GC-Safe Interprocedural Unboxing

Leaf Petersen and Neal Glew

Intel Labs, Santa Clara CA
`leaf.petersen@intel.com`
`neal.glew@intel.com`

Abstract. Modern approaches to garbage collection (*GC*) require information about which variables and fields contain GC-managed pointers. Interprocedural flow analysis can be used to eliminate otherwise unnecessary heap allocated objects (*unboxing*), but must maintain the necessary GC information. We define a core language which models compiler correctness with respect to the GC, and develop a correctness specification for interprocedural unboxing optimizations. We prove that any optimization which satisfies our specification will preserve GC safety properties and program semantics, and give a practical unboxing algorithm satisfying this specification.

1 Introduction

Precise garbage collection (*GC*) for managed languages is usually implemented by requiring the compiler to keep track of meta-data indicating which variables and fields contain GC-managed references and which should be ignored by the garbage collector. We refer to this information as the *traceability* of a field or variable: a field or variable is traceable if it should be treated as a pointer into the heap by the garbage collector.

In order to maintain this information in the presence of polymorphic functions (including subtype polymorphism), 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. All interprocedural use of native (non-heap) data therefore occurs only through fields of objects. This is commonly referred to as *boxing*, objects represented in this way are referred to as *boxed*, and projecting out of the uniform representation is referred to as *unboxing*. Boxing imposes substantial performance penalties for many reasons: the additional overhead of the allocation and projection is substantial, arrays of boxed objects exhibit poor locality, and the additional memory pressure can cause bottlenecks in the hardware. In this paper, we show how to use the results of interprocedural flow analyses (reaching definitions) to implement an interprocedural unboxing optimization while preserving the meta-data necessary for precise garbage collection.

In the following sections, we define a high-level core language that captures the essential aspects of GC meta-data and GC safety. We then give a high-level specification of what a reasonable flow analysis on this language must compute, and define a notion of a general unboxing optimization for this language. We give a specification for when such an optimization is acceptable for a given flow analysis. We show that the optimization induced by an acceptable unboxing preserves the semantics of the original program, including GC safety. Finally, we construct an algorithm closely based on one in use in our compiler and prove that it produces an unboxing that satisfies our correctness specification, and hence that it preserves the semantics of the program (including GC safety).

All the lemmas and theorems in the paper have been proven. Proofs are available in an extended technical report available from the first author's website[7].

2 GC safety

Consider the following 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:

$$\texttt{let}\ f\ =\ \lambda x.(\texttt{box}\,x)\ \ \texttt{in}\,\texttt{unbox}(\texttt{unbox}(f\ (\texttt{box}\,3)))$$

The only definition reaching the variable x is the boxed machine integer 3. Information from an interprocedural analysis can be used to rewrite this program to eliminate the boxing as follows:

$$\texttt{let}\ f\ =\ \lambda x.x\ \ \texttt{in}\,f\ 3$$

In the second version of this program, the traceability of the 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 garbage collection occuring while x is live must treat x as a root in the first program, and must ignore x in the second program. There are numerous approaches to communicating this information to the garbage collector. For example, some implementations choose to dynamically tag values in such a way as to allow the garbage collector to distinguish pointers from non-pointers by inspection. Such an implementation might steal a low bit from the machine integer representation to allow the machine integer 3 to be distinguished from a heap pointer.

Another very commonly used approach (particularly in more recent systems) is to require the compiler 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. It is this approach that we target in this paper.

The requirement that the compiler be able to annotate program variables with a single static traceability constrains the compiler's ability to rewrite programs in that it must do so in a way that preserves the correctness of the GC meta-data of the program. Consider an extension of the previous example.

$$\texttt{let}\ f\ =\ \lambda x.x\ \ \texttt{in}\,\texttt{unbox}((f\ f)\ (\texttt{box}\,3))$$

Assuming that functions are represented as heap-allocated objects then each variable in this program can be assigned a traceability, 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).

$$\texttt{let}\ f\ =\ \lambda x.x\ \ \texttt{in}(f\ f)\ 3$$

As this example shows, the concerns of maintaining garbage-collector meta-data constrain optimization[1] in ways not apparent in a GC-ignorant semantic model.

2.1 A core language for GC safety

In order to give a precise account of interprocedural unboxing, we begin by defining a core language capturing the essential features of GC safety. The motivation for the 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 must be tracked by the compiler. In our implementation, the compiler intermediate language under consideration is substantially more low-level: a control-flow graph based, static single assignment intermediate representation. We believe however that all of the key issues are captured faithfully in this higher-level representation.

Traceabilities t	$::=$	$\texttt{b} \mid \texttt{r}$	Terms m	$::=$	$x \mid \lambda x^i{:}t.e \mid e_1\ e_2 \mid \texttt{box}_t\, e \mid$
Term variables		x, y, z			$\texttt{unbox}\, e \mid \rho(e)$
Constants		c	Values v	$::=$	$c \mid \langle \rho, \lambda x^i{:}t.e \rangle \mid \langle v^i{:}t \rangle$
Labels i	$::=$	$0, 1, \ldots$	Environments ρ	$::=$	$x_1^{i_1}{:}t_1 = {v_1}^{j_1}, \ldots, x_n^{i_n}{:}t_n = {v_n}^{j_n}$
Labeled Terms e	$::=$	$m^i \mid v^i$	States M	$::=$	(ρ, e)

Fig. 1. Syntax

Figure 1 defines the syntax of our core language. The essence of the language is largely that of the standard untyped lambda calculus with an explicit environment semantics, extended with a form of degenerate type information we call *traceabilities*. Traceabilities describe the GC status of variables: the traceability `b` (for bits) indicates something that should be ignored by the garbage collector, while the traceability `r` (for reference) indicates a GC-managed pointer.

[1] It is worth noting that a serious compiler might be expected to duplicate the body of f in this simple example thereby eliminating this constraint and allowing the unboxing optimization to be more effective.

The traceability $\mathtt{b}$ is inhabited by an unspecified set of constants c while the traceability $\mathtt{r}$ 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 term, value, and variable binding site with an integer label. We do not assume that labels or variables are unique within a program.

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. Variable binding sites are decorated with traceability information ($\lambda x^i{:}t.e$), as are box introductions ($\mathtt{box}_t\, e$). We represent heap allocation in the language via the $\mathtt{box}_t\, e$ term, which corresponds to allocating a heap cell containing the value for e. The traceability t gives the meta-data with which the heap-cell will be tagged, allowing the garbage collector to trace the cell. Objects can be projected out of an allocated object by the $\mathtt{unbox}\, e$ operation. Frames $\rho(e)$ are discussed below.

Values consist of either constants, closures, or heap-allocated boxes. We distinguish between the introduction form ($\mathtt{box}_t\, e$) and the value form ($\langle v^i{:}t\rangle$) 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 the purposes of the dynamic semantics we also distinguish between functions ($\lambda x^i{:}t.e$) and the heap allocated closures that represent them at runtime ($\langle \rho, \lambda x^i{:}t.e\rangle$).

For notational convenience, we will sometimes use the notation $v_{\mathtt{b}}$ to indicate that a value v is a non-heap-allocated value (i.e. a constant c), and $v_{\mathtt{r}}$ to indicate that a value v is a heap-allocated value (i.e. either a lambda value or a boxed 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 `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 annotations on variables in the environments play the role of stack frame GC meta-data, indicating which slots of the frame are roots (traceability $\mathtt{r}$). The environments buried in closures ($\langle \rho, \lambda x^i{:}t.e\rangle$) similarly provide the traceabilities of values reachable from the closure, 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 takes steps in this semantics do not introduce ill-formed heap objects. 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. The

GC correctness criteria for a compiler optimization then is that the optimization map traceable programs to traceable programs, and that it not introduce new undefined behavior.

2.2 Operational semantics

We choose to use an explicit environment semantics rather than a standard substitution semantics since this makes the GC meta-data 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 traceability annotations on each of the variables mapped by the environment.

$$\frac{x^i{:}t = v^j \in \rho}{(\rho, x^k) \longmapsto (\rho, v^j)} \qquad \frac{}{(\rho, (\lambda x^i{:}t.e)^j) \longmapsto (\rho, \langle \rho, \lambda x^i{:}t.e\rangle^j)}$$

$$\frac{t = t'}{(\rho, (\mathtt{box}_t\, v_{t'}{}^i)^j) \longmapsto (\rho, \langle v_{t'}{}^i{:}t\rangle^j)} \qquad \frac{(\rho, e_1) \longmapsto (\rho, e_1')}{(\rho, (e_1\ e_2)^i) \longmapsto (\rho, (e_1'\ e_2)^i)}$$

$$\frac{(\rho, e_2) \longmapsto (\rho, e_2')}{(\rho, (v^i\ e_2)^j) \longmapsto (\rho, (v^i\ e_2')^j)} \qquad \frac{t = t'}{(\rho, (\langle \rho', \lambda x^i{:}t.e\rangle^j\ v_{t'}{}^k)^l) \longmapsto (\rho, (\rho', x^i{:}t = v_{t'}{}^k)(e)^l)}$$

$$\frac{(\rho, e) \longmapsto (\rho, e')}{(\rho, (\mathtt{box}_t\, e)^i) \longmapsto (\rho, (\mathtt{box}_t\, e')^i)}$$

$$\frac{(\rho, e) \longmapsto (\rho, e')}{(\rho, (\mathtt{unbox}\, e)^i) \longmapsto (\rho, (\mathtt{unbox}\, e')^i)} \qquad \frac{}{(\rho, (\mathtt{unbox}\, \langle v^i{:}t\rangle^j)^k) \longmapsto (\rho, v^i)}$$

$$\frac{(\rho', e) \longmapsto (\rho', e')}{(\rho, \rho'(e)^i) \longmapsto (\rho, \rho'(e')^i)} \qquad \frac{}{(\rho, \rho'(v^i)^j) \longmapsto (\rho, v^i)}$$

Fig. 2. Operational Semantics

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 reduction mapping a lambda term $\lambda x^i{:}t.e$ to an allocated closure $\langle \rho, \lambda x^i{:}t.e\rangle$, and similarly for boxed objects and values. More notably, beta-reduction is restricted to only permit construction of a stack frame when the meta-data attached to the parameter variable is appropriate 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 ($\texttt{box}_t\, v_{t'} \longmapsto \langle v_{t'}{:}t \rangle$) when the operation meta-data is appropriate for the value (i.e. $t = t'$).

It is important to note that this semantics does not model a dynamically checked language, in which there is an explicit check of the meta-data associated with these reductions. The point is simply that the semantics only specifies how programs behave when these conditions are met—in all other cases the behavior of the program is undefined.

2.3 Traceability

Labeled Terms $\boxed{\vdash e\ \mathbf{tr}}$

$$\frac{\vdash m\ \mathbf{tr}}{\vdash m^i\ \mathbf{tr}} \qquad \frac{\vdash_{\mathrm{v}} v{:}t}{\vdash v^i\ \mathbf{tr}}$$

Terms $\boxed{\vdash m\ \mathbf{tr}}$

$$\frac{}{\vdash x\ \mathbf{tr}} \qquad \frac{\vdash e\ \mathbf{tr}}{\vdash \lambda x^i{:}t.e\ \mathbf{tr}} \qquad \frac{\vdash e_1\ \mathbf{tr} \quad \vdash e_2\ \mathbf{tr}}{\vdash e_1\ e_2\ \mathbf{tr}}$$

$$\frac{\vdash e\ \mathbf{tr}}{\vdash \texttt{box}_t\, e\ \mathbf{tr}} \qquad \frac{\vdash e\ \mathbf{tr}}{\vdash \texttt{unbox}\, e\ \mathbf{tr}} \qquad \frac{\vdash \rho\ \mathbf{tr} \quad \vdash e\ \mathbf{tr}}{\vdash \rho(e)\ \mathbf{tr}}$$

Values $\boxed{\vdash_{\mathrm{v}} v{:}t}$

$$\frac{}{\vdash_{\mathrm{v}} c{:}b} \qquad \frac{\vdash \rho\ \mathbf{tr} \quad \vdash e\ \mathbf{tr}}{\vdash_{\mathrm{v}} \langle \rho, \lambda x^i{:}t.e \rangle{:}r} \qquad \frac{\vdash_{\mathrm{v}} v{:}t}{\vdash_{\mathrm{v}} \langle v^i{:}t \rangle{:}r}$$

Environments $\boxed{\vdash \rho\ \mathbf{tr}}$

$$\frac{\vdash_{\mathrm{v}} v_1{:}t_1 \quad \cdots \quad \vdash_{\mathrm{v}} v_n{:}t_n}{\vdash x_1^{i_1}{:}t_1 = {v_1}^{j_1}, \ldots, x_n^{i_n}{:}t_n = {v_n}^{j_n}\ \mathbf{tr}}$$

Machine States $\boxed{\vdash M\ \mathbf{tr}}$

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

Fig. 3. Traceability

The operational semantics ensures that no reduction step introduces mis-tagged values. In order to make use of this, we define a judgment 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 $\vdash_{\mathrm{v}} v{:}t$ asserts that a value v is well-formed, and has traceability t. In this simple language, this corresponds to having the meta-data on the environment of each lambda value be consistent and the meta-data on each boxed value be consistent with the traceability of the object nested in the box. An environment is consistent, $\vdash \rho\ \mathbf{tr}$, when the annotation on each variable agrees with the traceability of the value it is bound to. Since we cannot check the consistency of general terms with the first-order information available, the term judgement $\vdash e$ **tr** and machine state judgement $\vdash M$ **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* $\vdash M$ **tr** *and* $M \longmapsto M'$ *then* $\vdash M'$ **tr**.

There is of course no corresponding progress property for our notion of traceability, since programs can go wrong. Compiler optimizations are simply responsible for ensuring that they do not introduce new ways to go wrong.

3 Flow analysis

Our original motivation for this work was to apply interprocedural analysis to the problem of eliminating unnecessary boxing in programs. 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 terms flow to what use sites. This paper is not intended to provide any contribution to this body of work, which we will broadly refer to as *flow analysis*. Instead, we focus on how to use the results of such a generic analysis to implement an unboxing optimization that preserves GC safety.

In order to do this, we must provide some framework for describing what information a flow analysis must provide. For the purposes of our unboxing optimization, we are interested in finding (inter-procedurally) for every $(\texttt{unbox}\, v^j)^i$ operation the set of $(\texttt{box}_t\, e)^k$ terms that could possibly reach v. Under appropriate conditions, we can then eliminate both the box introductions and the box elimination, thereby improving the program. The core language defined in Section 2 provides labels serving as proxies for the terms and variables on which they occur – the question above can therefore be re-stated as finding the set of labels k that reach the position labeled with j.

More generally, 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 simply 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:

$$\text{Shapes: } s ::= c^i \mid (i{:}t \to j)^k \mid (\texttt{box}_t\, i)^j$$

The idea behind 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), $c^i \in \mathrm{C}(k)$ indicates that (according to the analysis) the constant c, labeled with i, might flow to a location labeled with k. Similarly, if $(i{:}t \to j)^k \in \mathrm{C}(l)$, then the analysis specifies that among the values flowing to locations labeled with l might be lambdas labeled with k, whose parameter variable is labeled with i and annotated with t and whose bodies are labeled with j. If $(\texttt{box}_t\, k)^i \in \mathrm{C}(l)$ 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 $\mathrm{C}(j) \subseteq \mathrm{C}(k)$.

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. [6], 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 completely decouple our optimization from the particulars of how the analysis is computed.

Our acceptable-analysis relation is given in Figure 4 – the judgement $\mathrm{C}; \varrho \vdash (\rho, e)$ 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 $\mathrm{lbl}(e)$ to denote the outermost label of e: that is, i where e is of the form m^i or v^i. 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 lambda value that flows to the applicand position, the set of shapes associated with the parameter of that lambda 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 lambda is a sub-set of the set of shapes associated with the application itself.

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

$$\frac{\mathrm{C}(\varrho(x)) \subseteq \mathrm{C}(i)}{\mathrm{C}; \varrho \vdash x^i} \qquad \frac{\varrho(x) = j \quad \mathrm{C}; \varrho \vdash e \quad (j{:}t \to \mathrm{lbl}(e))^i \in \mathrm{C}(i)}{\mathrm{C}; \varrho \vdash (\lambda x^j{:}t.e)^i}$$

$$\frac{\begin{array}{c}\mathrm{C}; \varrho \vdash e_1 \quad \mathrm{C}; \varrho \vdash e_2 \\ \forall (k{:}t \to l)^j \in \mathrm{C}(\mathrm{lbl}(e_1)) : \\ \mathrm{C}(\mathrm{lbl}(e_2)) \subseteq \mathrm{C}(k) \wedge \mathrm{C}(l) \subseteq \mathrm{C}(i)\end{array}}{\mathrm{C}; \varrho \vdash (e_1\ e_2)^i} \qquad \frac{\mathrm{C}; \varrho \vdash e \quad (\mathtt{box}_t\, j)^i \in \mathrm{C}(i) \quad \mathrm{C}(\mathrm{lbl}(e)) \subseteq \mathrm{C}(j)}{\mathrm{C}; \varrho \vdash (\mathtt{box}_t\, e)^i}$$

$$\frac{\begin{array}{c}\mathrm{C}; \varrho \vdash e \\ \forall (\mathtt{box}_t\, k)^j \in \mathrm{C}(\mathrm{lbl}(e)) : \mathrm{C}(k) \subseteq \mathrm{C}(i)\end{array}}{\mathrm{C}; \varrho \vdash (\mathtt{unbox}\, e)^i} \qquad \frac{\mathrm{C}; \varrho \vdash \rho \quad \mathrm{C}; \varrho \vdash e \quad \mathrm{C}(\mathrm{lbl}(e)) \subseteq \mathrm{C}(i)}{\mathrm{C}; \varrho \vdash \rho(e)^i}$$

$$\frac{c^i \in \mathrm{C}(i)}{\mathrm{C}; \varrho \vdash c^i} \qquad \frac{\varrho(x) = j \quad \mathrm{C}; \varrho \vdash \rho \quad \mathrm{C}; \varrho \vdash e \quad (j{:}t \to \mathrm{lbl}(e))^i \in \mathrm{C}(i)}{\mathrm{C}; \varrho \vdash \langle \rho, \lambda x^j{:}t.e \rangle^i}$$

$$\frac{\mathrm{C}; \varrho \vdash v^j \quad (\mathtt{box}_t\, k)^i \in \mathrm{C}(i) \quad \mathrm{C}(j) \subseteq \mathrm{C}(k)}{\mathrm{C}; \varrho \vdash \langle v^j{:}t \rangle^i}$$

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

$$\frac{\forall 1 \leq k \leq n : \varrho(x_k) = i_k \wedge \mathrm{C}(j_k) \subseteq \mathrm{C}(i_k) \wedge \mathrm{C}; \varrho \vdash {v_k}^{j_k}}{\mathrm{C}; \varrho \vdash x_1^{i_1}{:}t_1 = {v_i}^{j_1}, \ldots, x_n^{i_n}{:}t_n = {v_n}^{j_n}}$$

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

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

Fig. 4. Acceptable Analysis

Given this definition, we can show that the acceptability relation is preserved under reduction.

Lemma 2 (Many-step reduction preserves acceptability). *If* $\mathrm{C}; \varrho \vdash M$ *and* $M \longmapsto^* M'$ *then* $\mathrm{C}; \varrho \vdash M'$.

4 Unboxing

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. Doing so may change the traceability, since the object in the box may not be a GC-managed reference. Moreover, the object in the box may itself be a candidate for unboxing; consequently, determining the traceability of boxed objects depends on exactly which objects are unboxed. Function parameters may be instantiated with objects from multiple different definition sites, some of which may be unboxed and some of which may not.

Consider again the first example from Section 1, written out with explicit GC information and labels:

$$\texttt{let } f^0{:}\texttt{r} \;=\; (\lambda x^1{:}\texttt{r}.(\texttt{box}_{\texttt{r}}\, x^2)^3)^4 \;\texttt{in}\,(\texttt{unbox}\,(\texttt{unbox}\,(f^5\;(\texttt{box}_{\texttt{b}}\, 3^6)^7)^8)^9)^{10}$$

It is fairly easy to see that this program is unboxable. The binding site for x is only reached by the term labeled with 7 (the outer box introduction), and hence there should be no problems with changing its traceability annotation. Each box elimination is reached only by a single box introduction, and hence the box/unbox pairs in this program should be eliminable, yielding an optimized program:

$$\texttt{let } f^0{:}\texttt{r} \;=\; (\lambda x^1{:}\texttt{b}.x^2)^4 \;\texttt{in}\,(f^5\;3^6)^8$$

Notice that in order to rewrite the program, we have had to change the traceability annotation at the binding site for x, since we have eliminated the box introduction on its argument. This constraint is imposed on us by the need to keep the GC information consistent. If we choose (perhaps because of limitations on the precision of the analysis, or perhaps because of other constraints) to only eliminate the innermost box/unbox pair, then we must similarly adjust the traceability annotation on the remaining box introduction (labeled with 3).

$$\texttt{let } f^0{:}\texttt{r} \;=\; (\lambda x^1{:}\texttt{b}.(\texttt{box}_{\texttt{b}}\, x^2)^3)^4 \;\texttt{in}\,(\texttt{unbox}\,(f^5\;3^6)^8)^9$$

Not all programs can be consistently rewritten in this manner. If we consider again the second example from Section 1, we see an example of a program in which we must forgo optimization if we wish to preserve GC safety.

$$\texttt{let } f^0{:}\texttt{r} \;=\; (\lambda x^1{:}\texttt{r}.x^2)^3 \;\texttt{in}\,(\texttt{unbox}\,((f^4\;f^5)^6\;(\texttt{box}_{\texttt{b}}\, 3^7)^8)^9)^{10}$$

It is easy to see that any acceptable analysis must include the function labeled with 3 and the boxed term labeled with 8 in the set of terms reaching the binding site for x, labeled with 1. We might naively attempt to eliminate the box/unbox pair as follows:

$$\texttt{let } f^0{:}\texttt{r} \;=\; (\lambda x^1{:}\texttt{?}.x^2)^3 \;\texttt{in}\,((f^4\;f^5)^6\;3^7)^9$$

Unfortunately, there is no consistent choice of traceability annotation for the binding site for x. If we choose b as the traceability annotation then after reduction we arrive at a state that has no defined reduction:

$$((\langle\epsilon, \lambda x^1{:}\texttt{b}.x^2\rangle^3\;\langle\epsilon, \lambda x^1{:}\texttt{b}.x^2\rangle^3)^6\;3^7)^9$$

The first application leads to undefined behavior, since the traceability of the argument value does not match the traceability annotation on the parameter variable. If we had instead chosen `r` as the traceability annotation, then one further reduction would still lead us to undefined behavior.

$$(\langle \epsilon, \lambda x^1{:}\mathtt{r}.x^2 \rangle^3\ 3^7)^9$$

The requirement to preserve GC information imposes two burdens on us then: we must provide some mechanism for assigning new GC meta-data when we optimize the program, and we must also ensure that we do not optimize the program in a way that does not admit a consistent assignment of such meta-data. 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 reasonable one.

4.1 The unboxing optimization

We can divide the problem of specifying an unboxing into two sub-parts: choosing the particular box/unbox pairs that are valid to eliminate and assigning new traceability annotations to terms that are affected. An unboxing then is a pair (T, Υ), where Υ is a set of labels, and T is a partial function from labels to traceabilities. The unboxed set Υ is the set of labels to be unboxed, and the traceability map T specifies new traceabilities for labels affected by the unboxing. The fact that T is a partial function is essential for several reasons. On a technical level, we do not require that labels be unique in a program. Consequently, it is possible that there is no consistent choice for a specific label. More importantly, requiring that T be a total function would put unsatisfiable requirements on the flow analysis. For example, a program that allocates a mis-tagged object after going into an infinite loop is GC safe according to our specification since the bad allocation is never reached. Requiring the analysis to find a consistent traceability map for such a program is equivalent to requiring it to solve the halting problem, since it must statically prove that the set of values dynamically reaching the mis-tagged allocation site is empty. By allowing T to be a partial function, we allow for necessary imprecision in the analysis. Also of importance is the need to allow for relative imprecision in the analysis. In order to achieve faster compile times, we may choose to use less precise analyses that potentially over-approximate the set of terms reaching a use point. Consequently, even if a consistent traceability assignment exists, we may not have sufficiently precise information to construct it.

An unboxing pair defines a total function mapping labeled terms to labeled terms, as shown in Figure 5. For notational convenience, we take $\mathrm{T}(i) = t$ as asserting that i is in the domain of T, and that its image is t. We also say that: $\mathrm{T}(i) \geq t$ if and only if $\mathrm{T}(i) = t$ or $\mathrm{T}(i)$ is undefined; and $\underline{\mathrm{T}}(i, t) = \mathrm{T}(i)$ if T defined at i, otherwise t.

An important observation about the unboxing optimization as we have defined it is unlike many previous interprocedural approaches (Section 6), it only

$$\boxed{\downharpoonright e \downharpoonleft^{\mathrm{T}}_{\Upsilon}}$$

$$\begin{aligned}
\downharpoonright x^{i} \downharpoonleft^{\mathrm{T}}_{\Upsilon} &= x^{i} \\
\downharpoonright (\lambda x^{j}{:}t.e)^{i} \downharpoonleft^{\mathrm{T}}_{\Upsilon} &= (\lambda x^{j}{:}\underline{\mathrm{T}}(j,t).\downharpoonright e \downharpoonleft^{\mathrm{T}}_{\Upsilon})^{i} \\
\downharpoonright (e_1\ e_2)^{i} \downharpoonleft^{\mathrm{T}}_{\Upsilon} &= (\downharpoonright e_1 \downharpoonleft^{\mathrm{T}}_{\Upsilon}\ \downharpoonright e_2 \downharpoonleft^{\mathrm{T}}_{\Upsilon})^{i} \\
\downharpoonright (\mathtt{box}_t\ e)^{i} \downharpoonleft^{\mathrm{T}}_{\Upsilon} &= \downharpoonright e \downharpoonleft^{\mathrm{T}}_{\Upsilon} && i \in \Upsilon \\
&= (\mathtt{box}_{\underline{\mathrm{T}}(\mathrm{lbl}(e),t)}\ \downharpoonright e \downharpoonleft^{\mathrm{T}}_{\Upsilon})^{i} && i \notin \Upsilon \\
\downharpoonright (\mathtt{unbox}\, e)^{i} \downharpoonleft^{\mathrm{T}}_{\Upsilon} &= \downharpoonright e \downharpoonleft^{\mathrm{T}}_{\Upsilon} && \mathrm{lbl}(e) \in \Upsilon \\
&= (\mathtt{unbox}\, \downharpoonright e \downharpoonleft^{\mathrm{T}}_{\Upsilon})^{i} && \mathrm{lbl}(e) \notin \Upsilon
\end{aligned}$$

$$\begin{aligned}
\downharpoonright \rho(e)^{i} \downharpoonleft^{\mathrm{T}}_{\Upsilon} &= \downharpoonright \rho \downharpoonleft^{\mathrm{T}}_{\Upsilon}(\downharpoonright e \downharpoonleft^{\mathrm{T}}_{\Upsilon})^{i} \\
\downharpoonright c^{i} \downharpoonleft^{\mathrm{T}}_{\Upsilon} &= c^{i} \\
\downharpoonright \langle \rho, \lambda x^{j}{:}t.e \rangle^{i} \downharpoonleft^{\mathrm{T}}_{\Upsilon} &= \\
&\langle \downharpoonright \rho \downharpoonleft^{\mathrm{T}}_{\Upsilon}, \lambda x^{j}{:}\underline{\mathrm{T}}(j,t).\downharpoonright e \downharpoonleft^{\mathrm{T}}_{\Upsilon} \rangle^{i} \\
\downharpoonright \langle v^{j}{:}t \rangle^{i} \downharpoonleft^{\mathrm{T}}_{\Upsilon} &= \downharpoonright v^{j} \downharpoonleft^{\mathrm{T}}_{\Upsilon} && i \in \Upsilon \\
&= \langle \downharpoonright v^{j} \downharpoonleft^{\mathrm{T}}_{\Upsilon}{:}\underline{\mathrm{T}}(j,t) \rangle^{i} && i \notin \Upsilon
\end{aligned}$$

$$\boxed{\downharpoonright \rho \downharpoonleft^{\mathrm{T}}_{\Upsilon}} \quad \begin{aligned}
&\downharpoonright x_1^{i_1}{:}t_1 = {v_1}^{j_1}, \ldots, x_n^{i_n}{:}t_n = {v_n}^{j_n} \downharpoonleft^{\mathrm{T}}_{\Upsilon} = \\
&\qquad x_1^{i_1}{:}\underline{\mathrm{T}}(i_1,t_1) = \downharpoonright {v_1}^{j_1} \downharpoonleft^{\mathrm{T}}_{\Upsilon}, \ldots, x_n^{i_n}{:}\underline{\mathrm{T}}(i_n,t_n) = \downharpoonright {v_n}^{j_n} \downharpoonleft^{\mathrm{T}}_{\Upsilon}
\end{aligned}$$

$$\boxed{\downharpoonright M \downharpoonleft^{\mathrm{T}}_{\Upsilon}} \quad \downharpoonright (\rho, e) \downharpoonleft^{\mathrm{T}}_{\Upsilon} = (\downharpoonright \rho \downharpoonleft^{\mathrm{T}}_{\Upsilon}, \downharpoonright e \downharpoonleft^{\mathrm{T}}_{\Upsilon})$$

Fig. 5. Unboxing

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 (T, Υ) defines an unboxing, not every unboxing pair is reasonable in the sense that it defines a semantics preserving optimization. Just as we defined a notion of acceptable analysis in Section 3, we will 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, we can eliminate a box introduction if certain criteria are met. Firstly, we must be able to eliminate all of the unbox operations that it reaches. Secondly, we must be able to find a consistent traceability assignment covering each intermediate variable or field that it reaches, given all of the rest of our unboxing choices. We can eliminate an unbox operation if we can eliminate all of the box operations that reach it. Finally, we must also impose coherence requirements on traceability assignments. For every variable whose binding-site label occurs in the domain of T, we require that its new traceability assignment agree with the traceability assignment of all of its reaching definitions. Similarly, for every box introduction (or value form) that is not itself unboxed, we require that the traceability assignment for its contents agree with the traceability assignment for every reaching definition in the flow analysis.

This informal description is made precise in Figure 6. We use the notation $i \overset{\mathrm{T},\Upsilon}{\simeq} j$ to indicate when an unboxing *agrees* at two labels i and j.

$$
\begin{array}{ll}
i \overset{\mathrm{T}}{\simeq} j & \text{iff either } \mathrm{T}(i) = \mathrm{T}(j) \text{ (both defined) or } \mathrm{T}(i) \text{ and } \mathrm{T}(j) \text{ undefined} \\
i \overset{\Upsilon}{\simeq} j & \text{iff either } i, j \in \Upsilon \text{ or } i, j \notin \Upsilon \\
i \overset{\mathrm{T},\Upsilon}{\simeq} j & \text{iff } i \overset{\mathrm{T}}{\simeq} j \text{ and } i \overset{\Upsilon}{\simeq} j
\end{array}
$$

An unboxing pair (T, Υ) is acceptable relative to an analysis (C, ϱ) for a program M (judgement $\mathrm{C} \vdash M \downharpoonleft\downharpoonright (\mathrm{T}, \Upsilon)$) if the unboxing is *cache consistent* (judgement $\mathrm{C} \vdash (\mathrm{T}, \Upsilon)$), and *consistent* (judgement $\mathrm{T}, \Upsilon \vdash M$).

$$\boxed{\mathrm{T},\Upsilon \vdash e}$$

$$
\frac{}{\mathrm{T},\Upsilon \vdash x^i}
\qquad
\frac{\begin{array}{c}\mathrm{T},\Upsilon \vdash e \\ \mathrm{T}(j) \geq \mathtt{r}\end{array}}{\mathrm{T},\Upsilon \vdash (\lambda x^i{:}t.e)^j}
\qquad
\frac{\begin{array}{c}\mathrm{T},\Upsilon \vdash e_1 \\ \mathrm{T},\Upsilon \vdash e_2\end{array}}{\mathrm{T},\Upsilon \vdash (e_1\ e_2)^i}
\qquad
\frac{\begin{array}{c}i \in \Upsilon \\ \mathrm{T}(i) = \mathrm{T}(\mathrm{lbl}(e)) \\ \mathrm{T},\Upsilon \vdash e\end{array}}{\mathrm{T},\Upsilon \vdash (\mathtt{box}_t\ e)^i}
\qquad
\frac{\begin{array}{c}i \notin \Upsilon \quad \mathrm{T}(i) \geq \mathtt{r} \\ \mathrm{T},\Upsilon \vdash e\end{array}}{\mathrm{T},\Upsilon \vdash (\mathtt{box}_t\ e)^i}
$$

$$
\frac{\mathrm{T},\Upsilon \vdash e}{\mathrm{T},\Upsilon \vdash (\mathtt{unbox}\, e)^i}
\qquad
\frac{\mathrm{T},\Upsilon \vdash \rho \quad \mathrm{T},\Upsilon \vdash e}{\mathrm{T},\Upsilon \vdash \rho(e)^i}
\qquad
\frac{\mathrm{T}(i) \geq \mathtt{b}}{\mathrm{T},\Upsilon \vdash c^i}
\qquad
\frac{\mathrm{T},\Upsilon \vdash \rho \quad \mathrm{T},\Upsilon \vdash e \quad \mathrm{T}(j) \geq \mathtt{r}}{\mathrm{T},\Upsilon \vdash \langle \rho, \lambda x^i{:}t.e\rangle^j}
$$

$$
\frac{\begin{array}{c}j \in \Upsilon \quad \mathrm{T}(j) = \mathrm{T}(i) \\ \mathrm{T},\Upsilon \vdash v^i\end{array}}{\mathrm{T},\Upsilon \vdash \langle v^i{:}t\rangle^j}
\qquad
\frac{\begin{array}{c}j \notin \Upsilon \quad \mathrm{T}(j) \geq r \\ \mathrm{T},\Upsilon \vdash v^i\end{array}}{\mathrm{T},\Upsilon \vdash \langle v^i{:}t\rangle^j}
$$

$$
\boxed{\mathrm{T},\Upsilon \vdash \rho}
\qquad
\frac{\forall 1 \leq k \leq n : \underline{\mathrm{T}}(i_k, t_k) = \underline{\mathrm{T}}(j_k, t_k) \wedge \mathrm{T},\Upsilon \vdash {v_k}^{j_k}}{\mathrm{T},\Upsilon \vdash x_1^{i_1}{:}t_1 = {v_1}^{j_1}, \ldots, x_n^{i_n}{:}t_n = {v_n}^{j_n}}
$$

$$
\boxed{\mathrm{T},\Upsilon \vdash M}
\quad
\frac{\mathrm{T},\Upsilon \vdash \rho \quad \mathrm{T},\Upsilon \vdash e}{\mathrm{T},\Upsilon \vdash (\rho, e)}
\qquad
\boxed{\mathrm{C} \vdash (\mathrm{T},\Upsilon)}
\quad
\frac{\forall i : s \in \mathrm{C}(i) \implies i \overset{\mathrm{T},\Upsilon}{\simeq} \mathrm{lbl}(s)}{\mathrm{C} \vdash (\mathrm{T},\Upsilon)}
$$

$$
\boxed{\mathrm{C} \vdash M \downharpoonleft\downharpoonright (\mathrm{T},\Upsilon)}
\quad
\frac{\mathrm{C} \vdash (\mathrm{T},\Upsilon) \quad \mathrm{T},\Upsilon \vdash M}{\mathrm{C} \vdash M \downharpoonleft\downharpoonright (\mathrm{T},\Upsilon)}
$$

Fig. 6. Consistent and acceptable unboxing

Cache consistency $\mathrm{C} \vdash (\mathrm{T}, \Upsilon)$ encapsulates the requirement that an unbox can only be eliminated if all of the reaching definitions of its target are unboxed. It requires agreement between the label of the target of the unbox and the labels of everything in the cache of the target. The results from Section 3 ensure that under any evaluation, any term reaching the unbox is in the cache of the original target label, and hence that the unboxing approximation takes into account a sufficient set of terms[2].

Cache consistency does not put any constraints on the actual choice of traceabilities in T. The consistency judgement ($\mathrm{T}, \Upsilon \vdash M$) ensures that the traceability map T encodes choices that are compatible with the actual labeled terms in M, given a particular choice of terms to unbox Υ.

For environments, the consistency judgement insists that the traceability map assign consistent traceabilities to values and the variables to which they are bound. In this way we can ensure that the result of unboxing an environment still provides good traceability information for the garbage collector.

The term consistency judgement for the most part only requires that the traceability map be consistent with the labeled values. Variable uses incur no constraints, and neither do applications nor unbox operations (beyond requiring the consistency of their sub-terms). For constants c^i, we require that the traceability assignment T, if defined at i, maps i to `b`. That is, we require that the traceability assignment for i is consistent with the actual term inhabiting the label. Functions have a similar requirement: the traceability assignment for their label, if present, must be `r` since functions are represented by heap allocated closures. In the value form the closed over environment must be consistent as well.

The only particularly interesting rules are those covering the boxed introduction form and isomorphically the boxed value form. There are two cases: one for when the boxed value is selected for unboxing (that is, its label is in Υ), and one for when it has not been selected for unboxing.

If the term is not to be unboxed ($j \notin \Upsilon$), then the consistency rule requires that its traceability assignment (if any) be `r`. In the case that the term is to be unboxed ($j \in \Upsilon$) this is not required since the unboxed value may not in fact end up having traceability `r`. Instead, we require that the traceability map have an assignment both for the label of the box (j), and for the label of the contents of the box (i), and that it assign the same traceability to both. This requirement may be somewhat unexpected at first. The intuition behind it is that the end result of unboxing will replace the outer box by the inner boxed value; therefore we wish to treat the boxed value as having the same traceability as its contents.

Our goal is to show that the unboxing function induced by any acceptable unboxing is in some sense correct as an optimization. The first part of this is to show that unboxing preserves traceability.

Theorem 1 (Consistent unboxings preserve traceability).

- *If* $\mathrm{T}, \Upsilon \vdash v^i$ *and* $\vdash_{\mathrm{v}} v^i{:}t$ *then* $\vdash_{\mathrm{v}} \downharpoonleft v^i \downharpoonright_{\Upsilon}^{\mathrm{T}}{:}\underline{\mathrm{T}}(i,t)$.
- *If* $\mathrm{T}, \Upsilon \vdash e$ *and* $\vdash e\ \mathbf{tr}$ *then* $\vdash \downharpoonleft e \downharpoonright_{\Upsilon}^{\mathrm{T}}\ \mathbf{tr}$.
- *If* $\mathrm{T}, \Upsilon \vdash \rho$ *and* $\vdash \rho\ \mathbf{tr}$ *then* $\vdash \downharpoonleft \rho \downharpoonright_{\Upsilon}^{\mathrm{T}}\ \mathbf{tr}$.
- *If* $\mathrm{T}, \Upsilon \vdash M$ *and* $\vdash M\ \mathbf{tr}$ *then* $\vdash \downharpoonleft M \downharpoonright_{\Upsilon}^{\mathrm{T}}\ \mathbf{tr}$.

[2] See the cache refinement lemma in the extended technical report [7] for more detail.

Theorem 1 tells us that if we have a traceable program, then the result of unboxing it is still traceable. The second step to showing correctness is to show that unboxing does not introduce new undefined behavior.

Theorem 2 (Coherence).

- *If* $\mathrm{C}; \varrho \vdash M$, $\mathrm{C} \vdash M \downharpoonright\downharpoonleft (\mathrm{T}, \Upsilon)$, *and* $M \longmapsto^* (\rho, v^i)$ *then* $\downharpoonleft M \downharpoonleft^{\mathrm{T}}_{\Upsilon} \longmapsto^* (\downharpoonleft \rho \downharpoonleft^{\mathrm{T}}_{\Upsilon}, \downharpoonleft v^i \downharpoonleft^{\mathrm{T}}_{\Upsilon})$.
- *If* $\mathrm{C}; \varrho \vdash M$, $\mathrm{C} \vdash M \downharpoonright\downharpoonleft (\mathrm{T}, \Upsilon)$, *and* $M \longmapsto \cdots$ *then* $\downharpoonleft M \downharpoonleft^{\mathrm{T}}_{\Upsilon} \longmapsto \cdots$.

Theorem 2 shows that if two terms are related by reduction, then their images under the unboxing function are also related by the many step reduction relation given that the unboxing pair is acceptable; and that if a term diverges under reduction, then its image under the unboxing function also diverges. In other words, for an acceptable analysis and an acceptable unboxing, the induced unboxing function preserves the semantics of the original program up to elimination of boxes. Since the semantics of the core language only defines reduction steps that preserve GC safety, this theorem implies that the image of a GC safe program under unboxing is also GC safe.

5 Construction of An Acceptable Unboxing

The previous section gives a declarative specification for when an unboxing pair (T, Υ) is correct but does not specify how such a pair might be produced. In this section we give a simple algorithm for constructing an acceptable unboxing given an arbitrary acceptable flow analysis.

The idea behind the algorithm is that given a program and an acceptable flow analysis for it, we use the results of the flow analysis to construct the connected components of the inter-procedural flow graph of the program. Each connected component initially defines its own equivalence class. For each equivalence class, we then compute the least upper bound of the traceabilities of all of the introduction forms of all of the elements of the component except the box introductions. Box introductions are left initially unconstrained, since we intend to eliminate them. If the least upper bound is well-defined, then the equivalence class can potentially be eliminated. We then consider each box introduction in turn and attempt to eliminate it by combining the respective equivalence classes of the box and its contents. This is possible whenever doing so will not over-constrain the resulting combined equivalence class. When all possible boxes have been eliminated, the algorithm terminates. In the rest of the section, we make this informal algorithm concrete and show that the choice of unboxing that it produces is in fact acceptable.

For the purposes of this section we ignore environments and the intermediate forms $\rho(e)$, $\langle \rho, \lambda x^i{:}t.e \rangle^j$ and $\langle v^i{:}t \rangle^j$. These constructs are present in the language solely as mechanisms to discuss the dynamic semantics—in this sense 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), we define the induced undirected flow graph $\mathcal{FG}$ as an undirected graph with a node for every label in C, and edges as follows.

- For every label i and every shape $s \in \mathrm{C}(i)$, we add an edge between i and $\mathrm{lbl}(s)$.
- For every box introduction in the program $(\mathtt{box}_t\, e)^i$ and every shape in the cache $(\mathtt{box}_t\, j)^i \in \mathrm{C}(i)$ we add an edge between $\mathrm{lbl}(e)$ and j.

The first set of edges simply connects up each program point with all of its reaching definitions. The second set of edges is added to simplify the proofs in the pathological case that e has no reaching definitions (and hence the box itself is dynamically dead and unreachable): in the usual case where values reach e then the definition of an acceptable analysis implies that these edges are already present.

We define equivalence classes of labels as disjoint sets of labels in the usual way. The function $\mathcal{EC}$ maps labels i to the disjoint set containing i. We extend traceabilities t to a complete flat lattice $\hat{t}$ with a top element $\top$, a bottom element $\bot$ and the usual least upper bound function on $\hat{t}$.

$$
\begin{array}{cccc}
\top \sqcup t = \top & \bot \sqcup t = t & & \mathtt{b} \sqcup \mathtt{r} = \top \\
t \sqcup \top = \top & t \sqcup \bot = t & t \sqcup t = t & \mathtt{r} \sqcup \mathtt{b} = \top
\end{array}
$$

We initialize the mapping $\mathcal{EC}$ by finding the connected components of the induced undirected flow-graph $\mathcal{FG}$, and initializing $\mathcal{EC}(i)$ with the connected component containing i. As the algorithm proceeds, two equivalence classes may be collapsed into a single equivalence class requiring an updated mapping $\mathcal{EC}$.

We maintain a set of equivalence classes $\mathcal{CN}$ consisting of current candidates for unboxing. When equivalence classes are collapsed, the elements of $\mathcal{CN}$ are adjusted appropriately, as will be shown.

We maintain a set of labels Υ, which is an unboxing set in the sense of Section 4. The set Υ at all times contains all of the labels already selected for unboxing, and is initially empty.

We maintain an extended traceability map $\mathcal{T}$ that maps equivalence classes to extended traceabilities $\hat{t}$. For notational convenience we define $\mathcal{T}_{\mathcal{EC}}$ to be the derived function mapping labels to the extended traceabilities of their equivalence classes: $\mathcal{T}_{\mathcal{EC}}(k) = \mathcal{T}(\mathcal{EC}(k))$. The derivation of a standard traceability map T from an extended traceability map $\mathcal{T}$ is then given as follows.

$$
\begin{array}{ll}
\mathrm{T}(k) = \mathcal{T}_{\mathcal{EC}}(k) & \bot < \mathcal{T}_{\mathcal{EC}}(k) < \top \\
\mathrm{T}(k) = r & \mathcal{T}_{\mathcal{EC}}(k) = \bot \\
\mathrm{T}(k) = \text{undefined} & \mathcal{T}_{\mathcal{EC}}(k) = \top
\end{array}
$$

The general idea is that an equivalence class is mapped by the $\mathcal{T}$ function to the least upper bound of the traceabilities of all of the reaching definitions of all of the labels in the equivalence class. An equivalence class containing no reaching definitions will be unconstrained – for technical reasons we choose an arbitrary traceability ($\mathtt{r}$) for such classes. An equivalence class containing definitions with inconsistent traceabilities will have no defined traceability in the induced mapping.

During the algorithm traceability constraints imposed by box introductions in the candidate set are left out of the initial mapping and hence must be added back in before computing the induced traceability map. We write $\mathcal{T}^{\mathcal{CN}}$ for the extended traceability map obtained by adding in the delayed constraints for each equivalence class, and $\mathcal{T}^{\mathcal{CN}}_{\mathcal{EC}}$ for the extension of this to individual labels given by $\mathcal{T}^{\mathcal{CN}}_{\mathcal{EC}}(i) = \mathcal{T}^{\mathcal{CN}}(\mathcal{EC}(i))$.

$$\begin{array}{ll}\mathcal{T}^{\mathcal{CN}}(\mathcal{EC}(k)) = \mathcal{T}(\mathcal{EC}(k)) & \text{if } \mathcal{EC}(k) \notin \mathcal{CN} \\ \mathcal{T}^{\mathcal{CN}}(\mathcal{EC}(k)) = \mathcal{T}(\mathcal{EC}(k)) \sqcup \mathtt{r} & \text{if } \mathcal{EC}(k) \in \mathcal{CN}\end{array}$$

Note that by definition, if labels i and j are in the same equivalence class ($\mathcal{EC}(i) = \mathcal{EC}(j)$), then the traceability map T induced by an extended traceability map $\mathcal{T}$ agrees on i and j.

We define the immediate extended traceability of a labeled term $itr(e)$ as follows.

$$\begin{array}{llll} itr(c^i) & = b & itr((\mathtt{box}_t\, e)^i) & = r \\ itr((\lambda x^i{:}t.e)^j) & = r & itr(e) & = \bot \text{ otherwise}\end{array}$$

The algorithm starts with an empty unboxing set Υ. The candidate set $\mathcal{CN}$ is initialized by including $\mathcal{EC}(i)$ for each $(\mathtt{box}_t\, e)^i$ in the program. The extended traceability map is initialized by setting for each equivalence class S:

$$\mathcal{T}(S) = \bigsqcup_{i\in S,\ s\in \mathrm{C}(i),\ s\neq(\mathtt{box}_t\, k)^j} itr(s)$$

That is, we take the extended traceability associated with the equivalence class S to be least upper bound of the immediate traceabilities of all of the elements of the cache of all the labels in the equivalence class, except those which are box introductions. The practical effect of this is to make the extended traceability of every label be the least upper bound of the traceability of every introduction form in its connected component (again, excepting boxes). An equivalence class that is unconstrained ($\bot$) either counts only box introductions among its definitions, or contains no definitions at all and hence is uninhabited (this can arise because of unreachable code).

The result of the initialization phase is a $(\mathcal{T}, \Upsilon, \mathcal{CN}, \mathcal{EC})$ quadruple, which induces an unboxing pair (T, Υ) where $\mathrm{T} = \mathcal{T}^{\mathcal{CN}}_{\mathcal{EC}}$. It can be shown that the unboxing pair induced in this manner is acceptable.

Lemma 3 (The initial unboxing is acceptable). *If* $\mathrm{C}; \varrho \vdash e$ *then the unboxing quadruple* $(\mathcal{T}, \Upsilon, \mathcal{CN}, \mathcal{EC})$ *computed by the algorithm in this section induces an unboxing* (T, Υ) *(where* $\mathrm{T} = \mathcal{T}^{\mathcal{CN}}_{\mathcal{EC}}$*) such that* $\mathrm{C} \vdash (\mathrm{T}, \Upsilon)$ *and* $\mathrm{T}, \Upsilon \vdash e$.

The unboxing pair created by the initial phase is acceptable, but does no unboxing. The second phase of the algorithm proceeds by incrementally moving equivalence classes from the candidate set $\mathcal{CN}$ to the unboxing set Υ, while maintaining the invariant that at every step $(\mathcal{T}, \Upsilon, \mathcal{CN}, \mathcal{EC})$ define an acceptable (and increasingly useful) unboxing. Equivalence classes of boxes that get chosen for unboxing are collapsed into the same equivalence class as the contents of the box. We use the notation $\mathcal{EC}' = \cup_{\underline{i,j}}\mathcal{EC}$ to stand for combining the equivalence classes for i and j to get a new equivalence class in the usual way.

For the unboxing steps, we consider in turn each $(\mathtt{box}_t\, e)^i$ in the program. Let T be the traceability map induced by $\mathcal{T}$. The principal selection criterium for choosing which things to unbox is that $\mathcal{T}_{\mathcal{EC}}(i) \sqcup \mathcal{T}^{\mathcal{CN}}_{\mathcal{EC}}(\mathrm{lbl}(e)) < \top$. The idea is that under the assumption that no further unboxing is done, combining the equivalence classes for i and $\mathrm{lbl}(e)$ will not over-constrain the resulting equivalence class, and hence that the final induced traceability map will be well-defined at i and $\mathrm{lbl}(e)$. The extended traceability $\mathcal{T}_{\mathcal{EC}}(i)$ is the extended traceability associated with i under the assumption that $\mathcal{EC}(i)$ is unboxed, while $\mathcal{T}^{\mathcal{CN}}_{\mathcal{EC}}(\mathrm{lbl}(e))$ is the extended traceability associated with $\mathrm{lbl}(e)$ under the assumption that no further unboxing is done. If i is either unconstrained, or constrained to something compatible with $\mathrm{lbl}(e)$, then it is safe to unbox it.

Formally, if we have that $\mathcal{EC}(i) \in \mathcal{CN}$, $\mathcal{EC}(\mathrm{lbl}(e)) \neq \mathcal{EC}(i)$, and $\mathcal{T}_{\mathcal{EC}}(i) \sqcup \mathcal{T}^{\mathcal{CN}}_{\mathcal{EC}}(\mathrm{lbl}(e)) < \top$ then we select i for elimination. We then take the new unboxing to be the updated quadruple $(\mathcal{T}', \Upsilon', \mathcal{CN}', \mathcal{EC}')$ where:

$$\begin{array}{lll}
\mathcal{EC}' & = \cup_{\underline{\mathrm{lbl}(e),i}}\mathcal{EC} & \\
\mathcal{CN}' & = (\mathcal{CN} - \{\mathcal{EC}(\mathrm{lbl}(e)), \mathcal{EC}(i)\}) \cup \{\mathcal{EC}'(i)\} & \text{if } \mathcal{EC}(\mathrm{lbl}(e)) \in \mathcal{CN} \\
 & = (\mathcal{CN} - \{\mathcal{EC}(\mathrm{lbl}(e)), \mathcal{EC}(i)\}) & \text{if } \mathcal{EC}(\mathrm{lbl}(e)) \notin \mathcal{CN} \\
\Upsilon' & = \Upsilon \cup \{\mathcal{EC}(i)\} & \\
\mathcal{T}'(s) & = \mathcal{T}_{\mathcal{EC}}(i) \sqcup \mathcal{T}_{\mathcal{EC}}(\mathrm{lbl}(e)) & \text{if } s = \mathcal{EC}'(i) \\
 & = \mathcal{T}(s) & \text{otherwise}
\end{array}$$

For $\mathcal{EC}'$, we repartition the graph so that the equivalence classes for the box and its contents are combined into a single equivalence class. We remove the two original equivalence classes from the candidate set, and if the contents of the box was a candidate for unboxing we add back in the new equivalence class, which is the union of the two original classes. All of the elements of the original equivalence class of the box introduction are added to the unbox set. The extended traceability map is updated to map the new equivalence class (including both i and $\mathrm{lbl}(e)$) to the extended traceability of the contents of the box.

If the conditions for unboxing i are not satisfied, then we take $\mathcal{CN}' = \mathcal{CN} - \{\mathcal{EC}(i)\}$ and take $\mathcal{T}'(\mathcal{EC}(i)) = \mathcal{T}(\mathcal{EC}(i)) \sqcup r$ and leave the rest of the data structures unchanged. Since we only consider each box introduction in the program once, the algorithm terminates.

Lemma 4 (Unboxing steps preserve acceptability). *If $(\mathcal{T}, \Upsilon, \mathcal{CN}, \mathcal{EC})$ define an acceptable unboxing as constructed by the initial phase of the algorithm and maintained by the unboxing phase, then the $(\mathcal{T}', \Upsilon', \mathcal{CN}', \mathcal{EC}')$ quadruple produced by a single step of the algorithm above also define an acceptable unboxing.*

Lemma 4 states that each step of the unboxing phase of the algorithm preserves the property that the induced unboxing pair is acceptable. Consequently, the algorithm terminates with an acceptable unboxing.

Theorem 3 (The algorithm produces an acceptable unboxing). *If* $\mathrm{C}; \varrho \vdash e$ *then the algorithm defined in this section produces a quadruple* $(\mathcal{T}, \Upsilon, \mathcal{CN}, \mathcal{EC})$ *such that* $\mathrm{C} \vdash e \downharpoonright\downharpoonleft (\mathrm{T}, \Upsilon)$ *where* $\mathrm{T} = \mathcal{T}^{\mathcal{CN}}_{\mathcal{EC}}$.

This construction demonstrates that the specification defined in Section 4 is a useful one in the sense that it is satisfiable. While the algorithm defined here is unlikely to be optimal, it has proved very effective in our compiler: on floating-point intensive benchmarks we have measured an order of magnitude reduction in allocation and substantial performance and parallel scalability gains.

6 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. Our approach uses an arbitrary inter-procedural reaching definitions analysis to eliminate unnecessary heap allocation in an intermediate representation in which object representation has been made explicit. 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. For presentational purposes, we have restricted our attention to the core concern of GC safety, but additional issues such as value size, dynamic type tests, etc. are straightforward to incorporate.

There has been substantial previous work addressing the problem of unboxing. Peyton Jones [3] introduced an explicit distinction between boxed and unboxed objects to provide a linguistic account of unboxing, and hence to allow a high-level compiler to locally eliminate unboxes of syntactically apparent box introduction operations. Leroy [4] defined a type-driven approach to adding coercions into and out of specialized representations. The type driven translation represented monomorphic objects natively (unboxed, in our terminology), and then introduced 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.

Henglein and Jørgensen [2] defined a formal notion of optimality for local unboxings and gave two different choices of coercion placements that satisfy their notion of optimality. Their definition of optimality explicitly does not correspond in any way to reduced allocation or reduced instruction count and does not seem to provide uniform improvement over Leroy's approach.

The MLton compiler [10] 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, 9] 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.

Serrano and Feeley [8] described a flow analysis for performing unboxing substantially similar in spirit to our approach. Their algorithm attempts to find a monomorphic typing for 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. Their approach differs in that they define a dedicated analysis rather than using a generic reaching definitions analysis. They 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.

References

1. Harper, R., Morrisett, G.: Compiling polymorphism using intensional type analysis. In: Twenty-Second ACM Symposium on Principles of Programming Languages. pp. 130–141. San Francisco, CA (January 1995)
2. Henglein, F., Jørgensen, J.: Formally optimal boxing. In: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 213–226. POPL '94, ACM, New York, NY, USA (1994)
3. Jones, S.L.P., Launchbury, J.: 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. pp. 636–666. Springer-Verlag New York, Inc. (1991)
4. Leroy, X.: Unboxed objects and polymorphic typing. In: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 177–188. POPL '92, ACM, New York, NY, USA (1992)
5. Leroy, X.: The effectiveness of type-based unboxing. Tech. rep., Boston College, Computer Science Department (1997)
6. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
7. Petersen, L., Glew, N.: GC-safe interprocedural unboxing: Extended version (2012), http://leafpetersen.com/leaf/papers.html
8. Serrano, M., Feeley, M.: Storage use analysis and its applications. In: Proceedings of the first ACM SIGPLAN international conference on Functional programming. pp. 50–61. ICFP '96, ACM, New York, NY, USA (1996)
9. Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., Lee, P.: Til: a type-directed, optimizing compiler for ml. SIGPLAN Not. 39, 554–567 (April 2004)
10. Weeks, S.: Whole-program compilation in MLton. In: Proceedings of the 2006 workshop on ML. pp. 1–1. ML '06, ACM, New York, NY, USA (2006)