i_0 \to x]$ to denote $\langle e_i\rangle_i$ with e_{i_0} 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 any level $n > 0$ is any expression at level $< n$.

Remark. The n in e^n is a constraint on the shapes of terms that this metavariable ranges over and is not something that forms a part of any concrete term. Similar precautions apply to other superscripted metavariables in this formalism, like v^n and M^n.

We categorize classes (C) 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. The types (τ) include classes as well as the code type $\texttt{Code}\langle S, \tau\rangle$, which is considered distinct from the classes.

This code type $\texttt{Code}\langle S, \tau\rangle$ is indexed by a separability marker S, which indicates whether a code object is itself separable, and the type τ of the expression in the code object. Specifically, $\texttt{Code}\langle \texttt{sep}, \tau\rangle$ is the type of code objects containing separable code, which is a subtype of the standard code type, written $\texttt{Code}\langle \texttt{insep}, \tau\rangle$. This distinction is necessary in the case of a separable expression which itself contains a nested escape $\texttt{`}e$, since we must know for type preservation that $\texttt{`}e$ is guaranteed to reduce only to separable code. In this case, e must have type $\texttt{Code}\langle \texttt{sep}, \tau\rangle$.

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 ($\texttt{new}\ C\texttt{(...)}$) for instantiating ordinary (i.e., non-AIC) classes because one can write ($\texttt{let}\ x \Leftarrow \texttt{new}\ C\texttt{(...)}\ \texttt{in}\ x$) instead. Sequencing ($e_1; e_2$) is also omitted because this code can be written $seq.call(e_1, e_2)$, where $seq.call$ is a method that ignores its first argument and returns its second.

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.

6.2 Operational Semantics

Figure 5 shows the small-step semantics for Lightweight Mint. Ancillary definitions are found in Figure 4. Figure 5 defines the judgment $H_1, \widehat{e}_1 \overset{n}{\rightsquigarrow} H_2, \widehat{e}_2$ which states that heap H_1 and pseudo-expression $\widehat{e}_1$ evaluate in a single step at level n to H_2 and $\widehat{e}_2$, respectively. The notion of heap is standard; 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. The notion of a pseudo-expression, defined as either an expression or a method body, is used because execution can occur within bodies of methods defined at level > 0. Similarly, we define the notion of pseudo-value, which is either a value or a method body with no unresolved escapes at the current level.

Figure 5 contains a number of helper functions. $\mathrm{fields}()$ extracts the fields of a type. $\mathrm{method}()$ looks up a method, respecting overriding rules. $\mathrm{mbody}()$ extracts the specified method's formal arguments and body. Code types do not have methods (`run()` is formally not a method). mname extracts the method name from a method declaration.

The single-step evaluation judgment $\overset{n}{\rightsquigarrow}$ is defined as the closure of the primitive one-step relation $\underset{\text{prim}}{\overset{k}{\rightsquigarrow}}$ under n, k-evaluation contexts $\mathcal{E}^{n,k}$. These are pseudo-expressions with a hole $\bullet$. This hole can be filled with an expression e, written $\mathcal{E}^{n,k}[e]$; the superscripts n and k express that filling the hole with a level k expression yields a level n pseudo-expression. Most of the primitive reduction steps

operational terms

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

evaluation contexts

$$
\begin{aligned}
\mathcal{E}^{n,k} &::= \mathcal{E}_e^{n,k} \mid \mathcal{E}_M^{n,k} \\
\mathcal{E}_M^{n,k} &::= S\ \tau\ m\texttt{(}\langle \tau_i\ x_i\rangle_i\texttt{)\{}\mathcal{E}_e^{n,k}\texttt{\}}[n > 0] \\
\mathcal{E}_e^{n,k} &::= \bullet[n = k] \mid \mathcal{E}_e^{n,k}\texttt{.}f \mid (\mathcal{E}_e^{n,k}\texttt{.}f := e^n) \\
&\quad \mid (v^n\texttt{.}f := \mathcal{E}_e^{n,k}) \mid \mathcal{E}_e^{n,k}\texttt{.}m\texttt{(}\langle e_i^n\rangle_i\texttt{)} \\
&\quad \mid v^n\texttt{.}m\texttt{(}\langle v_i^n\rangle_i, \mathcal{E}_e^{n,k}, \langle e_j^n\rangle_j\texttt{)} \\
&\quad \mid \texttt{let}\ x \Leftarrow \texttt{new}\ C\texttt{(}\langle v_i^n\rangle_i, \mathcal{E}_e^{n,k}, \langle e_j^n\rangle_j\texttt{)}\ \texttt{in}\ e^n \\
&\quad \mid \texttt{let}\ x \Leftarrow \texttt{new}\ C\texttt{(}\langle v_i^n\rangle_i\texttt{)}\ \texttt{in}\ \mathcal{E}_e^{n,k}[n > 0] \\
&\quad \mid \texttt{new}\ D\texttt{(}\langle v_i^n\rangle_i, \mathcal{E}_e^{n,k}, \langle e_j^n\rangle_j\texttt{)\{}\langle M_a^n\rangle_a\texttt{\}} \\
&\quad \mid \texttt{new}\ D\texttt{(}\langle v_i^n\rangle_i\texttt{)\{}\langle M_j^{n-1}\rangle_j, \mathcal{E}_M^{n,k}, \langle M_a^n\rangle_a\texttt{\}}[n > 0] \\
&\quad \mid \langle\!\langle \mathcal{E}_e^{n+1,k} \rangle\!\rangle \mid \texttt{`}\mathcal{E}_e^{n-1,k}[n > 0] \mid \mathcal{E}_e^{n,k}\texttt{.run()}
\end{aligned}
$$

$\mathrm{fields}(T)$

$$
\begin{aligned}
&\mathrm{fields}(\texttt{Object}) = \mathrm{fields}(\texttt{Code}) = \mathrm{fields}(\texttt{Code}\langle S, \tau\rangle) = \langle\rangle \\
&\mathrm{fields}(\texttt{sub}\ D\ \texttt{\{}\langle M_i^0\rangle_i\texttt{\}}) = \mathrm{fields}(D) \\
&\mathrm{fields}(C) = \langle \tau_i\ f_i\rangle_i \circ \mathrm{fields}(D') \\
&\text{where } \texttt{class}\ C\ \texttt{extends}\ D'\ \texttt{\{}\langle \tau_i\ f_i\rangle_i\texttt{;...\}} \in P
\end{aligned}
$$

$\mathrm{mname}(M^n)$

$$
\mathrm{mname}(S\ \tau\ m\texttt{(...)\{...\}}) = m
$$

$\mathrm{method}(m, \tau)$ or $\mathrm{method}(m, T)$

$$
\begin{aligned}
&\mathrm{method}(m, \texttt{sub}\ D\ \texttt{\{}\langle M_i^0\rangle_i\texttt{\}}) \\
&\quad = \begin{cases} M_i^0 & \text{if } \mathrm{mname}(M_i^0) = m \\ \mathrm{method}(m, D) & \text{otherwise} \end{cases} \\
&\mathrm{method}(m, C) \\
&\quad = \begin{cases} M_j^0 & \text{if } \mathrm{mname}(M_j^0) = m \\ \mathrm{method}(m, D') & \text{otherwise} \end{cases} \\
&\text{assuming that } \texttt{class}\ C\ \texttt{extends}\ D'\ \texttt{\{...;}\langle M_j^0\rangle_j\texttt{\}} \in P.
\end{aligned}
$$

$\mathrm{mbody}(M^0)$ or $\mathrm{mbody}(m, \tau)$ or $\mathrm{mbody}(m, T)$

$$
\begin{aligned}
&\mathrm{mbody}(S\ \tau\ m\texttt{(}\langle \tau_i\ x_i\rangle_i\texttt{)\{}e^0\texttt{\}}) = (\langle x_i\rangle_i, e^0) \\
&\mathrm{mbody}(m, \tau) = \mathrm{mbody}(\mathrm{method}(m, \tau)) \\
&\mathrm{mbody}(m, T) = \mathrm{mbody}(\mathrm{method}(m, T)) \\
&\text{Variables returned by } \mathrm{mbody} \text{ are always fresh.}
\end{aligned}
$$

Figure 4. Preliminary definitions for operational semantics.

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. As local variables are immutable, we can model method invocation and `let`-execution by substitution. The local environment L found in LJ [25] is therefore unnecessary, and the small-step judgment is made between heap-term pairs rather than environment-heap-term triples. This is not the same as using a substitution-based semantics, because only heap locations are substituted for variables, i.e., variables are instantiated by their location on the heap. This is equivalent to an environment-based semantics, as discussed in the begin-

$\boxed{H, e^n \underset{\text{prim}}{\overset{n}{\rightsquigarrow}} H, e^n \text{ and } H, \widehat{e}^n \overset{n}{\rightsquigarrow} H, \widehat{e}^n}$

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

$$\frac{H(l) = (T, \langle l_i\rangle_i) \qquad \mathrm{fields}(T) = \langle f_i\rangle_i}{H, l.f_{i_0} \underset{\text{prim}}{\overset{0}{\rightsquigarrow}} H, l_{i_0}}$$

$$\frac{H(l) = (T, \langle l_i\rangle_i)}{H, (l.f_{i_0} := l') \underset{\text{prim}}{\overset{0}{\rightsquigarrow}} H[l \mapsto (T, \langle l_i\rangle_i[i_0 \to l'])], l'}$$

$$\frac{H(l) = (T, \ldots) \qquad \mathrm{mbody}(m, T) = (\langle x_i\rangle_i, e^0)}{H, l.m\texttt{(}\langle l_i\rangle_i\texttt{)} \underset{\text{prim}}{\overset{0}{\rightsquigarrow}} H, [\langle l_i\rangle_i/\langle x_i\rangle_i][l/\texttt{this}]e^0}$$

$$\frac{l \notin \mathrm{dom}\, H}{H, \texttt{let}\ x \Leftarrow \texttt{new}\ C\texttt{(}\langle l_i\rangle_i\texttt{)}\ \texttt{in}\ e^0 \underset{\text{prim}}{\overset{0}{\rightsquigarrow}} H[l \mapsto (C, \langle l_i\rangle_i)], [l/x]e^0}$$

$$\frac{H(l) = (\texttt{Code}, \langle\!|e^0|\!\rangle)}{H, \text{`}l \underset{\text{prim}}{\overset{1}{\rightsquigarrow}} H, e^0} \qquad \frac{H(l) = (\texttt{Code}, \langle\!|e^0|\!\rangle)}{H, l.\texttt{run()} \underset{\text{prim}}{\overset{0}{\rightsquigarrow}} H, e^0}$$

$$\frac{l \notin \mathrm{dom}\, H}{H, \langle\!|e^0|\!\rangle \underset{\text{prim}}{\overset{0}{\rightsquigarrow}} H[l \mapsto (\texttt{Code}, \langle\!|e^0|\!\rangle)], l}$$

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

Figure 5. Small-step semantics for Lightweight Mint.

ning of this section, but makes the formalism simpler because of the removal of the local binding L found in LJ.

Each staging construct induces a primitive reduction rule. Escape and run extract expressions from brackets. Escape reduces only at level 1 and run only reduces at level 0. These are standard in multi-stage languages [28], except that the code values are on the heap. Brackets allocate 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.

6.3 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 $\mathrm{iscf}_{\langle F_i\rangle}(\tau)$ means that τ is code-free assuming that the final classes $\langle F_i\rangle$ are. Thus $\mathrm{iscf}_{\langle\rangle}(\tau)$ means that τ is code-free, which is also simply written as $\mathrm{iscf}(\tau)$. Note, however, that $\mathrm{iscf}()$ does not require method types to be code-free because the formal language does not model class declarations within anonymous inner classes.The auxiliary functions $\mathrm{ftypes}()$, $\mathrm{ftype}_i()$, and $\mathrm{mtype}()$ are similar to $\mathrm{fields}()$ and $\mathrm{method}()$, but they extract type information.

Figure 7 shows the type system. The top-level judgment $\vdash p$ asserts that program p is a valid "initial state" of execution: the class hierarchy P contained in p must be well-formed and the expression e contained in p must be well-typed and free of heap

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

variable typing	$\Gamma \ : \ x \overset{\text{fin}}{\to} \tau^n$
store typing	$\Sigma \ : \ l \overset{\text{fin}}{\to} \tau$
variable typing pair	$\overline{\Gamma} ::= (\Gamma\vert\Gamma)$
pseudo-types	$\widehat{\tau} ::= \tau \vert \langle\tau_i\rangle_i \overset{S}{\to} \tau$

$\boxed{\mathrm{iscf}_{\langle F_i\rangle}(\tau)}$

$$\frac{F \in \langle F_i\rangle}{\mathrm{iscf}_{\langle F_i\rangle}(F)} \qquad \frac{\forall i.\ \mathrm{iscf}_{\langle F_j\rangle, F}(\mathrm{ftype}_i(F))}{\mathrm{iscf}_{\langle F_j\rangle}(F)}$$

$\boxed{\mathrm{cf}(\Sigma),\ \mathrm{locs}(\widehat{e}),\ \mathrm{ftypes}(\tau),\ \mathrm{ftype}_i(\tau),\ \mathrm{ftype}(f, \tau)}$

$\mathrm{cf}(\Sigma) = \Sigma|_L$ where $L = \{l \in \mathrm{dom}(\Sigma) : \mathrm{iscf}_{\langle\rangle}(\Sigma(l))\}$.
$\mathrm{locs}(\widehat{e}) = \{l : l \text{ is a subterm of } \widehat{e}\}$
$\mathrm{ftypes}(\tau) = \langle\tau_i\rangle_i$ assuming $\mathrm{fields}(\tau) = \langle\tau_i\ f_i\rangle_i$
$\mathrm{ftype}_i(\tau) = \mathrm{ftype}(f_i, \tau) = \tau_i$ assuming $\tau_i\ f_i \in \mathrm{fields}(\tau)$

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

$\mathrm{mtype}(S\ \tau\ m\texttt{(}\langle\tau_i\ x_i\rangle_i\texttt{)\{}e^0\texttt{\}}) = \langle\tau_i\rangle_i \overset{S}{\to} \tau$
$\mathrm{mtype}(m, \tau) = \mathrm{mtype}(\mathrm{method}(m, \tau))$ assuming
$\texttt{class}\ C\ \texttt{extends}\ D'\ \texttt{\{}\ldots\texttt{\}} \in P$

Figure 6. Preliminary definitions for the type system.

locations. A class hierarchy P is well-formed, written $\vdash P$, 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 $\mathrm{fields}()$ and $\mathrm{mtype}()$ are always unambiguous and that the sequence returned by $\mathrm{fields}()$ is finite and has no duplicates. Classes are well-formed if they contain no locations, their methods are well-typed, and any overridden methods have the same types as in the superclass.

The bottom half of Figure 7 defines typing for pseudo-expressions with the judgment $\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n \widehat{e}^n : \widehat{\tau}|S$, which states that the pseudo-expression $\widehat{e}^n$ has type $\widehat{\tau}$ at level n under the stack $\langle\Sigma_i\rangle_i$ of store typings and the pair $\overline{\Gamma}$ of contexts. If $S = \texttt{sep}$, this judgment further states that the pseudo-expression $\widehat{e}^n$ is weakly separable. Variable typing $\overline{\Gamma}$ is partitioned into two parts in order to check weak separability of field assignments. The right part contains the variables that were bound within the current method or enclosing escape, which are precisely the variables whose fields can be assigned to without violating weak separability. We always assume that no variables are repeated in $\overline{\Gamma}$.

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 variable and store typings, respectively. CSP is only allowed (by $k > 0$ or $n > 0$, respectively) if the associated type is code-free. The next rule is typing `let`-expressions by extending the current context with the `let`-bound variable, while the rule following is typing field lookups by typing the object and then looking up the relevant field type. In typing the body of a future-stage `let`, a new frame Σ is 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 that e_1 has some τ_1 and that the type τ_2 of e_2 matches the appropriate field type of τ_1. The first of these rules applies to

$\boxed{\tau \prec \tau'}$

$$\frac{}{C \prec C} \qquad \frac{\tau_1 \prec \tau_2 \qquad \tau_2 \prec \tau_3}{\tau_1 \prec \tau_3} \qquad \frac{}{\tau \prec \texttt{Object}}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{\{}\ldots\texttt{\}} \in P}{C \prec D}$$

$$\frac{}{\texttt{Code}\langle \texttt{sep}, \tau\rangle \prec \texttt{Code}\langle \texttt{insep}, \tau\rangle} \qquad \frac{\tau \prec \tau'}{\texttt{Code}\langle S, \tau\rangle \prec \texttt{Code}\langle S, \tau'\rangle}$$

$\boxed{\vdash p, \vdash P, \vdash CL}$

$$\frac{\vdash P \qquad \langle\rangle; \emptyset|\emptyset \vdash^0 e^0 : \tau|S \qquad \mathrm{locs}(e^0) = \emptyset}{\vdash P, e^0}$$

$$\frac{\text{inheritance is acyclic} \qquad \text{no field names clash} \qquad \langle \vdash CL_i\rangle_i}{\vdash \langle CL_i\rangle_i}$$

$$\frac{\langle \mathrm{locs}(M_i^0) = \emptyset\rangle_i \qquad \langle\langle\rangle; \emptyset|\texttt{this} : C^0 \vdash^0 M_i^0\rangle_i \qquad \langle \mathrm{mtype}(\mathrm{mname}(M_i), D) = \text{undef or } \mathrm{mtype}(M_i^0)\rangle_i}{\vdash \texttt{class}\ C\ \texttt{extends}\ D\ \texttt{\{}\langle \tau_j\ f_j\rangle_j; \langle M_i^0\rangle_i\texttt{\}}}$$

$\boxed{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n \texttt{sub}\ D\ \texttt{\{}\langle M_i^n\rangle\texttt{\}}}$

$$\frac{\langle\langle\Sigma_i\rangle_i; \Gamma_1|\Gamma_2, \texttt{this} : D^n \vdash^n M_j^n : \tau_j|S_j\rangle_j \qquad n > 0 \vee \mathrm{dom}(\cup_i\Sigma_i) \supseteq \mathrm{locs}(\texttt{sub}\ D\ \texttt{\{}\langle M_j^n\rangle_j\texttt{\}})}{\langle\Sigma_i\rangle_i; \Gamma_1|\Gamma_2 \vdash^n \texttt{sub}\ D\ \texttt{\{}\langle M_j^n\rangle_j\texttt{\}}}$$

where $\tau_j = \mathrm{mtype}(\mathrm{mname}(M_j^n), D)$.

$\boxed{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n M^n : \langle\tau_i\rangle_i \xrightarrow{S} \tau|S}$

$$\frac{\langle\Sigma_i\rangle_i, \Sigma; \Gamma_1, \Gamma_2, \langle x_i : \tau_i^n\rangle_i|\emptyset \vdash^n e^n : \tau|S}{\langle\Sigma_i\rangle_i; \Gamma_1|\Gamma_2 \vdash^n S\ \tau\ m\texttt{(}\langle\tau_i\ x_i\rangle_i\texttt{)\{}e^n\texttt{\}} : \langle\tau_i\rangle_i \xrightarrow{S} \tau|S'}$$

$\boxed{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash H}$

$$\frac{\forall l \in \mathrm{dom}(\cup_i\Sigma_i).\ \langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash H(l) : (\cup_i\Sigma_i)(l)}{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash H}$$

$\boxed{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash h : \tau}$

$$\frac{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash h : \tau \qquad \tau \prec \tau'}{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash h : \tau'} \qquad \frac{\langle(\cup_i\Sigma_i)(l_j) \prec \mathrm{ftype}_j(C)\rangle_j}{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash (C, \langle l_j\rangle) : C}$$

$$\frac{\langle\Sigma_i\rangle_i; \overline{\Gamma}^{\geq 1} \vdash^0 \langle\!|e^0|\!\rangle : \texttt{Code}\langle S, \tau\rangle|S'}{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash (\texttt{Code}, \langle\!|e^0|\!\rangle) : \texttt{Code}\langle S, \tau\rangle}$$

$$\frac{\langle\Sigma_i\rangle_i; \overline{\Gamma}^{\geq 1} \vdash^0 \texttt{sub}\ D\ \texttt{\{}\langle M_j^0\rangle_j\texttt{\}} \qquad \langle(\cup_i\Sigma_i)(l_k) \prec \mathrm{ftype}_k(D)\rangle_k}{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash (\texttt{sub}\ D\ \texttt{\{}\langle M_j^0\rangle_j\texttt{\}}, \langle l_k\rangle_k) : D}$$

where $\overline{\Gamma}^{\geq 1}(x) = \tau^n \iff \overline{\Gamma}(x) = \tau^n \wedge n \geq 1$.

$\boxed{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n e^n : \tau|S}$ Additional constraints on $\langle\Sigma_i\rangle_i$ are discussed in the text.

$$\frac{\tau' \prec \tau \qquad \langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n e^n : \tau'|S}{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n e^n : \tau|S} \qquad \frac{\overline{\Gamma}(x) = \tau^n \qquad \mathrm{iscf}(\tau) \vee k = 0}{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^{n+k} x : \tau|S} \qquad \frac{(\cup_i\Sigma_i)(l) = \tau \qquad \mathrm{iscf}(\tau) \vee n = 0}{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n l : \tau|S}$$

$$\frac{\langle\langle\Sigma_i\rangle_i; \Gamma_1|\Gamma_2 \vdash^n e_j^n : \mathrm{ftype}_j(C)|S\rangle_j \qquad \langle\Sigma_i\rangle_i, \Sigma; \Gamma_1, \Gamma_2|x : C^n \vdash^n e^n : \tau|S}{\langle\Sigma_i\rangle_i; \Gamma_1|\Gamma_2 \vdash^n (\texttt{let}\ x \Leftarrow \texttt{new}\ C\texttt{(}\langle e_j^n\rangle_j\texttt{)}\ \texttt{in}\ e^n) : \tau\,|S} \qquad \frac{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n e^n : \tau|S}{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n e^n\texttt{.}f : \mathrm{ftype}(f, \tau)|S}$$

$$\frac{\langle\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n e_j^n : \tau_j|S\rangle_{j=1}^2 \qquad \mathrm{ftype}(f, \tau_1) = \tau_2 \qquad S = \texttt{insep} \vee \mathrm{iscf}(\tau_2)}{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n (e_1\texttt{.}f := e_2) : \tau_2|S} \qquad \frac{\overline{\Gamma}(x) = \tau_1^n \qquad \langle\Sigma_i\rangle_i; \Gamma_1|\Gamma_2 \vdash^n e^n : \tau_2|\texttt{sep} \qquad x \in \mathrm{dom}\,\Gamma_2 \qquad \mathrm{ftype}(f, \tau_1) = \tau_2}{\langle\Sigma_i\rangle_i; \Gamma_1|\Gamma_2 \vdash^n (x\texttt{.}f := e^n) : \tau_2|\texttt{sep}}$$

$$\frac{(\cup_i\Sigma_i)(l) = \tau_1 \qquad \mathrm{iscf}(\tau_2) \vee (n = 0 \wedge l \in \mathrm{dom}\,\Sigma_I) \qquad \mathrm{ftype}(f, \tau_1) = \tau_2 \qquad \langle\Sigma_i\rangle_i^I; \overline{\Gamma} \vdash^n e^n : \tau_2|\texttt{sep}}{\langle\Sigma_i\rangle_i^I; \overline{\Gamma} \vdash^n (l\texttt{.}f := e^n) : \tau_2|\texttt{sep}} \qquad \frac{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n e^n : \tau|S \qquad \langle\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n e_j^n : \tau_j|S\rangle_j \qquad \mathrm{mtype}(m, \tau) = \langle\tau_j\rangle_j \xrightarrow{S} \tau}{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n e^n\texttt{.}m\texttt{(}\langle e_j^n\rangle_j\texttt{)} : \tau|S}$$

$$\frac{\langle\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n e_j^n : \mathrm{ftype}_j(D)|S\rangle_j \qquad \langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n \texttt{sub}\ D\ \texttt{\{}\langle M_k^n\rangle_k\texttt{\}}}{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n \texttt{new}\ D\texttt{(}\langle e_j^n\rangle_j\texttt{)}\ \texttt{\{}\langle M_k^n\rangle_k\texttt{\}} : D|S} \qquad \frac{\langle\Sigma_i\rangle_i; \Gamma_1, \Gamma_2|\emptyset \vdash^{n+1} e : \tau|S}{\langle\Sigma_i\rangle_i; \Gamma_1|\Gamma_2 \vdash^n \langle\!|e|\!\rangle : \texttt{Code}\langle S, \tau\rangle|S'} \qquad \frac{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n e^n : \texttt{Code}\langle S, \tau\rangle|\texttt{sep}}{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^{n+1} \texttt{`}e : \tau|S}$$

$$\frac{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n e : \texttt{Code}\langle S, \tau\rangle|S'}{\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n e\texttt{.run()}|\texttt{insep}}$$

Figure 7. Type system for Lightweight Mint.

arbitrary e_1 and types the assignment as weakly separable only if the field has code-free type. The second and third rule handle assignments of non-code-free type in a weakly separable context. The second rule requires e_1 to be a variable x and allows the assignment to be typed as separable only if the variable is in Γ_2 (recalling that Γ_2 gives precisely the variables whose fields can be modified without violating weak separability). The third rule essentially captures how a judgment using the first or the second rule transforms under substitution of a location l for the x in the second rule: it requires that either the location l is in the topmost store typing or the type of the field is code-free.

The rule following is typing method calls by looking up the type of the given method. The rule on the immediate right checks well-formedness of AIC definitions. 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 $\texttt{`}e$ requires typing $e$ at a code type on the previous level and removes the code type; and typing $e\texttt{.run()}$ 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 $\texttt{`}e$ are only weakly separable if e has type $\texttt{Code}\langle\texttt{sep}, \tau\rangle$.

The remainder of Figure 7 defines the following judgments. $\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n \texttt{sub}\ D\ \texttt{\{}\langle M_i^n\rangle\texttt{\}}$ states that an anonymous subclass of 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 $\mathrm{dom}(\cup_i \Sigma_i)$, ensuring that no new frames can be added to the stack of store typings. The judgment $\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n M^n : \langle\tau_i\rangle_i \xrightarrow{S} \tau|S$ states that method M has input types $\langle\tau_i\rangle$, output type τ, and further is weakly separable if $S = \texttt{sep}$. 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 $\overline{\Gamma}$ to the end, indicating that weakly separable expressions in the method cannot freely access variables bound at or before the method M.

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

6.4 Soundness

We now outline the key parts of our type soundness proof. Complete proofs can be found in the appendix. Type soundness is proved by the usual Preservation and Progress lemmas. Progress follows directly from Unique Decomposition, which states that any well-typed expression is either a value or contains a unique redex, which can be contracted by the operational rules. Uniqueness also ensures that our semantics is deterministic.

Lemma 1 (Unique Decomposition)**.** *If $\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n \widehat{e}^n : \tau|S$ and $\widehat{e}^n$ is not a pseudo-value then $\widehat{e}^n$ is uniquely decomposed as $\widehat{e}^n = \mathcal{E}^{n,m}[r^m]$, where $=$ denotes syntactic equality modulo α conversion.*

Proof. By straightforward induction on $\widehat{e}^n$. □

The proof of Preservation is more complicated. One technical difficulty is that the extra Σ added in the rules for binding constructs are unrestricted and therefore may include locations that are not in the current heap. To avoid this problem, we introduce typing for **configurations**—pairs of heaps and pseudo-expressions. The judgment $\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n (H, \widehat{e}^n) : \tau|S$ then specifies that the configuration $(H, \widehat{e}^n)$ is well-typed. Configuration typing rules are identical to pseudo-expression typing rules except that each rule also requires the heap H to be well-formed under the current context. For example, the rule for `let` becomes:

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

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 $\mathrm{dom}(\cup_i \Sigma_i)$ to reference a location in the new frame Σ. Then the resulting heap is ill-formed under $\langle\Sigma_i\rangle_i$, because this portion of the heap is not closed under reachability. The Smashing Lemma solves this problem by smashing the top two Σ's of $\langle\Sigma_i\rangle_i$ into one, giving a shorter store typing stack. Informally, it states that the side-effects of a small-step happening inside a dynamic binder are invisible from outside except for allocations of code-free locations.

Lemma 2 (Smashing)**.** *If*

1. $\Gamma_1' \cup \Gamma_2' \supseteq \Gamma_1 \cup \Gamma_2$
2. $\langle\Sigma_i\rangle_i^I; \Gamma_1|\Gamma_2 \vdash H_1$
3. $\langle\Sigma_i\rangle_i^I, \Sigma; \Gamma_1'|\Gamma_2' \vdash H_2$
4. $H_1|_L = H_2|_L$ *where* $L = \mathrm{dom}(\cup_{i=1}^I \Sigma_i) - \mathrm{dom}(\mathrm{cf}(\cup_{i=1}^I \Sigma_i))$

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

The different store typing stacks capture different views of the heap. The stack $\langle\Sigma_i\rangle_i^I$ captures the locations that can legally be referenced outside the binder (the outer view). The stack $\langle\Sigma_i\rangle_i^I, \Sigma$ captures what can be referenced from inside the binder (the inner view). Similarly, the variable typing pair $\Gamma_1|\Gamma_2$ is used outside the binder while $\Gamma_1'|\Gamma_2'$ is used inside the binder. The inner variable typing pair contains typings for the new variables in addition to those in the outer variable typing pair (condition 1). Code inside the binder may refer to locations that were illegal outside, namely those with code values that (may) contain free variables in $(\Gamma_1' \cup \Gamma_2') \setminus (\Gamma_1 \cup \Gamma_2)$. The extra Σ available in the inner view provides this extension to set of visible locations in the outer view.

Suppose a small-step in the inner context takes the heap from H_1 to H_2. Our concern is whether H_2 is still well-formed, especially in the outer view. The heap had better be well-formed at the beginning in both the inner and outer views; the outer one matters here (condition 2). After the small-step, we can easily show in the proof of Preservation that the new heap H_2 is well-formed in the inner view, by a suitable inductive hypothesis (condition 3). Recall that any new allocations during the small-step are attributed to the topmost store typing Σ, so the prefix $\langle\Sigma_i\rangle_i^I$ remains the same as in the check on H_1. All the code-free restrictions on side effects exist to ensure that the small-step could not have touched any non-code-free locations that are visible in the outer view (condition 4). The

lemma then asserts under these conditions that H_2 is indeed well-formed in the outer view, provided that we salvage any new code-free allocations from Σ into $\langle\Sigma_i\rangle_i^I$. We add these new allocations to the topmost store typing, Σ_I, in compliance with the convention of only modifying the topmost store typing.

The Smashing Lemma implies 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.

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

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

This statement is an abridged version. As mentioned at the beginning of this section, we need freshness assumptions about heap locations in Σ, and this lemma must be expanded to incorporate them. The Appendix contains a complete statement and a proof of this lemma.

7. Implementation

To validate both the expressivity of our type system as well as the potential usefulness of MSP for Java, we extended the Java OpenJDK compiler [17] with our proposed type system and with runtime support for staging. All examples presented earlier in the paper were type-checked using the implementation described in this section.

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.

7.1 OpenJDK Modifications

The compiler included in OpenJDK contains a pretty printer geared towards converting abstract syntax trees (ASTs) to Java source that can be compiled again. By using the pretty printer, we are able to generate source for future stages with minimal changes to the compiler. The fact that we are generating human-readable source also has debugging benefits.

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

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

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

7.2 Staging Translation

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

Mint also extends Java to include a `let` construct to bind values in an expression, as opposed to a statement block. For instance, the expression `let int x=1, y=2*x; 3*y` evaluates to 6, and the scope of `y` begins after the comma. Our `let` construct therefore matches LISP's `let*`. We employ the `let` construct in our implementation to store the freshly generated names when we rename variables in brackets to avoid accidental capture.

For example, in the term

```
public final class FinalClass {
  public int m() { return 5; }
}
//...
final int lifted = 1;
final FinalClass csp = new FinalClass();
Code<Integer> c = <| 123 |>;
Code<Integer> x = <| let int lv = 1;
  2 * lifted + 3 * ‘c + 4 * csp.m() + lv |>;
```

the body of the bracket in the last two lines is translated into a data structure containing:

- the string `"let int "`
- a gensym for `lv`
- the string `"= 1; 2 * "`
- the string `"1"` for the lifted variable
- the string `" + 3 * "`
- the AST of escaped expression `c`
- the string `" + 4 * "`
- the AST of CSP variable `csp`
- the string `".m() + "`
- and a gensym for `lv`.

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

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

```
Code<Integer> x = let
  final String gensym$$1 = varGenSym();
  new MSPTreeCode(new InteriorNode(
    new StrTree("let int "),
    new StrTree(gensym$$1),
    new StrTree("= 1; 2 * "),
    new StrTree("1"),
    new StrTree(" + 3 * "),
    MSPTreeCode.escape(c),
    new StrTree(" + 4 * "),
    new CSPTree(csp),
    new StrTree(".m() + "),
    new StrTree(gensym$$1)
  ));
```

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

7.3 Run

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

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

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

```
public class $$Code1$$
       implements Code<Integer> {
  private FinalClass cf0;

  public $$Code1$$(Object[] t) {
    cf0 = (FinalClass)t[0];
  }

  public Integer run() {
    return (let int var$$$1 = 1;
            2 * 1 +
            3 * (123) +
            4 * cf0.m() +
            var$$$1);
  }
}
```

The fresh symbol `var$$$1` has been substituted for all occurrences of variable `lv`. The contents of the escaped code object `c` and the lifted variable are present without overhead, as desired. The reference to the CSP variable `csp` has been replaced by an access to the CSP field `cf0`.

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

7.4 Class Loading

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

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

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

8. Performance

The timing results presented in this section confirm that MSP can affect the performance of Java programs in a way similar that observed in other languages.

8.1 Benchmarks

In order to measure the performance impact of MSP in Mint, we have benchmarked the following Mint examples:

- `power` is the power example from Section 2, called with base 2 and exponent 17.
- `fib` recursively computes the 17th element of the generalized Fibonacci function starting from 2 and 3.
- `mmult` performs sparse matrix multiplication, in which every 1 in the left matrix omits the floating-point multiplication at runtime and every 0 omits the multiplication and the addition. The benchmark is multiplies an 11-by-11 unsymmetric sparse matrix [6] with itself.
- `eval-fact` calculates the factorial of 10 using the `lint` interpreter discussed in Section 5.1.
- `eval-fib` calculates the 10th number in the standard Fibonacci sequence using the `lint` interpreter.
- `av-mmult` performs the same sparse matrix multiplication as `mmult`, but accesses the matrix using the array views described in Section 5.2.
- `av-mtrans` performs a matrix transpose using array views.
- `serialize` uses the serializer generator discussed in Section 5.4 to write the primitive fields contained in an object hierarchy two levels deep to an output stream.

For each operation in the benchmarking process (unstaged, staged), we first determine the number of repetitions that are required for the operation to run for 1-2 s. This calibration phase also allows the JIT compiler to finish optimizing the program both for the unstaged and the staged code. We then run as many repetitions of the operation as determined in the previous step and record the total time. The average runtime of a single repetition is calculated for each operation and used for the benchmark.

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

Benchmark	speedup	unstaged μs	staged μs
`power`	9.2	0.060	0.0065
`fib`	8.8	0.058	0.0065
`mmult`	4.7	13	2.7
`eval-fact`	20	0.83	0.042
`eval-fib`	24	18	0.73
`av-mmult`	65	20	0.30
`av-mtrans`	14	1.0	0.071
`serialize`	26	1.5	0.057

Figure 8. Benchmark results.

memory, running Mac OS 10.4.11 Tiger and the SoyLatte 1.0.3 JVM [24].

8.2 Results

The results are given in Figure 8. Performance improved in all cases. The speedups, defined as unstaged time divided by staged time, range from 4.7 to 65. The staged versions of `power` and `fib` executed approximately nine times faster than the unstaged code due to the removal of recursion. The `mmult` benchmark involved mostly tight `for` loops and could only be sped up by a factor of 4.7. Staging the `lint` interpreter removed call overhead and improved the performance of the `eval-fact` and `eval-fib` benchmarks by factors of 20 and 24, respectively. In the `av-mmult` and `av-mtrans` benchmarks, loops were unrolled and the layer of indirection in the form of array views was replaced by direct array accesses, resulting in speedups of 65 and 14, respectively. Finally, the `serializer` benchmark also benefited from staging through the removal of reflection, and execution time was reduced by a factor of 26. These improvements make it clear that the presence of JIT technology in Java does not subsume the need for staging techniques, and that the performance benefits reported in previous work [5, 26] on languages without JIT technology apply to Java as well.

9. Related Work

A distinguishing feature of Mint is a strong, expressive, and safe type system that permits both manipulation of open terms and imperative programming. Few multi-level imperative languages have such a type system, and fewer yet come with rigorous type safety proofs. 'C [19] and Jumbo [13] do not guarantee well-formedness of generated code. Cyclone [23] statically guarantees type safety (including well-formedness) of generated code, but does not treat code as first-class values. This design helps Cyclone's runtime code generator to be very fast and still produce high-quality code, but limits programmers in the way they can write generators. Other works on staging extensions to Java by Sestoft [21], Schultz et al. [20], and Zook et al. [30] focus on exploring novel uses of staging and/or quantifying performance benefits. As such, these authors do not attempt to establish safety properties of their extensions.

Some multi-stage systems based on Java offer safety properties, but formalizations and proofs are often absent or incomplete. Metaphore [15] comes with a core typed, Java-like calculus but its type soundness is left unproved. The calculus also leaves out side effects. SafeGen [10] is claimed to guarantee well-typedness of generated code, but the authors do not prove such a result or formalize their system. MorphJ [9] focuses on reflection and does not allow manipulation of arbitrary code values (in particular open terms). The paper proves soundness, but the system does not model side effects. Fähndrich et al. [7] propose a system similar to MorphJ that allows the user to perform limited manipulations of code values, using reflection, in a type-safe manner. DynJava [16] has static type checking for dynamically generated code based on annotations about the types of free variables in code fragments. The authors claim type safety, but do not appear to offer a rigorous proof. The type annotations also make their code generation system unhygienic (i.e. α equivalence fails for dynamic code).

Much of the work on type safety proofs in the literature are for functional languages, where imperative extensions similarly cause scope extrusion [2, 3, 11, 12, 14]. Mint either compares favorably or is competitive with all of these systems. Calcagno et al. [3] allow imperative operations on code but do not support imperative operations on open terms. Aktemur [1] and Kim et al. [14] support unrestricted imperative operations on open terms but give up α-equivalence for future-stage code. Kim et al. delegate hygiene to a specialized binder λ^*, whose semantics can be explained only in terms of a "gensym." Type terms also tend to be very large in their systems which may limit the programmer's ability to write down or interpret types and correct type errors; this problem is offset to some extent, however, by type inference. Aktemur's and Kim et al.'s systems allow storing open terms in global variables that can (potentially) outlive the scope of the variables they contain, which cannot be done in Mint. However, we are not aware of any use for this technique, apart from examples by Kim et al. that deliberately violate hygiene [14]. Unhygienic generation brings up the problem of inadvertent variable capture, and preventing it is a desirable feature of MSP.

Ancona and Moggi [2] and Kameyama et al. [11, 12] are the only works we know of that combine imperative operations on open terms and α-equivalence. Kameyama et al.'s approach [12] is closest to the weak separability approach presented in this paper, in that they limit the effects allowed within escapes. What distinguishes Mint from Ancona and Moggi's and Kameyama et al.'s systems is that Mint allows effects occurring in escapes to be visible outside the escapes as long as they do not involve code objects, whereas Ancona and Moggi and Kameyama et al. unconditionally prohibit such effects. This difference makes our system more expressive, which is demonstrated in Section 5.1, where we throw an exception in a code generator, and in Section 5.2, where we accumulate code in a for-loop. Neither Ancona and Moggi's nor Kameyama et al.'s system can directly express either of these examples. Even if we consider extending these systems, it is unclear how to extend them to handle Section 5.1's example. To allow the example in Section 5.2, their calculi must incorporate arrays, which is a nontrivial theoretical exercise. Our type system is thus better suited to Java programming, which in general makes heavy use of effects.

Furthermore, Kameyama et al. take delimited control operators as the primitives for effects, which are not found in mainstream languages like Java. In order to track the use of delimited control operators, they must use an effect type system, which complicates types and typing rules. It is true that delimited control allows Kameyama et al. to express computations that are not easy to transcribe to assignment-based systems like Mint; however, the advantage of their system is in the underlying imperative primitive and is not in an essential limitation of the weak separability idea itself.

Ancona and Moggi's system executes effects within dynamic binders not at code generation time but at the time the generated code is run. In other words if dynamic binders are involved, Ancona and Moggi's system cannot express effectful code generators although it can express generators that generate effectful code. Their system is similar to Aktemur's [1] and Kim et al.'s [14] in that they use fresh names. All such calculi currently require explicitly listing the free variables in the code type, and explicitly managing hygiene and free variables explicitly in terms. Most importantly, the extra constraints in the type require polymorphism and structural subtyping to be introduced in the language. Mainstream object-oriented languages such as Java and C# support nominal subtyping, not structural subtyping.

10. 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, Julia Lawall and Samuel Kamin for their helpful comments. We thank the anonymous reviewers for their valuable feedback.

References

[1] Baris Aktemur. 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] Tim Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. `http://www.cise.ufl.edu/research/sparse/matrices/Pajek/Tina_DisCal.html`.

[7] 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.

[8] Habanero Multicore Software Research Project. `http://habanero.rice.edu`.

[9] 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.

[10] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically safe program generation with safegen. In *GPCE '04: Proceedings of the 3rd International Conference on Generative Programming and Component Engineering*, pages 309–326, 2005.

[11] Yukiyoshi 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.

[12] Yukiyoshi 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.

[13] 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.

[14] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 257–268, New York, NY, USA, 2006. ACM.

[15] 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.

[16] Yutaka Oiwa, Hidehiko Masuhara, and Akinori Yonezawa. DynJava: Type safe dynamic code generation in Java. In *PPL '01: Proceedings of the 3rd JSSST Workshop on Programming and Programming Languages*, March 2001.

[17] OpenJDK Project. `http://openjdk.java.net`.

[18] Rice PLT. Mint Multi-stage Java Compiler. Available at `http://www.javamint.org`.

[19] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. C and tcc: a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, 1999.

[20] 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.

[21] Peter Sestoft. Runtime code generation with JVM and CLR. Available at `http://www.dina.dk/sestoft/publications.html`, 2002.

[22] Jun Shirako, Hironori Kasahara, and Vivek Sarkar. Language extensions in support of compiler parallelization. In *Languages and Compilers for Parallel Computing: 20th International Workshop, LCPC 2007, Urbana, IL, USA, October 11-13, 2007, Revised Selected Papers*, pages 78–94, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] Frederick Smith, Dan Grossman, J. Gregory Morrisett, Luke Hornof, and Trevor Jim. Compiling for template-based run-time code generation. *Journal of Functional Programming*, 13(3):677–708, 2003.

[24] SoyLatte Project. `http://landonf.bikemonkey.org/static/soylatte/`.

[25] 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.

[26] Walid Taha. A gentle introduction to multi-stage programming. In *DSPG '03: Proceedings of the International Seminar on Domain-Specific Program Generation*, 2003.

[27] Walid Taha. A gentle introduction to multi-stage programming, part ii. In *Generative and Transformational Techniques in Software Engineering II*, 2007.

[28] Walid Taha, Zine el-abidine Benaissa, 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.

[29] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT sym-*

posium on Principles of programming languages, pages 26–37, New York, NY, USA, 2003. ACM.

[30] 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.

A. Proofs

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

Notation. The components of a variable typing pair $\overline{\Gamma}$ are always named Γ_1 and Γ_2, so that $\overline{\Gamma} = \Gamma_1|\Gamma_2$, unless otherwise specified. Similarly, $\overline{\Gamma}' = \Gamma_1'|\Gamma_2'$.

We impose some well-formedness convention on the store typing stacks. The convention in Definition 3 is not essential to proving type safety but will significantly simplify the presentation as well as the proof. Definitions 4 and 5 by contrast are essential to the proof and are followed by a more in-depth discussion. Some of the lemmas' statements in the main text needs to be updated in order to exploit these convention—the updated statements are provided as necessary.

Definition 3. A store typing stack is well-formed iff all of its code-free bindings are gathered in the first Σ, and the individual store typings have pairwise disjoint domains:

$$\frac{\begin{array}{cc} I \geq 2 & \langle \mathrm{cf}(\Sigma_i) = \emptyset \rangle_{i=2}^{I} \\ \mathrm{cf}(\Sigma_1) = \Sigma_1 & \langle \mathrm{dom}(\Sigma_i) \cap \mathrm{dom}(\Sigma_j) = \emptyset \rangle_{i \neq j} \end{array}}{\vdash \langle \Sigma_i \rangle_i^I}.$$

NB: the requirement $I \geq 2$ is just to ensure that the stack has a nonempty code-free part and a nonempty non-code-free part.

Definition 4. The proof tree PT of a configuration typing judgment satisfies the *disjointness criterion* if any two store typing stacks $\langle \Sigma_i \rangle_i^I$ and $\langle \Sigma_j' \rangle_j^J$ that appear in PT have a common prefix of length at least 2 (i.e. $\exists K \geq 2.\ \langle \Sigma_k = \Sigma_k' \rangle_{k=1}^K$) and the rest have disjoint domains ($\langle \mathrm{dom}(\Sigma_i) \cap \mathrm{dom}(\Sigma_j) \rangle_{i,j>K}$). PT is well-formed iff it only uses well-formed store typing stacks and satisfies the disjointness criterion.

Definition 5. A location l is said to *appear in* PT iff PT uses a store typing stack that contains l in the domain of the stack's union. An l is said to be *fresh for* PT iff it does not appear in PT. An l is *local to* PT iff for any PT', if PT and PT' are disjoint subtrees of a well-formed tree, then l is fresh for PT'. An l is fresh or local in a configuration typing judgment iff it is fresh or local, respectively, for some proof tree of the typing judgment.

Note that l is local to PT iff it appears in a Σ that is introduced in PT as a result of a store typing stack extension within PT. Note also that this well-formedness concern for derivation trees is only for configurations, and expression typing derivations are always well-formed.

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

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

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

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

Proof. The expression part contains no locations, so the store typing stack is only used to type the heap which is empty and is therefore well-formed under, and only under, store typing stacks of the form $\langle \emptyset, \emptyset, \ldots, \emptyset \rangle$. Thus, every configuration typing judgment in the derivation of $(*)$ is of the form $\langle \emptyset \rangle_i^I; \overline{\Gamma} \vdash^n (\emptyset, \widehat{e}^n) : \widehat{\tau}|S$ and

every heap well-formedness judgment is of the form $\langle\emptyset\rangle_i^I; \overline{\Gamma} \vdash \emptyset$. If we consistently replace I by $I+2$ in all such judgments, then we have a derivation tree for $\langle\emptyset,\emptyset\rangle; \emptyset|\emptyset \vdash^0 (\emptyset, e^0) : \tau|S$. The typing derivation constructed here is clearly well-formed. □

Note that Proposition 4 starts by assuming typability under $\langle\rangle$ because that is what we used for program typing in Figure 7.

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

Proof. By hypothesis,

$$\langle\rangle; \emptyset|\texttt{this} : C^0 \vdash^0 M^0 : \mathrm{mtype}(M^0)|S.$$

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

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

Hereafter, the assumption that all store typing stacks and typing derivations are well-formed is in effect. The assumption is needed for typings of configurations under updated environments to propagate through congruence rules. This statement is formalized below as Lemma 6. It states that when a small step $H_1, e_1 \overset{n}{\rightsquigarrow} H_2, e_2$ has been taken on a subterm e_1 of some bigger term, the heap attached to the typing of any other disjoint subterm can be safely replaced by H_2. We use the well-formedness assumption to establish the precondition (iv) when we invoke this Lemma 6 in the Preservation Lemma.

We will also update the statements of the Smashing and Preservation Lemmas to observe the well-formedness assumption. The effect on the Smashing Lemma is mainly simplification rather than a change in its meaning, while the modification to the Preservation Lemma is mostly concerned with propagating the well-formedness conditions.

Lemma 6. *Suppose*

(i) $\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n (H_1, \widehat{e}^n) : \widehat{\tau}|S$
(ii) $\langle\Sigma'_j\rangle_j; \overline{\Gamma} \vdash H_2$
(iii) $\cup_j \Sigma'_j \supseteq \cup_i \Sigma_i$
(iv) $\forall l$ *s.t.* $H_1(l) \neq H_2(l)$, *either* l *is fresh for (i) or* $l \in \mathrm{dom}(\cup_j \Sigma'_j)$.

Then $\langle\Sigma'_j\rangle_j; \overline{\Gamma} \vdash^n (H_2, \widehat{e}^n) : \widehat{\tau}|S$.

Proof. Induction on $\widehat{e}^n$. Base cases are straightforward. For inductive cases, if inversion yields judgments of the form

$$\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n (H_1, \widehat{e}^n_s) : \widehat{\tau}_s|S$$

for each immediate subterm $\widehat{e}^n_s$ of $\widehat{e}^n$ and suitable $\widehat{\tau}_s$, then IH gives

$$\langle\Sigma'_j\rangle_j; \overline{\Gamma} \vdash^n (H_2, \widehat{e}^n_s) : \widehat{\tau}_s|S.$$

If no other form of judgment results from inversion, then we immediately obtain the conclusion.

The only case this argument fails is if a subterm is typed under an extended environment:

$$\langle\Sigma_i\rangle_i, \Sigma; \overline{\Gamma}' \vdash^n (H_1, \widehat{e}^n_s) : \widehat{\tau}_s|S$$

where $\Gamma'_1 \cup \Gamma'_2 \supseteq \Gamma_1 \cup \Gamma_2$. Then we must check that $\langle\Sigma'_j\rangle, \Sigma; \overline{\Gamma}' \vdash H_2$ before invoking the IH. Once we invoke IH, the conclusion is again immediate.

Take any $l \in \mathrm{dom}\,\Sigma$. Assuming (i) is well-formed, $l \notin \mathrm{dom}(\cup_i \Sigma_i)$. Σ is taken from the derivation of (i), so every location in $\mathrm{dom}\,\Sigma$ is *not* fresh for (i). Thus by (iv), $H_1(l) = H_2(l)$. Thus

$$\begin{aligned}
\text{(i)} &\Longrightarrow \langle\Sigma_i\rangle_i, \Sigma; \overline{\Gamma}' \vdash H_1(l) : \Sigma(l) && \text{by inversion}\\
&\Longrightarrow \langle\Sigma'_j\rangle_j, \Sigma; \overline{\Gamma}' \vdash H_2(l) : \Sigma(l). && \text{by } \Sigma_h \text{ weakening and } H_1(l) = H_2(l)
\end{aligned}$$

Now take any $l \in \mathrm{dom}(\cup_j \Sigma'_j)$. We have

$$\langle\Sigma'_j\rangle_j; \overline{\Gamma} \vdash H_2(l) : (\cup_j \Sigma'_j)(l)$$

by (ii). Then by Σ_h and Γ_h weakenings, we get $\langle\Sigma'_j\rangle_j, \Sigma; \overline{\Gamma}' \vdash H_2(l) : (\cup_j \Sigma'_j)(l)$. Therefore, $\langle\Sigma'_j\rangle_j, \Sigma; \overline{\Gamma}' \vdash H_2$.

If $\widehat{e}^n = \langle\!|e^{n+1}|\!\rangle$, the variable typing's partitioning bar (|) is moved but the store typing stack is not extended. In this case, the argument is essentially the same but we use just Γ_H weakening. □

Lemma 7 (Σ_e relevance)**.** *If we have $\langle\Sigma_i\rangle_i^I; \overline{\Gamma} \vdash^n \widehat{e}^n : \widehat{\tau}|S$ and $(\cup_i \Sigma_i)|_{\mathrm{locs}(\widehat{e}^n)} = (\cup_j \Sigma'_j)|_{\mathrm{locs}(\widehat{e}^n)}$, then $\langle\Sigma'_j\rangle_j^J; \overline{\Gamma} \vdash^n \widehat{e}^n : \widehat{\tau}|S$.*

Proof. Proof is by induction on $\widehat{e}^n$. When looking up a location the Σ's are always unioned together, so it clearly only matters what the union of the stack contains.

The only non-trivial inductive cases are the ones that extend the store typing stack. Take $\widehat{e}^n = S'\ \tau\ m(\langle\tau_k\ x_k\rangle_k)\{e^n\}$ for example, and let $L = \mathrm{locs}(e^n)$. By inversion we have

$$\exists\Sigma.\ \langle\Sigma_i\rangle_i^I, \Sigma; \Gamma_1, \Gamma_2, \langle x_k : \tau_k^n\rangle_k|\emptyset \vdash^n e^n : \tau|S'.$$

Then

$$\begin{aligned}
((\cup_i \Sigma_i) \cup \Sigma)|_L &= (\cup_i \Sigma_i)|_L \cup \Sigma|_L\\
&= (\cup_j \Sigma'_j)|_L \cup \Sigma|_L\\
&= ((\cup_j \Sigma'_j) \cup \Sigma)|_L
\end{aligned}$$

so we can use IH on e^n to obtain

$$\langle\Sigma'_j\rangle_j^J, \Sigma; \Gamma_1, \Gamma_2, \langle x_k : \tau_k^n\rangle_k|\emptyset \vdash^n e^n : \tau|S'.$$

The conclusion immediately follows. □

Lemma 8 (Γ_e weakening)**.** *If $\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash^n \widehat{e}^n : \widehat{\tau}|S$ and $\Gamma'_i \supseteq \Gamma_i$ $(i = 1, 2)$ then $\langle\Sigma_i\rangle_i; \overline{\Gamma}' \vdash^n \widehat{e}^n : \widehat{\tau}|S$.*

Proof. Straightforward induction on $\widehat{e}^n$, noting that part-wise containment $\Gamma'_i \supseteq \Gamma_i$ is preserved by manipulations of the form $\Gamma_1|\Gamma_2 \mapsto \Gamma_1, \Gamma_2, \Gamma''|\Gamma'''$ for Γ'' and Γ''' that are independent of Γ_1 and Γ_2. □

Lemma 9 (Γ_h weakening)**.** *If $(\Gamma'_1 \cup \Gamma'_2)^{\geq 1} \supseteq (\Gamma_1 \cup \Gamma_2)^{\geq 1}$ and $\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash h : \tau$ then $\langle\Sigma_i\rangle_i; \overline{\Gamma}' \vdash h : \tau$.*

Proof. The variable typing pair is used when $h = (\texttt{Code}, \langle\!|e^0|\!\rangle)$, where

$$\langle\Sigma_i\rangle_i; \overline{\Gamma}^{\geq 1} \vdash^0 \langle\!|e^0|\!\rangle : \texttt{Code}\langle S, \tau'\rangle|S'$$

and $\tau = \texttt{Code}\langle S, \tau'\rangle$, or $h = (\texttt{sub}\ D\ \{\langle M_j^0\rangle_j\})$, in which case

$$\langle\langle\Sigma_i\rangle_i; \overline{\Gamma}^{\geq 1} \vdash^0 M_j^0 : \mathrm{mtype}(\mathrm{mname}(M_j^0), D)\rangle_j.$$

In both cases, we can replace $\overline{\Gamma}$ with $\overline{\Gamma}'$ by Γ_e weakening. Notice that the partitioning bar is moved all the way to the right before the variable typing has a chance to be looked up, so that the assumption $(\Gamma'_1 \cup \Gamma'_2)^{\geq 1} \supseteq (\Gamma_1 \cup \Gamma_2)^{\geq 1}$ is turned into pair-wise containment, matching the hypothesis of Γ_e weakening. □

Lemma 10 (Γ_H weakening)**.** *If $(\Gamma'_1 \cup \Gamma'_2)^{\geq 1} \supseteq (\Gamma_1 \cup \Gamma_2)^{\geq 1}$ and $\langle\Sigma_i\rangle_i; \overline{\Gamma} \vdash H$ then $\langle\Sigma_i\rangle_i; \overline{\Gamma}' \vdash H$.*

Proof. Immediate consequence of Γ_h weakening. □

Lemma 11. *If* $\langle\Sigma_i\rangle_i;\overline{\Gamma} \vdash^{n+1} \widehat{e}^n : \widehat{\tau}|S$ *then* $\mathrm{locs}(\widehat{e}^n) \subseteq \mathrm{dom}\,\Sigma_1$.

Proof. Induction on $\widehat{e}^n$. It is evident from the typing rules that we maintain the invariant that the term can always be seen as lower-level than the level of the typing judgment, and therefore that the level of the typing judgment is > 0. Hence when we encounter a rule that looks up the store typing (i.e. $\widehat{e}^n = l$ or $(l.f := e^n)$), it is the case that $\mathrm{iscf}((\cup_i\Sigma_i)(l))$. By the assumption that the derivation is well-formed, the store typing stack is well-formed, hence $l \in \Sigma_1$. □

Lemma 12 (Σ_h weakening)**.** *If* $\langle\Sigma_i\rangle_i;\overline{\Gamma} \vdash h : \tau$ *and* $\cup_j\Sigma'_j \supseteq \cup_i\Sigma_i$ *then* $\langle\Sigma_j\rangle_j;\overline{\Gamma} \vdash h : \tau$.

Proof. If $h = (T, \langle l_k\rangle_k)$ then $\cup_j\Sigma'_j$ assigns the right types to the l_k since they are all in $\mathrm{dom}(\cup_i\Sigma_i)$, where $\cup_i\Sigma_i$ and $\cup_j\Sigma'_j$ coincide. If $T = \texttt{sub}\ D\ \texttt{\{}\langle M^0_a\rangle_a\texttt{\}}$ then

$$\langle\langle\Sigma_i\rangle_i;\Gamma_1|\Gamma_2, \texttt{this} : D^0 \vdash^0 M^0_a : \widehat{\tau}_a|S_a\rangle_a$$

where $\widehat{\tau}_a = \mathrm{mtype}(\mathrm{mname}(M^0_a), D)$, and

$$\mathrm{locs}(\texttt{sub}\ D\ \texttt{\{}\langle M^0_a\rangle_a\texttt{\}}) \subseteq \mathrm{dom}(\cup_i\Sigma_i) \subseteq \mathrm{dom}(\cup_j\Sigma'_j)$$

so $\cup_i\Sigma_i|_{\mathrm{locs}(T)} = \cup_j\Sigma'_j|_{\mathrm{locs}(T)}$. It follows by Σ_e relevance that

$$\langle\langle\Sigma'_j\rangle_j;\Gamma_1|\Gamma_2, \texttt{this} : D^0 \vdash^0 M^0_a : \widehat{\tau}_a|S_a\rangle_a$$

thus $\langle\Sigma'_j\rangle_j;\overline{\Gamma} \vdash \texttt{sub}\ D\ \texttt{\{}\langle M^0_a\rangle_a\texttt{\}}$. Therefore h is typable under $\langle\Sigma'_j\rangle_j$.

If $h = (\texttt{Code}, \langle\!|e^0|\!\rangle)$, then $\langle\Sigma_i\rangle_i;\overline{\Gamma}^{\geq 1} \vdash^0 \langle\!|e^0|\!\rangle : \tau'|S$ where $\tau = \texttt{Code}\langle S, \tau'\rangle$. By Lemma 11, $\mathrm{locs}(\langle\!|e^0|\!\rangle) \subseteq \mathrm{dom}(\cup_i\Sigma_i) \subseteq \mathrm{dom}(\cup_j\Sigma'_j)$, so $\langle\Sigma'_j\rangle_j;\overline{\Gamma}^{\geq 1} \vdash^0 \langle\!|e^0|\!\rangle : \tau'|S$ by Σ_e relevance. It follows that h is typable under $\langle\Sigma'_j\rangle_j$. □

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

Lemma 13 (Demotion)**.** *If* $\langle\Sigma_i\rangle_i;\emptyset|\emptyset \vdash (\texttt{Code}, \langle\!|e^0|\!\rangle) : \texttt{Code}\langle S, \tau\rangle$ *and* $\langle\Sigma_i\rangle_i;\emptyset|\emptyset \vdash H$ *then* $\langle\Sigma_i\rangle_i;\emptyset|\emptyset \vdash^0 (H, e^0) : \tau|S$.

Proof. Generalize to:

$$\langle\Sigma_i\rangle_i;\overline{\Gamma}^{\geq 1} \vdash^{n+1} \widehat{e}^n : \widehat{\tau}|S \wedge \langle\Sigma_i\rangle_i;\overline{\Gamma}\downarrow \vdash H$$
$$\implies \langle\Sigma_i\rangle_i;\overline{\Gamma}\downarrow \vdash^n (H, \widehat{e}^n) : \widehat{\tau}|S$$

where $\overline{\Gamma}\downarrow = \Gamma_1\downarrow\ |\Gamma_2\downarrow$ and $\Gamma\downarrow(x) = \tau^n \overset{\text{def}}{\iff} \Gamma(x) = \tau^{n+1}$. We prove this by induction on $\widehat{e}^n$.

If $\widehat{e}^n = x$ then $\overline{\Gamma}^{\geq 1}(x) = \widehat{\tau}^k$ and $0 < k \leq n$ so $\overline{\Gamma}\downarrow(x) = \widehat{\tau}^{k-1}$. Therefore $\langle\Sigma_i\rangle_i;\overline{\Gamma}\downarrow \vdash^n (H, x) : \widehat{\tau}|S$.

If $\widehat{e}^n = (\texttt{let}\ x \Leftarrow \texttt{new}\ C\texttt{(}\langle e^n_j\rangle_j\texttt{)}\ \texttt{in}\ e^n)$ then by IH, $\langle\langle\Sigma_i\rangle_i;\overline{\Gamma}\downarrow \vdash^n e^n_j : \tau_j|S_j\rangle_j$. By inversion we have

$$\exists\Sigma.\ \langle\Sigma_i\rangle_i, \Sigma; \Gamma^{\geq 1}_1, \Gamma^{\geq 1}_2|x : C^{n+1} \vdash^{n+1} e^n : \widehat{\tau}|S.$$

By Lemma 11 and Σ_e relevance, we may assume $\Sigma = \emptyset$, so by Γ_H weakening, $\langle\Sigma_i\rangle_i, \Sigma; (\Gamma^{\geq 1}_1, \Gamma^{\geq 1}_2|x : C^{n+1})\downarrow \vdash H$. Hence we can use IH to get $\langle\Sigma_i\rangle_i, \Sigma; (\Gamma_1, \Gamma_2)\downarrow|x : C^n \vdash^n (H, e^n) : \widehat{\tau}|S$. The desired conclusion follows immediately.

If $\widehat{e}^n = M^n$, the argument is essentially the same as for a `let`.

If $\widehat{e}^n = \texttt{new}\ D\texttt{(}\langle e^n_j\rangle_j\texttt{)\{}\langle M^0_k\rangle_k\texttt{\}}$, IH gives

$$\langle\langle\Sigma_i\rangle_i;\overline{\Gamma}\downarrow \vdash^n (H, M^0_k) : \mathrm{mtype}(\mathrm{mname}(M^0_k), D)|S_k\rangle_k.$$

For $n \neq 0$, this is enough to get the conclusion. If $n = 0$, then we need $\mathrm{dom}(\cup_i\Sigma_i) \supseteq \mathrm{locs}(\texttt{sub}\ D\ \texttt{\{}\langle M^0_k\rangle_k\texttt{\}})$ in addition; this is ensured by Lemma 11.

The remaining cases are straightforward. □

The statement of the Substitution Lemma is more or less standard.

Lemma 14 (Substitution Lemma)**.** *Let* $\overline{\Gamma}$ *and* $\overline{\Gamma}'$ *be identical, including the position of the partitioning bar* (|)*, except that* $\overline{\Gamma}(x) = \tau^0$ *and* $x \notin \mathrm{dom}\,\overline{\Gamma}'$. *If* $\langle\Sigma_i\rangle^I_i;\overline{\Gamma} \vdash^n (H, \widehat{e}^n) : \widehat{\tau}|S$ *and* $(\cup_i\Sigma_i)(l) = \tau$ *and* $x \notin \mathrm{dom}\,\Gamma_2 \vee l \in \mathrm{dom}(\Sigma_1 \cup \Sigma_I)$*, then* $\langle\Sigma_i\rangle^I_i;\overline{\Gamma}' \vdash^n (H, [l/x]\widehat{e}^n) : \widehat{\tau}|S$.

Proof. Induction on $\widehat{e}^n$. Note that in each case $\langle\Sigma_i\rangle_i;\overline{\Gamma}' \vdash H$ is assured by Γ_H weakening.

If $\widehat{e}^n = x$, inversion on the configuration typing gives $\mathrm{iscf}(\tau) \vee n = 0$, which is just the premise we need to justify $\langle\Sigma_i\rangle_i;\overline{\Gamma}' \vdash^n (H, l) : \widehat{\tau}|S$.

If $\widehat{e}^n = (x.f := e^n)$, then $\widehat{\tau} = \mathrm{ftype}(f, \tau)$ where $\tau = (\cup_i\Sigma_i)(l)$. The typing judgment must have been derived with one of two rules. If it used the rule for the generic form $(e^n_1.f := e^n_2)$, then $S = \texttt{insep}$ so by IH, the configuration after substitution can be typed by the same rule. Otherwise, $S = \texttt{sep}$, in which case the binding level 0 of x must equal the typing level n. IH gives

$$\langle\Sigma_i\rangle_i;\overline{\Gamma}' \vdash^n (H, [l/x]e^n) : \widehat{\tau}|\texttt{sep}.$$

Inversion gives $x \in \mathrm{dom}(\Gamma_2) \vee \mathrm{iscf}(\tau)$. If $x \in \mathrm{dom}\,\Gamma_2$ then $l \in \mathrm{dom}(\Sigma_1 \cup \Sigma_I)$ by hypothesis, thus $\mathrm{iscf}(\tau) \vee l \in \mathrm{dom}\,\Sigma_I$. Therefore, we have

$$\mathrm{iscf}(\tau) \vee (n = 0 \wedge l \in \mathrm{dom}\,\Sigma_I),$$

which establishes

$$\langle\Sigma_i\rangle_i;\overline{\Gamma}' \vdash^n (H, (l.f := [l/x]e^n)) : \widehat{\tau}|\texttt{sep}.$$

Suppose $\widehat{e}^n = (\texttt{let}\ y \Leftarrow \texttt{new}\ C\texttt{(}\langle e^n_j\rangle_j\texttt{)}\ \texttt{in}\ e^n)$. If $x = y$, then the substitution is the identity on this term so the conclusion is immediate. Otherwise, we have $\langle\langle\Sigma_i\rangle_i;\overline{\Gamma}' \vdash^n (H, [l/x]e^n_j)\widehat{\tau}|S\rangle_j$ by IH. By inversion,

$$\langle\Sigma_i\rangle_i, \Sigma; \Gamma_1, \Gamma_2|y : C^n \vdash^n (H, e^n) : \widehat{\tau}|S.$$

The store typing stack is extended, so the l is no longer in the rightmost Σ (if it was in Σ_I), but the binding for x is no longer to the right of the partitioning bar either, so we can apply IH. This gives $\langle\Sigma_i\rangle_i;\Gamma'_1, \Gamma'_2|y : C^n \vdash^n (H, [l/x]e^n) : \widehat{\tau}|S$. Thus $\langle\Gamma_i\rangle_i;\overline{\Gamma}' \vdash^n (H, [l/x]e^n) : \widehat{\tau}|S$.

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

If $\widehat{e}^n = \texttt{new}\ D\texttt{(}\langle e_j\rangle_j\texttt{)\{}\langle M^n_k\rangle_k\texttt{\}}$, the well-typedness of (H, e_j) and (H, M^n_k) follow directly from IH. The only concern left is whether $\mathrm{dom}(\cup_i\Sigma_i) \supseteq \mathrm{locs}([l/x](\texttt{sub}\ D\ \texttt{\{}\langle M^n_k\rangle_k\texttt{\}}))$ when $n = 0$. This clearly holds because the only addition to the set of locations is l, and $l \in \mathrm{dom}(\cup_i\Sigma_i)$.

The remaining cases are straightforward. □

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

Lemma 15 (Augmentation Lemma)**.** *If* $\langle\Sigma_i\rangle_i;\overline{\Gamma} \vdash^n \widehat{e}^n : \widehat{\tau}|S$ *and* $\langle\Sigma_i\rangle_i;\overline{\Gamma} \vdash H$ *and* $\mathrm{dom}(\cup_i\Sigma_i) \supseteq \mathrm{locs}(\widehat{e}^n)$ *then* $\langle\Sigma_i\rangle_i;\overline{\Gamma} \vdash^n (H, \widehat{e}^n) : \widehat{\tau}|S$.

Proof. Induction on $\widehat{e}^n$. This is just a matter of checking

$$\langle\Sigma_i\rangle_i \circ \langle\Sigma'_j\rangle_j;\overline{\Gamma}' \vdash H \qquad (*)$$

for every extended environment $\langle\Sigma_i\rangle_i \circ \langle\Sigma'_j\rangle_j;\overline{\Gamma}'$ that appears in the derivation of $\langle\Sigma_i\rangle_i;\overline{\Gamma} \vdash^n \widehat{e}^n : \widehat{\tau}|S$. By Σ_e relevance,

$\langle \Sigma'_j \rangle_j = \langle \emptyset, \emptyset, \ldots, \emptyset \rangle$ without loss of generality. Then $\cup_i \Sigma_i = (\cup_i \Sigma_i) \cup (\cup_j \Sigma'_j)$, so

$$\langle \Sigma_i \rangle_i \circ \langle \Sigma'_j \rangle_j ; \overline{\Gamma} \vdash H(l) : (\cup_i \Sigma_i)(l)$$

for every $l \in \mathrm{dom}(\cup_i \Sigma_i)$ $(= \mathrm{dom}((\cup_i \Sigma_i) \cup (\cup_j \Sigma'_j)))$ by Σ_h weakening. Therefore,

$$\langle \Sigma_i \rangle_i \circ \langle \Sigma'_j \rangle_j ; \overline{\Gamma} \vdash H,$$

and $(*)$ follows by Γ_H weakening. □

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

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

(I) $\langle \Sigma_i \rangle_i^I ; \overline{\Gamma} \vdash H_1$
(II) $\langle \Sigma'_i \rangle_i^{I+1} ; \overline{\Gamma}' \vdash H_2$
(III) $\forall l \in \mathrm{dom}(\cup_{i=1}^{I+1} \Sigma'_i).\ H_1(l) \neq H_2(l) \implies l \in \mathrm{dom}(\Sigma'_1 \cup \Sigma'_{I+1})$
(IV) $\Sigma'_1 \supseteq \Sigma_1 \wedge \Sigma'_{I+1} \supseteq \Sigma_{I+1} \wedge \langle \Sigma'_i = \Sigma_i \rangle_{i=2}^I$

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

$$\langle \Sigma'_i \rangle_i^I ; \overline{\Gamma} \vdash H_2.$$

Proof. For every location $l \in \mathrm{dom}(\cup_{i=1}^I \Sigma_i)$ such that $H_1(l) = H_2(l)$, we have

$$\langle \Sigma_i \rangle_i^I ; \overline{\Gamma} \vdash H_2(l) : (\cup_{i=1}^I \Sigma_i)(l)$$

by (I), and we can replace the store typing stack with $\langle \Sigma'_i \rangle_i^I$ due to Σ_h weakening.

For any other $l \in \mathrm{dom}(\cup_{i=1}^I \Sigma_i)$, (III) tells us that $l \in \mathrm{dom}\, \Sigma'_1$ so $(\cup_{i=1}^I \Sigma'_i)(l) = \Sigma'_1(l)$ and $\mathrm{iscf}(\Sigma'_1(l))$. It follows that $\Sigma'_1(l) = F$ and by (II), $H_2(l) = (F, \langle l_k \rangle_k)$. Moreover, $\forall k.\ \mathrm{iscf}(\mathrm{ftype}_k(F))$ so $\mathrm{iscf}((\cup_i \Sigma_i)(l))$, hence $l_k \in \mathrm{dom}\, \Sigma'_1$. Therefore, $H_2(l)$ is well-formed under any well-formed store typing stack starting with Σ'_1, including $\langle \Sigma'_i \rangle_i^I$.

Thus $\langle \Sigma'_i \rangle_i^I ; \overline{\Gamma} \vdash H_2$. □

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

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

(I) $\langle \Sigma_i \rangle_i^I ; \overline{\Gamma} \vdash^n (H_1, \widehat{e}_1^n) : \widehat{\tau} | S$
(II) $H_1, \widehat{e}_1^n \overset{n}{\rightsquigarrow} H_2, \widehat{e}_2^n$
(III) $S = \mathtt{sep} \vee \overline{\Gamma}^{>n} = \emptyset | \emptyset$
(IV) $\overline{\Gamma} = \overline{\Gamma}^{\geq 1}$

then $\exists \langle \Sigma'_i \rangle_i^I$ such that

(i) $\langle \Sigma'_i \rangle_i^I ; \overline{\Gamma} \vdash^n (H_2, \widehat{e}_2^n) : \widehat{\tau} | S$
(ii) $\Sigma'_1 \supseteq \Sigma_1 \wedge \Sigma'_I \supseteq \Sigma_I \wedge \langle \Sigma'_i = \Sigma_i \rangle_{i=2}^{I-1}$
(iii) $H_1(l) \neq H_2(l) \implies (l \notin \mathrm{dom}\, H_1) \vee (l \in \mathrm{dom}(\Sigma_1 \cup \Sigma_I)) \vee (l$ is local to (I)).

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

Proof. Induction on the evaluation context $\mathcal{E}^{n,k}$. Conclusion (ii) will be obvious for each case, so we will not explicitly write out its justification.

We first handle primitive reductions. There are three forms that extend the heap, all at $n = 0$. The only point of change on the heap for these case is the new location, which is fresh for H_1. Therefore (iii) holds, and only (i) remains to be proved.

- Suppose $\widehat{e}_1^n = \mathtt{new}\ D(\langle l_j \rangle_j)\{\langle M_a^0 \rangle_a\}$. Then $\widehat{e}_2^n = l \notin H_1$ and $\widehat{\tau} = D$. Take $\Sigma'_1 = \Sigma_1$ and $\Sigma'_I = \Sigma_I[l \mapsto D]$. For every $l' \neq l$, we have $H_1(l') = H_2(l')$ so by (I) and Σ_h weakening,

  $$\langle \Sigma'_i \rangle_i^I ; \overline{\Gamma} \vdash H_2(l') : D.$$

  For $H_2(l) = (\mathtt{sub}\ D\ \{\langle M_a^0 \rangle_a\}, \langle l_j \rangle_j)$, by inversion on (I) we know that both the tag and the fields are well-typed under $\langle \Sigma_i \rangle_i ; \overline{\Gamma}$, so that the heap element itself is well-typed. Thus by Σ_h weakening,

  $$\langle \Sigma'_i \rangle_i^I ; \overline{\Gamma} \vdash H_2(l) : D.$$

  This shows that $\langle \Sigma'_i \rangle_i^I ; \overline{\Gamma} \vdash H_2$. By inversion,

  $$\mathrm{locs}(\mathtt{sub}\ D\ \{\langle M_a^0 \rangle_a\}) \subseteq \mathrm{dom}(\cup_i \Sigma_i) \subseteq \mathrm{dom}(\cup_i \Sigma'_i).$$

  All the other premises needed to justify (i) is directly obtained from IH.
- Suppose $\widehat{e}_1^n = (\mathtt{let}\ x \Leftarrow \mathtt{new}\ C(\langle l_j \rangle_j)\ \mathtt{in}\ e^n)$. Then $\widehat{e}_2^n = [l/x]e^0$ and $l \notin \mathrm{dom}\, H_1$ and $\langle l_j \in \mathrm{dom}(\cup_i \Sigma_i) \rangle$. By inversion

  $$\exists \Sigma.\ \langle \Sigma_i \rangle_i, \Sigma ; \Gamma_1, \Gamma_2 | x : C^0 \vdash^0 (H_1, e^0) : \widehat{\tau} | S. \qquad (*1)$$

  If $\mathrm{iscf}(C)$, take $\Sigma'_1 = \Sigma_1[l \mapsto C] \wedge \Sigma'_I = \Sigma_I \cup \Sigma$. If not, take $\Sigma'_1 = \Sigma_1 \wedge \Sigma'_I = \Sigma_I \cup \Sigma[l \mapsto C]$. In either case, (iii) and $\cup_i \Sigma'_i \supseteq (\cup_i \Sigma_i) \cup \Sigma$.
  For every $l' \in \mathrm{dom}(\cup_i \Sigma'_i)$ such that $l' \neq l$, $H_1(l') = H_2(l')$ so

  $$\langle \Sigma'_i \rangle_i ; \Gamma_1, \Gamma_2 | x : C^0 \vdash H_2(l') : (\cup_i \Sigma'_i)(l)$$

  by $(*1)$ and Σ_h weakening. We also have $\langle \Sigma'_i \rangle_i^I ; \Gamma_1, \Gamma_2 | x : C^0 \vdash (C, \langle l_j \rangle_j) : C$ because $\langle (\cup_i \Sigma'_i)(l_j) = (\cup_i \Sigma_i)(l_j) \prec \mathrm{ftype}_j(C) \rangle_j$, so

  $$\langle \Sigma'_i \rangle_i ; \Gamma_1, \Gamma_2 | x : C^0 \vdash H_2. \qquad (*2)$$

  l is fresh in H_1 so it is fresh in $(*1)$. Thus by $(*1)$, $(*2)$, we can use Lemma 6 to get

  $$\langle \Sigma'_i \rangle_i ; \Gamma_1, \Gamma_2 | x : C^0 \vdash^0 (H_2, e^0) : \widehat{\tau} | S.$$

  Then, noting that $l \in \mathrm{dom}(\Sigma'_1 \cup \Sigma'_I)$, we have

  $$\langle \Sigma'_i \rangle_i ; \Gamma_1, \Gamma_2 | \emptyset \vdash^0 (H_2, [l/x]e^0) : \widehat{\tau} | S$$

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

  $$\langle \Sigma'_i \rangle_i ; \overline{\Gamma} \vdash^0 (H_2, [l/x]e^0) : \widehat{\tau} | S.$$
- Suppose $\widehat{e}_1^n = \langle\!|e^0|\!\rangle$. Then $\widehat{e}_2^n = l \notin \mathrm{dom}\, H_1$ and $\widehat{\tau} = \mathtt{Code}\langle S', \tau \rangle$. Take $\Sigma'_1 = \Sigma_1$ and $\Sigma'_I = \Sigma_I[l \mapsto \mathtt{Code}\langle S', \tau \rangle]$. By (I) and (IV), we have

  $$\langle \Sigma'_i \rangle_i ; \overline{\Gamma}^{\geq 1} \vdash^0 \langle\!|e^0|\!\rangle : \mathtt{Code}\langle S', \tau \rangle | S$$

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

  $$\langle \Sigma'_i \rangle_i ; \overline{\Gamma} \vdash^0 (H_2, l) : \mathtt{Code}\langle S', \tau \rangle | S.$$

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

- Suppose $\widehat{e}_1^n = (l.f_{j_0} := l')$. Then $\widehat{e}_2^n = l'$ and $H_1(l) = (T, \langle l_j \rangle_j)$. By inversion $\mathrm{iscf}((\cup_i \Sigma_i)(l)) \vee (l \in \mathrm{dom}\, \Sigma_I)$, and the first disjunct implies $l \in \Sigma_1$, so (iii) holds. Take $\langle \Sigma_i' = \Sigma_i \rangle_i$. The updated heap element $(T, \langle l_j \rangle_j [j_0 \mapsto l'])$ is well-formed because by inversion $(\cup_i \Sigma_i)(l') \prec \mathrm{ftype}_{j_0}(\tau)$, where $\tau = (\cup_i \Sigma_i)(l)$. The other locations are unchanged so they remain well-formed. Thus H_2 is well-formed, and we have

$$\langle \Sigma_i' \rangle_i; \overline{\Gamma} \vdash^0 (H_2, l') : \widehat{\tau} | S.$$

For all other primitive reductions, we have $H_1 = H_2$ so with $\Sigma_1' = \Sigma_1$ and $\Sigma_I' = \Sigma_I$, (ii) and (iii) hold.

- Suppose $\widehat{e}_1^n = l.f_{j_0}$, where $n = 0$ and $H_1(l) = (T, \langle l_j \rangle_j)$. By inversion $(\cup_i \Sigma_i)(l_{j_0}) \prec \widehat{\tau}$ so $\langle \Sigma_i' \rangle_i; \overline{\Gamma} \vdash^0 (H_2, l_{j_0}) : \widehat{\tau} | S$.
- Suppose $\widehat{e}_1^n = l.m(\langle l_j \rangle_j)$ where $H_1(l) = (T, \langle l_a' \rangle_a)$. Let $\mathrm{mbody}(m, T) = (\langle x_j \rangle_j^J, e^0)$ and $\tau = (\cup_i \Sigma_i)(l)$. By inversion, $\mathrm{mtype}(m, \tau) = \langle \tau_j \rangle_j^J \xrightarrow{S} \widehat{\tau}$ where $\langle (\cup_i \Sigma_i)(l_j) \prec \tau_j \rangle_j$. If $T = \mathtt{sub}\ D\ \{\langle M_c \rangle_c\}$ and $m = \mathrm{mname}(M_{c_0})$, then

  $$\mathrm{mtype}(m, \mathtt{sub}\ D\ \{\langle M_c \rangle_c\}) = \mathrm{mtype}(m, D) = \langle \tau_j \rangle_j^J \xrightarrow{S} \widehat{\tau}$$

  because class well-formedness rules require method overrides to preserve types. Thus

  $$\langle \Sigma_i \rangle_i; \Gamma_1, \Gamma_2 | \mathtt{this} : D^0 \vdash^0 M_{c_0}$$
  $$\Longrightarrow \langle \Sigma_i \rangle_i; \Gamma_1, \Gamma_2, \mathtt{this} : D^0, \langle x_j : \tau_j^0 \rangle_j^J | \emptyset \vdash^0 e^0 : \tau | S.$$

  We also have $\mathrm{dom}(\cup_i \Sigma_i) \supseteq \mathrm{locs}(e^0)$ from well-formedness of H_1. Γ_H weakening gives

  $$\langle \Sigma_i' \rangle_i; \Gamma_1, \Gamma_2, \mathtt{this} : D^0, \langle x_j : \tau_j^0 \rangle_j^J | \emptyset \vdash H_2$$

  so by the Augmentation Lemma,

  $$\langle \Sigma_i' \rangle_i; \Gamma_1, \Gamma_2, \mathtt{this} : D^0, \langle x_j : \tau_j^0 \rangle_j^J | \emptyset \vdash^0 (H_2, e^0) : \tau | S.$$

  Using the Substitution Lemma $J + 1$ times, we get

  $$\langle \Sigma_i' \rangle_i; \Gamma_1, \Gamma_2 | \emptyset \vdash^0 (H_2, [l/\mathtt{this}][\langle l_j \rangle_j^J / \langle x_j \rangle_j^J] e^0) : \tau | S.$$

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

  $$\langle \emptyset, \emptyset \rangle; \mathtt{this} : \tau^0, \langle x_j : \tau_j^0 \rangle_j | \emptyset \vdash^0 e^0 : \widehat{\tau} | S.$$

  By Σ_e relevance and Γ_e weakening,

  $$\langle \Sigma_i' \rangle_i; \Gamma_1, \Gamma_2, \mathtt{this} : \tau^0, \langle x_j : \tau_j^0 \rangle_j | \emptyset \vdash^0 e^0 : \widehat{\tau} | S.$$

  Then repeating the argument using the Augmentation and Substitution Lemmas gives (i).
- Suppose $\widehat{e}_1^n = \text{`}l$ and $H_1(l) = (\mathtt{Code}, \langle\!| e^0 |\!\rangle)$. Then $n = 1$, and since H_1 is well-formed, we have $\langle \Sigma_i' \rangle_i; \overline{\Gamma} \vdash^1 e^0 : \widehat{\tau} | S$ using Lemma 18. Then by Lemma 11 $\mathrm{locs}(e^0) \subseteq \mathrm{dom}(\cup_i \Sigma_i)$ so by the Augmentation Lemma, $\langle \Sigma_i' \rangle_i; \overline{\Gamma} \vdash^1 (H_2, e^0) : \widehat{\tau} | S$.
- Suppose $\widehat{e}_1^n = e^n.\mathtt{run()}$. Then $n = 0$ and $S = \mathtt{insep}$ and $H_1(l) = (\mathtt{Code}, \langle\!| \widehat{e}_2^n |\!\rangle)$. By (III) and (IV), $\overline{\Gamma} = \emptyset | \emptyset$. Then by well-formedness of H_1,

  $$\langle \Sigma_i' \rangle_i; \emptyset | \emptyset \vdash (\mathtt{Code}, \langle\!| \widehat{e}_2^n |\!\rangle) : \mathtt{Code}\langle S, \widehat{\tau} \rangle.$$

  $H_1 = H_2$ so by the Demotion Lemma $\langle \Sigma_i' \rangle_i; \overline{\Gamma} \vdash^0 \widehat{e}_2^n : \widehat{\tau} | S$.

We now consider non-trivial evaluation contexts. Let $\widehat{e}_1^n = \mathcal{E}^{n,k}[r^k]$ and $\widehat{e}_2^n = \mathcal{E}^{n,k}[e^k]$.

- Suppose $\mathcal{E}^{n,k} = (\mathtt{let}\ x \Leftarrow \mathtt{new}\ C(\langle v_j^n \rangle_j)\ \mathtt{in}\ \mathcal{E}_e^{n,k})$. Then necessarily $n > 0$. By inversion,

  $$\langle \langle \Sigma_i \rangle_i; \overline{\Gamma} \vdash^n (H_1, v_j^n) : \mathrm{ftype}_j(C) | S \rangle_j \quad (*3)$$
  $$\exists \Sigma.\ \langle \Sigma_i \rangle_i, \Sigma; \Gamma_1, \Gamma_2 | x : C^n \vdash^n (H_1, \mathcal{E}_e^{n,k}[r^k]) : \widehat{\tau} | S \quad (*4)$$

  and the derivations of these are disjoint subtrees of (I).
  We want to apply IH to $(*4)$, but to do so we must check (III) and (IV). Because $n > 0$ the $x : C^n$ is not a level-0 binding, hence (IV) holds for the subconfiguration. If $S = \mathtt{insep}$ then $\overline{\Gamma}^{>n} = \emptyset | \emptyset$ so $(\Gamma_1, \Gamma_2 | x : C^n)^{>n} = \emptyset | \emptyset$ so (III) is satisfied.
  Therefore, by IH(i) (that is, conclusion (i) of IH), $\exists \langle \Sigma_i' \rangle_i^{I+1}$ such that

  $$\langle \Sigma_i' \rangle_i^{I+1}; \Gamma_1, \Gamma_2 | x : C^n \vdash^n (H_2, \mathcal{E}_e^{n,k}[e^k]) : \widehat{\tau} | S.$$

  By IH(iii), $\forall l \in \mathrm{dom}(\cup_i \Sigma_i')$. $H_1(l) \neq H_2(l)$ implies $l \in \mathrm{dom}(\Sigma_1' \cup \Sigma_{I+1}')$. Thus by the refined Smashing Lemma, $\langle \Sigma_i' \rangle_i^I; \overline{\Gamma} \vdash H_2$.
  Also by IH(iii), any l that H_1 and H_2 disagree on satisfy one of:
  - $l \notin \mathrm{dom}\, H_1$, in which case l is fresh for $(*3)$ because the domain of store typings in a configuration typing is bounded by the domain of the heap.
  - l is local to $(*4)$, in which case it is fresh in $(*3)$ because they are disjoint subtrees.
  - $l \in \mathrm{dom}(\Sigma_1 \cup \Sigma)$. If $l \in \mathrm{dom}\, \Sigma_1$ then $l \in \mathrm{dom}(\Sigma_1' \cup \Sigma_I')$. Else $l \in \Sigma$ implies that l is fresh for $(*3)$ because it is a subtree of (I) that is disjoint from $(*4)$, and Σ is introduced at the root of $(*4)$.

  Hence l is fresh in $(*3)$ or $l \in \mathrm{dom}(\cup_{i=1}^I \Sigma_i')$. Therefore, by Lemma 6,

  $$\langle \langle \Sigma_i' \rangle_i^I; \overline{\Gamma} \vdash^n (H_2, e_j^n) : \mathrm{ftype}_j(C) | S \rangle_j$$

  which is the last piece needed for (i).
  $\langle \Sigma_i' \rangle_i^I$ satisfies (ii) because $\langle \Sigma_i' \rangle_i^{I+1}$ obeys IH(ii). It also satisfies (iii) because by IH(iii), $H_1(l) \neq H_2(l)$ ensures one of three conditions:
  - $l \notin \mathrm{dom}\, H_1$.
  - $l \in \mathrm{dom}(\Sigma_1 \cup \Sigma)$. If $l \in \mathrm{dom}\, \Sigma_1$ then $l \in \mathrm{dom}(\Sigma_1 \cup \Sigma_I)$. If $l \in \Sigma$, then it is local to (I).
  - l is local to $(*4)$. Then it is also local to the supertree, (I).
- If $\mathcal{E}^{n,k} = \mathcal{E}_M^{n,k}$, the argument is mostly a repetition of the previous case.
- The remaining cases are all straightforward. We simply use IH to obtain $\langle \Sigma_i' \rangle_i^I$, and apply Lemma 6 to see that the subterms that did not participate in the small step remain well-typed if we augment them with H_2.

This concludes the proof. □

Lemma 18. *If $\langle \Sigma_i \rangle_i; \Gamma_1, \Gamma_2 | \emptyset \vdash^n (H, \widehat{e}^n) : \widehat{\tau} | S$ holds, then $\langle \Sigma_i \rangle_i; \Gamma_1 | \Gamma_2 \vdash^n (H, \widehat{e}^n) : \widehat{\tau} | S$.*

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