

Type-Preserving Flow Analysis and Interprocedural Unboxing

Extended Version

Neal Glew

Intel Labs
neal.glew@intel.com

Leaf Petersen

Intel Labs
leaf.petersen@intel.com

Abstract

Interprocedural flow analysis can be used to eliminate otherwise unnecessary heap allocated objects (*unboxing*), and in previous work we have shown how to do so while maintaining correctness with respect to the garbage collector (GC). In this paper, we extend the notion of flow analysis to incorporate types, enabling optimization of typed programs. We apply this typed analysis to specify and construct a type-preserving interprocedural unboxing optimization, and prove that the optimization preserves type and GC safety along with program semantics. We also show that the unboxing optimization can be applied independently to separately compiled program modules, and prove via a contextual equivalence result that unboxing a module in isolation preserves program semantics.

1. Introduction

Many languages and compilers use a *uniform object representation* in which every source level object is represented at least initially by a heap allocated object. Such a representation allows polymorphic functions to be compiled once and enables the implementation of features that use runtime type information. In this representation machine integers and floating-point numbers are placed in a single-field object, a box, and this operation is called boxing. Operations such as addition require first projecting the number from the box (unboxing), then performing the operation, and finally creating a new box for the result. Boxing and unboxing operations add considerable overhead, and thus it is highly desirable to remove them when possible – e.g. when polymorphism or features requiring runtime type information are not being used. We refer to the general class of optimizations that attempt to remove unnecessary box and unbox operations as *unboxing* optimizations. We refer to unboxing optimizations that attempt to eliminate boxing and unboxing across function boundaries as *interprocedural unboxing*. We also include in this latter category optimizations (such as the one given in this paper) that attempt to unbox objects written to and read from other objects in the heap.

Interprocedural unboxing presents additional challenges in a typed setting, since type information must be updated to reflect any unboxing. A box might flow to an argument in an application, and the parameter of the called function might flow to an unbox operation. If the optimization decides to remove the box and unbox

operations then it must also remove the box type on the parameter. In other words, typed unboxing requires not just rewriting uses and definitions in the traditional sense, but also rewriting intermediate points in the program through which the unboxed values flow. At a high level then, the optimization can be viewed as selecting a set of box operations, unbox operations, and box types to remove. Such a selection has a global consistency requirement—a box type should only be removed if all boxes that flow to it are removed, a box operation should only be removed if all unbox operations it flows to are removed, and so on. Thus choosing a set of boxed objects to eliminate and rewriting the program to reflect this choice in a consistent manner requires knowing what things flow to what points in the program, a question that flow analyses are designed to answer. In this paper we use the results of flow analysis to formulate correctness conditions for unboxing and then prove that those conditions ensure correct optimization.

In previous work [8] we considered the simpler problem of rewriting garbage-collector (GC) meta-data rather than full types. An accurate GC requires specifying for each field of each object and each slot of each stack activation frame whether it contains a pointer into the GC heap or not (contains a machine integer, floating-point number, etc.). As with types, when interprocedurally unboxing such meta-data must be rewritten in a globally consistent manner. Our previous paper showed how to do this rewriting correctly using the results of a flow analysis in a whole program setting. In this paper we extend these ideas to develop a methodology for dealing with interprocedural optimization of statically typed languages (including universal polymorphism) in a type preserving fashion. We also show that this methodology does not depend on whole program compilation and can be extended to correctly optimize independent program modules.

In the following sections, we begin by defining a strongly typed polymorphic core language in which boxing has been made explicit. As in our previous paper we formalize a notion of GC safety for our language and show that well-typed programs are GC safe throughout execution. Next we specify a set of abstract conditions that a reasonable flow analysis must satisfy, with the property that any flow analysis that satisfies these conditions can be used in our framework to optimize programs. The main section of the paper defines an unboxing optimization parameterized over a choice of objects to unbox, and gives a set of correctness conditions under which such a choice is guaranteed to preserve typing and preserve semantics. We show that this set of correctness conditions is satisfiable by constructing a simple unboxing algorithm that satisfies these conditions. Finally we extend the system slightly by defining a notion of unboxing for modules and show that it is correct in the sense that a module is contextually equivalent to its unboxing.

While our paper is specifically about the concrete optimization of unboxing, the ideas used here generalize naturally to other optimizations that change the representation of objects in a non-

local fashion. Such optimizations include dead-field elimination, dead-parameter elimination, sum representation optimizations, and thunk elimination in lazy languages. All of these impose similar requirements for rewriting types and GC meta-data in a globally consistent fashion. Flow analyses can be used to specify and implement these (and others), and we believe (based on practical experience in our compiler) that the framework presented here extends naturally to such optimizations. As far as we know, this and our previous paper are the first to use a flow analysis to rewrite types and GC meta-data in a globally consistent fashion, and to use a flow analysis to formulate correctness conditions for this rewriting process and prove these conditions sound.

2. A type and GC safe core language

Consider the following untyped program (using informal notation), where `box` denotes a boxing operation that wraps its argument in a heap-allocated structure, and `unbox` denotes its elimination form that projects out the boxed item from the box:

```
let f = λx.(box x) in unbox(unbox(f (box 3)))
```

The only definition reaching the variable x is the boxed machine integer 3, and consequently it is easy to see that this program can be rewritten to eliminate the boxing as follows:

```
let f = λx.x in f 3
```

This second version is much better in that it does less allocation, and executes fewer instructions. In this optimized version of the program however, an important property has changed that is not reflected in this untyped syntax. Specifically, the GC status of values reaching x has changed: whereas in the original program all values reaching x are represented as heap allocated pointers, in the second program all values reaching x are represented as machine integers. From the standpoint of a garbage collector, a collection occurring while x is live must treat x as a root in the first program, and must ignore x in the second program.

The question of which variables should be treated as roots by the garbage collector is a subtle but crucial one for the purposes of optimization and compiler correctness. Consider a modification of the previous example in which the function f is used polymorphically:

```
let f = λx.(box x) in unbox(unbox((unbox(f f)) (box 3)))
```

In this variant, f is applied to itself, and the boxed result (itself) is unboxed and applied to a boxed integer. The resulting doubly boxed integer is then unboxed. Assuming that functions are represented as heap-allocated objects, each variable in this program has a concrete and statically known status as either a GC root or GC non-root, since all objects passed to f are heap references. However, an attempt to unbox this program as with the previous example results in f being applied to both heap references (f) and non-heap references (3).

```
let f = λx.x in (f f) 3
```

Consequently, a correct optimizer must decline to unbox this program (at least in entirety) to avoid incorrect GC behavior.¹

In our previous work [8] we developed a core language capturing the essential issues of GC safety, along with an analysis and optimization framework for reasoning about and correctly optimizing GC-safe programs in an untyped setting. However, the framework is essentially limited to untyped programs and consequently does not scale to typed core languages in which one must be able

¹ A conservative GC, or systems that tag pointers to distinguish them from non-pointers would not impose this restriction, but come with other drawbacks. See Section 2.2 for more discussion of the GC model.

to *check* the well-typedness (and hence the GC safety) of programs before and after optimization [6].

2.1 Type safety

How does the problem of unboxing change in a typed setting? Consider again the first example from this section using a still informal but now typed notation:

```
let
  f: box(int) → box(box(int)) = λx: box(int).(box x)
in unbox(unbox(f(box 3)))
```

As before, it is apparent that the only definition reaching the variable x is the boxed machine integer 3, and as before we can consider rewriting this program to eliminate (interprocedurally) the boxing. However, simply rewriting the terms of the program is inadequate from the standpoint of type-preserving compilation, since the result is not well-typed:

```
let
  f: box(int) → box(box(int)) = λx: box(int).x
in f 3
```

The types of both the actual argument and the return value of f have changed, and are no longer consistent with the type annotation for f and x . In order to correctly unbox this program then, it is necessary to rewrite not just the terms, but also the types:

```
let f: int → int = λx: int.x in f 3
```

The requirement to rewrite types is more imposing than might at first be apparent. In the untyped setting, it was sufficient to have information only about the direct definitions (boxes) and uses (unboxes) of objects. To rewrite types requires not just information about uses and definitions but also about intermediate program points (and other objects) through which the boxed objects flow. Notice in particular that in rewriting the type of f , we were forced to rewrite sub-components of the type that are not obviously syntactically connected to any box introduction or elimination.

In addition to incurring these additional rewriting requirements, the typed setting must still account for GC safety. Consider again the polymorphic variant of the previous untyped example (naming the first application of f for clarity).

```
let
  f: ∀α.α → box(α) = Λα.λx:α.(box x)
  g: box(∀α.α → box(α)) = f[∀α.α → box(α)](f)
in unbox(unbox((unbox g)[box(int)])(box 3)))
```

Here we have f applied to itself at a universal type to produce a boxed version of itself (g), which is then unboxed and applied to a boxed integer. Attempting to unbox this example (rewriting types as necessary) immediately illuminates the problem.

```
let
  f: ∀α.α → α = Λα.λx:α.x
  g: ∀α.α → α = f[∀α.α → α](f)
in g[int](3)
```

The function f is instantiated directly at a universal function type, and via its alias (g) at a machine integer type. As with the untyped example in the previous section, the compiler cannot assign a concrete GC status to the variable x . For correctness then, the compiler must not (fully) unbox this example, and must leave at least the boxing operation on the integer parameter to f .²

² It is worth noting that an optimizing compiler might duplicate the body of f to make it monomorphic, thereby enabling the unboxing. It is also possible to use a runtime type passing interpretation to relax the constraints on the garbage collector sufficiently to permit this example [1]. These

Traceabilities	$t ::= \mathbf{b} \mid \mathbf{r}$
Labels	$i ::= 0, 1, \dots$
Type variables	α, β
Labeled Types	$\tau ::= \sigma^i$
Types	$\sigma ::= \alpha \mid \mathbf{B} \mid \forall \alpha. \tau_1 \rightarrow \tau_2 \mid \mathbf{box}(\tau)$
Term variables	f, x, y, z
Constants	c
Labeled Terms	$e ::= m^i \mid v^i$
Terms	$m ::= x \mid \mathbf{fix} \ f[\alpha](x:\tau_1):\tau_2.e \mid e_1[\tau] \ e_2 \mid \mathbf{box}_\tau e \mid \mathbf{unbox} \ e \mid \rho(e)$
Values	$v ::= c \mid \langle \rho, \mathbf{fix} \ f[\alpha](x:\tau_1):\tau_2.e \rangle \mid \langle v^i:\tau \rangle$
Environments	$\rho ::= x_1:\tau_1 = v_1^{j_1}, \dots, x_n:\tau_n = v_n^{j_n}$
States	$M ::= (\rho, e)$

Figure 1. Syntax

In the rest of this paper we make these issues concrete and formal, and we show how to deal with them by extending the notion of flow analysis to incorporate types, thereby generating the necessary flow information to correctly rewrite types and terms in a consistent fashion. While we focus on a concrete optimization (unboxing), we believe that these ideas are generally applicable to representation optimizations based on flow analysis in typed intermediate languages.

2.2 A core language for GC safety

In order to give a precise account of typed flow analysis and interprocedural unboxing, we begin by defining a type-safe core language incorporating the essential features of GC safety. The motivation for the (small) idiosyncracies of this language lies in the requirements of the underlying model of garbage collection. We assume that pointers cannot be intrinsically distinguished from non-pointers, and hence the compiler is required to statically annotate the program with garbage collection meta-data such that at any garbage-collection point the garbage collector can reconstruct exactly which live variables are roots. Typically, this takes the form of annotations on variables and temporaries indicating which contain heap-pointers (the roots) and which do not (the non-roots), along with information at every allocation site indicating which fields of the allocated object contain traceable data. This approach is common in modern systems, and it is this approach that we target in this paper.

Figure 1 defines the syntax of our core language. The essence of the language is a standard polymorphic lambda calculus extended with a fixed-point operator, implemented via an explicit environment semantics. For the purposes of the semantics, we also include a form of degenerate type information we call *traceabilities*. Traceabilities describe the GC status of variables: the traceability \mathbf{b} (for bits) indicates something that should be ignored by the garbage collector, while the traceability \mathbf{r} (for reference) indicates a GC-managed pointer. The traceability \mathbf{b} is inhabited by an unspecified set of constants c while the traceability \mathbf{r} is inhabited by functions (anticipating their implementation by heap-allocated closures) and by boxed objects. Anticipating the needs of the flow analysis, we label each type, term, value, and variable binding site with an integer label. We do not assume that labels or variables are unique within a program.

Types, σ , consist of type variables, the base type of constants, \mathbf{B} , function types, $\forall \alpha. \tau_1 \rightarrow \tau_2$, and boxed types, $\mathbf{box}(\tau)$. In order to provide a concrete implementation strategy for the garbage col-

lector, we insist that every type correspond to a traceability so that we can extract the necessary garbage collection meta-data. Types are mapped to traceabilities using the function $tr(\tau)$, defined in Figure 2. Polymorphic functions are restricted by well-formedness rules to only be instantiated with types with the traceability \mathbf{r} , and consequently $tr(\alpha) = \mathbf{r}$. We define substitution of types in the standard way and define $\tau[\sigma^i/\alpha] = \tau[\sigma/\alpha]$.

Expressions e consist of labeled terms m^i and labeled values v^i . The terms m consist of variables, functions, applications, box introductions, box eliminations, and frames. Functions $\mathbf{fix} \ f[\alpha](x:\tau_1):\tau_2.e$ are polymorphic and recursive and variable-binding sites are decorated with types. We represent heap allocation in the language via the $\mathbf{box}_\tau e$ term, which corresponds to allocating a heap cell containing the value for e . The type τ is used by the dynamic semantics to provide the meta-data with which the heap-cell will be tagged, allowing the garbage collector to trace the cell. However, only the top-level traceability of the type (given by the $tr()$ function in Figure 2) is actually required by the dynamic semantics, and so the language can be erased into an untyped language in the obvious way. Objects can be projected out of an allocated object by the $\mathbf{unbox} \ e$ operation. Frames $\rho(e)$ are discussed further below.

Values consist of either constants, closures, or heap-allocated boxes. We distinguish between the introduction form ($\mathbf{box}_\tau e$) and the value form ($\langle v^i:\tau \rangle$) for allocated objects. The introduction form corresponds to the allocation instruction, whereas the value form corresponds to the allocated heap value. This distinction is key for the formulation of GC safety and the dynamic semantics. For this reason we also distinguish between functions ($\mathbf{fix} \ f[\alpha](x:\tau_1):\tau_2.e$) and their heap-allocated closures ($\langle \rho, \mathbf{fix} \ f[\alpha](x:\tau_1):\tau_2.e \rangle$).

For notational convenience, we will sometimes use the notation $v_{\mathbf{b}}$ to indicate that a value v is a non-heap-allocated value (i.e. a constant c), and $v_{\mathbf{r}}$ to indicate that a value v is a heap-allocated value. If t is a traceability meta-variable, then we use v_t to indicate that v is a value of the same traceability as t . In examples, we use a derived \mathbf{let} expression, taking it to be syntactic sugar for application in the usual manner.

Environments ρ map variables to values. The term $\rho(e)$ executes e in the environment ρ rather than the outer environment – all of the free variables of e are provided by ρ . The nested set of these environments at any point can be thought of as the activation stack frames of the executing program. The traceability of the typing annotations on variables in the environments play the role of stack-frame GC meta-data, indicating which slots of the frame are roots (traceability \mathbf{r}). The environments buried in closures ($\langle \rho, \mathbf{fix} \ f[\alpha](x:\tau_1):\tau_2.e \rangle$) similarly provide the traceabilities of values reachable from the closure via the type annotations on the variables in the environment, and hence provide the GC meta-data for tracing through closures. While we do not make the process of garbage collection explicit, it should be clear how to extract the appropriate set of GC roots from the environment and any active frames.

This core language contains the appropriate information to formalize a notion of GC safety consisting of two complementing pieces. First we define a dynamic semantics in which reductions that might lead to undefined garbage-collector behavior are explicitly undefined. Programs that take steps in this semantics do not introduce ill-formed heap objects or activation frames. Secondly, we define a notion of a traceable program: one in which all heap values have valid GC meta-data. Reduction steps in the semantics can then be shown to maintain the traceability property in addition to the usual well-typedness property. By showing that typable programs are both traceable programs and have well-defined semantics, we

optimizations are orthogonal (but complementary) to the issues addressed by this paper.

$$\begin{array}{l}
tr(\sigma^i) = tr(\sigma) \\
tr(\alpha) = \mathbf{r} \\
tr(\mathbf{B}) = \mathbf{b} \\
tr(\forall\alpha.\tau_1 \rightarrow \tau_2) = \mathbf{r} \\
tr(\mathbf{box}(\tau)) = \mathbf{r} \\
\\
\frac{x:\tau = v^j \in \rho}{(\rho, x^k) \mapsto (\rho, v^j)} \\
\\
\frac{(\rho, (\mathbf{fix} f[\alpha](x:\tau_1):\tau_2.e)^j) \mapsto (\rho, \langle \rho, \mathbf{fix} f[\alpha](x:\tau_1):\tau_2.e \rangle^j)}{tr(\tau) = t} \\
\frac{(\rho, (\mathbf{box}_\tau v_t^i)^j) \mapsto (\rho, \langle v_t^i:\tau \rangle^j)}{(\rho, e_1) \mapsto (\rho, e'_1)} \\
\frac{(\rho, (e_1[\tau] e_2)^i) \mapsto (\rho, (e'_1[\tau] e_2)^i)}{(\rho, e_2) \mapsto (\rho, e'_2)} \\
\frac{(\rho, (v^i[\tau] e_2)^j) \mapsto (\rho, (v^i[\tau] e'_2)^j)}{v_f = \langle \rho', \mathbf{fix} f[\alpha](x:\tau_1):\tau_2.e \rangle \quad \tau' = (\forall\alpha.\tau_1 \rightarrow \tau_2)^i} \\
\frac{\tau'_1 = \tau_1[\tau/\alpha] \quad tr(\tau'_1) = t}{(\rho, (v_f^i[\tau] v_t^j)^k) \mapsto (\rho, \langle \rho', f:\tau' = v_f^i, x:\tau'_1 = v_t^j \rangle(e[\tau/\alpha])^k)} \\
\frac{(\rho, e) \mapsto (\rho, e')}{(\rho, (\mathbf{box}_\tau e)^i) \mapsto (\rho, (\mathbf{box}_\tau e')^i)} \\
\frac{(\rho, e) \mapsto (\rho, e')}{(\rho, (\mathbf{unbox} e)^i) \mapsto (\rho, (\mathbf{unbox} e')^i)} \\
\frac{(\rho, (\mathbf{unbox} \langle v^i:\tau \rangle^j)^k) \mapsto (\rho, v^i)}{(\rho', e) \mapsto (\rho', e')} \\
\frac{(\rho, \rho'(e^i) \mapsto (\rho, \rho'(e')^i)}{(\rho, \rho'(v^i)^j) \mapsto (\rho, v^i)}
\end{array}$$

Figure 2. Operational Semantics

thereby show that GC correctness for a compiler optimization can be achieved simply by preserving well-typedness.

2.3 Operational semantics

We choose to use an explicit environment semantics rather than a standard substitution semantics since this makes the GC meta-data (as given by the types) for stack frames and closures explicit in the semantics. Thus a machine state (ρ, e) supplies an environment ρ for e that provides the values of the free variables of e during execution. Environments contain typing annotations on each of the variables mapped by the environment that provide the traceabilities of the variables.

Reduction in this language is for the most part fairly standard. We deviate somewhat in that we explicitly model the allocation of heap objects as a reduction step—hence there is an explicit reduc-

Labeled Terms

$$\frac{\vdash m \mathbf{tr}}{\vdash m^i \mathbf{tr}}$$

Terms

$$\frac{}{\vdash x \mathbf{tr}}$$

$$\frac{\vdash e_1 \mathbf{tr} \quad \vdash e_2 \mathbf{tr}}{\vdash e_1[\tau] e_2 \mathbf{tr}}$$

$$\frac{\vdash e \mathbf{tr}}{\vdash \mathbf{unbox} e \mathbf{tr}}$$

Values

$$\frac{}{\vdash_v c:\mathbf{b}}$$

$$\frac{\vdash_v v:\mathbf{tr}(\tau)}{\vdash_v \langle v^i:\tau \rangle:\mathbf{x}}$$

$$\vdash \rho \mathbf{tr} \quad \vdash e \mathbf{tr}$$

$$\frac{}{\vdash_v \langle \rho, \mathbf{fix} f[\alpha](x:\tau_1):\tau_2.e \rangle:\mathbf{r}}$$

Environments

$$\vdash_v v_1:\mathbf{tr}(\tau_1) \quad \dots \quad \vdash_v v_n:\mathbf{tr}(\tau_n)$$

$$\frac{}{\vdash x_1:\tau_1 = v_1^{j_1}, \dots, x_n:\tau_n = v_n^{j_n} \mathbf{tr}}$$

Machine States

$$\vdash \rho \mathbf{tr} \quad \vdash e \mathbf{tr}$$

$$\frac{}{\vdash (\rho, e) \mathbf{tr}}$$

Figure 3. Traceability

tion mapping a function term $\mathbf{fix} f[\alpha](x:\tau_1):\tau_2.e$ to an allocated closure $\langle \rho, \mathbf{fix} f[\alpha](x:\tau_1):\tau_2.e \rangle$, and similarly for boxed objects and values. More notably, beta-reduction is restricted to only permit construction of a stack frame when the type for the parameter variable has an appropriate traceability for the actual argument value. This captures the requirement that stack frames have correct meta-data for the garbage collector. In actual practice, incorrect meta-data for stack frames leads to undefined behavior (since incorrect meta-data may cause arbitrary memory corruption by the garbage collector)—similarly here in the meta-theory we leave the behavior of such programs undefined. In a similar fashion, we only define the reduction of the allocation operation to an allocated value ($\mathbf{box}_\tau v_t \mapsto \langle v_t:\tau \rangle$) when the operation meta-data is appropriate for the value (i.e. $tr(\tau) = t$).

It is important to note that this semantics does not imply an explicit check of the meta-data associated with these reductions. The point is rather that the semantics only specifies how programs behave when these conditions are met—in all other cases the behavior of the program is undefined (stuck).

2.4 Traceability

The operational semantics ensures that no reduction step introduces mis-tagged values. In order to make use of this, we define a judgement for checking that a program does not have a mis-tagged value in the first place. Implicitly this judgement defines what a well-formed heap and activation stack looks like; however, since our heap and stack are implicit in our machine states, it takes the form of a judgement on terms, values, environments, and machine states.

The value judgement $\vdash_v v:t$ asserts that a value v is well-formed, and has traceability t . This corresponds to having the types on the variables in the environment of each function value have traceabilities that are consistent with the values to which they are bound, and the type on each boxed value be consistent with the traceability of the object nested in the box. An environment is consistent, $\vdash \rho \text{ tr}$, when the annotation on each variable agrees with the traceability of the value it is bound to. The term judgement $\vdash e \text{ tr}$ and machine state judgement $\vdash M \text{ tr}$ simply check that all values and environments (and hence stack frames) contained in the term or machine state are well-formed.

The key result for traceability is that it is preserved under reduction. That is, if a traceable term takes a well-defined reduction step, then the resulting term will be traceable.

Lemma 1 (Preservation of traceability)

If $\vdash M \text{ tr}$ and $M \mapsto M'$ then $\vdash M' \text{ tr}$.

Proof: If $\vdash (\rho, e) \text{ tr}$ then $\vdash \rho \text{ tr}$ and $\vdash e \text{ tr}$. If $(\rho, e) \mapsto (\rho', e')$ then the result follows if we can show $\vdash e' \text{ tr}$. The proof of that is by induction on the derivation of $(\rho, e) \mapsto (\rho', e')$. Consider the cases for the last rule used to derive it (the cases are in the same order as in the figure):

- In this case, $e = x^k$ for some x and k , and $e' = v^j$ where $x:\tau = v^j \in \rho$ for some τ, v , and j . Since $\vdash \rho \text{ tr}$ then $\vdash_v v:tr(\tau)$, so by the traceability rules $\vdash v^j \text{ tr}$ as required.
- In this case, $e = (\text{fix } f[\alpha](x:\tau_1):\tau_2.e'')^j$ for some $f, \alpha, x, \tau_1, \tau_2, e''$, and j , and $e' = \langle \rho, \text{fix } f[\alpha](x:\tau_1):\tau_2.e'' \rangle^j$. The first hypothesis is that $\vdash (\text{fix } f[\alpha](x:\tau_1):\tau_2.e'')^j \text{ tr}$. There is only one rule to derive this judgement and that rule requires that $\vdash \text{fix } f[\alpha](x:\tau_1):\tau_2.e'' \text{ tr}$, which in turn can only be derived by one rule that requires that $\vdash e'' \text{ tr}$. Then, and since $\vdash \rho \text{ tr}$, by the rules for traceability, $\vdash_v \langle \rho, \text{fix } f[\alpha](x:\tau_1):\tau_2.e'' \rangle:\mathbf{x}$, and by the traceability rules again $\vdash \langle \rho, \text{fix } f[\alpha](x:\tau_1):\tau_2.e'' \rangle^j \text{ tr}$, as we are required to prove.
- In this case, $e = (\text{box}_\tau v_t^i)^j$ for some τ, v_t, i , and j , $e' = \langle v_t^i:\tau \rangle^j$, and $tr(\tau) = t$. The first hypothesis is that $\vdash (\text{box}_\tau v_t^i)^j \text{ tr}$. There is only one rule to derive this judgement and that rule requires that $\vdash \text{box}_\tau v_t^i \text{ tr}$, which in turn can only be derived by one rule that requires that $\vdash v_t^i \text{ tr}$. There is only one rule to derive the latter judgement and it requires that $\vdash_v v_t:t'$ for some t' . By inspection of the rules for value traceability, we see that $t = t'$. Since $tr(\tau) = t = t'$, by the rules for traceability, $\vdash_v \langle v_t^i:\tau \rangle:\mathbf{x}$, and by the traceability rules again $\vdash \langle v_t^i:\tau \rangle^j \text{ tr}$, as we are required to prove.
- In this case, $e = (e_1[\tau] e_2)^i$ for some e_1, τ, e_2 , and i , $e' = (e'_1[\tau] e'_2)^i$ for some e'_1 , and $(\rho, e_1) \mapsto (\rho, e'_1)$ is a subderivation. The first hypothesis is that $\vdash (e_1[\tau] e_2)^i \text{ tr}$. There is only one rule to derive this judgement and that rule requires that $\vdash e_1[\tau] e_2 \text{ tr}$, which in turn can only be derived by one rule that requires both $\vdash e_1 \text{ tr}$ and $\vdash e_2 \text{ tr}$. Thus, by the induction hypothesis, $\vdash e'_1 \text{ tr}$. Then, by the rules for traceability, $\vdash e'_1[\tau] e_2 \text{ tr}$, and by the traceability rules again, $\vdash (e'_1[\tau] e_2)^i \text{ tr}$, as we are required to prove.
- In this case, $e = (v^i[\tau] e_2)^j$ for some v, i, τ, e_2 , and j , $e' = (v^i[\tau] e'_2)^j$ for some e'_2 , and $(\rho, e_2) \mapsto (\rho, e'_2)$ is a subderivation. The first hypothesis is that $\vdash (v^i[\tau] e_2)^j \text{ tr}$. There is only one rule to derive this judgement and that rule requires that $\vdash v^i[\tau] e_2 \text{ tr}$, which in turn can only be derived by one rule that requires both $\vdash v^i \text{ tr}$ and $\vdash e_2 \text{ tr}$. Thus, by the induction hypothesis, $\vdash e'_2 \text{ tr}$. Then, by the rules for traceability, $\vdash v^i[\tau] e'_2 \text{ tr}$, and by the traceability rules again, $\vdash (v^i[\tau] e'_2)^j \text{ tr}$, as we are required to prove.
- In this case:

$$\begin{aligned} e &= (v_f^j[\tau] v_t^k)^l \\ v_f &= \langle \rho', \text{fix } f[\alpha](x:\tau_1):\tau_2.e'' \rangle \\ e' &= \rho''(e''[\tau/\alpha])^l \\ \rho'' &= \rho', f:\tau' = v_f^j, x:\tau'_1 = v_t^k \\ \tau' &= (\forall \alpha. \tau_1 \rightarrow \tau_2)^j \\ \tau'_1 &= \tau_1[\tau/\alpha] \\ tr(\tau'_1) &= t \end{aligned} \quad (6)$$
 for some $\rho', f, \alpha, x, \tau_1, \tau_2, e'', j, \tau, v_t, k$, and l . The first hypothesis is that $\vdash (v_f^j[\tau] v_t^k)^l \text{ tr}$. There is only one rule to derive that judgement and that rule requires that $\vdash v_f^j[\tau] v_t^k \text{ tr}$, which in turn can only be derived by one rule that requires both $\vdash v_f^j \text{ tr}$ and $\vdash v_t^k \text{ tr}$. Both of these latter derivations can only be derived by one rule and those rules require that $\vdash_v v_f:\mathbf{x}$ (1) and $\vdash_v v_t:t$ (2) (a simple inspection reveals the traceabilities to be \mathbf{x} and t). Judgement 1 can only be derived by one rule and that rule requires that $\vdash \rho' \text{ tr}$ (3) and $\vdash e'' \text{ tr}$ (4). By (3), (1), $tr(\tau') = \mathbf{x}$, (2), and (6) we can derive $\vdash \rho'' \text{ tr}$ (5). By (5) and (4) we can derive $\vdash \rho''(e'')^l \text{ tr}$, and then $\vdash e' \text{ tr}$, as required.
- In this case, $e = (\text{box}_\tau e'')^i$ for some τ, e'' , and i , $e' = (\text{box}_\tau e''')^i$, and $(\rho, e'') \mapsto (\rho, e''')$ is a subderivation. The first hypothesis is that $\vdash (\text{box}_\tau e'')^i \text{ tr}$. There is only one rule to derive this judgement and that rule requires that $\vdash \text{box}_\tau e'' \text{ tr}$, which in turn can only be derived by one rule that requires that $\vdash e'' \text{ tr}$. Thus, by the induction hypothesis, $\vdash e''' \text{ tr}$. Then, by the rules for traceability, $\vdash \text{box}_\tau e''' \text{ tr}$, and by the traceability rules again, $\vdash (\text{box}_\tau e''')^i \text{ tr}$, as we are required to prove.
- In this case, $e = (\text{unbox } e'')^i$ for some e'' and i , $e' = (\text{unbox } e''')^i$, and $(\rho, e'') \mapsto (\rho, e''')$ is a subderivation. The first hypothesis is that $\vdash (\text{unbox } e'')^i \text{ tr}$. There is only one rule to derive this judgement and that rule requires that $\vdash \text{unbox } e'' \text{ tr}$, which in turn can only be derived by one rule that requires that $\vdash e'' \text{ tr}$. Thus, by the induction hypothesis, $\vdash e''' \text{ tr}$. Then, by the rules for traceability, $\vdash \text{unbox } e''' \text{ tr}$, and by the traceability rules again, $\vdash (\text{unbox } e''')^i \text{ tr}$, as we are required to prove.
- In this case, $e = (\text{unbox } \langle v^i:\tau \rangle^j)^k$ for some τ, v, i, j , and k , and $e' = v^i$. The first hypothesis is that $\vdash (\text{unbox } \langle v^i:\tau \rangle^j)^k \text{ tr}$. There is only one rule to derive this judgement and that rule requires that $\vdash \text{unbox } \langle v^i:\tau \rangle^j \text{ tr}$, which in turn can only be derived by one rule that requires that $\vdash \langle v^i:\tau \rangle^j \text{ tr}$. There is only one rule to derive this latter judgement and that rule requires that $\vdash_v \langle v^i:\tau \rangle:t$ for some t , which in turn can only be derived by one rule that requires that $\vdash_v v:tr(\tau)$. Then, by the rules for traceability, $\vdash v^i \text{ tr}$, as we are required to prove.
- In this case, $e = \rho'(e'')^i$ for some ρ', e'' and i , $e' = \rho'(e''')^i$ for some e''' , and $(\rho', e'') \mapsto (\rho', e''')$. The hypothesis $\vdash e \text{ tr}$ can only be derived in a certain way, unpacking that we see that $\vdash \rho' \text{ tr}$ and $\vdash e'' \text{ tr}$. Then by the induction hypothesis,

$$\begin{array}{c}
\Delta ::= \alpha_1, \dots, \alpha_n \\
\Gamma ::= x_1:\tau_1, \dots, x_n:\tau_n \\
\boxed{\Delta \vdash \tau \text{ wf}} \\
\frac{ftw(\tau) \subseteq \Delta}{\Delta \vdash \tau \text{ wf}} \\
\boxed{\vdash \tau_1 = \tau_2} \\
\frac{\frac{\frac{\frac{\vdash \alpha^i = \alpha^j}{\vdash \tau_{11} = \tau_{21}} \quad \frac{\vdash B^i = B^j}{\vdash \tau_{12} = \tau_{22}}}{\vdash (\forall \alpha. \tau_{11} \rightarrow \tau_{12})^i = (\forall \alpha. \tau_{21} \rightarrow \tau_{22})^j} \quad \vdash \tau_1 = \tau_2}{\vdash \text{box}(\tau_1)^i = \text{box}(\tau_2)^j} \\
\boxed{\vdash \rho : \Gamma} \\
\frac{\frac{\frac{\emptyset \vdash \tau_1 \text{ wf} \quad \dots \quad \emptyset \vdash \tau_n \text{ wf}}{\emptyset; \emptyset \vdash v_1^{i_1} : \tau'_1 \quad \dots \quad \emptyset; \emptyset \vdash v_n^{i_n} : \tau'_n} \quad \vdash \tau_1 = \tau'_1 \quad \dots \quad \vdash \tau_n = \tau'_n}{\vdash (x_1:\tau_1 = v_1^{i_1}, \dots, x_n:\tau_n = v_n^{i_n}) : (x_1:\tau_1, \dots, x_n:\tau_n)}} \\
\boxed{\vdash M : \tau} \\
\frac{\vdash \rho : \Gamma \quad \emptyset; \Gamma \vdash e : \tau}{\vdash (\rho, e) : \tau}
\end{array}$$

Figure 4. Type rules, other constructs

$\vdash e''' \text{ tr}$. So applying the rules, we derive that $\vdash \rho'(e''')$ and then $\vdash e' \text{ tr}$, as required.

- In this case, $e = \rho'(v^i)^j$ for some ρ' , v , i , and j , and $e' = v^i$. The hypothesis, $\vdash e \text{ tr}$ can only be derived in one way and unpacking that we see that $\vdash v^i \text{ tr}$, which is what we are required to prove. ■

There is no corresponding progress property for our notion of traceability, since in the absence of further guarantees, programs can go wrong. However, well-typed programs are both traceable and do not go wrong as we will see in the next section, and so preserving typability ensures GC correctness.

2.5 Typing

The typing rules appear in Figures 4 and 5. They are for the most part standard except for two modifications. First, as types are labelled, we must sometimes ignore the labels in typing. Judgement $\vdash \tau_1 = \tau_2$ states that types τ_1 and τ_2 are syntactically equivalent except that the labels on their sub-terms might differ. This is important in (for example) the rule for application, where we require only that the parameter type τ_1 and the actual argument type τ_2 satisfy $\vdash \tau_1 = \tau_2$ rather than $\tau_1 = \tau_2$; similarly in the rule for environments. Second, the instantiation rule for polymorphic functions enforces the property that the type argument have traceability \mathbf{r} , justifying a type variable always having traceability \mathbf{r} and allowing the compiler to compute GC meta-data for all types.

One particularly important aspect of our language is that we assume a type-erasure semantics. For this interpretation to be cor-

$$\begin{array}{c}
\boxed{\Delta; \Gamma \vdash e : \tau} \\
\frac{x:\tau \in \Gamma}{\Delta; \Gamma \vdash x^i : \tau} \\
\frac{\frac{\Delta \vdash (\forall \alpha. \tau_1 \rightarrow \tau_2)^i \text{ wf}}{\Delta, \alpha; \Gamma, f:(\forall \alpha. \tau_1 \rightarrow \tau_2)^i, x:\tau_1 \vdash e : \tau_2}}{\Delta; \Gamma \vdash (\mathbf{fix} f[\alpha](x:\tau_1):\tau_2.e)^i : (\forall \alpha. \tau_1 \rightarrow \tau_2)^i} \\
\frac{\frac{\Delta; \Gamma \vdash e_1 : (\forall \alpha. \tau_1 \rightarrow \tau')^j \quad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta \vdash \tau \text{ wf} \quad tr(\tau) = \mathbf{r} \quad \vdash \tau_1[\tau/\alpha] = \tau_2}}{\Delta; \Gamma \vdash (e_1[\tau] e_2)^i : \tau'[\tau/\alpha]} \\
\frac{\Delta \vdash \tau \text{ wf} \quad \Delta; \Gamma \vdash e : \tau' \quad \vdash \tau = \tau'}{\Delta; \Gamma \vdash (\mathbf{box}_\tau e)^i : \mathbf{box}(\tau)^i} \\
\frac{\Delta; \Gamma \vdash e : \mathbf{box}(\tau)^j}{\Delta; \Gamma \vdash (\mathbf{unbox} e)^i : \tau} \\
\frac{\vdash \rho : \Gamma' \quad \emptyset; \Gamma' \vdash e : \tau}{\Delta; \Gamma \vdash \rho(e)^i : \tau} \\
\frac{\Delta; \Gamma \vdash c^i : B^i}{\vdash \rho : \Gamma' \quad \emptyset \vdash (\forall \alpha. \tau_1 \rightarrow \tau_2)^i \text{ wf}} \\
\frac{\alpha; \Gamma', f:(\forall \alpha. \tau_1 \rightarrow \tau_2)^i, x:\tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \langle \rho, \mathbf{fix} f[\alpha](x:\tau_1):\tau_2.e \rangle^i : (\forall \alpha. \tau_1 \rightarrow \tau_2)^i} \\
\frac{\emptyset \vdash \tau \text{ wf} \quad \Delta; \Gamma \vdash v^j : \tau' \quad \vdash \tau = \tau'}{\Delta; \Gamma \vdash \langle v^j : \tau \rangle^i : \mathbf{box}(\tau)^i}
\end{array}$$

Figure 5. Type rules, expressions

rect, we must show that we can compute the correct GC meta-data when erasing types. The operational semantics have the application of a polymorphic function step to a frame where the annotation on the function's parameter is a substituted type. We need that the GC meta-data for this substituted type equal the GC meta-data for the unsubstituted parameter type of the function. The requirement $tr(\tau) = \mathbf{r}$ in the typing rule for application is crucial to that equality, and the following lemma proves it.

Lemma 2

If $tr(\tau) = \mathbf{r}$ then $tr(\tau') = tr(\tau'[\tau/\alpha])$.

Proof: The proof is by inspection of the definitions. ■

We can prove type safety for this language in the standard way, via progress and preservations lemmas. First we need several lemmas: that type equality is an equivalence relation, that equal types have the same traceabilities, that a well-typed value has the same traceability as its type, that type equality respects type substitution, that value typing is independent of the typing context, and a type substitution lemma.

Lemma 3

Type equality is an equivalence relation, that is, $\vdash \tau = \tau$, $\vdash \tau_1 = \tau_2$ implies $\vdash \tau_2 = \tau_1$, and $\vdash \tau_1 = \tau_2$ and $\vdash \tau_2 = \tau_3$ implies $\vdash \tau_1 = \tau_3$.

Proof: The proof is by a simple induction on the structure of τ for reflexivity or the structure of the derivation(s) for symmetry and transitivity and inspection of the rules. ■

Lemma 4

If $\vdash \tau_1 = \tau_2$ then $tr(\tau_1) = tr(\tau_2)$.

Proof: The proof is by inspection of the last rule used. ■

Lemma 5

If $\vdash \tau_1 = \tau_2$ then $\vdash \tau_1[\tau/\alpha] = \tau_2[\tau/\alpha]$.

Proof: The proof is by an easy induction on the derivation of $\vdash \tau_1 = \tau_2$. ■

Lemma 6

If $\Delta; \Gamma \vdash v^i : \tau$ then $tr(\tau) = t$.

Proof: The proof is by inspection of the three rules for value typing. ■

Lemma 7

If $\Delta; \Gamma \vdash v^i : \tau$ then $\Delta'; \Gamma' \vdash v^i : \tau$ for any Δ' and Γ' .

Proof: The proof is by any easy induction on the typing derivation and inspection of the three rules for value typing. ■

Lemma 8

If $\Delta, \alpha, \Delta'; \Gamma \vdash e : \tau$, $\Delta \vdash \tau' \text{ wf}$, and $tr(\tau') = \mathbf{r}$ then $\Delta, \Delta'; \Gamma[\tau'/\alpha] \vdash e[\tau'/\alpha] : \tau[\tau'/\alpha]$.

Proof: The proof is a straight forward induction over the derivation of $\Delta, \alpha, \Delta'; \Gamma \vdash e : \tau$. It uses Lemma 2 in the case of the rule for application. ■

With all these lemmas we can prove Type Preservation and Progress.

Lemma 9 (Type Preservation)

If $\vdash M_1 : \tau_1$ and $M_1 \mapsto M_2$ then $\vdash M_2 : \tau_2$ and $\vdash \tau_1 = \tau_2$ for some τ_2 .

Proof: Assume that $\vdash (\rho, e_1) : \tau_1$ and $(\rho, e_1) \mapsto (\rho, e_2)$. We will show by induction on the derivation of the latter that $\vdash (\rho, e_2) : \tau_2$ and $\vdash \tau_1 = \tau_2$ for some τ_2 . By the typing rules, $\vdash \rho : \Gamma$ and $\emptyset; \Gamma \vdash e_1 : \tau_1$ for some Γ . By the typing rules, we just need to show that $\emptyset; \Gamma \vdash e_2 : \tau_2$ and $\vdash \tau_1 = \tau_2$ for some τ_2 . Consider the cases, in the same rule as the figure, for the last rule used to derive the reduction:

- (Variable) In this case, $e_1 = x^i$, $e_2 = v^j$, and $x:\tau = v^j \in \rho$ for some x, i, v, j , and τ' . The typing judgement can only be derived with one rule and it requires that $x:\tau \in \Gamma$. The typing judgement (for ρ) can only be derived in one way and it requires that $\tau = \tau'$, $\emptyset; \emptyset \vdash v^j : \tau''$, and $\vdash \tau = \tau''$. Thus the desired τ_2 is τ'' . We just need to show that $\emptyset; \Gamma \vdash v^j : \tau_2$, which follows by Lemma 7.
- (Fix expression) In this case, $e_1 = (\text{fix } f[\alpha](x:\tau'_1):\tau'_2.e')^i$ and $e_2 = \langle \rho, \text{fix } f[\alpha](x:\tau'_1):\tau'_2.e' \rangle^i$. The typing judgement can only be derived with one rule and it requires that $\emptyset \vdash \tau_1 \text{ wf}$, $\alpha; \Gamma, f:\tau_1, x:\tau'_1 \vdash e' : \tau'_2$ and $\tau_1 = (\forall \alpha. \tau'_1 \rightarrow \tau'_2)^i$. Thus by the typing rules, $\emptyset; \Gamma \vdash e_2 : \tau_1$. By Lemma 3, $\vdash \tau_1 = \tau_1$, so the result follows by setting $\tau_2 = \tau_1$.

- (Box expression) In this case, $e_1 = (\text{box}_\tau v^i)^j$ and $e_2 = \langle v^i:\tau \rangle^j$ for some τ, v, i , and j . The typing judgement can only be derived with one rule and it requires that $\emptyset \vdash \tau \text{ wf}$, $\emptyset; \Gamma \vdash v^i : \tau'$, $\vdash \tau = \tau'$, and $\tau_1 = \text{box}(\tau)^j$. By the typing rules, $\emptyset; \Gamma \vdash e_2 : \tau_1$. By Lemma 3, $\vdash \tau_1 = \tau_1$, so the result follows by setting $\tau_2 = \tau_1$.

- (Application function) In this case, $e_1 = (e_3[\tau] e_4)^i$, $e_2 = (e_5[\tau] e_4)^i$, and $(\rho, e_3) \mapsto (\rho, e_5)$ for some e_3, τ, e_4 , and e_5 . The typing judgement can only be derived with one rule and it requires that $\emptyset; \Gamma \vdash e_3 : (\forall \alpha. \tau_3 \rightarrow \tau')^j$, $\emptyset; \Gamma \vdash e_4 : \tau_4$, $\emptyset \vdash \tau \text{ wf}$, $tr(\tau) = \mathbf{r}$, $\vdash \tau_3[\tau/\alpha] = \tau_4$, and $\tau = \tau'[\tau/\alpha]$ for some τ_3, τ', j , and τ_4 . By the induction hypothesis, $\emptyset; \Gamma \vdash e_5 : \tau_5$ and $\vdash (\forall \alpha. \tau_3 \rightarrow \tau')^j = \tau_5$ for some τ_5 . There is only one rule to derive the latter and it requires that $\tau_5 = (\forall \alpha. \tau_{51} \rightarrow \tau_{52})^k$, $\vdash \tau_3 = \tau_{51}$, and $\vdash \tau' = \tau_{52}$ for some τ_{51}, τ_{52} , and k . By Lemma 5, $\vdash \tau_3[\tau/\alpha] = \tau_{51}[\tau/\alpha]$ and $\vdash \tau'[\tau/\alpha] = \tau_{52}[\tau/\alpha]$. By Lemma 3, $\vdash \tau_{51}[\tau/\alpha] = \tau_4$. So by the typing rules, $\emptyset; \Gamma \vdash e_2 : \tau_{52}[\tau/\alpha]$. The result follows by setting $\tau_2 = \tau_{52}[\tau/\alpha]$.

- (Application argument) In this case, $e_1 = (e_3[\tau] e_4)^i$, $e_2 = (e_5[\tau] e_5)^i$, and $(\rho, e_3) \mapsto (\rho, e_5)$ for some e_3, e_4 , and e_5 . The typing judgement can only be derived with one rule and it requires that $\emptyset; \Gamma \vdash e_3 : (\forall \alpha. \tau_3 \rightarrow \tau')^j$, $\emptyset; \Gamma \vdash e_4 : \tau_4$, $\emptyset \vdash \tau \text{ wf}$, $tr(\tau) = \mathbf{r}$, $\vdash \tau_3[\tau/\alpha] = \tau_4$, and $\tau_1 = \tau'[\tau/\alpha]$ for some τ_3, τ', j , and τ_4 . By the induction hypothesis, $\emptyset; \Gamma \vdash e_5 : \tau_5$ and $\vdash \tau_4 = \tau_5$. By Lemma 3, $\vdash \tau_3[\tau/\alpha] = \tau_5$. So by the typing rules, $\emptyset; \Gamma \vdash e_2 : \tau_1$. By Lemma 3, $\vdash \tau_1 = \tau_1$, so the result follows by setting $\tau_2 = \tau_1$.

- (Application beta) In this case:

$$\begin{aligned} e_1 &= (v_1^i[\tau] v_2^j)^k \\ v_1 &= \langle \rho', \text{fix } f[\alpha](x:\tau'_1):\tau'_2.e' \rangle \\ e_2 &= \rho''(e'[\tau/\alpha])^k \\ \rho'' &= \rho', f:\tau' = v_1^i, x:\tau'_1[\tau/\alpha] = v_2^j \\ \tau' &= (\forall \alpha. \tau'_1 \rightarrow \tau'_2)^i \end{aligned}$$

for some $\rho', f, \alpha, x, \tau'_1, \tau'_2, e', i, v_2, j$, and k . Unpacking the typing judgement, which can only be derived in one way, $\emptyset; \Gamma \vdash v_1^i : \tau' (1)$, $\vdash \rho' : \Gamma' (2)$, $\emptyset \vdash \tau' \text{ wf} (12)$, $\alpha; \Gamma', f:\tau', x:\tau'_1 \vdash e' : \tau'_2 (3)$, $\tau_1 = \tau'_2[\tau/\alpha] (4)$, $\emptyset; \Gamma \vdash v_2^j : \tau'' (5)$, $\vdash \tau'_1[\tau/\alpha] = \tau'' (6)$, $\emptyset \vdash \tau \text{ wf}$, and $tr(\tau) = \mathbf{r}$ for some Γ' and τ'' . By (1) and Lemma 7, $\emptyset; \emptyset \vdash v_1^i : \tau' (7)$. By Lemma 3, $\vdash \tau' = \tau' (8)$. By (5) and Lemma 7, $\emptyset; \emptyset \vdash v_2^j : \tau'' (9)$. By (2), (7), (8), (9), and (6), the typing rules give $\vdash \rho'' : \Gamma', f:\tau', x:\tau'_1[\tau/\alpha] (10)$. By (3) and Lemma 8, $\emptyset; (\Gamma', f:\tau', x:\tau'_1[\tau/\alpha]) \vdash e'[\tau/\alpha] : \tau'_2[\tau/\alpha] (11)$. By (2) and (12), by inspection of the typing rules, $\Gamma'[\tau/\alpha] = \Gamma'$ and $\tau'[\tau/\alpha] = \tau'$. Thus, $\emptyset; \Gamma', f:\tau', x:\tau'_1[\tau/\alpha] \vdash e'[\tau/\alpha] : \tau'_2[\tau/\alpha] (13)$. By (10) and (13), the typing rules give $\emptyset; \Gamma \vdash \rho''(e'[\tau/\alpha])^k : \tau'_2[\tau/\alpha] (14)$. By (4), the result follows by setting $\tau_2 = \tau'_2[\tau/\alpha]$.

- (Box argument) In this case, $e_1 = (\text{box}_\tau e)^i$, $e_2 = (\text{box}_\tau e')^i$, and $(\rho, e) \mapsto (\rho, e')$ for some τ, e, i , and e' . The typing judgement can only be derived with one rule and it requires that $\emptyset \vdash \tau \text{ wf}$, $\emptyset; \Gamma \vdash e : \tau'$, $\vdash \tau = \tau'$ and $\tau_1 = \text{box}(\tau)^i$ for some τ' . By the induction hypothesis, $\emptyset; \Gamma \vdash e' : \tau''$ and $\vdash \tau' = \tau''$ for some τ'' . By Lemma 3, $\vdash \tau = \tau''$. By the typing rules, $\emptyset; \Gamma \vdash e_2 : \text{box}(\tau)^i$. By Lemma 3, $\vdash \tau_1 = \tau_1$, so the result follows by setting $\tau_2 = \tau_1$.
- (Unbox argument) In this case, $e_1 = (\text{unbox } e)^i$, $e_2 = (\text{unbox } e')^i$, and $(\rho, e) \mapsto (\rho, e')$ for some e, i , and e' . The typing judgement can only be derived with one rule and it re-

quires that $\emptyset; \Gamma \vdash e : \text{box}(\tau_1)^j$ for some j . By the induction hypothesis, $\emptyset; \Gamma \vdash e' : \tau'$ and $\vdash \text{box}(\tau_1)^j = \tau'$ for some τ' . The latter can only be derived with one rule and it requires that $\tau' = \text{box}(\tau'')^k$ and $\vdash \tau_1 = \tau''$ for some τ'' and k . By the typing rules, $\emptyset; \Gamma \vdash e_2 : \tau''$, so the result follows by setting $\tau_2 = \tau''$.

- (Unbox beta) In this case, $e_1 = (\text{unbox } \langle v^i : \tau \rangle^j)^k$ and $e_2 = v^i$ for some τ, v, i, j , and k . The typing judgement can only be derived with one rule and it requires that $\emptyset; \Gamma \vdash \langle v^i : \tau \rangle^j : \text{box}(\tau_1)^l$ for some l . The latter can only be derived with one rule and it requires $\tau = \tau_1, \emptyset; \Gamma \vdash v^i : \tau'$, and $\vdash \tau = \tau'$. So the result follows by setting $\tau_2 = \tau'$.
- (Frame step) In this case, $e_1 = \rho'(e)^i, e_2 = \rho'(e')^i$, and $(\rho', e) \mapsto (\rho', e')$ for some ρ', e, i , and e' . The typing judgement can only be derived with one rule and it requires that $\rho' \vdash \Gamma' : \text{and } \emptyset; \Gamma' \vdash e : \tau_1$ for some Γ' . By the induction hypothesis, $\emptyset; \Gamma' \vdash e' : \tau'$ and $\vdash \tau_1 = \tau'$ for some τ' . By the typing rules, $\emptyset; \Gamma \vdash e_2 : \tau'$, so the result follows by setting $\tau_2 = \tau'$.
- (Frame return) In this case, $e_1 = \rho'(v^i)^j$ and $e_2 = v^i$ for some ρ', v, i , and j . The typing judgement can only be derived with one rule and it requires that $\rho' \vdash \Gamma' : \text{and } \emptyset; \Gamma' \vdash v^i : \tau_1$. By Lemma 7, $\emptyset; \Gamma \vdash v^i : \tau_1$. By Lemma 3, $\vdash \tau_1 = \tau_1$, so the result follows by setting $\tau_2 = \tau_1$. ■

Lemma 10 (Progress)

If $\vdash M : \tau$ then either M has the form (ρ, v^i) or $M \mapsto M'$ for some M' .

Proof: The result follows from: If $\vdash \rho : \Gamma$ and $\emptyset; \Gamma \vdash e : \tau$ then either e has the form v^i or $(\rho, e) \mapsto (\rho, e')$ for some e' . We will prove this by induction on the typing derivation for e . Consider the last rule, in the same order as the figure, used to derive the judgement:

- (Variable) In this case $e = x^i$ and $x:\tau \in \Gamma$. There is only one rule to derive $\vdash \rho : \Gamma$ and it requires that $x:\tau = v^j \in \rho$ and other conditions for some v and j . Then by the variable rule, $(\rho, e) \mapsto v^j$, as required.
- (Fix expression) In this case $e = (\text{fix } f[\alpha](x:\tau_1):\tau_2.e')^i$. Clearly by the fix expression rule:

$$(\rho, e) \mapsto (\rho, \langle \rho, \text{fix } f[\alpha](x:\tau_1):\tau_2.e' \rangle^i)$$

- (Application) In this case, $e = (e_1[\tau'] e_2)^i$. The typing rule requires that $\emptyset; \Gamma \vdash e_1 : (\forall \alpha. \tau_1 \rightarrow \tau_3)^j$ (1), $\emptyset; \Gamma \vdash e_2 : \tau_2$ (2), and $\vdash \tau_1[\tau'/\alpha] = \tau_2$ (3) for some τ_1, j , and τ_2 . By the induction hypothesis, either e_1 is a value or reduces, and e_2 is a value or reduces. There are three subcases:
 - Case 1, e_1 reduces: In this case there is e'_1 such that $(\rho, e_1) \mapsto (\rho, e'_1)$. Then by the application function rule, $(\rho, e) \mapsto (\rho, (e'_1[\tau'] e_2)^i)$, as required.
 - Case 2, e_1 is a value and e_2 reduces: In this case there is e'_2 such that $(\rho, e_2) \mapsto (\rho, e'_2)$. Then by the application function rule, $(\rho, e) \mapsto (\rho, (e_1[\tau'] e'_2)^i)$, as required.
 - Case 3, $e_1 = v_1^k$ and $e_2 = v_2^l$ for some v_1, k, v_2 , and l : There is only one typing rule to derive (1) and it requires that v_1 have the form $\langle \rho', \text{fix } f[\alpha](x:\tau_1):\tau_3.e' \rangle$ for some ρ', f, x , and e' . Let t be the traceability of v_2 . By Lemma 6 and (2), $tr(\tau_2) = t$. By Lemma 4 and (3), $tr(\tau_1[\tau'/\alpha]) = t$.

Then by the application beta rule:

$$\begin{aligned} (\rho, e) &\mapsto \\ (\rho, (\rho', f:\tau' = v_1^k, x:\tau_1[\tau'/\alpha] = v_2^l)(e'[\tau'/\alpha])^i) & \end{aligned}$$

where $\tau' = (\forall \alpha. \tau_1 \rightarrow \tau_3)^k$, as required.

- (Box expression) In this case, $e = (\text{box}_{\tau'} e')^i$ for some τ', e' , and i . The typing rule requires that $\tau = \text{box}(\tau')^i, \emptyset; \Gamma \vdash e' : \tau''$ (1), and $\vdash \tau' = \tau''$ (2) for some τ'' . By the induction hypothesis, either e' is a value or reduces:
 - If $e' = v_i^j$ then by Lemma 6 and (1), $tr(\tau'') = t$. By (2) and Lemma 4, $tr(\tau') = t$. So by the box reduction rule, $(\rho, e) \mapsto (\rho, \langle v_i^j : \tau' \rangle^i)$, as required.
 - If $(\rho, e') \mapsto (\rho, e'')$ then $(\rho, e) \mapsto (\rho, (\text{box}_{\tau'} e'')^i)$, as required.
- (Unbox) In this case, $e = (\text{unbox } e')^i$ for some e' and i . The typing rule requires that $\emptyset; \Gamma \vdash e' : \text{box}(\tau)^j$ (1) for some j . By the induction hypothesis, e' is a value or reduces:
 - If $e' = v^k$ then (1) can be derived by only one rule and it requires that $v = \langle v^l : \tau' \rangle$ for some v', l , and τ' . By the unbox beta rule, $(\rho, e) \mapsto (\rho, v^l)$, as required.
 - If $(\rho, e') \mapsto (\rho, e'')$ then by the unbox argument rule, $(\rho, e) \mapsto (\rho, (\text{unbox } e'')^i)$, as required.
- (Frame) In this case, $e = \rho'(e')^i$ for some ρ', e' , and i . The typing rule requires that $\vdash \rho' : \Gamma'$ and $\emptyset; \Gamma' \vdash e' : \tau$ for some Γ' . By the induction hypothesis, e' is a value or reduces:
 - If $e' = v^j$ then by the frame return rule, $(\rho, e) \mapsto (\rho, v^j)$, as required.
 - If $(\rho, e') \mapsto (\rho, e'')$ then by the frame step rule, $(\rho, e) \mapsto (\rho, \rho'(e'')^i)$, as required.
- (Constant) In this case $e = c^i$ for some c and i and is clearly a value.
- (Fix value) In this case $e = \langle \rho', \text{fix } f[\alpha](x:\tau_1):\tau_2.e' \rangle^i$ for some $\rho', f, x, \tau_1, \tau_2, e'$ and i and is clearly a value.
- (Box value) In this case $e = \langle v^i : \tau' \rangle^j$ for some τ', v, i , and j and is clearly a value. ■

Finally, we can also prove that typability implies traceability and thus typable programs are GC safe and remain so throughout execution.

Lemma 11

- If $\vdash M : \tau$ then $\vdash M \text{ tr}$.
- If $\vdash \rho : \Gamma$ then $\vdash \rho \text{ tr}$.
- If $\Delta; \Gamma \vdash e : \tau$ then $\vdash e \text{ tr}$.
- If $\Delta; \Gamma \vdash v^i : \tau$ then $\vdash_v v:tr(\tau)$.

Proof: The results are proven simultaneously by induction on the structure of the typing derivation. The cases for the last rule used, in the same order as the figure, are:

- (Variable) In this case clearly $\vdash e \text{ tr}$.
- (Fix expression) In this case $e = (\text{fix } f[\alpha](x:\tau_1):\tau_2.e')^i$ for some $x, \alpha, \tau_1, \tau_2, e'$, and i . Then by the typing rule, $\Delta, \alpha; \Gamma, f:\tau, x:\tau_1 \vdash e' : \tau_2$ is a subderivation. By the induction hypothesis, $\vdash e' \text{ tr}$. So by the rules for traceability, $\vdash e \text{ tr}$, as required.
- (Application) In this case $e = (e_1[\tau'] e_2)^i$ for some e_1, τ', e_2 , and i . By the typing rule, $\Delta; \Gamma \vdash e_1 : \tau_1$ and $\Delta; \Gamma \vdash e_2 : \tau_2$

for some τ_1 and τ_2 . By the induction hypothesis, $\vdash e_1 \text{ tr}$ and $\vdash e_2 \text{ tr}$. By the rules for traceability $\vdash e \text{ tr}$, as required.

- (Box expression) In this case $e = (\text{box}_{\tau'} e')^i$ for some τ' , e' , and i . By the typing rule, $\Delta; \Gamma \vdash e' : \tau''$ for some τ'' . By the induction hypothesis, $\vdash e' \text{ tr}$. By the rules for traceability, $\vdash e \text{ tr}$, as required.
- (Unbox) In this case $e = (\text{unbox } e')^i$ for some e' and i . By the typing rule, $\Delta; \Gamma \vdash e' : \tau'$ for some τ' . By the induction hypothesis, $\vdash e' \text{ tr}$. By the rules for traceability, $\vdash e \text{ tr}$, as required.
- (Frame) In this case, $e = \rho(e')^i$ for some ρ , e' , and i . By the typing rule, $\vdash \rho : \Gamma'$ and $\emptyset; \Gamma' \vdash e' : \tau$ for some Γ' . By the induction hypothesis, $\vdash \rho \text{ tr}$ and $\vdash e' \text{ tr}$. By the rules for traceability, $\vdash e \text{ tr}$, as required.
- (Constant) In this case $e = c^i$. By the typing rule, $\tau = \mathbf{B}^i$ and so clearly $\text{tr}(\tau) = \mathbf{b}$. By the rules for traceability, $\vdash_{\mathbf{v}} c:\mathbf{b}$, proving the fourth result. By the rules for traceability again, $\vdash e \text{ tr}$, proving the third result.
- (Fix value) In this case $e = \langle \rho, \text{fix } f[\alpha](x:\tau_1):\tau_2.e' \rangle^i$ for some $\rho, f, \alpha, x, \tau_1, \tau_2, e'$, and i . By the typing rule, $\vdash \rho : \Gamma'$ and $\alpha; \Gamma', f:\tau, x:\tau_1 \vdash e' : \tau_2$ for some Γ' . Also by the typing rules, τ is a function type, so $\text{tr}(\tau) = \mathbf{r}$. By the induction hypothesis, $\vdash \rho \text{ tr}$ and $\vdash e' \text{ tr}$. By the rules for traceability, $\vdash_{\mathbf{v}} \langle \rho, \text{fix } f[\alpha](x:\tau_1):\tau_2.e' \rangle:\mathbf{r}$, proving the fourth result. By the rules for traceability again, $\vdash e \text{ tr}$, proving the third result.
- (Box value) In this case $e = \langle v^i:\tau' \rangle^j$. By the typing rule, $\Delta; \Gamma \vdash v^i : \tau''$ and $\vdash \tau' = \tau''$ for some τ'' . Also by the typing rule, τ is a box type, so $\text{tr}(\tau) = \mathbf{r}$. By the induction hypothesis, $\vdash_{\mathbf{v}} v:\text{tr}(\tau'')$. By Lemma 4, $\vdash_{\mathbf{v}} v:\text{tr}(\tau')$. By the rules for traceability, $\vdash_{\mathbf{v}} \langle v^i:\tau' \rangle:\mathbf{r}$, proving the third result. By the rules for traceability again, $\vdash e \text{ tr}$, as required.
- (Environment) In this case $\rho = x_1:\tau_1 = v_1^{i_1}, \dots, x_n:\tau_n = v_n^{i_n}$. By the typing rule, $\emptyset; \emptyset \vdash v_j^{i_j} : \tau'_j$ and $\vdash \tau_j = \tau'_j$ for $1 \leq j \leq n$ and some τ'_j s. By the induction hypothesis, $\vdash_{\mathbf{v}} v_j:\text{tr}(\tau'_j)$ for $1 \leq j \leq n$. By Lemma 4, $\text{tr}(\tau_j) = \text{tr}(\tau'_j)$ for $1 \leq j \leq n$. Thus $\vdash_{\mathbf{v}} v_j:\text{tr}(\tau_j)$ for $1 \leq j \leq n$. By the traceability rules, $\vdash \rho \text{ tr}$, as required.
- (Machine state) In this case $M = (\rho, e)$. By the typing rule, $\vdash \rho : \Gamma$ and $\emptyset; \Gamma \vdash e : \tau$ for some Γ . By the induction hypothesis, $\vdash \rho \text{ tr}$ and $\vdash e \text{ tr}$. By the rules for traceability, $\vdash M \text{ tr}$, as required. ■

3. Flow analysis

There is a vast body of literature on interprocedural analysis and optimization, and it is generally fairly straightforward to use these approaches to obtain information about what definitions flow to what use sites. Without committing to a particular approach or implementation, we refer to this body of work broadly as *flow analysis*. Our contribution in this paper lies in showing how to extend a specification of a general flow analysis to the type level, and in showing that any flow analysis so extended can be used to implement an unboxing optimization that preserves type safety and program semantics.

In order to do this, we must provide some framework for describing what information a flow analysis must provide. The core language defined in Section 2 provides labels serving as proxies for the terms, types, and variables on which they occur – the question above can therefore be thought of in terms of finding the set of labels k that reach the position labeled with j .

Following previous work we begin by defining an abstract notion of analysis. We say that an analysis is a pair (C, ϱ) . Binding environments ϱ serve to map variables to the label of their binding sites. The mappings are, as usual, global for the program. Consequently, a given environment may not apply to alpha-variants of a term. We do not require that labels be unique within a program—as usual however, analyses will be more precise if this is the case. Variables are also not required to be unique (since reduction may duplicate terms and hence binding sites). However, duplicate variable bindings in a program must be labeled consistently according to ϱ or else no analysis of the program can be acceptable according to our definition. This can always be avoided by alpha-varying or relabeling appropriately.

A cache C is a mapping from labels to sets of shapes. Shapes are given by the grammar:

$$\text{Shapes: } s ::= c^i \mid (\forall i'.j \rightarrow k)_v^l \mid (\text{box}_t i)_v^j \mid \mathbf{B}^i \mid (\forall i'.j \rightarrow k)_t^l \mid (\text{box } i)_t^j$$

There are two classes of shapes—term shapes and type shapes. The idea behind term shapes is that each shape provides a proxy for a set of terms that might flow to a given location, describing both the shape of the values that might flow there and the labels of the sub-components of those values. For example, for an analysis (C, ϱ) , $c^i \in C(k)$ indicates that (according to the analysis) the constant c , labeled with i , might flow to a location labeled with k . Similarly, if $(\forall i'.i \rightarrow j)_v^k \in C(l)$, then the analysis specifies that among the values flowing to locations labeled with l might be functions labeled with k , whose type parameter is labeled with i' , parameter type is labeled with i , and whose bodies are labeled with j . If $(\text{box}_t k)_v^i \in C(l)$ then among the values that might flow to l (according to the analysis) are boxed values labeled with i , with meta-data t and whose bodies are labeled by some j such that $C(j) \subseteq C(k)$.

Where term shapes provide a proxy for the set of values that might flow to a given location, type shapes provide a proxy for the types of the locations that values might flow through to get to a given location. For example, for an analysis (C, ϱ) , $\mathbf{B}^i \in C(k)$ indicates that (according to the analysis) objects that reach location k might flow through a variable or term of type \mathbf{B} , labeled with i . The function type and box type shapes similarly correspond to the flow of values through variables of function or box types.

It is important to note that the shapes in the cache may not correspond exactly to the terms in the program, since reduction may change program terms (e.g. by instantiating variables with values). However, reduction does not change the outer shape and labeling of values—it is this reduction invariant information that is captured by shapes.

Clearly, not every choice of analysis pairs is meaningful for program optimization. While in general it is reasonable (indeed, unavoidable) for an analysis to overestimate the set of terms associated with a label, it is unacceptable for an analysis to underestimate the set of terms that flow to a label—most optimizations will produce incorrect results, since they are designed around the idea that the analysis is telling them everything that could possibly flow to them. In order to capture the notion of when an analysis pair gives a suitable approximation of the flow of values in a program we follow the general spirit of Nielson et al. [7], and define a notion of an *acceptable analysis*. That is, we give a declarative specification that gives sufficient conditions for specifying when a given analysis does not underestimate the set of terms flowing to a label, without committing to a particular analysis. We arrange the subsequent meta-theory such that our results apply to any analysis that is *acceptable*. In this way, we decouple our optimization from the particulars of how the analysis is computed and the relative precision of the result.

Our acceptable-analysis relation is given in Figures 6 and 7—the judgement $C; \varrho \vdash (\rho, e)$ determines that an analysis pair (C, ϱ) is *acceptable* for a machine state (ρ, e) , and similarly for the environment and expression forms of the judgement. We use the notation $\text{lbl}(e)$ to denote the outermost label of e : that is, i where e is of the form m^i or v^i and similarly for types. The acceptability judgement generally indicates for each syntactic form what the flow of values is. For example, in the application rule, the judgment insists that for every function value that flows to the applicand position, the set of shapes associated with the parameter of that function is a super-set of the set of shapes associated with the argument of the application; and that the set of shapes associated with the result of the function is a sub-set of the set of shapes associated with the application itself.

The judgement $C; \varrho \vdash \tau$ determines that an analysis pair (C, ϱ) is acceptable for a labeled type τ . In particular, if a function flows to a function type $\forall \alpha. \tau_1 \rightarrow \tau_2$ then the set of values that flow to the function's parameter can flow to the argument type τ_1 , and the set of values that can flow from the result of the function can flow to the result type τ_2 . Similarly for a box type, for every box value that might flow to the box type the contents of the box value flows to the contents of the box type.

Given this definition, we can show a correctness result for analyses satisfying this specification by showing that the acceptability relation is preserved under reduction.

Lemma 12 (Cache refinement under reduction)

If $C; \varrho \vdash \rho$, $C; \varrho \vdash e_1$, and $(\rho, e_1) \mapsto (\rho, e_2)$ then $C(\text{lbl}(e_1)) \supseteq C(\text{lbl}(e_2))$.

Proof: The proof is by induction on the derivation of $(\rho, e_1) \mapsto (\rho, e_2)$. Consider the cases for the last rule used to it (the cases are in the same order as in the figure):

- (Variable instantiation.) In this case, $e_1 = x^k$, $e_2 = v^j$, and $x:\tau = v^j \in \rho$. The assumption $C; \varrho \vdash \rho$ requires that $C(j) \subseteq C(\text{lbl}(\tau))$ and $\varrho(x) = \text{lbl}(\tau)$. The assumption $C; \varrho \vdash e_1$ requires that $C(\varrho(x)) \subseteq C(k)$. Thus $C(j) \subseteq C(k)$. Clearly, $\text{lbl}(e_1) = k$ and $\text{lbl}(e_2) = j$ and the result follows.
- (Fix introduction.) In this case, clearly $\text{lbl}(e_1) = \text{lbl}(e_2)$ and the result immediately follows.
- (Box introduction.) In this case, clearly $\text{lbl}(e_1) = \text{lbl}(e_2)$ and the result immediately follows.
- (Application left.) In this case, clearly $\text{lbl}(e_1) = \text{lbl}(e_2)$ and the result immediately follows.
- (Application right.) In this case, clearly $\text{lbl}(e_1) = \text{lbl}(e_2)$ and the result immediately follows.
- (Application beta.) In this case, clearly $\text{lbl}(e_1) = \text{lbl}(e_2)$ and the result immediately follows.
- (Under box.) In this case, clearly $\text{lbl}(e_1) = \text{lbl}(e_2)$ and the result immediately follows.
- (Under unbox.) In this case, clearly $\text{lbl}(e_1) = \text{lbl}(e_2)$ and the result immediately follows.
- (Unbox beta.) In this case, $e_1 = (\text{unbox } \langle v^i:\tau \rangle^j)^k$ and $e_2 = v^i$. The first hypothesis can be derived only by one rule and it requires that $C; \varrho \vdash \langle v^i:\tau \rangle^j$ (1), and $\text{box}(C; j, k)$ (2). Judgement 1 can only be derived by one rule and it requires that $C; \varrho \vdash v^i$ (4), $(\text{box}_{tr(\tau)} i'')^j \in C(j)$ (5) for some i'' , and $C(i) \subseteq C(i'')$ (6). Instantiating Fact 2 with Fact 5 we get that $C(i'') \subseteq C(k)$ (7). Combining Facts 6 and 7, $C(i) \subseteq C(k)$, as we are required to prove.

$$\boxed{\text{fun}(C; i, j, k, l) \quad \text{box}(C; i, j)}$$

$$\begin{aligned} \text{fun}(C; i, j, k, l) = & \wedge \forall (\forall j'. k' \rightarrow l')_v^{i'} \in C(i) : \\ & C(j) = C(j') \wedge C(k) \subseteq C(k') \wedge C(l') \subseteq C(l) \\ & \wedge \forall (\forall j'. k' \rightarrow l')_t^{i'} \in C(i) : \\ & C(j) = C(j') \wedge C(k) \subseteq C(k') \wedge C(l') \subseteq C(l) \\ \text{box}(C; i, j) = & \wedge \forall (\text{box}_t j')_y^{i'} \in C(i) : C(j') \subseteq C(j) \\ & \wedge \forall (\text{box}_t j')_t^{i'} \in C(i) : C(j') \subseteq C(j) \end{aligned}$$

$$\boxed{C; \varrho \vdash e}$$

$$\frac{C(\varrho(x)) \subseteq C(i)}{C; \varrho \vdash x^i}$$

$$\frac{\varrho(f) = i \quad \varrho(x) = \text{lbl}(\tau_1) \quad C; \varrho \vdash (\forall \alpha. \tau_1 \rightarrow \tau_2)^i \quad C; \varrho \vdash e \quad (\forall \varrho(\alpha). \text{lbl}(\tau_1) \rightarrow \text{lbl}(e))_v^i \in C(i)}{C; \varrho \vdash (\text{fix } f[\alpha](x:\tau_1):\tau_2.e)^i}$$

$$\frac{C; \varrho \vdash e_1 \quad C; \varrho \vdash \tau \quad C; \varrho \vdash e_2 \quad \text{fun}(C; \text{lbl}(e_1), \text{lbl}(\tau), \text{lbl}(e_2), i)}{C; \varrho \vdash (e_1[\tau] e_2)^i}$$

$$C; \varrho \vdash (\text{unbox } e)^i$$

$$\frac{C; \varrho \vdash \text{box}(\tau)^i \quad C; \varrho \vdash e \quad (\text{box}_{tr(\tau)} j)_v^i \in C(i) \quad C(\text{lbl}(e)) \subseteq C(j)}{C; \varrho \vdash (\text{box}_\tau e)^i}$$

$$C; \varrho \vdash e \quad \text{box}(C; \text{lbl}(e), i)$$

$$C; \varrho \vdash (\text{unbox } e)^i$$

$$\frac{C; \varrho \vdash \rho \quad C; \varrho \vdash e \quad C(\text{lbl}(e)) \subseteq C(i)}{C; \varrho \vdash \rho(e)^i}$$

$$\frac{C; \varrho \vdash B^i \quad c^i \in C(i)}{C; \varrho \vdash c^i}$$

$$\frac{\varrho(f) = i \quad \varrho(x) = \text{lbl}(\tau_1) \quad C; \varrho \vdash (\forall \alpha. \tau_1 \rightarrow \tau_2)^i \quad C; \varrho \vdash \rho \quad C; \varrho \vdash e \quad (\forall \varrho(\alpha). \text{lbl}(\tau_1) \rightarrow \text{lbl}(e))_v^i \in C(i)}{C; \varrho \vdash \langle \rho, \text{fix } f[\alpha](x:\tau_1):\tau_2.e \rangle^i}$$

$$\frac{C; \varrho \vdash \text{box}(\tau)^i \quad C; \varrho \vdash v^j \quad (\text{box}_{tr(\tau)} k)_v^i \in C(i) \quad C(j) \subseteq C(k)}{C; \varrho \vdash \langle v^j:\tau \rangle^i}$$

Figure 6. Acceptable Analysis, Expressions

- (Under frame.) In this case, clearly $\text{lbl}(e_1) = \text{lbl}(e_2)$ and the result immediately follows.
- (Frame return.) In this case, $e_1 = \rho'(v^i)^j$ and $e_2 = v^i$. The assumption $C; \varrho \vdash e_1$ requires that $C(i) \subseteq C(j)$. Since $\text{lbl}(e_1) = j$ and $\text{lbl}(e_2) = i$, the result is immediate.

$$\boxed{C; \varrho \vdash \tau}$$

$$\frac{C(\varrho(\alpha)) = C(i) \quad \mathbf{B}^j \in C(i)}{C; \varrho \vdash \alpha^i} \quad \frac{\mathbf{B}^j \in C(i)}{C; \varrho \vdash \mathbf{B}^i}$$

$$\frac{C; \varrho \vdash \tau_1 \quad C; \varrho \vdash \tau_2 \quad (\forall \varrho(\alpha). \text{lbl}(\tau_1) \rightarrow \text{lbl}(\tau_2))_{\mathfrak{t}}^k \in C(i) \quad \text{fun}(C; i, \varrho(\alpha), \text{lbl}(\tau_1), \text{lbl}(\tau_2))}{C; \varrho \vdash (\forall \alpha. \tau_1 \rightarrow \tau_2)^i}$$

$$\frac{C; \varrho \vdash \tau \quad (\text{box } \text{lbl}(\tau))_{\mathfrak{t}}^k \in C(i) \quad \text{box}(C; i, \text{lbl}(\tau))}{C; \varrho \vdash \text{box}(\tau)^i}$$

$$\boxed{C; \varrho \vdash \Gamma}$$

$$\frac{\forall 1 \leq j \leq n : \varrho(x_j) = \text{lbl}(\tau_j) \wedge C; \varrho \vdash \tau_j}{C; \varrho \vdash x_1:\tau_1, \dots, x_n:\tau_n}$$

$$\boxed{C; \varrho \vdash \rho}$$

$$\frac{C; \varrho \vdash x_1:\tau_1, \dots, x_n:\tau_n \quad \forall 1 \leq j \leq n : C(i_j) \subseteq C(\text{lbl}(\tau_j)) \wedge C; \varrho \vdash v_k^{i_k}}{C; \varrho \vdash x_1:\tau_1 = v_1^{i_1}, \dots, x_n:\tau_n = v_n^{i_n}}$$

$$\boxed{C; \varrho \vdash M}$$

$$\frac{C; \varrho \vdash \rho \quad C; \varrho \vdash e}{C; \varrho \vdash (\rho, e)}$$

Figure 7. Acceptable Analysis, Other Constructs

Next we show a type substitution lemma for acceptability.

Lemma 13

If $C; \varrho \vdash \tau$ and $C(\text{lbl}(\tau)) = C(\varrho(\alpha))$ then:

- If $C; \varrho \vdash \tau'$ then $C; \varrho \vdash \tau'[\tau/\alpha]$.
- If $C; \varrho \vdash \Gamma$ then $C; \varrho \vdash \Gamma[\tau/\alpha]$.
- If $C; \varrho \vdash e$ then $C; \varrho \vdash e[\tau/\alpha]$.
- If $C; \varrho \vdash \rho$ then $C; \varrho \vdash \rho[\tau/\alpha]$.

Proof: The proof is by induction on the derivation of the $C; \varrho \vdash \tau'$ and $C; \varrho \vdash e$. Consider the cases for the rules used to derive it (in the same order as in the figures):

- The cases for expressions are straight forward.
- (Type variable) In this case $\tau' = \beta^i$. If $\beta \neq \alpha$ then $\tau'[\tau/\alpha] = \tau'$ and the result is immediate. Otherwise, by the rules for acceptability, $C(\varrho(\alpha)) = C(i)$. If $\tau = \sigma^j$ then $\tau'[\tau/\alpha] = \sigma^i$. Consider the cases for σ :
 - Subcase 1, $\sigma = \alpha'$: Then by the rules for acceptability, $C(\varrho(\alpha')) = C(j)$. Since $C(j) = C(\varrho(\alpha)) = C(i)$, $C(\varrho(\alpha')) = C(i)$, and thus $C; \varrho \vdash \alpha'^i$, as required.
 - Subcase 2, $\sigma = \forall \alpha'. \tau_1 \rightarrow \tau_2$: Since $C; \varrho \vdash \tau$, the rules require:

$$\begin{aligned}
C; \varrho \vdash \tau_1 & \quad (1) \\
C; \varrho \vdash \tau_2 & \quad (2) \\
(\forall \varrho(\alpha'). \text{lbl}(\tau_1) \rightarrow \text{lbl}(\tau_2))_{\mathfrak{t}}^k \in C(j) & \quad (3) \\
\text{fun}(C; j, \varrho(\alpha'), \text{lbl}(\tau_1), \text{lbl}(\tau_2)) & \quad (4)
\end{aligned}$$

By (3) and $C(j) = C(i)$:

$$(\forall \varrho(\alpha'). \text{lbl}(\tau_1) \rightarrow \text{lbl}(\tau_2))_{\mathfrak{t}}^k \in C(i) \quad (5)$$

By (4) and $C(j) = C(i)$:

$$\text{fun}(C; i, \varrho(\alpha'), \text{lbl}(\tau_1), \text{lbl}(\tau_2)) \quad (6)$$

By (1), (2), (5), and (6), by the rules for acceptability, $C; \varrho \vdash \sigma^i$.

- Subcase 3, $\sigma = \text{box}(\tau'')$: Since $C; \varrho \vdash \tau$, the rules require:

$$\begin{aligned}
C; \varrho \vdash \tau'' & \quad (1) \\
(\text{box } \text{lbl}(\tau''))_{\mathfrak{t}}^k \in C(j) & \quad (2) \\
\text{box}(C; j, \text{lbl}(\tau'')) & \quad (3)
\end{aligned}$$

By (2) and $C(j) = C(i)$, $(\text{box } \text{lbl}(\tau''))_{\mathfrak{t}}^k \in C(i)$ (4). By (3) and $C(j) = C(i)$, $\text{box}(C; i, \text{lbl}(\tau''))$ (5). By (1), (4), and (5), by the rules for acceptability, $C; \varrho \vdash \sigma^i$, as required.

- (Base type) In this case $\tau'[\tau/\alpha] = \tau$ and the result is immediate.
- (Function type) In this case $\tau' = (\forall \alpha'. \tau_1 \rightarrow \tau_2)^i$. The rules for acceptability require:

$$\begin{aligned}
C; \varrho \vdash \tau_1 & \quad (1) \\
C; \varrho \vdash \tau_2 & \quad (2) \\
(\forall \varrho(\alpha'). \text{lbl}(\tau_1) \rightarrow \text{lbl}(\tau_2))_{\mathfrak{t}}^k \in C(i) & \quad (3) \\
\text{fun}(C; i, \varrho(\alpha'), \text{lbl}(\tau_1), \text{lbl}(\tau_2)) & \quad (4)
\end{aligned}$$

By (1), (2), and the induction hypothesis:

$$\begin{aligned}
C; \varrho \vdash \tau_1[\tau/\alpha] & \quad (5) \\
C; \varrho \vdash \tau_2[\tau/\alpha] & \quad (6)
\end{aligned}$$

Since $\text{lbl}(\tau_1[\tau/\alpha]) = \text{lbl}(\tau_1)$ and $\text{lbl}(\tau_2[\tau/\alpha]) = \text{lbl}(\tau_2)$:

$$\begin{aligned}
(\forall \varrho(\alpha'). \text{lbl}(\tau_1[\tau/\alpha]) \rightarrow \text{lbl}(\tau_2[\tau/\alpha]))_{\mathfrak{t}}^k \in C(i) & \quad (7) \\
\text{fun}(C; i, \varrho(\alpha'), \text{lbl}(\tau_1[\tau/\alpha]), \text{lbl}(\tau_2[\tau/\alpha])) & \quad (8)
\end{aligned}$$

Since $\text{lbl}(\tau'[\tau/\alpha]) = i$, by (5), (6), (7), and (8), $C; \varrho \vdash \tau'[\tau/\alpha]$, as required.

- (Box type) In this case, $\tau' = \text{box}(\tau'')$. The rules for acceptability require:

$$\begin{aligned}
C; \varrho \vdash \tau'' & \quad (1) \\
(\text{box } \text{lbl}(\tau''))_{\mathfrak{t}}^k \in C(i) & \quad (2) \\
\text{box}(C; i, \text{lbl}(\tau'')) & \quad (3)
\end{aligned}$$

By (1) and the induction hypothesis:

$$C; \varrho \vdash \tau''[\tau/\alpha] \quad (4)$$

Since $\text{lbl}(\tau''[\tau/\alpha]) = \text{lbl}(\tau'')$:

$$\begin{aligned}
(\text{box } \text{lbl}(\tau''[\tau/\alpha]))_{\mathfrak{t}}^k \in C(i) & \quad (5) \\
\text{box}(C; i, \text{lbl}(\tau''[\tau/\alpha])) & \quad (6)
\end{aligned}$$

Since $\text{lbl}(\tau'[\tau/\alpha]) = i$, by (4), (5), and (6), $C; \varrho \vdash \tau'[\tau/\alpha]$, as required.

- The cases for type and value environments are straight forward. ■

With these lemmas we can prove that reduction preserves acceptability of the flow analysis.

Lemma 14 (Preservation of acceptability under reduction)

If $C; \varrho \vdash M$ and $M \mapsto M'$ then $C; \varrho \vdash M'$.

Proof: If $C; \varrho \vdash (\rho, e)$ then $C; \varrho \vdash \rho$ and $C; \varrho \vdash e$. If $(\rho, e) \mapsto (\rho, e')$ then the result follows if we show that $C; \varrho \vdash e'$. The proof of the latter is by induction on the derivation of $(\rho, e) \mapsto (\rho, e')$.

Consider the cases for the last rule used to derive it (the cases are in the same order as in the figure):

- In this case $e = x^k$, $e' = v^j$, and $x:\tau = v^j \in \rho$. The assumption $C; \varrho \vdash \rho$ requires that $C; \varrho \vdash v^j$, which is what we need to prove.
- In this case $e = (\text{fix } f[\alpha](x:\tau_1):\tau_2.e'')^j$ for some $f, \alpha, x, \tau_1, \tau_2, e''$, and j , and $e' = \langle \rho, \text{fix } f[\alpha](x:\tau_1):\tau_2.e'' \rangle^j$. Let $i = \text{lbl}(\tau_1)$. The first hypothesis can only be derived by one rule and it requires that $\varrho(f) = j$, $\varrho(x) = i$, $C; \varrho \vdash \tau$ where $\tau = (\forall \alpha. \tau_1 \rightarrow \tau_2)^j$, $C; \varrho \vdash e''$, and $(\forall \varrho(\alpha). i \rightarrow \text{lbl}(e''))_v^j \in C(j)$. Then, and noting $C; \varrho \vdash \rho$ by assumption, by the rules for acceptable analysis, $C; \varrho \vdash \langle \rho, \text{fix } f[\alpha](x:\tau_1):\tau_2.e'' \rangle^j$, as we are required to prove.
- In this case $e = (\text{box}_\tau v^i)^j$ for some τ, v, i , and j , and $e' = \langle v^i:\tau \rangle^j$. The first hypothesis can only be derived by one rule and it requires that $C; \varrho \vdash \text{box}(\tau)^j$, $C; \varrho \vdash v^i$, $(\text{box}_{tr(\tau)} k)_v^j \in C(j)$ for some k , and $C(i) \subseteq C(k)$. Then by the rules for acceptable analysis, $C; \varrho \vdash \langle v^i:\tau \rangle^j$, as we are required to prove.
- In this case $e = (e_1[\tau] e_2)^i$ for some e_1, τ, e_2 , and i , $e' = (e'_1[\tau] e'_2)^i$, and $(\rho, e_1) \mapsto (\rho, e'_1)$ is a subderivation. The first hypothesis can only be derived by one rule and it requires that $C; \varrho \vdash e_1$ (1), $C; \varrho \vdash \tau$ (7), $C; \varrho \vdash e_2$ (2), and $\text{fun}(C; \text{lbl}(e_1), \text{lbl}(\tau), \text{lbl}(e_2), i)$ (3). By the induction hypothesis and Judgement 1, $C; \varrho \vdash e'_1$ (4). By Lemma 12, $C(\text{lbl}(e'_1)) \subseteq C(\text{lbl}(e_1))$ (5). Combining Facts 3 and 5, $\text{fun}(C; \text{lbl}(e'_1), \text{lbl}(\tau), \text{lbl}(e_2), i)$ (6). Combining Facts 4, 7, 2, and 6, and using the rules for acceptable analysis, we see that $C; \varrho \vdash (e'_1[\tau] e'_2)^i$, as we are required to prove.
- In this case $e = (v^j[\tau] e_2)^i$ for some v, j, τ, e_2 , and i , $e' = (v^j[\tau] e'_2)^i$, and $(\rho, e_2) \mapsto (\rho, e'_2)$ is a subderivation. The first hypothesis can only be derived by one rule and it requires that $C; \varrho \vdash v^j$ (1), $C; \varrho \vdash \tau$ (7), $C; \varrho \vdash e_2$ (2), and $\text{fun}(C; j, \text{lbl}(\tau), \text{lbl}(e_2), i)$ (3). By the induction hypothesis and Judgement 1, $C; \varrho \vdash e'_2$ (4). By Lemma 12, $C(\text{lbl}(e'_2)) \subseteq C(\text{lbl}(e_2))$ (5). Combining Facts 3 and 5, $\text{fun}(C; j, \text{lbl}(\tau), \text{lbl}(e'_2), i)$ (6). Combining Facts 1, 7, 4, and 6, and using the rules for acceptable analysis, we see that $C; \varrho \vdash (v^j[\tau] e'_2)^i$, as we are required to prove.
- In this case:

$$\begin{aligned} e &= (v_1^j[\tau] v_2^k)^l \\ v_1 &= \langle \rho', \text{fix } f[\alpha](x:\tau_1):\tau_2.e'' \rangle \\ e' &= \rho''(e''[\tau/\alpha])^l \\ \rho'' &= \rho', f:\tau' = v_1^j, x:\tau_1[\tau/\alpha] = v_2^k \\ \tau' &= (\forall \alpha. \tau_1 \rightarrow \tau_2)^j \end{aligned}$$

for some $\rho', f, \alpha, x, \tau_1, \tau_2, e'', j, \tau, v_2, k$, and l . The first hypothesis can only be derived by one rule and it requires that $C; \varrho \vdash v_1^j$ (1), $C; \varrho \vdash \tau$ (2), $C; \varrho \vdash v_2^k$ (3), and $\text{fun}(C; j, \text{lbl}(\tau), k, l)$ (4). Let $i = \text{lbl}(\tau_1)$. Judgement 1 can only be derived by one rule and it requires that $\varrho(f) = j$ (5), $\varrho(x) = i$ (6), $C; \varrho \vdash \tau'$ (7), $C; \varrho \vdash e''$ (8), and $(\forall i. \varrho(\alpha) \rightarrow \text{lbl}(e''))_v^j \in C(j)$ (9). Instantiating Fact 4 with Fact 9, $C(\text{lbl}(\tau)) = C(\varrho(\alpha))$ (10), $C(k) \subseteq C(i)$ (11), and $C(\text{lbl}(e'')) \subseteq C(l)$ (12). Judgement 7 requires that $C; \varrho \vdash \tau_1$ (13). By (13), (2), and (10), by Lemma 13, $C; \varrho \vdash \tau_1[\tau/\alpha]$ (14). Since $C; \varrho \vdash \rho$, (5), (7), $C(j) \subseteq C(j)$, (1), $\text{lbl}(\tau_1[\tau/\alpha]) = \text{lbl}(\tau_1) = i$ and (6), (14), (11), and (3), we can derive $C; \varrho \vdash \rho''$ (15). By (13), (2), and (10), by Lemma 13, $C; \varrho \vdash e''[\tau/\alpha]$

(16). By (15), (16), and $\text{lbl}(e''[\tau/\alpha]) = \text{lbl}(e'')$ and (12), we can derive $C; \varrho \vdash e'$, as required.

- In this case $e = (\text{box}_\tau e_1)^i$ for some t, e_1 , and i , $e' = (\text{box}_\tau e_2)^i$ for some e_2 , and $(\rho, e_1) \mapsto (\rho, e_2)$ is a subderivation. The first hypothesis can only be derived by one rule and it requires that $C; \varrho \vdash \text{box}(\tau)^i$ (7), $C; \varrho \vdash e_1$ (1), $(\text{box}_{tr(\tau)} j)_v^i \in C(i)$ (2) for some j , and $C(\text{lbl}(e_1)) \subseteq C(j)$ (3). By the induction hypothesis and Judgement 1, $C; \varrho \vdash e_2$ (4). By Lemma 12, $C(\text{lbl}(e_2)) \subseteq C(\text{lbl}(e_1))$ (5). Combining Facts 3 and 5 gives $C(\text{lbl}(e_2)) \subseteq C(j)$ (6). Then by Facts 7, 4, 2, and 6, and using the rules for acceptable analysis, $C; \varrho \vdash (\text{box}_\tau e_2)^i$, as we are required to prove.
- In this case $e = (\text{unbox } e_1)^i$ for some e_1 and i , $e' = (\text{unbox } e_2)^i$ for some e_2 , and $(\rho, e_1) \mapsto (\rho, e_2)$ is a subderivation. The first hypothesis can only be derived by one rule and it requires that $C; \varrho \vdash e_1$ (1) and $\text{box}(C; \text{lbl}(e_1), i)$ (2). By Judgement (1) and the induction hypothesis, $C; \varrho \vdash e_2$ (3). By Lemma 12, $C(\text{lbl}(e_2)) \subseteq C(\text{lbl}(e_1))$ (4). Combining Facts 4 and 2, $\text{box}(C; \text{lbl}(e_2), i)$ (5). Combining Facts 3 and 5, by the rules for acceptable analysis, $C; \varrho \vdash (\text{unbox } e_2)^i$, as we are required to prove.
- In this case $e = (\text{unbox } \langle v^i:\tau \rangle^j)^k$ for some τ, v, i, j , and k , and $e' = v^i$. The first hypothesis can only be derived by one rule and it requires that $C; \varrho \vdash \langle v^i:\tau \rangle^j$, which in turn can only be derived by one rule that requires that $C; \varrho \vdash v^i$, as we are required to prove.
- In this case $e = \rho'(e'')^i$, $e' = \rho'(e''')^i$, and $(\rho', e'') \mapsto (\rho', e''')$ is a subderivation. Assumption $C; \varrho \vdash e$ requires that $C; \varrho \vdash \rho'$ (1), $C; \varrho \vdash e''$ (2), and $C(\text{lbl}(e'')) \subseteq C(i)$ (3). By (1), (2), and the induction hypothesis, $C; \varrho \vdash e'''$ (4). By Lemma 12, $C(\text{lbl}(e''')) \subseteq C(\text{lbl}(e''))$ (5). Combining (3) and (5), $C(\text{lbl}(e''')) \subseteq C(i)$ (6). Using (1), (4), and (6) we derive $C; \varrho \vdash \rho'(e''')^i$, as required.
- In this case $e = \rho'(v^i)^j$ and $e' = v^i$. The assumption $C; \varrho \vdash e$ unpacks to requiring that $C; \varrho \vdash v^i$, as required. ■

Lemma 15 (Many-step reduction preserves acceptability)

If $C; \varrho \vdash M$ and $M \mapsto^* M'$ then $C; \varrho \vdash M'$.

Proof: By induction on reduction sequences and Lemma 14. ■

The key implication of this result is that any acceptable analysis correctly approximates the set of values to which a term might evaluate.

Theorem 1 (Correctness)

If $C; \varrho \vdash (\rho, e)$ and $(\rho, e) \mapsto^* (\rho, v^i)$, then $\exists s : \text{lbl}(s) = i \wedge s \in C(\text{lbl}(e))$.

Proof: By Lemma 15, $C; \varrho \vdash (\rho, v^i)$. By inspection of the acceptability rules, there exists a shape s such that $\text{lbl}(s) = i$ and $s \in C(i)$ (that is, the rules for value forms always require that an appropriate shape with the same label as the value appear in the cache for the value). By Lemma 12, $C(\text{lbl}(e)) \supseteq C(i)$, and so $s \in C(i)$. ■

We can also show an important connection between typing and acceptable flow analysis—namely that the cache of an expression's type is a contained in the cache of that expression.

Lemma 16

If $\Delta; \Gamma \vdash e : \tau$, $C; \varrho \vdash \Gamma$, and $C; \varrho \vdash e$ then $C(\text{lbl}(\tau)) \subseteq C(\text{lbl}(e))$ and $C; \varrho \vdash \tau$.

Proof: The proof is by induction on the derivation of $\Gamma \vdash e : \tau$. Consider the cases for the last rule used (in same order as figure):

- (Variable) In this case $e = x^i$ and $x:\tau \in \Gamma$. By the rules for acceptable analysis, $\varrho(x) = \text{lbl}(\tau)$ (1), $C; \varrho \vdash \tau$ (2), and $C(\varrho(x)) \subseteq C(i)$ (3). By (1) and (3), $C(\text{lbl}(\tau)) \subseteq C(i)$ (4). The result is (4) and (2).
- (Fix expression) In this case, $e = (\text{fix } f[\alpha](x:\tau_1):\tau_2.e')$ and $\tau = (\forall\alpha.\tau_1 \rightarrow \tau_2)^i$. The first part is immediate since $\text{lbl}(\tau) = \text{lbl}(e)$. The second part is required by $C; \varrho \vdash e$.
- (Application) In this case:

$$e = (e_1[\tau'] e_2)^i$$

$$\Delta; \Gamma \vdash e_1 : (\forall\alpha.\tau_1 \rightarrow \tau_3)^j \quad (1)$$

$$\tau = \tau_3[\tau'/\alpha]$$
 By the rules for acceptability, $C; \varrho \vdash e_1$ (2), $C; \varrho \vdash \tau'$ (3), $C; \varrho \vdash e_2$, and $\text{fun}(C; \text{lbl}(e_1), \text{lbl}(\tau'), \text{lbl}(e_2), i)$ (4). By (1), (2), and the induction hypothesis, $C(j) \subseteq C(\text{lbl}(e_1))$ (5) and $C; \varrho \vdash (\forall\alpha.\tau_1 \rightarrow \tau_3)^j$ (6). By (6) and the rules for acceptability, $C; \varrho \vdash \tau_3$ (7) and $(\forall\varrho(\alpha).\text{lbl}(\tau_1) \rightarrow \text{lbl}(\tau_3))_t^k \in C(j)$ (8). By (5), instantiating (4) with (8), $C(\text{lbl}(\tau')) = C(\varrho(\alpha))$ (9) and $C(\text{lbl}(\tau_3)) \subseteq C(i)$, so since $\text{lbl}(\tau) = \text{lbl}(\tau_3)$, $C(\text{lbl}(\tau)) \subseteq C(i)$ (10). By (3), (7), and (9), $C; \varrho \vdash \tau_3[\tau'/\alpha]$ (11). The result is (10) and (11).
- (Box expression) In this case $e = (\text{box}_{\tau'} e')^i$ and $\tau = \text{box}(\tau')^i$. The first part holds as $\text{lbl}(e) = \text{lbl}(\tau)$. The second part is required by $C; \varrho \vdash e$.
- (Unbox) In this case $e = (\text{unbox } e')^i$ and $\Delta; \Gamma \vdash e' : \text{box}(\tau)^j$ (1) is a subderivation. By the rules for acceptability, $C; \varrho \vdash e'$ (2) and $\text{box}(C; \text{lbl}(e'), i)$ (3). By (1), (2), and the induction hypothesis, $C(j) \subseteq C(\text{lbl}(e'))$ (4) and $C; \varrho \vdash \text{box}(\tau)^j$ (5). By (5) and the rules for acceptability, $C; \varrho \vdash \tau$ (6) and $(\text{box } \text{lbl}(\tau))_t^k \in C(j)$ (7). By (4), instantiating (3) with (7), $C(\text{lbl}(\tau)) \subseteq C(i)$ (8). The result is (8) and (6).
- (Frame) In this case $e = \rho(e')^i$, $\vdash \rho : \Gamma'$ (1), and $\emptyset; \Gamma' \vdash e' : \tau$ (2). By the rules for acceptability, $C; \varrho \vdash \rho$ (3), $C; \varrho \vdash e'$ (4), and $C(\text{lbl}(e')) \subseteq C(i)$ (5). By (1), (3), the rules for typing, and the rules for acceptability, $C; \varrho \vdash \Gamma'$ (6). By (6), (2), (4), and the induction hypothesis, $C(\text{lbl}(\tau)) \subseteq C(\text{lbl}(e'))$ (7) and $C; \varrho \vdash \tau$ (8). By (7) and (5), $C(\text{lbl}(\tau)) \subseteq C(i)$ (9). The result is (9) and (8).
- (Constant) In this case $e = c^i$ and $\tau = B^i$. The first part clearly holds as $\text{lbl}(e) = \text{lbl}(\tau)$. The second part is required by $C; \varrho \vdash e$.
- (Fix value) In this case, $e = \langle \rho, \text{fix } f[\alpha](x:\tau_1):\tau_2.e' \rangle^i$, $\tau = (\forall\alpha.\tau_1 \rightarrow \tau_2)^i$. The first part holds as $\text{lbl}(e) = \text{lbl}(\tau)$. The second part is required by $C; \varrho \vdash e$.
- (Box value) In this case $e = \langle v^j : \tau' \rangle^i$ and $\tau = \text{box}(\tau')^i$. The first part holds as $\text{lbl}(e) = \text{lbl}(\tau)$. The second part is required by $C; \varrho \vdash e$.

4. Unboxing

In the previous section, we developed a notion of flow analysis that incorporated types into the analysis. In this section, we demonstrate that this notion of analysis can be used for optimization of typed programs by developing a type-preserving global unboxing optimization. As we discussed informally in Section 2, the goal of the unboxing optimization is to use the information provided by a flow analysis to replace a boxed object with the contents of the box, and

$$\boxed{\downarrow \tau \downarrow_{\Upsilon}}$$

$$\begin{aligned} \downarrow \alpha^i \downarrow_{\Upsilon} &= \alpha^i \\ \downarrow B^i \downarrow_{\Upsilon} &= B^i \\ \downarrow (\forall\alpha.\tau_1 \rightarrow \tau_2)^i \downarrow_{\Upsilon} &= (\forall\alpha.\downarrow \tau_1 \downarrow_{\Upsilon} \rightarrow \downarrow \tau_2 \downarrow_{\Upsilon})^i \\ \downarrow \text{box}(\tau)^i \downarrow_{\Upsilon} &= \downarrow \tau \downarrow_{\Upsilon} & i \in \Upsilon \\ \downarrow \text{box}(\tau)^i \downarrow_{\Upsilon} &= \text{box}(\downarrow \tau \downarrow_{\Upsilon})^i & i \notin \Upsilon \end{aligned}$$

$$\boxed{\downarrow \Gamma \downarrow_{\Upsilon}}$$

$$\downarrow x_1:\tau_1, \dots, x_n:\tau_n \downarrow_{\Upsilon} = x_1:\downarrow \tau_1 \downarrow_{\Upsilon}, \dots, x_n:\downarrow \tau_n \downarrow_{\Upsilon}$$

$$\boxed{\downarrow e \downarrow_{\Upsilon}}$$

$$\begin{aligned} \downarrow x^i \downarrow_{\Upsilon} &= x^i \\ \downarrow m^i \downarrow_{\Upsilon} &= (\text{fix } f[\alpha](x:\downarrow \tau_1 \downarrow_{\Upsilon}):\downarrow \tau_2 \downarrow_{\Upsilon}.\downarrow e \downarrow_{\Upsilon})^i \\ &\quad \text{where } m = \text{fix } f[\alpha](x:\tau_1):\tau_2.e \\ \downarrow (e_1[\tau'] e_2)^i \downarrow_{\Upsilon} &= (\downarrow e_1 \downarrow_{\Upsilon} [\downarrow \tau' \downarrow_{\Upsilon}] \downarrow e_2 \downarrow_{\Upsilon})^i \\ \downarrow (\text{box}_{\tau'} e)^i \downarrow_{\Upsilon} &= \downarrow e \downarrow_{\Upsilon} & i \in \Upsilon \\ &= (\text{box}_{\downarrow \tau' \downarrow_{\Upsilon}} \downarrow e \downarrow_{\Upsilon})^i & i \notin \Upsilon \\ \downarrow (\text{unbox } e)^i \downarrow_{\Upsilon} &= \downarrow e \downarrow_{\Upsilon} & \text{lbl}(e) \in \Upsilon \\ &= (\text{unbox } \downarrow e \downarrow_{\Upsilon})^i & \text{lbl}(e) \notin \Upsilon \\ \downarrow \rho(e)^i \downarrow_{\Upsilon} &= \downarrow \rho \downarrow_{\Upsilon} (\downarrow e \downarrow_{\Upsilon})^i \\ \downarrow c^i \downarrow_{\Upsilon} &= c^i \\ \downarrow v^i \downarrow_{\Upsilon} &= \langle \downarrow \rho \downarrow_{\Upsilon}, \text{fix } f[\alpha](x:\downarrow \tau_1 \downarrow_{\Upsilon}):\downarrow \tau_2 \downarrow_{\Upsilon}.\downarrow e \downarrow_{\Upsilon} \rangle^i \\ &\quad \text{where } v = \langle \rho, \text{fix } f[\alpha](x:\tau_1):\tau_2.e \rangle \\ \downarrow \langle v^j : \tau \rangle^i \downarrow_{\Upsilon} &= \downarrow v^j \downarrow_{\Upsilon} & i \in \Upsilon \\ &= \langle \downarrow v^j \downarrow_{\Upsilon} : \downarrow \tau \downarrow_{\Upsilon} \rangle^i & i \notin \Upsilon \end{aligned}$$

$$\boxed{\downarrow \rho \downarrow_{\Upsilon}}$$

$$\begin{aligned} \downarrow x_1:\tau_1 = v_1^{j_1}, \dots, x_n:\tau_n = v_n^{j_n} \downarrow_{\Upsilon} &= \\ x_1:\downarrow \tau_1 \downarrow_{\Upsilon} = \downarrow v_1^{j_1} \downarrow_{\Upsilon}, \dots, x_n:\downarrow \tau_n \downarrow_{\Upsilon} = \downarrow v_n^{j_n} \downarrow_{\Upsilon} & \end{aligned}$$

$$\boxed{\downarrow M \downarrow_{\Upsilon}}$$

$$\downarrow (\rho, e) \downarrow_{\Upsilon} = (\downarrow \rho \downarrow_{\Upsilon}, \downarrow e \downarrow_{\Upsilon})$$

Figure 8. Unboxing

to rewrite the other types and terms of the program in a manner consistent with this replacement. In the rest of this section, we first develop a framework for specifying an unboxing assignment regardless of any correctness concerns, and then separately define a judgement specifying when such an assignment is a (provably) reasonable one.

4.1 The unboxing optimization

We specify a particular choice of unboxing via an unboxing set Υ that contains the set of labels of terms and types to be unboxed. A choice of a particular Υ then induces an unboxing function as defined in Figure 8. The induced unboxing function is defined in a straightforward compositional manner. Box introductions are dropped when their labels are in the unboxing set, box type constructors are dropped when their labels are in the unboxing set, box eliminations are dropped when the labels of their arguments are in the unboxing set, and all other terms and types are left unchanged.

Consider again the example from Section 2, annotated with labels:

$$\begin{aligned} \text{let } f(x:\text{box}(\text{int}^0)^1) : \text{box}(\text{box}(\text{int}^2)^3)^4 &= (\text{box } x^5)^6 \\ \text{in } (\text{unbox } (\text{unbox } (f(\text{box } 3^7)^8)^9)^{10} \text{ }^{11}) & \end{aligned}$$

If we choose as the unboxing set $\Upsilon = \{1, 3, 4, 6, 8, 9, 10\}$, the induced unboxing function produces the following optimized program:

$$\text{let } f(x:\text{int}^0) : \text{int}^2 = x^5 \text{ in } f(3^7)^9$$

If we choose instead the smaller unboxing set $\Upsilon = \{4, 6, 10\}$, the induced unboxing function produces instead the following optimized program:

$$\begin{aligned} &\text{let } f(x:\text{box}(\text{int}^0)^1) : \text{box}(\text{int}^2)^3 = x^5 \\ &\text{in } (\text{unbox } (f(\text{box } 3^7)^8)^9)^{10} \end{aligned}$$

An important observation about the unboxing optimization as we have defined it here is that unlike many previous interprocedural approaches (Section 7), it only improves programs and never introduces instructions or allocation. This is easy to see, since the unboxing function only removes boxes (which allocate and have an instruction cost), and unboxes (which have an instruction cost) and never introduces any new operations at all.

4.2 Acceptable unboxings

While any choice of Υ defines an unboxing, not every unboxing set is sensible. For example, if we elide 10 from the first example unboxing set above, the resulting program is left with an extra unboxing operation in the body of the let and is not well-typed, nor semantically valid. Just as we defined a notion of acceptable analysis in Section 3, we can define a judgement that captures sufficient conditions for ensuring correctness of an unboxing, without specifying a particular method of choosing such an unboxing. By using analyses of different precisions or choosing different optimization strategies we may end up with quite different choices of unboxings; however, so long as they satisfy our notion of acceptability we can be sure that they will preserve correctness.

Informally, a choice of an unboxing set is reasonable if it meets two criteria. Firstly, it must make uniform choices in the sense that if a box introduction is eliminated, then all of the types and elimination forms to which it flows must also be unboxed, and vice versa. Secondly, we must ensure that types remain consistent with their uses in polymorphic instantiations, since we do not allow polymorphism over base types.

We use the notation $i \overset{\Upsilon}{\simeq} j$ to indicate when an unboxing *agrees* at two labels i and j .

$$i \overset{\Upsilon}{\simeq} j \quad \text{iff} \quad \text{either } i, j \in \Upsilon \text{ or } i, j \notin \Upsilon$$

The first requirement is then specified via the *cache consistency* judgement, $C \vdash \Upsilon$, given in Figure 9. This judgement enforces that for any label i , the unboxing set must agree on i and the labels of any shapes in the cache of i . Returning again to the first example above, any acceptable analysis must include a box shape labeled with 6 in the cache of label 10. The first choice of Υ above is valid (in part) because it agrees on 6 and 10. If we elide 10 from Υ , then the cache consistency criterion is no longer satisfied, since there is a shape labeled with 6 in the cache for 10, but Υ does not agree on the two labels. In this way, cache consistency enforces the global property that if we choose to eliminate a box, we must eliminate all unboxes and box types to which it flows.

The second requirement is specified via the *consistent unboxing* judgements of Figure 9. These rules enforce the property that the result of the unboxing contains no polymorphic instantiations at non-pointer types. In the judgement, the rules for types relate a type to the traceability of its unboxing: that is, the judgement $\Upsilon \vdash \tau : t$ indicates that unboxing τ with Υ will result in a type of traceability t . The key use of this type judgement is in the term-level polymorphic-instantiation rule, which requires that the traceability of the unboxed type be \mathbf{r} .

$$\begin{array}{c} \boxed{\Upsilon \vdash \tau : t} \\ \frac{\Upsilon \vdash \alpha^i : \mathbf{r} \quad \Upsilon \vdash \mathbf{B}^i : \mathbf{b}}{\Upsilon \vdash (\forall \alpha. \tau_1 \rightarrow \tau_2)^i : \mathbf{r}} \\ \frac{i \in \Upsilon \quad \Upsilon \vdash \tau : t \quad i \notin \Upsilon}{\Upsilon \vdash \text{box}(\tau)^i : t} \quad \Upsilon \vdash \text{box}(\tau)^i : \mathbf{r} \\ \boxed{\Upsilon \vdash e} \\ \frac{\Upsilon \vdash e}{\Upsilon \vdash x^i} \quad \frac{\Upsilon \vdash e}{\Upsilon \vdash (\mathbf{fix } f[\alpha](x:\tau_1):\tau_2.e)^i} \\ \frac{\Upsilon \vdash e_1 \quad \Upsilon \vdash e_2 \quad \Upsilon \vdash \tau : \mathbf{r}}{\Upsilon \vdash (e_1[\tau] e_2)^i} \\ \frac{\Upsilon \vdash e}{\Upsilon \vdash (\text{box}_\tau e)^i} \quad \frac{\Upsilon \vdash e}{\Upsilon \vdash (\text{unbox } e)^i} \\ \frac{\Upsilon \vdash \rho \quad \Upsilon \vdash e}{\Upsilon \vdash \rho(e)^i} \quad \frac{\Upsilon \vdash e}{\Upsilon \vdash c^i} \\ \frac{\Upsilon \vdash \rho \quad \Upsilon \vdash e}{\Upsilon \vdash \langle \rho, \mathbf{fix } f[\alpha](x:\tau_1):\tau_2.e \rangle^i} \quad \frac{\Upsilon \vdash v^i}{\Upsilon \vdash \langle v^i : \tau \rangle^j} \\ \boxed{\Upsilon \vdash \rho} \\ \frac{\forall 1 \leq j \leq n : \Upsilon \vdash v_j^{i_j}}{\Upsilon \vdash x_1:\tau_1 = v_1^{i_1}, \dots, x_n:\tau_n = v_n^{i_n}} \\ \boxed{\Upsilon \vdash M} \\ \frac{\Upsilon \vdash \rho \quad \Upsilon \vdash e}{\Upsilon \vdash (\rho, e)} \\ \boxed{C \vdash \Upsilon} \\ \frac{\forall i, s : s \in C(i) \implies i \overset{\Upsilon}{\simeq} \text{lbl}(s)}{C \vdash \Upsilon} \end{array}$$

Figure 9. Cache consistency, and consistent unboxing

4.3 Type Preservation

Our goal is to show that the unboxing function induced by any consistent unboxing is in some sense correct as an optimization. The first part of this is to show that unboxing preserves typing. One key property is that types have non-empty caches.

Lemma 17 (Type Inhabitation)

If τ is not a type variable and $C; \varrho \vdash \tau$ then $C(\text{lbl}(\tau)) \neq \emptyset$.

Proof: The proof is by inspection of the rules for acceptability. ■

We also need several technical properties: labels agree when their caches intersect, unboxing preserves type well formedness, type traceability, and type equality, and unboxing commutes with type substitution.

Lemma 18 (Agreement)

If $C \vdash \Upsilon$ and $C(i) \cap C(j) \neq \emptyset$ then $i \stackrel{\Upsilon}{\simeq} j$.

Proof: The proof is by inspection of the rules for cache consistency. ■

Lemma 19

If $\Delta \vdash \tau$ wf then $\Delta \vdash \tau|_{\Upsilon}$ wf.

Proof: The proof is a straight forward induction on the structure of τ . ■

Lemma 20

If $C; \varrho \vdash \tau_2$ and $C(\text{lbl}(\tau_2)) = C(\varrho(\alpha))$ then:

- If $C; \varrho \vdash \tau_1$ then $\downarrow \tau_1[\tau_2/\alpha]|_{\Upsilon} = \downarrow \tau_1|_{\Upsilon}[\downarrow \tau_2|_{\Upsilon}/\alpha]$.
- If $C; \varrho \vdash e$ then $\downarrow e[\tau_2/\alpha]|_{\Upsilon} = \downarrow e|_{\Upsilon}[\downarrow \tau_2|_{\Upsilon}/\alpha]$.

Proof:

- The proof is by induction on the structure of τ_1 . Consider the cases for τ_1 :

- Case 1, $\tau_1 = \alpha^i$: If $\tau_2 = \sigma^j$ and $\downarrow \sigma^j|_{\Upsilon} = \sigma'^k$ then $\tau_1[\tau_2/\alpha] = \sigma^i$, thus $\downarrow \tau_1[\tau_2/\alpha]|_{\Upsilon} = \downarrow \sigma^i|_{\Upsilon}$, and also $\downarrow \tau_1|_{\Upsilon}[\downarrow \tau_2|_{\Upsilon}/\alpha] = \sigma'^i$. Thus I need to show that $\downarrow \sigma^i|_{\Upsilon} = \sigma'^i$. When σ is not a box type, this condition follows easily from the definitions. When σ is a box type, this condition follows if $i \stackrel{\Upsilon}{\simeq} j$. By $C; \varrho \vdash \tau_2$ and Lemma 17, $C(j) \neq \emptyset$. By $C; \varrho \vdash \tau_1$, $C(\text{lbl}(\tau_2)) = C(\varrho(\alpha))$, and the rules for acceptability, $C(i) = C(j)$. By Lemma 18, $i \stackrel{\Upsilon}{\simeq} j$, as required.

- Case 2, $\tau_1 = \beta^i$ and $\alpha \neq \beta$: In this case $\tau_1[\tau_2/\alpha] = \tau_1$, $\downarrow \tau_1|_{\Upsilon} = \tau_1$, and the result is immediate.

- Case 3, $\tau_1 = (\forall \alpha'. \tau_3 \rightarrow \tau_4)^i$: Then $C; \varrho \vdash \tau_1$ requires $C; \varrho \vdash \tau_3$ and $C; \varrho \vdash \tau_4$. By the induction hypothesis, $\downarrow \tau_3[\tau_2/\alpha]|_{\Upsilon} = \downarrow \tau_3|_{\Upsilon}[\downarrow \tau_2|_{\Upsilon}/\alpha]$ and $\downarrow \tau_4[\tau_2/\alpha]|_{\Upsilon} = \downarrow \tau_4|_{\Upsilon}[\downarrow \tau_2|_{\Upsilon}/\alpha]$. Thus:

$$\begin{aligned} & \downarrow \tau_1[\tau_2/\alpha]|_{\Upsilon} \\ &= \downarrow (\forall \alpha'. \tau_3[\tau_2/\alpha] \rightarrow \tau_4[\tau_2/\alpha])^i|_{\Upsilon} \\ &= (\forall \alpha'. \downarrow \tau_3[\tau_2/\alpha]|_{\Upsilon} \rightarrow \downarrow \tau_4[\tau_2/\alpha]|_{\Upsilon})^i \\ &= (\forall \alpha'. \downarrow \tau_3|_{\Upsilon}[\downarrow \tau_2|_{\Upsilon}/\alpha] \rightarrow \downarrow \tau_4|_{\Upsilon}[\downarrow \tau_2|_{\Upsilon}/\alpha])^i \\ &= (\forall \alpha'. \downarrow \tau_3|_{\Upsilon} \rightarrow \downarrow \tau_4|_{\Upsilon})^i[\downarrow \tau_2|_{\Upsilon}/\alpha] \\ &= \downarrow \tau_1|_{\Upsilon}[\downarrow \tau_2|_{\Upsilon}/\alpha] \end{aligned}$$

- Case 4, $\tau_1 = \text{box}(\tau)^i$: Then $C; \varrho \vdash \tau_1$ requires $C; \varrho \vdash \tau$. The induction hypothesis is $\downarrow \tau[\tau_2/\alpha]|_{\Upsilon} = \downarrow \tau|_{\Upsilon}[\downarrow \tau_2|_{\Upsilon}/\alpha]$. If $i \in \Upsilon$ then $\downarrow \tau_1|_{\Upsilon} = \downarrow \tau|_{\Upsilon}$ and $\downarrow \tau_1[\tau_2/\alpha]|_{\Upsilon} = \downarrow \tau[\tau_2/\alpha]|_{\Upsilon}$, as required. If $i \notin \Upsilon$ then:

$$\begin{aligned} & \downarrow \tau_1[\tau_2/\alpha]|_{\Upsilon} \\ &= \downarrow \text{box}(\tau[\tau_2/\alpha])^i|_{\Upsilon} \\ &= \text{box}(\downarrow \tau[\tau_2/\alpha]|_{\Upsilon})^i \\ &= \text{box}(\downarrow \tau|_{\Upsilon}[\downarrow \tau_2|_{\Upsilon}/\alpha])^i \\ &= \text{box}(\downarrow \tau|_{\Upsilon})^i[\downarrow \tau_2|_{\Upsilon}/\alpha] \\ &= \downarrow \tau_1|_{\Upsilon}[\downarrow \tau_2|_{\Upsilon}/\alpha] \end{aligned}$$

- The proof is a straight forward induction on the structure of e . ■

Lemma 21

If $\Upsilon \vdash \tau : t$ then $\text{tr}(\downarrow \tau|_{\Upsilon}) = t$.

Proof: The proof is a straight forward induction on the derivation of $\Upsilon \vdash \tau : t$. ■

Lemma 22

If $\vdash \tau_1 = \tau_2$, $C \vdash \Upsilon$, $C; \varrho \vdash \tau_1$, $C; \varrho \vdash \tau_2$, and either $C(\text{lbl}(\tau_1)) \subseteq C(\text{lbl}(\tau_2))$ or $C(\text{lbl}(\tau_2)) \subseteq C(\text{lbl}(\tau_1))$ then $\vdash \tau_1|_{\Upsilon} = \downarrow \tau_2|_{\Upsilon}$.

Proof: The proof is by induction on the derivation of $\vdash \tau_1 = \tau_2$. Consider the last rule used (in the same order as the figure):

- (Type variable) In this case $\tau_1 = \alpha^i$ and $\tau_2 = \alpha^j$. By definition, $\downarrow \tau_1|_{\Upsilon} = \tau_1$ and $\downarrow \tau_2|_{\Upsilon} = \tau_2$, and the result is immediate.
- (Base) In this case $\tau_1 = B^i$ and $\tau_2 = B^j$. By definition, $\downarrow \tau_1|_{\Upsilon} = \tau_1$ and $\downarrow \tau_2|_{\Upsilon} = \tau_2$, and the result is immediate.
- (Function) In this case:

$$\begin{aligned} \tau_1 &= (\forall \alpha. \tau_{11} \rightarrow \tau_{12})^{i_1} \\ \tau_2 &= (\forall \alpha. \tau_{21} \rightarrow \tau_{22})^{i_2} \\ \vdash \tau_{11} &= \tau_{21} & (1) \\ \vdash \tau_{12} &= \tau_{22} & (2) \end{aligned}$$

WLOG, assume $C(i_1) \subseteq C(i_2)$ (3). By the rules for acceptability:

$$\begin{aligned} C; \varrho \vdash \tau_{11} & & (4) \\ C; \varrho \vdash \tau_{12} & & (5) \\ (\forall \varrho(\alpha). \text{lbl}(\tau_{11}) \rightarrow \text{lbl}(\tau_{12}))^j \in C(i_1) & & (6) \\ C; \varrho \vdash \tau_{21} & & (7) \\ C; \varrho \vdash \tau_{22} & & (8) \\ \text{fun}(C; i_2, \varrho(\alpha), \text{lbl}(\tau_{21}), \text{lbl}(\tau_{22})) & & (9) \end{aligned}$$

By (6), (3), and (9), $C(\text{lbl}(\tau_{21})) \subseteq C(\text{lbl}(\tau_{11}))$ (10) and $C(\text{lbl}(\tau_{12})) \subseteq C(\text{lbl}(\tau_{22}))$ (11). By (1), (4), (7), (10), and the induction hypothesis, $\vdash \tau_{11}|_{\Upsilon} = \downarrow \tau_{21}|_{\Upsilon}$ (12). By (2), (5), (8), (11), and the induction hypothesis, $\vdash \tau_{12}|_{\Upsilon} = \downarrow \tau_{22}|_{\Upsilon}$ (13). By (12), (13), and the typing rules:

$$\vdash (\forall \alpha. \downarrow \tau_{11}|_{\Upsilon} \rightarrow \downarrow \tau_{12}|_{\Upsilon})^{i_1} = (\forall \alpha. \downarrow \tau_{21}|_{\Upsilon} \rightarrow \downarrow \tau_{22}|_{\Upsilon})^{i_2}$$

By definition:

$$\vdash \downarrow (\forall \alpha. \tau_{11} \rightarrow \tau_{12})^{i_1}|_{\Upsilon} = \downarrow (\forall \alpha. \tau_{21} \rightarrow \tau_{22})^{i_2}|_{\Upsilon}$$

as required.

- (Box) In this case $\tau_1 = \text{box}(\tau'_1)^{i_1}$, $\tau_2 = \text{box}(\tau'_2)^{i_2}$, and $\vdash \tau'_1 = \tau'_2$ (1). WLOG, assume $C(i_1) \subseteq C(i_2)$ (2). By the rules for acceptability, $C; \varrho \vdash \tau'_1$ (3), $(\text{box } \text{lbl}(\tau'_1))^j \in C(i_1)$ (4), $C; \varrho \vdash \tau'_2$ (5), and $\text{box}(C; i_2, \text{lbl}(\tau'_2))$ (6). By (4), (2), and (6), $C(\text{lbl}(\tau'_1)) \subseteq C(\text{lbl}(\tau'_2))$ (7). By (1), (3), (5), (7), and the induction hypothesis, $\vdash \downarrow \tau'_1|_{\Upsilon} = \downarrow \tau'_2|_{\Upsilon}$ (8). By Lemmas 17 and 18, $i_1 \stackrel{\Upsilon}{\simeq} i_2$. There are two cases:

- Case 1, $i_1 \in \Upsilon$: In this case, $\downarrow \tau_1|_{\Upsilon} = \downarrow \tau'_1|_{\Upsilon}$, $\downarrow \tau_2|_{\Upsilon} = \downarrow \tau'_2|_{\Upsilon}$, and the result is (8).

- Case 2, $i_2 \notin \Upsilon$: In this case, $\downarrow \tau_1|_{\Upsilon} = \text{box}(\downarrow \tau'_1|_{\Upsilon})^{i_1}$, $\downarrow \tau_2|_{\Upsilon} = \text{box}(\downarrow \tau'_2|_{\Upsilon})^{i_2}$, and the result follows from (8) and the typing rules. ■

Now we can prove that unboxing preserves typing.

Theorem 2 (Consistent unboxings preserve typing)

If $C \vdash \Upsilon$ then:

- If $\Delta; \Gamma \vdash e : \tau$, $C; \varrho \vdash \Gamma$, $C; \varrho \vdash e$, and $\Upsilon \vdash e$ then $\Delta; \downarrow \Gamma|_{\Upsilon} \vdash \downarrow e|_{\Upsilon} : \downarrow \tau|_{\Upsilon}$.
- If $\vdash \rho : \Gamma$, $C; \varrho \vdash \rho$, and $\Upsilon \vdash \rho$ then $\vdash \rho|_{\Upsilon} : \downarrow \Gamma|_{\Upsilon}$.
- If $\vdash M : \tau$, $C; \varrho \vdash M$, and $\Upsilon \vdash M$ then $\vdash M|_{\Upsilon} : \downarrow \tau|_{\Upsilon}$.

Proof: The proof is by induction on the structure of the typing judgement. Consider the cases, in the same order as the figure, for the last rule used in the derivation:

- (Variable) In this case $e = x^i$ and $x:\tau \in \Gamma$. Then $\downarrow e|_{\Upsilon} = x^i$ and clearly $x:\tau|_{\Upsilon} \in \downarrow \Gamma|_{\Upsilon}$, so the result follows by the typing rules.
- (Fix expression) In this case $e = (\mathbf{fix} f[\alpha](x:\tau_1):\tau_2.e')^i$. The typing rule requires that both $\tau = (\forall\alpha.\tau_1 \rightarrow \tau_2)^i$ and $\Delta; \Gamma, f:\tau, x:\tau_1 \vdash e' : \tau_2$. The assumption $C; \varrho \vdash e$ requires $\varrho(f) = i$, $\varrho(x) = \text{lbl}(\tau_1)$, $C \vdash \tau$, and $C; \varrho \vdash e'$. From $C \vdash \tau$ and the rules for acceptability, $C \vdash \tau_1$. From these facts, $C; \varrho \vdash \Gamma, f:\tau, x:\tau_1$. The assumption $\Upsilon \vdash e$ requires that $\Upsilon \vdash e'$. By the induction hypothesis, $\Delta; \downarrow \Gamma, f:\tau, x:\tau_1|_{\Upsilon} \vdash \downarrow e'|_{\Upsilon} : \downarrow \tau_2|_{\Upsilon}$. Since:

$$\downarrow \Gamma, f:\tau, x:\tau_1|_{\Upsilon} = \downarrow \Gamma|_{\Upsilon}, f:(\forall\alpha.\downarrow \tau_1|_{\Upsilon} \rightarrow \downarrow \tau_2|_{\Upsilon})^i, x:\downarrow \tau_2|_{\Upsilon}$$

by the typing rules:

$$\Delta; \downarrow \Gamma|_{\Upsilon} \vdash (\mathbf{fix} f[\alpha](x:\downarrow \tau_1|_{\Upsilon}):\downarrow \tau_2|_{\Upsilon}.\downarrow e'|_{\Upsilon})^i : (\forall\alpha.\downarrow \tau_1|_{\Upsilon} \rightarrow \downarrow \tau_2|_{\Upsilon})^i$$

The result follows since:

$$\begin{aligned} \downarrow e|_{\Upsilon} &= (\mathbf{fix} f[\alpha](x:\downarrow \tau_1|_{\Upsilon}):\downarrow \tau_2|_{\Upsilon}.\downarrow e'|_{\Upsilon})^i \\ \downarrow \tau|_{\Upsilon} &= (\forall\alpha.\downarrow \tau_1|_{\Upsilon} \rightarrow \downarrow \tau_2|_{\Upsilon})^i \end{aligned}$$

- (Application) In this case, $e = (e_1[\tau'] e_2)^i$. The typing rule, $C; \varrho \vdash e$, and $\Upsilon \vdash e$ require that:

$$\begin{aligned} \Delta; \Gamma \vdash e_1 : (\forall\alpha.\tau_1 \rightarrow \tau_3)^j & \quad (1) \\ \tau = \tau_3[\tau'/\alpha] & \quad (2) \\ \Delta; \Gamma \vdash e_2 : \tau_2 & \quad (3) \\ \Delta \vdash \tau' \text{ wf} & \quad (4) \\ \vdash \tau_1[\tau'/\alpha] = \tau_2 & \quad (5) \\ C; \varrho \vdash e_1 & \quad (6) \\ C; \varrho \vdash \tau' & \quad (7) \\ C; \varrho \vdash e_2 & \quad (8) \\ \text{fun}(C; \text{lbl}(e_1), \text{lbl}(\tau), \text{lbl}(e_2), i) & \quad (9) \\ \Upsilon \vdash e_1 & \quad (10) \\ \Upsilon \vdash e_2 & \quad (11) \\ \Upsilon \vdash \tau' : \mathbf{r} & \quad (12) \end{aligned}$$

for some τ_1, τ_3, j , and τ_2 . By (1), (5), (10), (2), (7), (11), and the induction hypothesis:

$$\begin{aligned} \Delta; \downarrow \Gamma|_{\Upsilon} \vdash \downarrow e_1|_{\Upsilon} : \downarrow (\forall\alpha.\tau_1 \rightarrow \tau_3)^j|_{\Upsilon} & \quad (13) \\ \Delta; \downarrow \Gamma|_{\Upsilon} \vdash \downarrow e_2|_{\Upsilon} : \downarrow \tau_2|_{\Upsilon} & \quad (14) \end{aligned}$$

By definition $\downarrow (\forall\alpha.\tau_1 \rightarrow \tau_3)^j|_{\Upsilon} = (\forall\alpha.\downarrow \tau_1|_{\Upsilon} \rightarrow \downarrow \tau_3|_{\Upsilon})^j$. By (3) and Lemma 19, $\Delta \vdash \downarrow \tau'|_{\Upsilon} \text{ wf}$ (15). By (12) and Lemma 21, $\text{tr}(\downarrow \tau'|_{\Upsilon}) = \mathbf{r}$ (16). By (1), (2), and Lemma 16:

$$\begin{aligned} C(j) \subseteq C(\text{lbl}(e_1)) & \quad (17) \\ C; \varrho \vdash (\forall\alpha.\tau_1 \rightarrow \tau_3)^j & \quad (18) \\ C(\text{lbl}(\tau_2)) \subseteq C(\text{lbl}(e_2)) & \quad (19) \\ C \vdash \tau_2 & \quad (20) \end{aligned}$$

By (18) and the rules for acceptability:

$$\begin{aligned} C; \varrho \vdash \tau_1 & \quad (21) \\ C; \varrho \vdash \tau_3 & \quad (22) \\ (\forall\varrho(\alpha).\text{lbl}(\tau_1) \rightarrow \text{lbl}(\tau_3))_t^k \in C(j) & \quad (23) \end{aligned}$$

By (23), (17), and (8), $C(\text{lbl}(\tau')) = C(\varrho(\alpha))$ (24) and $C(\text{lbl}(e_2)) \subseteq C(\text{lbl}(\tau_1))$. Hence by (19), $C(\text{lbl}(\tau_2)) \subseteq C(\text{lbl}(\tau_1))$. Since $\text{lbl}(\tau_1[\tau'/\alpha]) = \text{lbl}(\tau_1)$, $C(\text{lbl}(\tau_2)) \subseteq C(\text{lbl}(\tau_1[\tau'/\alpha]))$ (25). By (21), (6), (24), and Lemma 13, $C; \varrho \vdash \tau_1[\tau'/\alpha]$ (26). By (4), (26), (20), (25), and Lemma 22, $\vdash \downarrow \tau_1[\tau'/\alpha]|_{\Upsilon} = \downarrow \tau_2|_{\Upsilon}$. By (21), (6), (24), and Lemma 20, $\vdash \downarrow \tau_1|_{\Upsilon}[\downarrow \tau'|_{\Upsilon}/\alpha] = \downarrow \tau_2|_{\Upsilon}$ (27). Thus by (13), (14), (15), (16), (27), and the typing rules, $\Delta; \downarrow \Gamma|_{\Upsilon} \vdash (\downarrow e_1|_{\Upsilon}[\downarrow \tau'|_{\Upsilon}] \downarrow e_2|_{\Upsilon})^i : \downarrow \tau_3|_{\Upsilon}[\downarrow \tau'|_{\Upsilon}/\alpha]$. By definition, (22), (6), (24), and Lemma 20, $\Delta; \downarrow \Gamma|_{\Upsilon} \vdash \downarrow (e_1[\tau'] e_2)^i|_{\Upsilon} : \downarrow \tau|_{\Upsilon}$, as required.

- (Box expression) In this case, $e = (\mathbf{box}_{\tau''} e')^i$ for some e' and i . The typing rule requires that $\tau = \mathbf{box}(\tau'')^i$, $\Delta \vdash \tau'' \text{ wf}$, $\Delta; \Gamma \vdash e' : \tau'$, and $\vdash \tau'' = \tau'$ for some τ' . The assumption $C; \varrho \vdash e$ requires that $C \vdash \tau$ and $C; \varrho \vdash e'$. The assumption $\Upsilon \vdash e$ requires that $\Upsilon \vdash e'$. By the induction hypothesis, $\Delta; \downarrow \Gamma|_{\Upsilon} \vdash \downarrow e'|_{\Upsilon} : \downarrow \tau'|_{\Upsilon}$. There are two subcases:

- If $i \in \mathbf{T}$ then $\downarrow e|_{\Upsilon} = \downarrow e'|_{\Upsilon}$ and $\downarrow \tau|_{\Upsilon} = \downarrow \tau'|_{\Upsilon}$ and the result is immediate.
- If $i \notin \mathbf{T}$ then $\downarrow e|_{\Upsilon} = (\mathbf{box}_{\downarrow \tau''|_{\Upsilon}} \downarrow e'|_{\Upsilon})^i$ and $\downarrow \tau|_{\Upsilon} = \mathbf{box}(\downarrow \tau''|_{\Upsilon})^i$. The result follows by the typing rules if $\Delta \vdash \downarrow \tau''|_{\Upsilon} \text{ wf}$, which holds by Lemma 19, and $\vdash \downarrow \tau''|_{\Upsilon} = \downarrow \tau'|_{\Upsilon}$, which holds by Lemma 22 if its other three premises hold. Since $C \vdash \tau$, by the rules for acceptability, $\mathbf{box}(C; i, \text{lbl}(\tau''))$ (1) and $C \vdash \tau''$, showing the first premise. Since $\Delta; \Gamma \vdash e' : \tau'$, by Lemma 16, $C(\text{lbl}(\tau'')) \subseteq C(\text{lbl}(e'))$ (2) and $C \vdash \tau'$, showing the second premise. By $C; \varrho \vdash e$, $(\mathbf{box}_{\text{tr}(\tau')} \text{lbl}(e'))_v^j \in C(i)$. Thus by (1), $C(\text{lbl}(e')) \subseteq C(\text{lbl}(\tau''))$, so by (2), $C(\text{lbl}(\tau'')) \subseteq C(\text{lbl}(\tau''))$, showing the third premise, as required.

- (Unbox) In this case, $e = (\mathbf{unbox} e')^i$ for some e' and i . The typing rule requires that $\Delta; \Gamma \vdash e' : \mathbf{box}(\tau)^j$ for some j . The assumption $C; \varrho \vdash e$ requires $C; \varrho \vdash e'$. The assumption $\Upsilon \vdash e$ requires $\Upsilon \vdash e'$. By the induction hypothesis, $\Delta; \downarrow \Gamma|_{\Upsilon} \vdash \downarrow e'|_{\Upsilon} : \downarrow \mathbf{box}(\tau)^j|_{\Upsilon}$. By Lemma 16, $C(j) \subseteq C(\text{lbl}(e'))$. By Lemmas 17 and 18, $j \stackrel{\Upsilon}{\simeq} \text{lbl}(e')$. There are two subcases:

- If $j \in \Upsilon$ then $\downarrow e|_{\Upsilon} = \downarrow e'|_{\Upsilon}$ and $\downarrow \mathbf{box}(\tau)^j|_{\Upsilon} = \downarrow \tau|_{\Upsilon}$ and the result is immediate.
- If $j \notin \Upsilon$ then $\downarrow e|_{\Upsilon} = (\mathbf{unbox} \downarrow e'|_{\Upsilon})^i$ and $\downarrow \mathbf{box}(\tau)^j|_{\Upsilon} = \mathbf{box}(\downarrow \tau|_{\Upsilon})^j$. The result then follows by the typing rules.

- (Frame) In this case, $e = \rho'(e')^i$ for some ρ' , e' , and i . The typing rule requires that $\vdash \rho' : \Gamma'$ and $\Delta; \Gamma' \vdash e' : \tau$ for some Γ' . The assumption $C; \varrho \vdash e$ requires that $C; \varrho \vdash \rho'$ and $C; \varrho \vdash e'$. The former requires that $C; \varrho \vdash \Gamma'$. The assumption $\Upsilon \vdash e$ requires $\Upsilon \vdash e'$. By the induction hypothesis, $\vdash \rho'|_{\Upsilon} : \downarrow \Gamma'|_{\Upsilon}$ and $\Delta; \downarrow \Gamma'|_{\Upsilon} \vdash \downarrow e'|_{\Upsilon} : \downarrow \tau|_{\Upsilon}$. So by the typing rules, $\Delta; \downarrow \Gamma|_{\Upsilon} \vdash \downarrow \rho'|_{\Upsilon}(\downarrow e'|_{\Upsilon})^i : \downarrow \tau|_{\Upsilon}$. The result follows since $\downarrow e|_{\Upsilon} = \downarrow \rho'|_{\Upsilon}(\downarrow e'|_{\Upsilon})^i$.

- (Constant) In this case $e = c^i$ for some c and i . The typing rule requires that $\tau = \mathbf{B}^i$. Clearly, $\downarrow e|_{\Upsilon} = c^i$, $\downarrow \tau|_{\Upsilon} = \mathbf{B}^i$, and the result follows by the typing rules.

- (Fix value) In this case $e = \langle \rho, \mathbf{fix} f[\alpha](x:\tau_1):\tau_2.e' \rangle^i$. The typing rule require that $\tau = (\forall\alpha.\tau_1 \rightarrow \tau_2)^i$, $\vdash \rho : \Gamma'$, and $\Delta, \alpha; \Gamma', f:\tau, x:\tau_1 \vdash e' : \tau_2$. The assumption $C; \varrho \vdash e$ requires $C; \varrho \vdash \rho$, from which $C; \varrho \vdash \Gamma'$, $\varrho(f) = i$, $\varrho(x) = \text{lbl}(\tau_1)$, $C \vdash \tau$, and $C; \varrho \vdash e'$. From $C \vdash \tau$ and the rules for acceptability, $C \vdash \tau_1$. From these facts, $C; \varrho \vdash \Gamma', f:\tau, x:\tau_1$. The assumption $\Upsilon \vdash e$ requires $\Upsilon \vdash e'$. By the induction hypothesis, $\vdash \rho|_{\Upsilon} : \downarrow \Gamma'|_{\Upsilon}$ and $\Delta, \alpha; \downarrow \Gamma', f:\tau, x:\tau_1|_{\Upsilon} \vdash \downarrow e'|_{\Upsilon} : \downarrow \tau_2|_{\Upsilon}$. Since:

$$\downarrow \Gamma', f:\tau, x:\tau_1|_{\Upsilon} = \downarrow \Gamma'|_{\Upsilon}, f:(\forall\alpha.\downarrow \tau_1|_{\Upsilon} \rightarrow \downarrow \tau_2|_{\Upsilon})^i, x:\downarrow \tau_2|_{\Upsilon}$$

by the typing rules:

$$\Delta; \downarrow \Gamma|_{\Upsilon} \vdash \langle \rho|_{\Upsilon}, \mathbf{fix} f[\alpha](x:\downarrow \tau_1|_{\Upsilon}):\downarrow \tau_2|_{\Upsilon}.\downarrow e'|_{\Upsilon} \rangle^i : (\forall\alpha.\downarrow \tau_1|_{\Upsilon} \rightarrow \downarrow \tau_2|_{\Upsilon})^i$$

The result follows since:

$$\begin{aligned} \downarrow e|_{\Upsilon} &= \langle \rho|_{\Upsilon}, \mathbf{fix} f[\alpha](x:\downarrow \tau_1|_{\Upsilon}):\downarrow \tau_2|_{\Upsilon}.\downarrow e'|_{\Upsilon} \rangle^i \\ \downarrow \tau|_{\Upsilon} &= (\forall\alpha.\downarrow \tau_1|_{\Upsilon} \rightarrow \downarrow \tau_2|_{\Upsilon})^i \end{aligned}$$

- (Box value) In this case, $e = \langle v^j : \tau'' \rangle^i$ for some v, i , and j . The typing rule requires that $\tau = \text{box}(\tau'')^i$, $\Delta \vdash \tau'' \text{ wf}$, $\Delta; \Gamma \vdash v^j : \tau'$, and $\vdash \tau'' = \tau'$ for some τ' . The assumption $C; \varrho \vdash e$ requires that $C \vdash \tau$ and $C; \varrho \vdash v^j$. The assumption $\Upsilon \vdash e$ requires $\Upsilon \vdash v^j$. By the induction hypothesis, $\Delta; \downarrow \Gamma \downarrow_{\Upsilon} \vdash \downarrow v^j \downarrow_{\Upsilon} : \downarrow \tau' \downarrow_{\Upsilon}$. There are two subcases:

- If $i \in \mathbb{T}$ then $\downarrow e \downarrow_{\Upsilon} = \downarrow v^j \downarrow_{\Upsilon}$ and $\downarrow \tau \downarrow_{\Upsilon} = \downarrow \tau' \downarrow_{\Upsilon}$ and the result is immediate.
- If $i \notin \mathbb{T}$ then $\downarrow e \downarrow_{\Upsilon} = \langle \downarrow v^j \downarrow_{\Upsilon} : \downarrow \tau'' \downarrow_{\Upsilon} \rangle^i$ and $\downarrow \tau \downarrow_{\Upsilon} = \text{box}(\downarrow \tau' \downarrow_{\Upsilon})^i$. The result follows by the typing rules if $\Delta \vdash \downarrow \tau'' \downarrow_{\Upsilon} \text{ wf}$, which holds by Lemma 19, and $\vdash \downarrow \tau'' \downarrow_{\Upsilon} = \downarrow \tau' \downarrow_{\Upsilon}$, which holds by Lemma 22 if its other three premises hold. Since $C \vdash \tau$, by the rules for acceptability, $\text{box}(C; i, \text{lbl}(\tau''))$ (1) and $C \vdash \tau''$, showing the first premise. Since $\Delta; \Gamma \vdash v^j : \tau'$, by Lemma 16, $C(\text{lbl}(\tau')) \subseteq C(j)$ (2) and $C \vdash \tau'$, showing the second premise. By $C; \varrho \vdash e$, $(\text{box}_{tr(\tau')} j)_v^k \in C(i)$. Thus by (1), $C(j) \subseteq C(\text{lbl}(\tau''))$, so by (2), $C(\text{lbl}(\tau')) \subseteq C(\text{lbl}(\tau''))$, showing the third premise, as required.

- (Environment) In this case $\rho = x_1 : \tau_1 = v_1^{i_1}, \dots, x_n : \tau_n = v_n^{i_n}$ and $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$. The typing rule requires that $\emptyset; \emptyset \vdash v_j^{i_j} : \tau_j^i$ and $\vdash \tau_j = \tau_j^i$ for $1 \leq j \leq n$ and some τ_j^i s. The assumption $C; \varrho \vdash \rho$ requires $C; \varrho \vdash \tau_j$ and $C; \varrho \vdash v_j^{i_j}$ for $1 \leq j \leq n$. Clearly $C; \varrho \vdash \Gamma'$ where Γ' is empty. The assumption $\Upsilon \vdash \rho$ requires $\Upsilon \vdash v_j^{i_j}$ for $1 \leq j \leq n$. By the induction hypothesis, $\emptyset; \emptyset \vdash \downarrow v_j^{i_j} \downarrow_{\Upsilon} : \downarrow \tau_j^i \downarrow_{\Upsilon}$ for $1 \leq j \leq n$. By the rules for acceptability, $C(i_j) \subseteq C(\text{lbl}(\tau_j^i))$ for $1 \leq j \leq n$. By Lemma 16, $C(\tau_j^i) \subseteq C(i_j)$ and $C \vdash \tau_j^i$ for $1 \leq j \leq n$. Thus $C(\tau_j^i) \subseteq C(\tau_j)$ for $1 \leq j \leq n$. By Lemma 22, $\vdash \downarrow \tau_j \downarrow_{\Upsilon} = \downarrow \tau_j^i \downarrow_{\Upsilon}$ for $1 \leq j \leq n$. Then by the typing rules $\vdash x_1 : \downarrow \tau_1 \downarrow_{\Upsilon} = \downarrow v_1^{i_1} \downarrow_{\Upsilon}, \dots, x_n : \downarrow \tau_n \downarrow_{\Upsilon} = \downarrow v_n^{i_n} \downarrow_{\Upsilon} : x_1 : \downarrow \tau_1 \downarrow_{\Upsilon}, \dots, x_n : \downarrow \tau_n \downarrow_{\Upsilon}$. The result follows since $\downarrow \rho \downarrow_{\Upsilon} = x_1 : \downarrow \tau_1 \downarrow_{\Upsilon} = \downarrow v_1^{i_1} \downarrow_{\Upsilon}, \dots, x_n : \downarrow \tau_n \downarrow_{\Upsilon} = \downarrow v_n^{i_n} \downarrow_{\Upsilon}$ and $\downarrow \Gamma \downarrow_{\Upsilon} = x_1 : \downarrow \tau_1 \downarrow_{\Upsilon}, \dots, x_n : \downarrow \tau_n \downarrow_{\Upsilon}$.
- (Machine state) In this case $M = (\rho, e)$. By the typing rule, $\vdash \rho : \Gamma$ and $\emptyset; \Gamma \vdash e : \tau$ for some Γ . The assumption $C; \varrho \vdash M$ requires both $C; \varrho \vdash \rho$ and $C; \varrho \vdash e$. The former requires $C; \varrho \vdash \Gamma$. The assumption $\Upsilon \vdash M$ requires $\Upsilon \vdash \rho$ and $\Upsilon \vdash e$. By the induction hypothesis, $\vdash \downarrow \rho \downarrow_{\Upsilon} : \downarrow \Gamma \downarrow_{\Upsilon}$ and $\emptyset; \downarrow \Gamma \downarrow_{\Upsilon} \vdash \downarrow e \downarrow_{\Upsilon} : \downarrow \tau \downarrow_{\Upsilon}$. So by the typing rules, $\vdash (\downarrow \rho \downarrow_{\Upsilon}, \downarrow e \downarrow_{\Upsilon}) : \downarrow \tau \downarrow_{\Upsilon}$. The result follows since $\downarrow M \downarrow_{\Upsilon} = (\downarrow \rho \downarrow_{\Upsilon}, \downarrow e \downarrow_{\Upsilon})$.

A consequence of type preservation is that unboxed well-typed programs are traceable.

Theorem 3

If $\vdash M : \tau, C; \varrho \vdash M, C \vdash \Upsilon$, and $\Upsilon \vdash M$ then $\vdash \downarrow M \downarrow_{\Upsilon} \text{ tr}$.

Proof: The proof follows from Theorem 2 and Lemma 11. ■

4.4 Coherence

The other part of proving correctness is to show that unboxing preserves semantics in some appropriate sense. That requires two key lemmas—that a step of the program can be matched by zero or more steps of the unboxed program and that consistency is preserved under reduction.

To show the first lemma, we need three technical lemmas—that a value's cache is nonempty, that reduction preserves the unboxing or not of the outermost label, and a multistep compositionality property.

Lemma 23 (Inhabitation)

If $C; \varrho \vdash v^k$ then $\exists s \in C(k)$ such that $\text{lbl}(s) = k$.

Proof: By inspection of the acceptable analysis and acceptable instantiation rules. ■

Lemma 24 (Unboxing set preservation)

If $C; \varrho \vdash \rho, C; \varrho \vdash e, C \vdash \Upsilon$, and $(\rho, e_1) \mapsto (\rho, e_2)$ then $\text{lbl}(e) \stackrel{\Upsilon}{\simeq} \text{lbl}(e')$.

Proof: All of the cases for which $\text{lbl}(e_1) = \text{lbl}(e_2)$ follow immediately. For the remaining cases:

- If $(\rho, x^k) \mapsto (\rho, v^j)$ where $x : \tau = v^j \in \rho$ then by the assumptions we have that $\varrho(x) = \text{lbl}(\tau)$ (1), $C(j) \subseteq C(\text{lbl}(\tau))$ (2) and $C(\varrho(x)) \subseteq C(k)$ (3), so by transitivity we have $C(j) \subseteq C(k)$ (4). By Inhabitation (Lemma 23) we have an $s \in C(j)$ (5) such that $\text{lbl}(s) = j$ (6), and so by Agreement (Lemma 18) we have $k \stackrel{\Upsilon}{\simeq} j$. Since $\text{lbl}(e) = k$ and $\text{lbl}(e') = j$, the result follows.
- If $(\rho, (\text{unbox } \langle v^i : t \rangle^j)^k) \mapsto (\rho, v^i)$ then we must show that $k \stackrel{\Upsilon}{\simeq} i$. By Inhabitation we have $s \in C(i)$ with $\text{lbl}(s) = i$, so by Agreement, it suffices to show that $C(i) \subseteq C(k)$. By the box rule for an acceptable analysis, there is a $s = (\text{box}_t l)_v^j \in C(j)$ such that $C(i) \subseteq C(l)$. Since $s \in C(j)$, by the rule for unbox, $C(l) \subseteq C(k)$, so $C(i) \subseteq C(k)$ and we're done.
- If $(\rho, \rho'(v^i)^j) \mapsto (\rho, v^i)$ then we must show that $j \stackrel{\Upsilon}{\simeq} i$. By Inhabitation, there is an $s \in C(i)$, and by the acceptable analysis rule for frames we have that $C(i) \subseteq C(j)$, so by Agreement we have that $j \stackrel{\Upsilon}{\simeq} i$. ■

Lemma 25 (Many step compositionality)

If $(\rho, e_1) \mapsto^* (\rho, e_2)$ then:

- $(\rho, (e_1 e)^i) \mapsto^* (\rho, (e_2 e)^i)$
- $(\rho, (v^j e_1)^i) \mapsto^* (\rho, (v^j e_2)^i)$
- $(\rho, (\text{box}_{\tau} e_1)^i) \mapsto^* (\rho, (\text{box}_{\tau} e_2)^i)$
- $(\rho, (\text{unbox } e_1)^i) \mapsto^* (\rho, (\text{unbox } e_2)^i)$
- $(\rho, \rho'(e_1)^i) \mapsto^* (\rho, \rho'(e_2)^i)$

Proof: The proof is by an easy induction on the length of the reduction sequences. ■

Theorem 4 (Single step reduction coherence)

If $\vdash M : \tau, C; \varrho \vdash M, C \vdash \Upsilon, \Upsilon \vdash M$, and $M \mapsto M'$ then $\downarrow M \downarrow_{\Upsilon} \mapsto^* \downarrow M' \downarrow_{\Upsilon}$.

Proof: The proof is by induction on the derivation of $M \mapsto M'$, consider the cases for the last rule used to derive it:

- If $(\rho, x^k) \mapsto (\rho, v^j)$ where $x : \tau = v^j \in \rho$ then by definition $\downarrow M \downarrow_{\Upsilon} = (\downarrow \rho \downarrow_{\Upsilon}, x^k)$, $\downarrow M' \downarrow_{\Upsilon} = (\downarrow \rho \downarrow_{\Upsilon}, \downarrow v^j \downarrow_{\Upsilon})$, and $x : \downarrow \tau \downarrow_{\Upsilon} = \downarrow v^j \downarrow_{\Upsilon} \in \downarrow \rho \downarrow_{\Upsilon}$. Thus $\downarrow M \downarrow_{\Upsilon} \mapsto \downarrow M' \downarrow_{\Upsilon}$ by the same rule.

• If:

$$\begin{aligned} & (\rho, (\text{fix } f[\alpha](x : \tau_1) : \tau_2 . e_1)^j) \mapsto \\ & (\rho, \langle \rho, \text{fix } f[\alpha](x : \tau_1) : \tau_2 . e_1 \rangle^j) \end{aligned}$$

then the unboxings of the e and e' are of the same form, and the same reduction step applies.

- If $(\rho, (\text{box}_{\tau'} v^i)^j) \mapsto (\rho, \langle v^i : \tau' \rangle^j)$ where $tr(\tau') = t$ then:

- If $j \notin \Upsilon$ then:

By the definition of unboxing, $\downarrow e \downarrow_{\Upsilon} = (\text{box}_{\downarrow \tau' \downarrow_{\Upsilon}} v_{t'}^{k'})^j$ where $v_{t'}^{k'} = \downarrow v_{t'}^i \downarrow_{\Upsilon}$.

By hypothesis, $\vdash \rho : \Gamma, \emptyset; \Gamma \vdash v_{t'}^i : \tau''$ and $\vdash \tau' = \tau''$ for some τ'' . By hypothesis, $\Upsilon \vdash v_{t'}^i$. By Theorem 2, $\emptyset; \downarrow \Gamma \downarrow_{\Upsilon} \vdash v_{t'}^{k'} : \downarrow \tau'' \downarrow_{\Upsilon}$. The proof of that theorem also showed that $\vdash \downarrow \tau' \downarrow_{\Upsilon} = \downarrow \tau'' \downarrow_{\Upsilon}$, so by Lemma 4, $\text{tr}(\downarrow \tau' \downarrow_{\Upsilon}) = \text{tr}(\downarrow \tau'' \downarrow_{\Upsilon})$. By Lemma 6, $\text{tr}(\downarrow \tau'' \downarrow_{\Upsilon}) = t'$, so $\text{tr}(\downarrow \tau' \downarrow_{\Upsilon}) = t'$.

By definition of reduction $(\downarrow \rho \downarrow_{\Upsilon}, (\text{box}_{\downarrow \tau' \downarrow_{\Upsilon}} v_{t'}^{k'})^j) \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, \langle v_{t'}^{k'} : \downarrow \tau' \downarrow_{\Upsilon} \rangle^j)$.

- If $j \in \Upsilon$ then:

By definition of unboxing

$$\downarrow e \downarrow_{\Upsilon} = v_{t'}^{k'} \text{ where } v_{t'}^{k'} = \downarrow v_{t'}^i \downarrow_{\Upsilon}$$

By definition of reduction

$$(\downarrow \rho \downarrow_{\Upsilon}, v_{t'}^{k'}) \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, v_{t'}^{k'})$$

- If $(\rho, (e_1[\tau'] e_2)^j) \mapsto (\rho, (e_1'[\tau'] e_2')^j)$ then:

By definition of C ; $\rho \vdash M$ we have that $C; \rho \vdash \rho$ and $C; \rho \vdash e_1$. Hence we have that $C; \rho \vdash (\rho, e_1)$. By the typing rules we also have that $\vdash \rho : \Gamma$ and $\emptyset; \Gamma \vdash e_1 : \tau_1$ for some Γ and τ_1 , so $\vdash (\rho, e_1) : \tau_1$. By the rules for consistency, $\Upsilon \vdash \rho$ and $\Upsilon \vdash e_1$, so $\Upsilon \vdash (\rho, e_1)$. Hence by induction we have that $(\downarrow \rho \downarrow_{\Upsilon}, \downarrow e_1 \downarrow_{\Upsilon}) \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, \downarrow e_1' \downarrow_{\Upsilon})$.

By Lemma 25

$$(\downarrow \rho \downarrow_{\Upsilon}, (\downarrow e_1 \downarrow_{\Upsilon} [\downarrow \tau' \downarrow_{\Upsilon}] \downarrow e_2 \downarrow_{\Upsilon})^j) \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, (\downarrow e_1' \downarrow_{\Upsilon} [\downarrow \tau' \downarrow_{\Upsilon}] \downarrow e_2 \downarrow_{\Upsilon})^j)$$

By definition of unboxing

$$(\downarrow \rho \downarrow_{\Upsilon}, \downarrow (e_1[\tau'] e_2)^j \downarrow_{\Upsilon}) \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, \downarrow (e_1'[\tau'] e_2')^j \downarrow_{\Upsilon})$$

- If $(\rho, (e_1[\tau'] e_2)^j) \mapsto (\rho, (e_1[\tau'] e_2')^j)$ then the argument follows by the symmetric argument to the previous case.
- If $(\rho, (v_f^j[\tau'] v_t^k)^l) \mapsto (\rho, \rho''(e''[\tau'/\alpha])^l)$ where:

$$\begin{aligned} v_f &= \langle \rho', \text{fix } f[\alpha](x:\tau_1):\tau_2.e'' \rangle \\ \rho'' &= \rho', f:\tau = v_f^j, x:\tau_1 = v_t^k \\ \tau &= (\forall \alpha. \tau_1 \rightarrow \tau_2)^j \\ \tau_1' &= \tau_1[\tau'/\alpha] \end{aligned}$$

then:

By definition of unboxing we have that:

$$\begin{aligned} \downarrow e \downarrow_{\Upsilon} &= (v_f^j[\downarrow \tau' \downarrow_{\Upsilon}] v_t^{k'})^l \\ v_f^j &= \langle \downarrow \rho' \downarrow_{\Upsilon}, \text{fix } f[\alpha](x:\downarrow \tau_1 \downarrow_{\Upsilon}):\downarrow \tau_2 \downarrow_{\Upsilon}. \downarrow e'' \downarrow_{\Upsilon} \rangle \\ v_t^{k'} &= \downarrow v_t^k \downarrow_{\Upsilon} \\ \downarrow e' \downarrow_{\Upsilon} &= (\downarrow \rho' \downarrow_{\Upsilon}, f:\downarrow \tau \downarrow_{\Upsilon} = v_f^j, x:\downarrow \tau_1 \downarrow_{\Upsilon} = v_t^{k'}) \\ &\quad (\downarrow e''[\tau'/\alpha] \downarrow_{\Upsilon})^l \end{aligned}$$

By hypothesis, $\vdash \rho : \Gamma, \emptyset; \Gamma \vdash v_t^k : \tau_1'$, and $\vdash \tau_1' = \tau_1''$. By Theorem 2, $\emptyset; \downarrow \Gamma \downarrow_{\Upsilon} \vdash v_t^{k'} : \downarrow \tau_1'' \downarrow_{\Upsilon}$. The proof of that theorem also showed that $\vdash \downarrow \tau_1' \downarrow_{\Upsilon} = \downarrow \tau_1'' \downarrow_{\Upsilon}$. By Lemma 4, $\text{tr}(\downarrow \tau_1' \downarrow_{\Upsilon}) = \text{tr}(\downarrow \tau_1'' \downarrow_{\Upsilon})$. By Lemma 6, $\text{tr}(\downarrow \tau_1'' \downarrow_{\Upsilon}) = t'$. Thus $\text{tr}(\downarrow \tau_1' \downarrow_{\Upsilon}) = t'$. So by the application beta rule:

$$(\downarrow \rho \downarrow_{\Upsilon}, \downarrow e \downarrow_{\Upsilon}) \mapsto (\downarrow \rho \downarrow_{\Upsilon}, \rho'''(e''')^l)$$

where:

$$\begin{aligned} \rho''' &= \downarrow \rho \downarrow_{\Upsilon}, f:\tau'' = v_f^j, x:\tau_1'' = v_t^{k'} \\ \tau'' &= (\forall \alpha. \downarrow \tau_1 \downarrow_{\Upsilon} \rightarrow \downarrow \tau_2 \downarrow_{\Upsilon})^j \\ \tau_1'' &= \downarrow \tau_1 \downarrow_{\Upsilon} [\downarrow \tau' \downarrow_{\Upsilon} / \alpha] \\ e''' &= \downarrow e'' \downarrow_{\Upsilon} [\downarrow \tau' \downarrow_{\Upsilon} / \alpha] \end{aligned}$$

By definition of unboxing, $\tau'' = \downarrow \tau \downarrow_{\Upsilon}$. The proof of the theorem above also showed that $C; \rho \vdash \tau_1$, $C; \rho \vdash \tau'$, and $C(\text{lbl}(\tau')) = C(\rho(\alpha))$. By the rules for acceptability, clearly $C; \rho \vdash e''$. By Lemma 20, $\tau_1'' = \downarrow \tau_1 \downarrow_{\Upsilon}$ and $e''' = \downarrow e''[\tau'/\alpha] \downarrow_{\Upsilon}$. Putting that altogether, $\rho'''(e''')^l = \downarrow e' \downarrow_{\Upsilon}$, as required.

- If $(\rho, (\text{box}_{\tau'} e)^j) \mapsto (\rho, (\text{box}_{\tau'} e')^j)$ then: By definition of acceptability $C; \rho \vdash \rho$ and $C; \rho \vdash e$, so $C; \rho \vdash (\rho, e)$. By the typing rules, $\vdash \rho : \Gamma$ and $\emptyset; \Gamma \vdash e : \tau''$ for some Γ and τ'' , so $\vdash (\rho, e) : \tau''$. By the rules for consistency, $\Upsilon \vdash \rho$ and $\Upsilon \vdash e$, so $\Upsilon \vdash (\rho, e)$. So by the induction hypothesis, $(\downarrow \rho \downarrow_{\Upsilon}, \downarrow e \downarrow_{\Upsilon}) \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, \downarrow e' \downarrow_{\Upsilon})$.

- If $j \in \Upsilon$ then:

By definition of unboxing

$$\downarrow (\text{box}_{\tau'} e)^j \downarrow_{\Upsilon} = \downarrow e \downarrow_{\Upsilon}$$

By definition of unboxing

$$\downarrow (\text{box}_{\tau'} e')^j \downarrow_{\Upsilon} = \downarrow e' \downarrow_{\Upsilon}$$

By induction

$$(\downarrow \rho \downarrow_{\Upsilon}, \downarrow e \downarrow_{\Upsilon}) \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, \downarrow e' \downarrow_{\Upsilon})$$

- If $j \notin \Upsilon$ then:

By definition of unboxing:

$$\begin{aligned} \downarrow (\text{box}_{\tau'} e)^j \downarrow_{\Upsilon} &= (\text{box}_{\downarrow \tau' \downarrow_{\Upsilon}} \downarrow e \downarrow_{\Upsilon})^j \\ \downarrow (\text{box}_{\tau'} e')^j \downarrow_{\Upsilon} &= (\text{box}_{\downarrow \tau' \downarrow_{\Upsilon}} \downarrow e' \downarrow_{\Upsilon})^j \end{aligned}$$

Hence by the induction hypothesis and Lemma 25 we have that:

$$(\downarrow \rho \downarrow_{\Upsilon}, (\text{box}_{\downarrow \tau' \downarrow_{\Upsilon}} \downarrow e \downarrow_{\Upsilon})^j) \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, (\text{box}_{\downarrow \tau' \downarrow_{\Upsilon}} \downarrow e' \downarrow_{\Upsilon})^j)$$

- If $(\rho, (\text{unbox } e)^j) \mapsto (\rho, (\text{unbox } e')^j)$ then let $i = \text{lbl}(e)$ and $i' = \text{lbl}(e')$. By Lemma 24 we have that $i \simeq^{\Upsilon} i'$. By the rules for acceptability, $C; \rho \vdash \rho$ and $C; \rho \vdash e$, so $C; \rho \vdash (\rho, e)$. By the typing rules, $\vdash \rho : \Gamma$ and $\emptyset; \Gamma \vdash e : \tau'$ for some Γ and τ' , so $\vdash (\rho, e) : \tau'$. By the rules for consistency, $\Upsilon \vdash \rho$ and $\Upsilon \vdash e$, so $\Upsilon \vdash (\rho, e)$. So by the induction hypothesis, $(\downarrow \rho \downarrow_{\Upsilon}, \downarrow e \downarrow_{\Upsilon}) \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, \downarrow e' \downarrow_{\Upsilon})$.

- If $i, i' \in \Upsilon$ then:

By definition of unboxing

$$\downarrow (\text{unbox } e)^j \downarrow_{\Upsilon} = \downarrow e \downarrow_{\Upsilon}$$

By definition of unboxing

$$\downarrow (\text{unbox } e')^j \downarrow_{\Upsilon} = \downarrow e' \downarrow_{\Upsilon}$$

By induction

$$(\downarrow \rho \downarrow_{\Upsilon}, \downarrow e \downarrow_{\Upsilon}) \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, \downarrow e' \downarrow_{\Upsilon})$$

- If $i, i' \notin \Upsilon$ then:

By definition of unboxing

$$\downarrow (\text{unbox } e)^j \downarrow_{\Upsilon} = (\text{unbox } \downarrow e \downarrow_{\Upsilon})^j$$

By definition of unboxing

$$\downarrow (\text{unbox } e')^j \downarrow_{\Upsilon} = (\text{unbox } \downarrow e' \downarrow_{\Upsilon})^j$$

By induction

$$(\downarrow \rho \downarrow_{\Upsilon}, \downarrow e \downarrow_{\Upsilon}) \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, \downarrow e' \downarrow_{\Upsilon})$$

By Lemma 25

$$(\downarrow \rho \downarrow_{\Upsilon}, \downarrow (\text{unbox } e)^j \downarrow_{\Upsilon}) \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, \downarrow (\text{unbox } e')^j \downarrow_{\Upsilon})$$

- If $(\rho, (\text{unbox } \langle v^i : \tau \rangle^k)^j) \mapsto (\rho, v^i)$ then:

- If $j \in \Upsilon$ then:

By definition of unboxing

$$\downarrow(\text{unbox } \langle v^i : \tau \rangle^j)^k \downarrow_{\Upsilon} = \downarrow \langle v^i : \tau \rangle^j \downarrow_{\Upsilon} = \downarrow v^i \downarrow_{\Upsilon}$$

So in zero steps

$$(\downarrow \rho \downarrow_{\Upsilon}, \downarrow(\text{unbox } \langle v^i : \tau \rangle^j)^k \downarrow_{\Upsilon}) \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, \downarrow v^i \downarrow_{\Upsilon})$$

▪ If $j \notin \Upsilon$ then:

By definition of unboxing

$$\downarrow(\text{unbox } \langle v^i : \tau \rangle^j)^k \downarrow_{\Upsilon} = (\text{unbox } \langle \downarrow v^i : \tau \rangle^j \downarrow_{\Upsilon})^k$$

By definition of unboxing

$$(\text{unbox } \langle \downarrow v^i : \tau \rangle^j \downarrow_{\Upsilon})^k = (\text{unbox } \langle \downarrow v^i \downarrow_{\Upsilon} : \downarrow \tau \downarrow_{\Upsilon} \rangle^j)^k$$

By definition of reduction

$$(\downarrow \rho \downarrow_{\Upsilon}, (\text{unbox } \langle \downarrow v^i \downarrow_{\Upsilon} : \downarrow \tau \downarrow_{\Upsilon} \rangle^j)^k) \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, \downarrow v^i \downarrow_{\Upsilon})$$

• If $(\rho, \rho'(e_1)^i) \mapsto (\rho, \rho'(e_2)^i)$ then:

By the rules for acceptability, $C; \varrho \vdash \rho'$ and $C; \varrho \vdash e_1$, so $C; \varrho \vdash (\rho', e_1)$. By the typing rules, $\vdash \rho' : \Gamma'$ and $\emptyset; \Gamma' \vdash e_1 : \tau$ for some Γ' , so $\vdash (\rho', e_1) : \tau$. By the rules for consistency, $\Upsilon \vdash \rho'$ and $\Upsilon \vdash e_1$, so $\Upsilon \vdash (\rho', e_1)$. So by the induction hypothesis, $(\downarrow \rho' \downarrow_{\Upsilon}, \downarrow e_1 \downarrow_{\Upsilon}) \mapsto^* (\downarrow \rho' \downarrow_{\Upsilon}, \downarrow e_2 \downarrow_{\Upsilon})$.

By definition of unboxing

$$\downarrow \rho'(e_1)^i \downarrow_{\Upsilon} = \downarrow \rho' \downarrow_{\Upsilon} (\downarrow e_1 \downarrow_{\Upsilon})^i$$

By definition of unboxing

$$\downarrow \rho'(e_2)^i \downarrow_{\Upsilon} = \downarrow \rho' \downarrow_{\Upsilon} (\downarrow e_2 \downarrow_{\Upsilon})^i$$

By induction

$$(\downarrow \rho' \downarrow_{\Upsilon}, \downarrow e_1 \downarrow_{\Upsilon}) \mapsto^* (\downarrow \rho' \downarrow_{\Upsilon}, \downarrow e_2 \downarrow_{\Upsilon})$$

By Lemma 25

$$(\downarrow \rho \downarrow_{\Upsilon}, \downarrow \rho' \downarrow_{\Upsilon} (\downarrow e_1 \downarrow_{\Upsilon})^i) \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, \downarrow \rho' \downarrow_{\Upsilon} (\downarrow e_2 \downarrow_{\Upsilon})^i)$$

• If $(\rho, \rho'(v^i)^j) \mapsto (\rho, v^i)$ then:

By definition of unboxing

$$\downarrow \rho'(v^i)^j \downarrow_{\Upsilon} = \downarrow \rho' \downarrow_{\Upsilon} (\downarrow v^i \downarrow_{\Upsilon})^j$$

Unboxed value is a value, so by reduction rules

$$(\downarrow \rho \downarrow_{\Upsilon}, \downarrow \rho' \downarrow_{\Upsilon} (\downarrow v^i \downarrow_{\Upsilon})^j) \mapsto (\downarrow \rho \downarrow_{\Upsilon}, \downarrow v^i \downarrow_{\Upsilon})$$

■

To show preservation of consistency we need a type substitution lemma.

Lemma 26

If $C \vdash \Upsilon, \Upsilon \vdash \tau : \mathbf{r}$, τ is not a type variable, and $C(\text{lbl}(\tau)) = C(\varrho(\alpha))$ then:

- If $\Upsilon \vdash \tau' : \mathbf{r}$ and $C; \varrho \vdash \tau'$ then $\Upsilon \vdash \tau'[\tau/\alpha] : \mathbf{r}$.
- If $\Upsilon \vdash e$ and $C; \varrho \vdash e$ then $\Upsilon \vdash e[\tau/\alpha]$.
- If $\Upsilon \vdash \rho$ and $C; \varrho \vdash \rho$ then $\Upsilon \vdash \rho[\tau/\alpha]$.

Proof: The proof is by simultaneous induction on the derivation of $\Upsilon \vdash \tau' : \mathbf{r}$, $\Upsilon \vdash e$, and $\Upsilon \vdash \rho$. The cases for expressions and environments are straight forward. Consider the cases for types:

- Case 1, $\tau' = \alpha^i$: If $\tau = \sigma^j$ then $\tau'[\tau/\alpha] = \sigma^i$. Since $C; \varrho \vdash \tau'$, $C(i) = C(\varrho(\alpha))$, so $C(i) = C(j)$. By Lemmas 17 and 18, $i \stackrel{\Upsilon}{\simeq} j$. Then by inspection of the rules, $\Upsilon \vdash \sigma^i : \mathbf{r}$ as $\Upsilon \vdash \sigma^j : \mathbf{r}$.
- Case 2, $\tau' = \beta^i$ and $\alpha \neq \beta$: Then $\tau'[\tau/\alpha] = \tau'$ and the result is immediate.
- Case 3, $\tau' = \mathbf{B}^i$: Then $\Upsilon \vdash \tau' : \mathbf{r}$ is not possible.
- Case 4, $\tau' = (\forall \beta. \tau_1 \rightarrow \tau_2)^i$: Then $\Upsilon \vdash \tau'[\tau/\alpha] : \mathbf{r}$, as required.

• Case 5, $\tau' = (\text{box}(\tau''))^i$, $i \in \Upsilon$: By the rule, $\Upsilon \vdash \tau'' : \mathbf{r}$. Assumption $C; \varrho \vdash \tau'$ requires $C; \varrho \vdash \tau''$. By the induction hypothesis, $\Upsilon \vdash \tau''[\tau/\alpha] : \mathbf{r}$. By the consistency rules, $\Upsilon \vdash \text{box}(\tau''[\tau/\alpha])^i : \mathbf{r}$. By definition of substitution, $\Upsilon \vdash \text{box}(\tau'')^i[\tau/\alpha] : \mathbf{r}$, as required.

• Case 6, $\tau' = (\text{box}(\tau''))^i$, $i \notin \Upsilon$: Then $\Upsilon \vdash \tau'[\tau/\alpha] : \mathbf{r}$, as required. ■

Lemma 27

If $\Upsilon \vdash M_1, C \vdash \Upsilon, C; \varrho \vdash M_1, \vdash M_1 : \tau$, and $M_1 \mapsto M_2$ then $\Upsilon \vdash M_2$.

Proof: The proof is by induction on the derivation of $M_1 \mapsto M_2$. Let $M_1 = (\rho, e_1)$ and $M_2 = (\rho, e_2)$. By the rules for consistency, $\Upsilon \vdash \rho$ and $\Upsilon \vdash e_1$. The result follows if $\Upsilon \vdash e_2$. The typing rules require $\vdash \rho : \Gamma$ and $\emptyset; \Gamma \vdash e_1 : \tau'$ for some Γ and τ' . Assumption $C; \varrho \vdash M$ requires $C; \varrho \vdash \rho$ and $C; \varrho \vdash e_1$. Consider the cases for the last rule used (in the same order as the figure):

- (Variable) In this case: $e_1 = x^i$, $e_2 = v^j$, and $x:\tau' = v^j \in \rho$. By $\Upsilon \vdash \rho$, $\Upsilon \vdash v^j$, as required.
- (Fix expression) In this case: $e_1 = (\text{fix } f[\alpha](x:\tau_1):\tau_2.e)^i$ and $e_2 = \langle \rho, \text{fix } f[\alpha](x:\tau_1):\tau_2.e \rangle^i$. Then $\Upsilon \vdash e_1$ requires $\Upsilon \vdash e$, then since $\Upsilon \vdash \rho$, $\Upsilon \vdash e_2$, as required.
- (Box expression) In this case: $e_1 = (\text{box}_{\tau} v^i)^j$ and $e_2 = \langle v^i : \tau \rangle^j$. Then $\Upsilon \vdash e_1$ requires $\Upsilon \vdash v^i$, so $\Upsilon \vdash e_2$, as required.
- (Application left) In this case, we have $e_1 = (e_3[\tau] e_4)^i$, $e_2 = \langle \rho, e_3[\tau] e_4 \rangle^i$, and $(\rho, e_3) \mapsto (\rho, e_5)$ is a subderivation. Then $\Upsilon \vdash e_1$ requires $\Upsilon \vdash e_3$, $\Upsilon \vdash e_4$, and $\Upsilon \vdash \tau : \mathbf{r}$; $\emptyset; \Gamma \vdash e_1 : \tau'$ requires $\emptyset; \Gamma \vdash e_3 : \tau''$ for some τ'' ; $C; \varrho \vdash e_1$ requires $C; \varrho \vdash e_3$. By the induction hypothesis, $\Upsilon \vdash e_5$, so by the consistency rules, $\Upsilon \vdash e_2$, as required.
- (Application right) In this case: $e_1 = (e_3[\tau] e_4)^i$, $e_2 = \langle \rho, e_3[\tau] e_4 \rangle^i$, and $(\rho, e_3) \mapsto (\rho, e_5)$ is a subderivation. Then $\Upsilon \vdash e_1$ requires $\Upsilon \vdash e_3$, $\Upsilon \vdash e_4$, and $\Upsilon \vdash \tau : \mathbf{r}$; $\emptyset; \Gamma \vdash e_1 : \tau'$ requires $\emptyset; \Gamma \vdash e_4 : \tau''$ for some τ'' ; $C; \varrho \vdash e_1$ requires $C; \varrho \vdash e_4$. By the induction hypothesis, $\Upsilon \vdash e_5$, so by the consistency rules, $\Upsilon \vdash e_2$, as required.
- (Application beta) In this case:

$$\begin{aligned} e_1 &= (v_1^i[\tau] v_2^j)^k \\ v_1 &= \langle \rho', \text{fix } f[\alpha](x:\tau_1):\tau_2.e \rangle \\ e_2 &= \rho''(e[\tau/\alpha])^k \\ \rho'' &= \rho', f:\tau' = v_1^i, x:\tau'_1 = v_2^j \\ \tau' &= (\forall \alpha. \tau_1 \rightarrow \tau_2)^i \\ \tau'_1 &= \tau_1[\tau/\alpha] \end{aligned}$$

By $\Upsilon \vdash e_1$ and the rules for consistency, $\Upsilon \vdash v_1^i$, $\Upsilon \vdash \rho'$, $\Upsilon \vdash e$, $\Upsilon \vdash \tau : \mathbf{r}$, and $\Upsilon \vdash v_2^j$. Thus by the rules for consistency, $\Upsilon \vdash \rho''$. By the typing rule $\emptyset \vdash \tau' \text{ wf}$, so τ' cannot be a type variable. Assumption $C; \varrho \vdash M$ requires $C; \varrho \vdash e$ and, as in previous proofs, $C(\text{lbl}(\tau)) = C(\varrho(i))$. By Lemma 26, $\Upsilon \vdash e[\tau/\alpha]$. By the rules for consistency, $\Upsilon \vdash e_2$, as required.

- (Under box) Similar to application left.
- (Under unbox) Similar to application left.
- (Unbox beta) In this case: $e_1 = (\text{unbox } \langle v^i : \tau \rangle^j)^k$ and $e_2 = v^i$. Then $\Upsilon \vdash e_1$ requires $\Upsilon \vdash v^i$, as required.
- (Under frame) Similar to application left.

- (Frame return) In this case: $e_1 = \rho'(v^i)^j$ and $e_2 = v^i$. Then $\Upsilon \vdash e_1$ requires $\Upsilon \vdash v^i$, as required. ■

With these lemmas we can prove our semantics preservation result.

Theorem 5 (Coherence)

- If $\vdash M : \tau, C; \varrho \vdash M, C \vdash \Upsilon, \Upsilon \vdash M$, and $M \mapsto^* (\rho, v^i)$ then $\downarrow M \downarrow_{\Upsilon} \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, \downarrow v^i \downarrow_{\Upsilon})$.
- If $\vdash M : \tau, C; \varrho \vdash M, C \vdash \Upsilon, \Upsilon \vdash M$, and $M \mapsto \dots$ then $\downarrow M \downarrow_{\Upsilon} \mapsto \dots$.

Proof:

- By induction on reduction derivations, using Theorem 4.
 1. If $M \mapsto^* (\rho, v^i)$ in zero steps, then the result follows immediately.
 2. If $M \mapsto^* (\rho, v^i)$ in n steps, then by definition, $M \mapsto M'$ and $M' \mapsto^* (\rho, v^i)$ in $n - 1$ steps.

By Theorem 4
 $\downarrow M \downarrow_{\Upsilon} \mapsto \downarrow M' \downarrow_{\Upsilon}$

By Theorem 2
 $\vdash M' : \tau'$

By Lemma 15
 $C; \varrho \vdash M'$

By Lemma 27
 $\Upsilon \vdash M'$

By induction
 $\downarrow M' \downarrow_{\Upsilon} \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, \downarrow v^i \downarrow_{\Upsilon})$

By the definition of many step reduction
 $\downarrow M \downarrow_{\Upsilon} \mapsto^* (\downarrow \rho \downarrow_{\Upsilon}, \downarrow v^i \downarrow_{\Upsilon})$

- In the operational semantics, there are six leaf reductions. Two of them take expression forms to value forms, but otherwise leave the term unchanged. One of them takes unbox of box of a value to that value. One of them takes a frame of a value to that value. Thus if we measure a term by adding its size, number of lambda expressions, and number of box expressions, then this metric strictly decreases for these three leaf reductions. Therefore, in any infinite reduction sequence, there must be an infinite number of steps whose leaf reduction is a variable reduction or an application beta reduction. Then observe in the proof of Theorem 4 that the unboxing of a variable redex or of an application beta redex will always take a step, and that Lemma 25 preserves this. Thus the unboxing will also take an infinite number of steps. ■

Theorem 5 shows that if a program reduces to a value then its unboxing reduces to the unboxed value given that the analysis is acceptable and the unboxing is consistent; and that if a program diverges then its unboxing diverges. In other words, for an acceptable analysis and a consistent unboxing, the induced unboxing function preserves the semantics of the original program up to elimination of boxes on final values.

5. Construction of an acceptable unboxing

The previous section gives a declarative specification for when an unboxing set Υ is correct but does not specify how such a set might be chosen. In this section we give a simple algorithm for constructing an unboxing given an arbitrary acceptable flow analysis, and show that the unboxing produced by this algorithm is consistent, and hence correct.

The idea behind the algorithm is that we use the results of a flow analysis to construct the connected components of the interprocedural flow graph of a program. All of the elements of a connected component will then either be unboxed together, or not unboxed at all. Any such choice of unboxing (as we will show) satisfies the cache coherence property. The only remaining requirement is that the choice of unboxing set be consistent, which is easily satisfied by ensuring that any connected component that includes a type passed to a polymorphic function is only unboxed if the unboxing of the type argument still has traceability \mathbf{r} .

For the purposes of this section we ignore environments and the intermediate forms $\rho(e)$, $\langle \rho, \text{fix } f[\alpha](x:\tau_1):\tau_2.e \rangle^j$ and $\langle v^i:t \rangle^j$. These constructs are present in the language solely as mechanisms to discuss the operational semantics—they can be thought of as intermediate terms, rather than source terms. It is straightforward to incorporate these into the algorithm if desired.

Given a flow analysis (C, ϱ) and program e such that $C; \rho \vdash e$, we define the induced undirected flow graph \mathcal{FG} as an undirected graph with a node for every label in C . For every label i and every shape $s \in C(i)$, we add an edge to \mathcal{FG} between i and $\text{lbl}(s)$. Informally, these edges simply connect up each program point with all of its reaching definitions.

Given a flow graph \mathcal{FG} , we can find the connected components in the usual way. Let \mathcal{CC} be a mapping that maps labels to the connected component in which they occur. Note that by definition each label occurs in exactly one connected component. It is easy to show that any connected component is cache consistent, and therefore that any set consisting of a union of connected components of the induced flow graph is cache consistent.

Lemma 28 (Cache consistency of a connected component)

Given any acceptable analysis (C, ϱ) with induced flow graph \mathcal{FG} , and any connected component S of \mathcal{FG} , S is cache consistent: that is, $C \vdash S$.

Proof: To show that $C \vdash S$ we must show that $\forall i, s : s \in C(i) \implies i \stackrel{S}{\simeq} \text{lbl}(s)$. But note that by the construction of the induced flow graph \mathcal{FG} , whenever $s \in C(i)$ there is an edge between i and $\text{lbl}(s)$, and consequently by definition of a connected component, i and $\text{lbl}(s)$ must be in the same connected component. Since every label occurs in exactly one connected component, either both i and $\text{lbl}(s)$ are in S or both are not in S . By definition then, $i \stackrel{S}{\simeq} \text{lbl}(s)$. ■

Lemma 29 (Cache consistency (unary) closure)

Given any acceptable analysis (C, ϱ) and disjoint label sets S_1 and S_2 , then if $C \vdash S_1$ and $C \vdash S_2$ then $C \vdash S_1 \cup S_2$

Proof: To show that $C \vdash S_1 \cup S_2$ we must show that $\forall i, s : s \in C(i) \implies i \stackrel{S_1 \cup S_2}{\simeq} \text{lbl}(s)$. Consider an arbitrary label i . If i is not in $S_1 \cup S_2$, then we have that i is not in S_1 and not in S_2 , and hence by assumption, $\text{lbl}(s)$ is not in S_1 and not in S_2 , and hence we have agreement. If i is in $S_1 \cup S_2$, then it must be in either S_1 or S_2 . WLOG, assume that $i \in S_1$. By assumption, $i \stackrel{S_1}{\simeq} \text{lbl}(s)$, and so $\text{lbl}(s) \in S_1$, and hence $\text{lbl}(s) \in S_1 \cup S_2$ and we have agreement. ■

Lemma 30 (Cache consistency closure)

Given any acceptable analysis (C, ϱ) with induced flow graph \mathcal{FG} , and any set \mathcal{SS} of connected components of \mathcal{FG} , $\bigcup \mathcal{SS}$ is cache consistent.

Proof: By Lemma 28, each connected component is cache consistent. By definition, any two connected components are disjoint, and

so by Lemma 29 the union of any two connected components are cache consistent, and are disjoint from any other connected component. The cache consistency of $\bigcup SS$ follows directly by induction. ■

5.1 The algorithm

Given the set of connected components for the induced flow graph, the algorithm begins with an initial unboxing set Υ consisting of the union of all of the connected components. By Lemma 30, we have that $C \vdash \Upsilon$. The algorithm then proceeds by considering in turn each application sub-term $e_1[\tau]e_2$ as follows:

- For each sub-term of e of the form $e_1[\tau]e_2$:
- if $\text{lbl}(\tau) \in \Upsilon$, and if $\Upsilon \vdash \tau : \mathbf{b}$, then:
 - $\Upsilon \leftarrow \Upsilon - \mathcal{CC}(\text{lbl}(\tau))$.

That is, for any application for which the current unboxing results in the type argument being unboxed to a non-reference type, we remove the connected component for the type from the unboxing set. Note that after removing a connected component from Υ , the new unboxing set Υ is still cache consistent since it is still a union of connected components.

With the help of some technical lemmas, it is straightforward to show that the final unboxing set Υ computed by the algorithm is a consistent unboxing for the program, and hence that by construction the specification defined in Section 4 is a useful one in the sense that it is satisfiable.

To begin with, we observe that if a type's label is not in the unboxing set Υ , then it is consistent and its traceability is unchanged by the unboxing.

Lemma 31 (Type consistency)

For any unboxing set Υ and type τ , if $\text{lbl}(\tau) \notin \Upsilon$ then $\Upsilon \vdash \tau : \text{tr}(\tau)$.

Proof: By inspection.

- (Variable) $\text{tr}(\alpha^i) = \mathbf{r}$, and $\Upsilon \vdash \alpha^i : \mathbf{r}$.
- (Base type) $\text{tr}(\mathbf{B}^i) = \mathbf{b}$, and $\Upsilon \vdash \mathbf{B}^i : \mathbf{b}$.
- (Fun type) $\text{tr}(\forall \alpha. \tau_1 \rightarrow \tau_2^i) = \mathbf{r}$, and $\Upsilon \vdash \forall \alpha. \tau_1 \rightarrow \tau_2^i : \mathbf{r}$.
- (Box type) $\text{tr}(\text{box}(\tau')^i) = \mathbf{r}$, and by assumption $i \notin \Upsilon$, so we have that $\Upsilon \vdash \text{box}(\tau')^i : \mathbf{r}$. ■

It is also the case that the consistent type judgement defines a total function on types, and hence for any type we either have that it is consistent at traceability \mathbf{r} or that it is consistent at traceability \mathbf{b} .

Lemma 32 (Type consistency is a total function)

For any unboxing set Υ and type τ , either $\Upsilon \vdash \tau : \mathbf{b}$, or $\Upsilon \vdash \tau : \mathbf{r}$.

Proof: By induction on types. All of the cases follow immediately except when $\tau = \text{box}(\tau')^i$ and $i \in \Upsilon$. In that case, by induction we have that either $\Upsilon \vdash \tau' : \mathbf{b}$, or $\Upsilon \vdash \tau' : \mathbf{r}$, and so by construction either $\Upsilon \vdash \tau : \mathbf{b}$, or $\Upsilon \vdash \tau : \mathbf{r}$. ■

Theorem 6

If $\Delta; \Gamma \vdash e : \tau$, $C; \varrho \vdash \Gamma$, and $C; \varrho \vdash e$ and if Υ is the unboxing set computed by the algorithm in this section, then Υ is a consistent unboxing for e . That is, $C \vdash \Upsilon$ and $\Upsilon \vdash e$.

Proof: The conclusion that $C \vdash \Upsilon$ follows almost immediately from Lemma 30. The initial choice of Υ is a union of connected components, and hence is cache consistent. At every step of the algorithm, we may remove a single connected component from Υ .

The result is still a union of connected components (since connected components are disjoint), and hence the result of removing a connected component is still cache consistent by Lemma 30.

The conclusion that $\Upsilon \vdash e$ follows by induction on the structure of the typing derivation.

- (Variable) In this case, $e = x^i$, consistency is immediate.
- (Fix) In this case $e = (\text{fix } f[\alpha](x:\tau_1):\tau_2.e')^i$. To get consistency, we must show that $\Upsilon \vdash e'$. The last rule applied in the typing judgement must have been the fix rule, and by its premises we have that $\Delta \vdash \forall \alpha. \tau_1 \rightarrow \tau_2^i$ wf (1), and that $\Delta, \alpha; \Gamma, f:\forall \alpha. \tau_1 \rightarrow \tau_2^i, x:\tau_1 \vdash e' : \tau_2$ (2). The last rule applied in the acceptable analysis judgement must also have been the fix rule, and by its premises we have that $C; \varrho \vdash e'$ (3). To apply the induction hypothesis, we need (1), (3), and that $C; \varrho \vdash \Gamma, f:\forall \alpha. \tau_1 \rightarrow \tau_2^i, x:\tau_1$ (4). To show (4), it is sufficient to show that:
 - $\varrho(f) = i$ which is a premise of the acceptable analysis derivation
 - $C; \varrho \vdash \forall \alpha. \tau_1 \rightarrow \tau_2^i$ which is a premise of the acceptable analysis derivation
 - $\varrho(x) = \text{lbl}(\tau_1)$ which is a premise of the acceptable analysis derivation
 - $C; \varrho \vdash \tau_1$ which is a sub-premise of the derivation of $C; \varrho \vdash \forall \alpha. \tau_1 \rightarrow \tau_2^i$.

So by (1), (3), and (4), we have by induction that $\Upsilon \vdash e'$.

- (Application) In this case $e = (e_1[\tau]e_2)^i$. To prove consistency, we need that $\Upsilon \vdash e_1$ (1), $\Upsilon \vdash e_2$ (2), and $\Upsilon \vdash \tau : \mathbf{r}$ (3). Inverting the typing derivation and the acceptable analysis derivation immediately gives us the premises we need to apply the induction hypothesis to get (1) and (2). To prove (3), note that a premise of the typing derivation gives us that $\text{tr}(\tau) = \mathbf{r}$ (4). If $\text{lbl}(\tau) \notin \Upsilon$, then by Lemma 31 we have that $\Upsilon \vdash \tau : \text{tr}(\tau)$ and so by (4) we're done. If $\text{lbl}(\tau) \in \Upsilon$, then by the definition of the algorithm, we must have that $\Upsilon \vdash \tau : \mathbf{b}$ does not hold (since otherwise the algorithm would have removed the connected component containing $\text{lbl}(\tau)$ from Υ), and so by Lemma 32 we must have that $\Upsilon \vdash \tau : \mathbf{r}$ and we're done.
- (Box) All of the premises need to apply the induction hypothesis are available immediately by inverting the typing derivation and the acceptable analysis derivation.
- (Unbox) All of the premises need to apply the induction hypothesis are available immediately by inverting the typing derivation and the acceptable analysis derivation.
- (Constant) Follows immediately. ■

6. Open Terms

Up to this point the framework we have developed has implicitly been restricted to whole-program optimization in the sense that it is built around closed terms. In practice, it is important to be able to optimize program fragments (modules) where we have a part of the program that may refer to other pieces not available for analysis, and may in turn export itself for use by other program fragments. Since we do not wish to assume anything about the compilation of the code to which a fragment is linked, such a setting adds the additional correctness criterion that since we do not have access to the rest of the program, anything that flows across the boundary to or from the rest of the program must retain its original boxed representation. Informally, we can easily ensure this by simply

$$\begin{array}{c}
\boxed{\vdash (\Gamma \Rightarrow e : \tau) \text{ wf}} \\
\frac{\emptyset \vdash \Gamma \text{ wf} \quad \emptyset; \Gamma \vdash e : \tau}{\vdash (\Gamma \Rightarrow e : \tau) \text{ wf}} \\
\boxed{C; \varrho \vdash (\Gamma \Rightarrow e : \tau)} \\
\frac{C; \varrho \vdash^s \Gamma \quad C; \varrho \vdash^s e \quad C; \varrho \vdash^s \tau}{C; \varrho \vdash (\Gamma \Rightarrow e : \tau)} \\
\boxed{\downarrow (\Gamma \Rightarrow e : \tau) \downarrow_{\Upsilon}} \\
\downarrow (\Gamma \Rightarrow e : \tau) \downarrow_{\Upsilon} = (\Gamma \Rightarrow \downarrow e \downarrow_{\Upsilon} : \tau) \\
\boxed{\Upsilon \vdash \tau \text{ not unboxed}} \\
\frac{i \notin \Upsilon}{\Upsilon \vdash \alpha^i \text{ not unboxed}} \quad \frac{i \notin \Upsilon}{\Upsilon \vdash \mathbf{B}^i \text{ not unboxed}} \\
\frac{i \notin \Upsilon \quad \Upsilon \vdash \tau_1 \text{ not unboxed} \quad \Upsilon \vdash \tau_2 \text{ not unboxed}}{\Upsilon \vdash (\forall \alpha. \tau_1 \rightarrow \tau_2)^i \text{ not unboxed}} \\
\frac{i \notin \Upsilon \quad \Upsilon \vdash \tau \text{ not unboxed}}{\Upsilon \vdash \mathbf{box}(\tau)^i \text{ not unboxed}} \\
\boxed{\Upsilon \vdash \Gamma \text{ not unboxed}} \\
\frac{\forall 1 \leq j \leq n : \Upsilon \vdash \tau_j \text{ not unboxed}}{\Upsilon \vdash x_1 : \tau_1, \dots, x_n : \tau_n \text{ not unboxed}} \\
\boxed{\Upsilon \vdash (\Gamma \Rightarrow e : \tau)} \\
\frac{\Upsilon \vdash \Gamma \text{ not unboxed} \quad \Upsilon \vdash e \quad \Upsilon \vdash \tau \text{ not unboxed}}{\Upsilon \vdash (\Gamma \Rightarrow e : \tau)}
\end{array}$$

Figure 10. Judgements for modules

$$\begin{array}{c}
\boxed{C; \varrho \vdash^s \tau} \\
\frac{C(\varrho(\alpha)) = C(i) \quad s \in C(i)}{C; \varrho \vdash^s \alpha^i} \\
\boxed{C; \varrho \vdash^s e} \\
\frac{C; \varrho \vdash^s \mathbf{box}(\tau)^i \quad C; \varrho \vdash^s e \quad (\mathbf{box}_{tr(\tau)} \text{lbl}(e))_v^i \in C(i)}{C; \varrho \vdash^s (\mathbf{box}_{\tau} e)^i} \\
\frac{C; \varrho \vdash^s \mathbf{box}(\tau)^i \quad C; \varrho \vdash^s v^j \quad (\mathbf{box}_{tr(\tau)} j)_v^i \in C(i)}{C; \varrho \vdash^s \langle v^j : \tau \rangle^i}
\end{array}$$

Figure 11. Stronger Analysis

requiring that nothing on the boundary is in the unboxing set. In this section we make this requirement precise, define a notion of unboxing for program modules, and prove this extended unboxing correct.

For the purposes of this section, a program fragment is a module of the form $(\Gamma \Rightarrow e : \tau)$. Here Γ specifies the imports of the module, e specifies the body of the module, and τ gives the type of the body. The module is considered to export only one thing—the value that e evaluates to; generalization to multiple exports is straightforward. As discussed above, the requirements of independent compilation forbid us from unboxing any of the imports and the exported value. In this typed setting, this can be achieved directly by forbidding the unboxing of any part of any type in Γ and in τ . Since the types determine the representation of the values that inhabit them, this is sufficient to ensure that values that flow across module boundaries are not unboxed.

To make this precise, we extend the definitions of well typedness, acceptability of flow analysis, unboxing, and consistency of unboxing to modules in Figure 10. For technical reasons, we must very slightly strengthen the definition of acceptability for flow analyses. Specifically, the rules for type variables, box expressions, and box values are replaced with those in Figure 11 while leaving all other rules the same. The new rules for boxes require a more precise shape in the cache in which the labels for the contents of the box match directly. It is likely that any actual flow analysis would use such a shape, and so we do not believe that this requirement is an undue burden. These stronger module rules are not closed under reduction, and hence the rules for programs must be weaker. The stronger condition for type variables is a technical requirement to ensure consistency even in the case that the caches for the type variables would be otherwise uninhabited. It is easy to arrange it such that caches for type variables are always non-empty, and hence to trivially satisfy this requirement.

A suitable notion of correctness for modular unboxing is that a module and its unboxing are contextually equivalent. Rather than define contextual equivalence directly, we use a notion that is usually proven equivalent to contextual equivalence as our definition. Specifically, we say that two expressions are contextually equivalent if in any environment that closes them and in any elimination context for their type, they are observably equivalent. The formal definition is given in Figure 12.

The strategy is that we will take the context and alpha vary it and relabel it so that it is sufficiently distinct from the module. Then we will argue that we can modify the flow analysis and unboxing to cover the context without unboxing any of it. Then by coherence the module in context will behave the same as the unboxing of the module in context, which because the context is not unboxed, will act the same as the unboxed module in context.

First we formalize and prove that the operational semantics is insensitive to the alpha variant and labels used. Let $x \sim_s y$ mean that x and y are alpha variants and possibly relabeled.

Lemma 33

If $M_1 \sim_s M_2$ and $M_1 \mapsto M_3$ then there exists M_4 such that $M_3 \sim_s M_4$ and $M_2 \mapsto M_4$.

Proof: The proof is by a straight forward induction on the derivation of $M_1 \mapsto M_3$. ■

Next we prove three lemmas about unboxing preservation. In the first two we show that something's unboxing is that something because either the not unboxed judgement (the first lemma) or the labels in the something are not in the unboxing set (the second lemma). In the third we show the unboxing of an expression is the same if the unboxing set is the same on the labels in the expression. To state and prove these and subsequent lemmas we need a function

$$\begin{array}{c}
E ::= \langle \rangle \mid (E[\tau] e)^i \mid (\text{unbox } E)^i \\
\hline
\Gamma \vdash \langle \rangle : \tau \langle \tau \rangle \\
\hline
\Gamma \vdash E : (\forall \alpha. \tau_1 \rightarrow \tau_2)^j \langle \tau \rangle \quad \emptyset \vdash \tau \text{ wf} \quad \text{tr}(\tau) = \mathbf{r} \quad \emptyset; \Gamma \vdash e : \tau'_1 \quad \vdash \tau_1[\tau/\alpha] = \tau'_1 \quad \Gamma \vdash E : \text{box}(\tau')^j \langle \tau \rangle \\
\hline
\Gamma \vdash (E[\tau] e)^i : \tau_2[\tau/\alpha] \langle \tau \rangle \quad \Gamma \vdash (\text{unbox } E)^i : \tau' \langle \tau \rangle \\
\hline
M_1 \stackrel{\text{obs}}{\equiv} M_2 \quad \stackrel{\text{def}}{=} (\forall c, i : (M_1 \mapsto^* c^i \Leftrightarrow M_2 \mapsto^* c^i)) \wedge (M_1 \mapsto \dots \Leftrightarrow M_2 \mapsto \dots) \\
\Gamma \vdash e_1 \equiv e_2 : \tau \quad \stackrel{\text{def}}{=} \emptyset; \Gamma \vdash e_1 : \tau \wedge \emptyset; \Gamma \vdash e_2 : \tau \wedge \\
\forall \rho, E : \vdash \rho : \Gamma \wedge \Gamma \vdash E : \mathbf{B}^i \langle \tau \rangle \implies (\rho, E \langle e_1 \rangle) \stackrel{\text{obs}}{\equiv} (\rho, E \langle e_2 \rangle) \\
\vdash (\Gamma_1 \Rightarrow e_1 : \tau_1) \equiv (\Gamma_2 \Rightarrow e_2 : \tau_2) \quad \stackrel{\text{def}}{=} \Gamma_1 = \Gamma_2 \wedge \tau_1 = \tau_2 \wedge (\Gamma_1 \vdash e_1 \equiv e_2 : \tau_1)
\end{array}$$

Figure 12. Contextual Equivalence

$$\begin{array}{ll}
\text{lbls}(\sigma^i) & = \{i\} \cup \text{lbls}(\sigma) \\
\text{lbls}(\alpha) & = \emptyset \\
\text{lbls}(\mathbf{B}) & = \emptyset \\
\text{lbls}(\forall \alpha. \tau_1 \rightarrow \tau_2) & = \text{lbls}(\tau_1) \cup \text{lbls}(\tau_2) \\
\text{lbls}(\text{box}(\tau)) & = \text{lbls}(\tau) \\
\text{lbls}(m^i) & = \{i\} \cup \text{lbls}(m) \\
\text{lbls}(v^i) & = \{i\} \cup \text{lbls}(v) \\
\text{lbls}(x) & = \emptyset \\
\text{lbls}(\mathbf{fix} \ f[\alpha](x:\tau_1):\tau_2.e) & = \text{lbls}(\tau_1) \cup \text{lbls}(\tau_2) \cup \text{lbls}(e) \\
\text{lbls}(e_1[\tau] e_2) & = \text{lbls}(e_1) \cup \text{lbls}(\tau) \cup \text{lbls}(e_2) \\
\text{lbls}(\text{box}_\tau e) & = \text{lbls}(\tau) \cup \text{lbls}(e) \\
\text{lbls}(\text{unbox } e) & = \text{lbls}(e) \\
\text{lbls}(\rho(e)) & = \text{lbls}(\rho) \cup \text{lbls}(e) \\
\text{lbls}(c) & = \emptyset \\
\text{lbls}(\langle \rho, \mathbf{fix} \ f[\alpha](x:\tau_1):\tau_2.e \rangle) & = \text{lbls}(\rho) \cup \text{lbls}(\tau_1) \cup \text{lbls}(\tau_2) \cup \text{lbls}(e) \\
\text{lbls}(\langle v^i:\tau \rangle) & = \text{lbls}(\tau) \cup \text{lbls}(v^i) \\
\text{lbls}(x_1:\tau_1 = v_1^{i_1}, \dots, x_n:\tau_n = v_n^{i_n}) & = \cup_{1 \leq i \leq n} \text{lbls}(\tau_j) \cup \text{lbls}(v_j^{i_j})
\end{array}$$

Figure 13. The labels in an type, expression, or environment

to return all the labels in an expressions, type, or environment. It is defined in Figure 13. ■

Lemma 34

- If $\Upsilon \vdash \tau$ not unboxed then $\downarrow \tau \downarrow_{\Upsilon} = \tau$.
- If $\Upsilon \vdash \Gamma$ not unboxed then $\downarrow \Gamma \downarrow_{\Upsilon} = \Gamma$.

Proof:

- The proof is by induction on the structure of τ . Consider the cases:
 - Case 1, $\tau = \alpha^i$: Then by definition $\downarrow \tau \downarrow_{\Upsilon} = \tau$, as required.
 - Case 2, $\tau = \mathbf{B}^i$: Then by definition $\downarrow \tau \downarrow_{\Upsilon} = \tau$, as required.
 - Case 3, $\tau = (\forall \alpha. \tau_1 \rightarrow \tau_2)^i$: Then $\Upsilon \vdash \tau$ not unboxed requires $\Upsilon \vdash \tau_1$ not unboxed and $\Upsilon \vdash \tau_2$ not unboxed. By the induction hypothesis, $\downarrow \tau_1 \downarrow_{\Upsilon} = \tau_1$ and $\downarrow \tau_2 \downarrow_{\Upsilon} = \tau_2$. By definition, $\downarrow \tau \downarrow_{\Upsilon} = \tau$, as required.
 - Case 4, $\tau = \text{box}(\tau')^i$: Then $\Upsilon \vdash \tau$ not unboxed requires $i \notin \Upsilon$ and $\Upsilon \vdash \tau'$ not unboxed. By the induction hypothesis, $\downarrow \tau' \downarrow_{\Upsilon} = \tau'$. By definition, $\downarrow \tau \downarrow_{\Upsilon} = \tau$, as required.
- If $\Gamma = x_1:\tau_1, \dots, x_n:\tau_n$ then: $\Upsilon \vdash \Gamma$ not unboxed requires $\Upsilon \vdash \tau_j$ not unboxed for $1 \leq j \leq n$. So by the first item, $\downarrow \tau_j \downarrow_{\Upsilon} = \tau_j$ for $1 \leq j \leq n$. Then by definition, $\downarrow \Gamma \downarrow_{\Upsilon} = \Gamma$, as required.

Lemma 35

- If $\text{lbls}(\rho) \cap \Upsilon = \emptyset$ then $\downarrow \rho \downarrow_{\Upsilon} = \rho$.
- If $\text{lbls}(E) \cap \Upsilon = \emptyset$ then $\downarrow E \downarrow_{\Upsilon} = E$.

Proof: The proof is a straight forward induction on the structure of ρ and E . ■

Lemma 36

If $\Upsilon_1 \cap \text{lbls}(e) = \Upsilon_2 \cap \text{lbls}(e)$ then $\downarrow e \downarrow_{\Upsilon_1} = \downarrow e \downarrow_{\Upsilon_2}$.

Proof: The proof is a straight forward induction on the structure of e . ■

Next we state and prove our main technical lemma. This lemma states that we can rewrite the context and flow analysis to have certain desirable properties, namely that the flow analysis covers the context and the module, that the context is not unboxed, that the module is unboxed as before, and the unboxing set and flow analysis remain consistent and consistent with the module and context.

Lemma 37

If:

$$\begin{aligned}
& \emptyset \vdash \Gamma \text{ wf} \\
& \emptyset; \Gamma \vdash e : \tau \\
& C; \varrho \vdash \Gamma \\
& C; \varrho \vdash e \\
& C; \varrho \vdash \tau \\
& C \vdash \Upsilon \\
& \Upsilon \vdash \Gamma \text{ not unboxed} \\
& \Upsilon \vdash e \\
& \Upsilon \vdash \tau \text{ not unboxed} \\
& \vdash \rho : \Gamma \\
& \Gamma \vdash E : \mathbb{B}^i \langle \tau \rangle
\end{aligned}$$

then there exists ρ', E', C', ϱ' , and Υ' such that:

$$\begin{aligned}
& \rho \sim_s \rho' \\
& E \sim_s E' \\
& \vdash \rho' : \Gamma \\
& \Gamma \vdash E' : \mathbb{B}^j \langle \tau \rangle \\
& C'; \varrho' \vdash (\rho', E' \langle e \rangle) \\
& C' \vdash \Upsilon' \\
& \Upsilon' \vdash (\rho', E' \langle e \rangle) \\
& \text{lbls}(\rho') \cap \Upsilon' = \emptyset \\
& \text{lbls}(E') \cap \Upsilon' = \emptyset \\
& \Upsilon \cap \text{lbls}(e) = \Upsilon' \cap \text{lbls}(e)
\end{aligned}$$

Proof: Let V be the set of variables that occur in e . Let A be the set of type variables that occur in Γ , e , or τ . Both these sets are finite.

The derivation of $C; \varrho \vdash \Gamma$, $C; \varrho \vdash e$, and $C; \varrho \vdash \tau$ will for each type that is not a type variable require a particular type shape with some label on it in the cache of the label of that type, similarly for each box expression and box value require a box shape with some label of its contents in the cache. Let L be one such label for each such type and such box as well as $\varrho(V) \cup \varrho(A) \cup \text{lbls}(\Gamma) \cup \text{lbls}(e) \cup \text{lbls}(\tau)$. Note that L is a finite set.

Let ϱ' be ϱ on V and A and on every other variable or type variable let it map to a fresh label (distinct from each other and from L). Define:

$$C''(i) = \begin{cases} \{s \mid s \in C(i) \wedge \text{lbls}(s) \subseteq L\} & i \in L \\ \emptyset & i \notin L \end{cases}$$

Claim: $C''; \varrho' \vdash \Gamma$, $C''; \varrho' \vdash e$, and $C''; \varrho' \vdash \tau$. The proof is by induction on the derivation, consider the last rule used:

- (Variable) In this case $e = x^i$ and $C(\varrho(x)) \subseteq C(\varrho(x))$. Since $x \in V$, $\varrho'(x) = \varrho(x)$ and $\varrho(x) \in L$. Also $i \in L$. Therefore, $C''(\varrho'(x)) \subseteq C''(i)$, as required. Thus by the same rule, $C''; \varrho' \vdash e$, as required.
- (Fix expression) In this case $e = (\text{fix } f[\alpha](x:\tau_1):\tau_2.e')^i$, $\varrho(f) = i$, $\varrho(x) = \text{lbl}(\tau_1)$, $C; \varrho \vdash (\forall \alpha.\tau_1 \rightarrow \tau_2)^i$, $C; \varrho \vdash e'$, and $(\forall \varrho(\alpha).\text{lbl}(\tau_1) \rightarrow \text{lbl}(e'))_v^{\subseteq} C(i)$. Since $f, x \in V$ and $\alpha \in A$, $\varrho'(f) = \varrho(f)$, $\varrho'(x) = \varrho(x)$, $\varrho'(\alpha) = \varrho(\alpha)$, and $\varrho(\alpha) \in L$. By the induction hypothesis, $C''; \varrho' \vdash (\forall \alpha.\tau_1 \rightarrow \tau_2)^i$ and $C''; \varrho' \vdash e'$. Since $\varrho'(\alpha) \in L$, $\text{lbl}(\tau_1) \in L$, $\text{lbl}(e) \in L$, and $i \in L$, $(\forall \varrho'(\alpha).\text{lbl}(\tau_1) \rightarrow \text{lbl}(e'))_v^{\subseteq} C''(i)$. Thus by the same rule, $C''; \varrho' \vdash e$, as required.
- (Application) In this case $e = (e_1[\tau] e_2)^i$, $C; \varrho \vdash e_1$, $C; \varrho \vdash \tau$, $C; \varrho \vdash e_2$, and $\text{fun}(C; \text{lbl}(e_1), \text{lbl}(\tau), \text{lbl}(e_2), i)$. By the induction hypothesis, $C'; \varrho' \vdash e_1$, $C'; \varrho' \vdash \tau$, and $C'; \varrho' \vdash e_2$. Since $\text{lbl}(e_1) \in L$, $\text{lbl}(\tau) \in L$, $\text{lbl}(e_2) \in L$, and $i \in L$, it is easy to see that $\text{fun}(C; \text{lbl}(e_1), \text{lbl}(\tau), \text{lbl}(e_2), i)$ (for C''). Thus by the same rule, $C''; \varrho' \vdash e$, as required.
- Other cases are similar ...

Let A' be the set of type variables that appear in Γ and τ . We construct ρ' and E' as alpha variants and relabelings of ρ and E as follows. Since $\vdash \rho : \Gamma$, ρ contains Γ , so we keep that part the same. Type variables that are in A' we keep the same. All other type variables and variables we pick an alpha variant that is fresh (distinct from each other and from A respectively V). The outermost label on types on variables we relabel to the binding label for that variable. All other labels we relabel to be fresh. Clearly $\rho \sim_s \rho'$ and $E \sim_s E'$.

Claim: $\vdash \rho' : \Gamma$ and $\Gamma \vdash E' : \mathbb{B}^j \langle \tau \rangle$ for some j . The proof is a straight forward induction on the structure of ρ' and E' .

Now we need to build a C' such that $C'; \varrho' \vdash (\rho', E' \langle e \rangle)$. We start from C'' . First we add into the caches, shapes required directly for the rules for $C'; \varrho' \vdash \rho'$ and $C'; \varrho' \vdash E'$ (such things are already there for e). In the case of types we add shapes using the label of the type as the label of the shape. In the case of box expressions and values we use the label of the contents of the box as the label of the contents of the shape. What remains is a bunch of subset and equality constraints between cache entries, so we pick C' to be the smallest larger cache that satisfies these constraints. Clearly such a C' exists and by construction, $C'; \varrho' \vdash (\rho', E' \langle e \rangle)$.

Set $\Upsilon' = \Upsilon \cap L$. Clearly, $\Upsilon \cap \text{lbls}(e) = \Upsilon' \cap \text{lbls}(e)$ as $\text{lbls}(e) \subseteq L$. By construction, the labels of ρ' and E' are in the labels of Γ or τ or are not in L . Since $\Upsilon \vdash \Gamma$ not unboxed and $\Upsilon \vdash \tau$ not unboxed, the labels of Γ and τ are not in Υ . Therefore, $\text{lbls}(\Gamma) \cap \Upsilon' = \emptyset$ and $\text{lbls}(\tau) \cap \Upsilon' = \emptyset$. In fact, if A'' is a set of type variables in ρ' and E' then $\varrho'(A'') \cap \Upsilon' = \emptyset$ too.

Claim: any flow from the interface to a $\text{box}(C; i, j)$ condition has a box type at the interface (**), and similarly for $\text{fun}(C; i, j, k, l)$. The proof is by induction on the flow conditions noting that in all cases the two end points have the same type.

Claim: $C' \vdash \Upsilon'$. Let s and i be such that $s \in C'(i)$. If $s \in C''(i)$ then $s \in C(i)$, $i \in L$, and $\text{lbls}(s) \subseteq L$, and in particular, $\text{lbl}(s) \in L$. Since $C \vdash \Upsilon$, $i \stackrel{\Upsilon'}{\simeq} \text{lbl}(s)$. Since $\Upsilon' = \Upsilon \cap L$, $i \in L$, and $\text{lbl}(s) \in L$, $i \stackrel{\Upsilon'}{\simeq} \text{lbl}(s)$, as required. Otherwise, we claim that $i, \text{lbl}(s) \notin \Upsilon'$. Let $L_C = \text{lbls}(\rho') \cup \text{lbls}(E') \cup \varrho'(A'')$, $L_I = \text{lbls}(\Gamma) \cup \text{lbls}(\tau)$, $L_M = L - L_C$, and $L_A = L_C \cup L$. First notice that C'' has entries only for labels in L and with shapes whose labels are in L . The first part of computing C' added shapes to cache entries for labels in L_C with shapes whose labels are in L_C . The second part of computing C' only propagates existing shapes from one cache entry to another, and only from/to cache entries in L_A or in labels in shapes in the cache entries. Thus, the cache entries of C' are only for L_A with shapes with labels in L_A . If $\text{lbl}(s) \in L_C$ then by previous argument $\text{lbl}(s) \notin \Upsilon'$, as required. If $\text{lbl}(s) \in L_M$ then we will show that $s \in C''(j)$ for some $j \in L_I$. Then $s \in C(j)$, and since $C \vdash \Upsilon$ and $j \notin \Upsilon$, $\text{lbl}(s) \notin \Upsilon$ so $\text{lbl}(s) \notin \Upsilon'$, as required. If $i \in L_C$ then by previous argument $i \notin \Upsilon'$, as required. If $i \in L_M$ then we will show that $C''(j) \subseteq C''(i)$ for some $j \in L_I$. Then since $C''(j)$ is inhabited because it labels a type checked in $C''; \varrho' \vdash \Gamma$ or $C''; \varrho' \vdash \tau$, and since that required type shape has labels in L , $\emptyset \neq C(j) \subseteq C(i)$. Then since $C \vdash \Upsilon$ and $j \notin \Upsilon$, $i \notin \Upsilon$, so $i \notin \Upsilon'$, as required. It remains to show the two conditions we claimed. Since C' was computed using a least fixed point, we prove these claims by induction on when s was added to $C'(i)$. Consider the cases:

- Case 1, s was added to i because $s \in C'(j)$, $s \notin C'(i)$, and $C'(j) \subseteq C(i)$ is required by the rules for variables, box expressions, frames, box values, or environments. In this case, i and j have to come from the same term, that is, either $i, j \in L_C$ or $i, j \in L$. If $\text{lbl}(s) \in L_M$ then s must have been added to $C(j)$ previously in the second phase of constructing C' , so by the induction hypothesis, $\text{lbl}(s) \in C''(k)$ for some $k \in L_I$.

If $i \in L_M$ then $j \in L$. First note that the condition on j and i is also required to show that $C''; \varrho' \vdash \Gamma$, $C''; \varrho' \vdash e$, or $C''; \varrho' \vdash \tau$, so $C''(j) \subseteq C''(i)$. If $j \in L_I$ then we have what we need. Otherwise $j \in L_M$, so s was added to $C'(j)$ previously in the second phase of constructing $C'(j)$, so by the induction hypothesis, $C''(k) \subseteq C''(j)$ for some $k \in L_I$. Then $C''(k) \subseteq C''(i)$, as required.

- Case 2, s was added to i because $C(\varrho'(\alpha)) = C(i)$, required by the rule for type variables, did not hold and $s \in C(\varrho'(\alpha))$. Similar to Case 1.
- Case 3, s was added to i because $\text{box}(C; j, i)$ is required, either $(\text{box}_t i')_v^{j'} \in C'(j)$ or $(\text{box}_t i')_t^{j'} \in C'(j)$, and s was already in $C(i')$. In this case, i and j have to come from the same term, that is, either $i, j \in L_C$ or $i, j \in L$. Note that the labels in any shape under consideration come from the same term, that is, either they are all in L or they are all in L_C .
 - If $\text{lbl}(s) \in L_M$ then:
 - If s was added to $C'(i')$ previously in the second phase of constructing C' then by the induction hypothesis, $s \in C''(k)$ for some $k \in L_I$.
 - Otherwise $s \in C''(i')$ and $i', j' \in L$. Since s was not already in $C'(i)$ then the box shape was added to $C'(j)$ previously in the second phase of constructing C' , so by the induction hypothesis, the box shape is in $C''(k)$ for some $k \in L_I$. By (**), k labels a box type. By the rules for acceptability, $\text{box}(C; k, k')$ for C'' for some $k' \in L_I$. Thus, $C''(i') \subseteq C''(k')$. Thus $s \in C''(k')$, as required.
 - If $i \in L_M$ then $j \in L$ and $\text{box}(C; j, i)$ holds for C'' .
 - If the box shape was added to $C'(j)$ previously in the second phase of the constructing C' then by the induction hypothesis, $C''(k) \subseteq C''(j)$ for some $k \in L_I$. By (**), k labels a box type. Then by the rules for acceptability, $(\text{box}_t i'')_t^{j''} \in C''(k)$ for some $i'' \in L_I$, so $(\text{box}_t i'')_t^{j''} \in C''(j)$. By $\text{box}(C; j, i)$, $C''(i'') \subseteq C''(i)$, as required.
 - Otherwise, the box shape was in $C''(j)$ and $i', j' \in L$. By $\text{box}(C; j, i)$, $C''(i') \subseteq C''(i)$. Since s was not already in $C'(i)$ then it was previously added to $C'(i')$ in the second phase of constructing C' , so by the induction hypothesis, $C''(k) \subseteq C''(i')$ for some $k \in L_I$. Then by transitivity $C''(k) \subseteq C''(i)$, as required.
- Case 4, s was added because a $\text{fun}(C; j_1, j_2, j_3, j_4)$ is required. Similar to Case 3.

Claim: $\Upsilon' \vdash (\rho', E'\langle e \rangle)$. The proof is by a straight forward induction on the structure of $(\rho', E'\langle e \rangle)$. The only interesting case is application. In that case, we have $(e_1[\tau] e_2)^i$. By the induction hypothesis we get $\Upsilon' \vdash e_1$ and $\Upsilon' \vdash e_2$. We just need to show that $\Upsilon' \vdash \tau : \mathbf{r}$. If the application came from e then since the labels of τ are in L , the result follows from $\Upsilon \vdash \tau : \mathbf{r}$, which holds by assumption ($\Upsilon \vdash e$). Otherwise, the labels of τ are not in Υ' so clearly $\Upsilon' \vdash \tau \text{tr}(\tau)$. Then $\text{tr}(\tau) = \mathbf{r}$ holds by the typing rules. ■

With these definitions we can prove that unboxing for modules is correct via the following theorem.

Theorem 7

If $\vdash (\Gamma \Rightarrow e : \tau)$ wf, $C; \varrho \vdash (\Gamma \Rightarrow e : \tau)$, $C \vdash \Upsilon$, and $\Upsilon \vdash (\Gamma \Rightarrow e : \tau)$ then $\vdash (\Gamma \Rightarrow e : \tau) \equiv \downarrow(\Gamma \Rightarrow e : \tau)|_{\Upsilon}$.

Proof: By definition, $\downarrow(\Gamma \Rightarrow e : \tau)|_{\Upsilon} = (\Gamma \Rightarrow \downarrow e|_{\Upsilon} : \tau)$. Clearly $\Gamma = \Gamma$ and $\tau = \tau$, so it remains to show that $\Gamma \vdash e \equiv \downarrow e|_{\Upsilon} : \tau$. By $\vdash (\Gamma \Rightarrow e : \tau)$ wf, $\emptyset \vdash \Gamma$ wf and $\Gamma \vdash e : \tau$. By $C; \varrho \vdash (\Gamma \Rightarrow e : \tau)$, $C; \varrho \vdash \Gamma$, $C; \varrho \vdash e$, and $C; \varrho \vdash \tau$. By $\Upsilon \vdash (\Gamma \Rightarrow e : \tau)$, $\Upsilon \vdash \Gamma$ not unboxed, $\Upsilon \vdash e$, and $\Upsilon \vdash \tau$ not unboxed. By Theorem 2, $\downarrow\Gamma|_{\Upsilon} \vdash \downarrow e|_{\Upsilon} : \downarrow\tau|_{\Upsilon}$. By Lemma 34, $\Gamma \vdash \downarrow e|_{\Upsilon} : \tau$. Let ρ and E be such that $\vdash \rho : \Gamma$ and $\Gamma \vdash E : \mathbf{B}'(\tau)$. Then by Lemma 37, there exists ρ', E', C', ϱ' , and Υ' such that:

$$\begin{aligned} \rho &\sim_s \rho' \\ E &\sim_s E' \\ \vdash \rho' : \Gamma \\ \Gamma &\vdash E' : \mathbf{B}'(\tau) \\ C' &; \varrho' \vdash (\rho', E'\langle e \rangle) \\ C' &\vdash \Upsilon' \\ \Upsilon' &\vdash (\rho', E'\langle e \rangle) \\ \text{lbls}(\rho') \cap \Upsilon' &= \emptyset \\ \text{lbls}(E') \cap \Upsilon' &= \emptyset \\ \Upsilon \cap \text{lbls}(e) &= \Upsilon' \cap \text{lbls}(e) \end{aligned}$$

Since the operational semantics is deterministic, we just need to show that $(\rho, E\langle \downarrow e|_{\Upsilon} \rangle)$ matches $(\rho, E'\langle e \rangle)$ in behaviour. There are two cases:

- If $(\rho, E\langle e \rangle) \mapsto^* (\rho, c^j)$ then by Lemma 33, $(\rho', E'\langle e \rangle) \mapsto^* (\rho', c^{j'})$ for some j' . By Theorem 5, $\downarrow(\rho', E'\langle e \rangle)|_{\Upsilon'} \mapsto^* \downarrow(\rho', c^{j'})|_{\Upsilon'}$. By both Lemma 35 and definition of unboxing, $(\rho', E'\langle \downarrow e|_{\Upsilon} \rangle) \mapsto^* (\rho', c^{j'})$. Hence by Lemma 36, $(\rho', E'\langle \downarrow e|_{\Upsilon} \rangle) \mapsto^* (\rho', c^{j'})$. Therefore by Lemma 33 again, $(\rho, E\langle \downarrow e|_{\Upsilon} \rangle) \mapsto^* (\rho', c^{j''})$, for some j'' . It is not too hard to see that $j = j''$, as required.
- If $(\rho, E\langle e \rangle) \mapsto \dots$ then by Lemma 33, $(\rho', E'\langle e \rangle) \mapsto \dots$. By Theorem 5, $\downarrow(\rho', E'\langle e \rangle)|_{\Upsilon'} \mapsto \dots$. By Lemma 35, $(\rho', E'\langle \downarrow e|_{\Upsilon} \rangle) \mapsto \dots$. By Lemma 36, $(\rho', E'\langle \downarrow e|_{\Upsilon} \rangle) \mapsto \dots$. By Lemma 33, $(\rho, E\langle \downarrow e|_{\Upsilon} \rangle) \mapsto \dots$, as required. ■

7. Related work

This paper provides a modular approach to showing correctness of a realistic compiler optimization that rewrites the structure of program data structures in significant ways. We show how to use an arbitrary interprocedural reaching definitions analysis to eliminate unnecessary heap allocation in type-preserving fashion. Our optimization can be staged freely with other optimizations. Unlike any previous work that we are aware of, we account for correctness with respect to the meta-data requirements of the garbage collector, and we believe that additional issues such as different value sizes, dynamic type tests, etc. are straightforward to incorporate.

There has been substantial previous work addressing the problem of unboxing in general, and typed compilation specifically. Peyton Jones [3] introduced an explicit distinction between boxed and unboxed objects to allow a high-level compiler to locally eliminate boxing. Henglein and Jørgensen [2] defined a formal notion of optimality for local unboxings, albeit one that does not correspond to reduced allocation or reduced instruction count.

Leroy [4] defined a type-driven approach to adding coercions into and out of specialized representations. The type driven translation represents monomorphic objects natively (unboxed, in our terminology), and then introduces wrappers to coerce polymorphic uses into an appropriate form. To a first-order approximation, instead of boxing at definition sites this approach boxes objects at polymorphic use sites. This style of approach has the problem that it is not necessarily beneficial, since allocation is introduced in

places where it would not otherwise be present. This is reflected in the slowdowns observed on some benchmarks described in the original paper. This approach also has the potential to introduce space leaks. In a later paper [5] Leroy argued that a simple untyped approach gives better and more predictable results.

The MLton compiler [11] largely avoids the issue of a uniform object representation by completely monomorphizing programs before compilation. This approach requires whole-program compilation. More limited monomorphization schemes could be considered in an incremental compilation setting. Monomorphization does not eliminate the need for boxing in the presence of dynamic type tests or reflection. Just in time compilers (e.g. for .NET) may monomorphize dynamically at runtime.

The TIL compiler [1, 10] uses intensional type analysis in a whole-program compiler to allow native data representations without committing to whole-program compilation. As with the Leroy coercion approach, polymorphic uses of objects require conditionals and boxing coercions to be inserted at use sites, and consequently there is the potential to slow down, rather than speed up, the program. This approach can also be used by the garbage collector to relax the restriction that type variables range only over types of a single traceability. This idea is complementary to our approach in the sense that it merely relaxes some of the correctness restrictions placed on the unboxing by the underlying GC model.

Serrano and Feeley [9] described a flow analysis for performing unboxing similar in spirit to our approach, albeit in an untyped setting. Their algorithm essentially attempts to find a monomorphic typing for (parts of) a program in which object representations have not been made explicit, which they then use selectively to choose whether to use a uniform or non-uniform representation for each particular object. They do not deal with type information and type preserving compilation. They also assume a conservative garbage collector and hence do not need to account for the requirements of GC safety, and they do not prove a correctness result. The effectiveness of their algorithm would seem to be incomparable in that while they are not limited by the restrictions of GC and type safety, their algorithm can only unbox objects used monomorphically – the algorithm presented here can unbox objects used polymorphically subject to the restriction that the unboxed type have the appropriate traceability.

References

- [1] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.
- [2] F. Henglein and J. Jørgensen. Formally optimal boxing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 213–226, New York, NY, USA, 1994. ACM. ISBN 0-89791-636-0.
- [3] S. L. P. Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 636–666. Springer-Verlag New York, Inc., 1991.
- [4] X. Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pages 177–188, New York, NY, USA, 1992. ACM. ISBN 0-89791-453-8.
- [5] X. Leroy. The effectiveness of type-based unboxing. Technical report, Boston College, Computer Science Department, 1997.
- [6] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21:527–568, May 1999. ISSN 0164-0925.
- [7] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

ISBN 3540654100.

- [8] L. Petersen and N. Glew. GC-safe interprocedural unboxing. In *Proceedings of the 21st International Conference on Compiler Construction*, CC 2012, 2012.
- [9] M. Serrano and M. Feeley. Storage use analysis and its applications. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, ICFP '96, pages 50–61, New York, NY, USA, 1996. ACM. ISBN 0-89791-770-7.
- [10] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: a type-directed, optimizing compiler for ML. *SIGPLAN Not.*, 39(4):554–567, April 2004. ISSN 0362-1340.
- [11] S. Weeks. Whole-program compilation in MLton. In *Proceedings of the 2006 workshop on ML*, ML '06, pages 1–1, New York, NY, USA, 2006. ACM. ISBN 1-59593-483-9.