

Connecting Effects and Uniqueness with Adoption

John Tang Boyland*
University of Wisconsin—Milwaukee
Department of EECS
P O Box 784
Milwaukee, WI, USA 53201
boyland@cs.uwm.edu

William Retert*
University of Wisconsin—Milwaukee
Department of EECS
P O Box 784
Milwaukee, WI, USA 53201
williamr@cs.uwm.edu

ABSTRACT

“Adoption” is when one piece of state is logically embedded in another piece of state. Adoption provides information hiding (the adopter can be used as a proxy for the adoptee) and with linear existentials, provides a way to store unique pointers in shared state. In this paper, we give an operational semantics of adoption in a simple procedural language with pointers to records. We define a “permission” type-system that uses adoption to model both effects and uniqueness. We prove type soundness (well-typed programs don’t go wrong) and state separation (separately-typed statements cannot access the same state). Then we show how high-level effects and uniqueness annotations can be expressed in the type-system. The distinction between read and write effects is ignored in the body of this paper.

Categories and Subject Descriptors: F.3.2 [Semantics of Programming Languages] : program analysis

General Terms: Languages

Keywords: uniqueness, adoption, permissions, ownership

1. INTRODUCTION

In a previous paper [1], we discussed how the concepts of uniqueness and effects are interdependent. If one wishes to check uniqueness, it can be best done when considering effects, because read effects on a unique variable cannot be permitted while it is temporarily aliased. Checking effects on the other hand may require uniqueness, because an effect on a unique object can be transferred to the object that currently has the only unique reference. In retrospect, this interdependence could be expected because the better-known problems of data-dependence determination and aliasing are similarly related. Uniqueness involves an aliasing property and effects can be used to determine data dependencies.

*Work supported in part by the National Science Foundation (CCR-9984681) and the NASA/Ames High Dependability Computing Program (NCC-2-1298).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL’05, January 12–14, 2005, Long Beach, California, USA.
Copyright 2005 ACM 1-58113-830-X/05/0001 ...\$5.00.

1.1 Background on Uniqueness

Unique references that may occur in shared structures can be modeled soundly using “destructive reads” in which the stored pointer variable (often a field of an object) is nullified atomically with the read of that variable. This solution goes back at least to Hogg’s Islands [2] and has been variously used by Baker [3], Minsky’s Eiffel* [4], Aldrich et al’s AliasJava [5] (in their proofs), and Clarke and Wrigstad’s External Uniqueness [6].

However, destructive reads have several problems: (1) they make it difficult to query information about a unique object without losing it; (2) they make it impossible to have sound invariants about the non-nullness of unique variables; (3) they are inappropriate for use in “const” methods, which are supposed to treat their object as read only; (4) they require a language change. Thus several researchers have independently proposed the use of what we call “borrowing” reads, which do not nullify the field. Unfortunately borrowing reads greatly impact the benefits of uniqueness unless it can be shown that a unique field is not read (or at least not considered unique) during the lifetime of the borrowing. For example, AliasJava permits borrowing without checking for possible reads during the lifetime of the borrow and thus can only guarantee that a “unique” pointer stored in a field will not be also available in another field. In particular, a class cannot prevent “outsiders” from modifying the contents of supposedly unique sub-objects of instances of this class. Consider Figure 1. The method `badBorrow()` is able to use its borrowed `Point` to modify a point which had been transferred to a `Rectangle`. In particular, the assignment deforms the `Rectangle` such that the `area()` method is no longer correct.

A related problem occurs in Vault [7] where one cannot store unique (“linear”) values inside shared (“nonlinear”) structures. Fähndrich and DeLine solve this problem with “adoption and focus” [8] without using destructive reads, but rather temporary removal of the rights to access data that might be in an inconsistent state. The authors give a type system but no formal statement or proof of what properties are preserved by the system.

Leino, Nelson and Stata [9, 10] propose a pointer property, “virginity,” related to uniqueness that is required of all initialization of “pivot” fields (fields that refer to wholly-owned subsidiary objects or “sub-objects”). The rules have a similar weakness to that of AliasJava: they are (intentionally) not strong enough to prevent a client of an object from having access to its sub-objects.

Leino and others’ recent work [11] and “strong” ownership

```

class Rectangle {
  group Looks;
  group Dims in Looks;
  unique Point tl in Dims;
  unique Point br in Dims;
  shared String nm in Looks = "";

  Rectangle(unique Point topLeft,
            unique Point bottomRight) {
    tl = topLeft; br = bottomRight;
    assert(tl.x < br.x);
    assert(tl.y < br.y);
  }

  accesses Dims
  int area() {
    return (br.x-tl.x)*(tl.y-br.y);
  }
  accesses Looks
  void setName(shared String n) {
    nm = n;
  }
}

class Point {
  group Loc;
  int x in Loc;
  int y in Loc;
  Point(int a,int b) { x=a; y=b; }
}

class RMaker {
  unique Point p;
  RMaker() {
    p = new Point(0,0);
  }
  accesses p
  unique Rectangle rmake() {
    unique Point tl = p;
    p = new Point(0,0);
    unique Point br = new Point(5,5)
    return new Rectangle(tl,br)
  }
  accesses p
  Rectangle badBorrow() {
    borrowed Point q = p;
    Rectangle r = rmake();
    q.x = 17;
    return r;
  }
}

```

Figure 1: Example of dubious borrowing

type systems [12, 13] avoid these aliasing problems by not permitting clients to create objects that will be internal to a container. In essence, the system forbids objects from being transferred from one container to another. External uniqueness extends ownership to cover transferable uniqueness, but does so through “temporary” destructive reads. With external uniqueness, the example in Figure 1 will pass all static checks and crash with a null pointer exception when a null `Point` is passed to the `Rectangle` constructor. There is no obvious connection between the uniqueness/borrowing error and the null pointer exception.

In our earlier work on “alias burying” [14], we proposed using effects annotations to prevent borrowing reads from weakening the semantics of uniqueness. With alias burying, no destructive reads are needed. The paper suggested annotating every method with the list of fields of this or any other object that is read during the dynamic extent of the method call. Clearly, this not only exposes too much of object’s internal structure, but is overly conservative since it does not distinguish different objects. We intended to use our object-oriented effects system [15] instead, but until this work, no one has been able to come up with a satisfactory system to combine these two areas. This paper contributes such a type system, based on adoption and focus, and proves correctness with respect to an underlying semantics.

1.2 Contribution

We can check that program accesses meet declared effects by using a *permission* system, in which the context used to check program elements indicates which parts of the state we are allowed to access. “Fractional permissions” [16] can be used to distinguish reads from writes, but this paper does not formalize this aspect. Effects are in terms of *state* in the program: variables, and fields of objects on the heap. For encapsulation purposes, we aggregate fields into “data groups” [15, 17]. This is modeled by having the permission to access the field “nested” within the permission to access the data group. For instance, in Figure 1 the effects of the `area()` method reference the `Dims` data group, which includes the references for both points. As these references are unique, they in turn contain permission for the state of the objects referenced. “Nesting” is defined by extending “adoption” [8] to permit fine-grained adoption between fields (rather than between objects). The extension also permits multiple (distinct) keys with the same guard to be “focused” on as well.

In a permission system, there is only one permission for each state. Thus permissions can also be used to model uniqueness: a unique pointer is one which is packaged along with the permission to access the state pointed to. This conception of uniqueness is weaker than the usual sense of uniqueness in which there are no other pointers to the state in the store. However, in our system, any other potential pointers to the same state come without permission to access it, and thus even the limited sense of uniqueness provided by permissions is sufficient for analysis purposes. The problem is not the *existence* of aliasing pointers, but rather the *access* of the state through these aliasing pointers.

Let’s return to the method `badBorrow()` in Figure 1. The borrowing read of `p` is legal, as `q` can borrow `p`’s permission to access the state of the `Point` object. However, the call to `rmake()` is annotated with an access of `p`’s state. Therefore,

permission to access p is passed to `rmake()` and lost to q .¹ Once q loses the permission it had borrowed from p , the access $q.x$ becomes statically illegal.

Of course not all pointers are unique. A *shared* pointer is modeled by having the permission to access the pointed-to state “nested” in a globally accessible permission. Now, we want to make sure that it is impossible for two separate parts of the code to grab the permissions to the same shared state. This could be done using dynamic checks, but in our system we use the type system to prevent the same permission from being removed twice at the same time.

Languages often permit a “null” pointer to be used in the place of a pointer to actual state. Our system makes this detail explicit: the permissions associated with a possibly-null pointer are conditional on the boolean formula that the pointer is not null.

Thus the type system described in this paper has the following properties:

- Permissions model both effects and uniqueness.
- The “nesting” (adoption) of permissions models shared state and aggregation of state in “data groups.”
- “Permission closures” model uniqueness.
- Conditional permissions make explicit the use of “null” or other terminating pointers.

In the following section, we describe this type system, and then in Section 3 we show how some of our earlier effects annotations together with the uniqueness annotations of our work on Alias Burying can be interpreted in the system proposed in this paper. After some extensions to the type system are considered in Section 4, Section 5 compares our system to closely related work.

2. THE SYSTEM

We define our type system over a simple low-level imperative language.

2.1 Operational Semantics

The source language has four kinds of values: the unit value, booleans, integers and pointer values (that is, object references):

$$\text{(value)} \quad v ::= () \mid \mathbf{true} \mid \mathbf{false} \mid n \mid \nu$$

Here ν refers to an object reference, an absolute memory address. The only absolute memory address that occurs normally in unevaluated programs is $\$0$ —the value of null pointers.

In this simple language, the only kind of variable is the field of an object (global variables are handled by treating them as fields of the null object). Array elements could be handled with dependent types [18] in a similar manner.

Each field f has a declared default value and type chosen

¹Actually, the method also returns a permission to access p , but wrapped in an existential, making it impossible to prove that q refers to the same object. In this example, of course, q does *not* refer to the same object as p after the call.

from the following limited set:

v_f	τ_f
$()$	unit
\mathbf{false}	bool
0	int
$\$0$	ptr($\$0$)

Expressions include values, simple arithmetic (represented by addition), comparisons, allocation, field reads and writes, sequential composition, conditionals, procedure calls and “nesting” (adoption):

$$v \mid e+e \mid e=e \mid \mathbf{new}\{f, \dots, f\} \mid e.f \mid \\ s ::= e.f := e \mid s; s \mid \mathbf{if} \ b \ \mathbf{then} \ s \ \mathbf{else} \ s \mid \mathbf{call} \ p \mid \\ \mathbf{nest} \ e.f \ \mathbf{in} \ e.f$$

A program consists of a mapping of procedure names to procedure expressions. A memory consists of a binding of fields to their values (a function). We also record “adoption” (the nesting relation).

$$\begin{aligned} \text{(program)} \quad g & ::= \{p \rightarrow e, \dots\} \\ \text{(location)} \quad l & ::= \nu.f \\ \text{(memory)} \quad \mu & ::= \{l \rightarrow v, \dots\} \\ \text{(adoption)} \quad a & ::= \{l \prec l, \dots\} \end{aligned}$$

There is no requirement that adoption is a functional relation, although we will require it to be acyclic.

Figure 2 defines a small-step semantics for evaluating this language. The three interesting parts are (1) the evaluation of `new` expressions in which an address is found unused in the memory or adoption information, and (2) for writes to fields, we require that the new value of the field be storage compatible with the default value of the field (the relation is defined using storage compatibility for types), and (3) nesting which adds an adoption relation fact. Adoption can never be undone. An actual implementation of a runtime system would not need to check storage compatibility or track the adoption relation; including these properties here enables us to prove that well-typed programs preserve them.

2.2 Permission Types

The environment $E = (\Delta; \Pi)$ in which code is checked has two parts: a type context Δ which is a set of address variables r ; and a bag (or multiset) of permissions Π that indicates what state we are permitted to access and what type that state will have. Notationally, we use \emptyset for the empty bag, $\{\dots\}$ to indicate bag values, and $+$ for bag union. As syntactic sugar, we use a comma operator to represent the flattened bag union operator

$$a, b \triangleq \{\{a\}\} + \{\{b\}\} \quad a, B \triangleq \{\{a\}\} + B \quad A, B \triangleq A + B$$

Bag unions are commutative, associative and have \emptyset as the identity element. Using a bag instead of an ordinary set enables us to identify duplicate permissions.

A related operator is \uplus , the union of disjoint sets. This operator is commutative, associative, and feature the empty set $\{\}$ as an identity element. The union of disjoint sets, unlike the standard disjoint union operation, is only partially defined; thus $A \uplus A = A$ is true only for $A = \emptyset$. Additionally, cancellation is well-defined:

$$A \uplus B = A \uplus C \Rightarrow B = C.$$

$$\begin{array}{c}
\text{E-PLUS1} \\
\frac{(\mu; a; e_1) \rightarrow_g (\mu'; a'; e'_1)}{(\mu; a; e_1 + e_2) \rightarrow_g (\mu'; a'; e'_1 + e_2)} \\
\\
\text{E-EQUAL1} \\
\frac{(\mu; a; e_1) \rightarrow_g (\mu'; a'; e'_1)}{(\mu; a; e_1 = e_2) \rightarrow_g (\mu'; a'; e'_1 = e_2)} \\
\\
\text{E-NEW} \\
\frac{\nu \notin \text{Rng}(\mu) \quad \forall f \in F \nu.f \notin \text{Dom}(\mu) \cup \text{Dom}(a) \cup \text{Rng}(a)}{(\mu; a; \text{new}\{f_1, \dots, f_n\}) \rightarrow_g (\mu[\nu.f_i \rightarrow v_{f_i} \mid 1 \leq i \leq n]; a; \nu)} \\
\\
\text{E-WRITE1} \\
\frac{(\mu; a; e_1) \rightarrow_g (\mu'; a'; e'_1)}{(\mu; a; e_1.f := e_2) \rightarrow_g (\mu'; a'; e'_1.f := e_2)} \\
\\
\text{E-IF1} \\
\frac{(\mu; a; e_1) \rightarrow_g (\mu'; a'; e'_1)}{(\mu; a; \text{if } e_1 \text{ then } e_2 \text{ else } e_3) \rightarrow_g (\mu'; a'; \text{if } e'_1 \text{ then } e_2 \text{ else } e_3)} \\
\\
\text{E-IFFALSE} \\
(\mu; a; \text{if false then } e_2 \text{ else } e_3) \rightarrow_g (\mu; a; e_3) \\
\\
\text{E-ADOPT1} \\
\frac{(\mu; a; e_1) \rightarrow_g (\mu'; a'; e'_1)}{(\mu; a; \text{nest } e_1.f_1 \text{ in } e_2.f_2) \rightarrow_g (\mu'; a'; \text{nest } e'_1.f_1 \text{ in } e_2.f_2)} \\
\\
\text{E-ADOPT2} \\
\frac{(\mu; a; e_2) \rightarrow_g (\mu'; a'; e'_2)}{(\mu; a; \text{nest } v_1.f_1 \text{ in } e_2.f_2) \rightarrow_g (\mu'; a'; \text{nest } v_1.f_1 \text{ in } e'_2.f_2)} \\
\\
\text{E-PLUS2} \\
\frac{(\mu; a; e_2) \rightarrow_g (\mu'; a'; e'_2)}{(\mu; a; v + e_2) \rightarrow_g (\mu'; a'; v + e'_2)} \\
\\
\text{E-EQUAL2} \\
\frac{(\mu; a; e_2) \rightarrow_g (\mu'; a'; e'_2)}{(\mu; a; v = e_2) \rightarrow_g (\mu'; a'; v = e'_2)} \\
\\
\text{E-READ1} \\
\frac{(\mu; a; e) \rightarrow_g (\mu'; a'; e')}{(\mu; a; e.f) \rightarrow_g (\mu'; a'; e'.f)} \\
\\
\text{E-READ} \\
\frac{\mu(\nu.f) = v}{(\mu; a; \nu.f) \rightarrow_g (\mu; a; v)} \\
\\
\text{E-WRITE2} \\
\frac{(\mu; a; e_2) \rightarrow_g (\mu'; a'; e'_2)}{(\mu; a; v_1.f := e_2) \rightarrow_g (\mu'; a'; v_1.f := e'_2)} \\
\\
\text{E-WRITE} \\
\frac{v_2 \sim v_f \quad \mu' = \mu[\nu.f \rightarrow v_2]}{(\mu; a; \nu.f := v_2) \rightarrow_g (\mu'; a; ())} \\
\\
\text{E-SEQ1} \\
\frac{(\mu; a; e_1) \rightarrow_g (\mu'; a'; e'_1)}{(\mu; a; e_1; e_2) \rightarrow_g (\mu'; a'; e'_1; e_2)} \\
\\
\text{E-SEQ} \\
(\mu; a; () ; e_2) \rightarrow_g (\mu; a; e_2) \\
\\
\text{E-IFTRUE} \\
(\mu; a; \text{if true then } e_2 \text{ else } e_3) \rightarrow_g (\mu; a; e_2) \\
\\
\text{E-CALL} \\
(\mu; a; \text{call } p) \rightarrow_g (\mu; a; gp)
\end{array}$$

Figure 2: Operational Semantics

$$\vdash () : \text{unit} \qquad \vdash b : \text{bool} \qquad \vdash n : \text{int} \qquad \vdash \nu : \text{ptr}(\nu)$$

Figure 3: Atomic types for values: $\vdash v : \tau$

$$\frac{\vdash v_1 : \tau_1 \quad \vdash v_2 : \tau_2 \quad \tau_1 \sim \tau_2}{v_1 \sim v_2} \qquad \frac{\tau \in \{\text{unit}, \text{bool}, \text{int}, \text{ptr}(\rho) \mid \rho \in N \cup R\}}{\tau \sim \tau} \qquad \text{ptr}(\rho) \sim \text{ptr}(\rho')$$

Figure 4: Storage compatibility for values and types

For any environment $E = (\Delta; \Pi)$, we require that all free variables in Π are in Δ . For brevity purposes, this restriction is left implicit.

We treat Π in a somewhat linear fashion, in that permissions cannot be duplicated. However, making use of permissions does not consume them. The basic permission is the permission for a key k . As mentioned earlier, a key here means a *field* of an object, whereas earlier proposals use a whole object as a key. The permission to access a field names the object reference ρ (which is either a literal object reference ν or a reference variable r) and also includes the type stored there. We have four atomic types: (The use of “...” here means there are other forms for types and permissions.)

$$\begin{aligned} \text{(object reference)} \quad \rho &::= \nu \mid r \\ \text{(key)} \quad k &::= \rho.f \\ \text{(type)} \quad \tau &::= \text{unit} \mid \text{bool} \mid \text{int} \mid \text{ptr}(\rho) \mid \dots \\ \text{(base permission)} \quad \beta &::= k : \tau \\ \text{(permission)} \quad \pi &::= \beta \mid \dots \end{aligned}$$

Two atomic types τ and τ' are “storage compatible” (written $\tau \sim \tau'$) if space storing a value of type τ can be reused to store a τ' . This relation (see Fig. 4) is only true for the four atomic types listed here, and thus $\tau \sim \tau$ if and only if $\tau \in \{\text{unit}, \text{bool}, \text{int}, \text{ptr}(\rho) \mid \rho \in N \cup R\}$, where N is the set of all object references and R is the set of all reference variables. Additionally all pointer types are interchangeable: $\text{ptr}(\rho) \sim \text{ptr}(\rho')$.

Fields of unit type do not store any information, but instead are used to model “data groups” (we use the terminology of Leino and others [17, 11] with the semantics of our “regions” [15]). Aggregation of fields into data groups is accomplished by nesting, for instance $\rho.x : \text{int} \prec \rho.\text{loc}$ means the “ x ” field of the object at ρ is part of the “loc” data group. We assume the existence of a field “All” with type $\tau_{\text{All}} = \text{unit}$. Conventionally, an object constructor may ensure that all of its fields are adopted into its “All” data group, directly or indirectly.

Permissions may be combined using the comma operator, as explained above.² We denote a general bag of permissions $\Pi = \{\{\pi, \dots\}\}$, or a bag of base permissions $B = \{\{\beta, \dots\}\}$.

The type system described here uses a simple logic that can be represented by boolean formulae over equalities $k = k'$ and typed adoption facts $k : \tau \prec k'$:

$$\begin{aligned} \text{(formula)} \quad \Gamma &::= \text{true} \mid \Gamma \wedge \Gamma \mid \neg \Gamma \mid k = k' \mid \\ &k : \tau \prec k' \mid \dots \end{aligned}$$

We write $k \neq k'$ as shorthand for $\neg(k = k')$, and $\rho = \rho'$ as shorthand for $\rho.\text{All} = \rho'.\text{All}$, where “All” is a data group in which all of the fields of the object are nested, perhaps indirectly. A boolean formula can be used as a permission. It does not give any permission to access the heap, but it does constrain which situations can occur.³ Boolean formulae represent fixed truth, since the equality of two addresses is immutable and adoption cannot be undone. The value of a formula (especially adoption) may be unknown, but once it gains a value, that value cannot change during execution.

²The comma operator is roughly the equivalent of the \star in separation logic, while the empty bag is similar to “**emp**.”

³Thus our system can be seen as a particular model of the logic of Bunched Implications [19, 20].

Since adoptions (nesting) can happen at any later time, it will be seen that the consistency rules never give a false value to an adoption fact; either it is known to be true (and thus unchangeable) or it is not known. The syntax also provides a construct for named recursive formulae t over object references:

$$\text{(formula)} \quad \Gamma ::= \dots \mid t(l_1, \dots, l_n)$$

The formula t is given a definition by an implicit global environment T : $T(t)$ is a formula with free variables in $\{r_1, \dots, r_n\}$. We implicitly require/assume that all uses of a formula supply the correct number of arguments.

Following “Adoption & Focus” [8], a *conditional permission* is written in the form of a linear implication using \multimap .⁴ There are two forms of this operator, $\Gamma \multimap \Pi$ and $B \multimap \Pi$. The first form means that we have all the permissions implied by Π if the formula Γ is true and no permissions otherwise. The second form means that we have all the permissions implied by Π except for those base permissions contained in B , which must be contained directly or indirectly within Π . We don’t permit the general form $\Pi_1 \multimap \Pi_2$ because of technical difficulties. All permissions are *precise* [21]: given a particular assignment to the address variables, and a particular assignment of values in the heap, the permissions can be satisfied in only a single way. We exploit precision in the way we check and manipulate (conditional) permissions.

Thus we have the following three remaining forms for permissions:

$$\text{(permission)} \quad \pi ::= \dots \mid \Gamma \mid \Gamma \multimap \Pi \mid B \multimap \Pi$$

For example, suppose \mathbf{x} is a global variable (represented by a field of the null object) that may be null, but if it is not null, we have permission to access all of the fields of the pointed-to object. This situation is represented by the following bag of permissions:

$$\$0.\mathbf{x} : \text{ptr}(\rho), \rho \neq \$0 \multimap \rho.\text{All} : \text{unit}$$

Now, we often do not know what the actual pointer value of a variable is and thus need to use some form of quantified types. For this reason a variable may have existential type, for example $\mathbf{x} : \exists r.\text{ptr}(r)$, but this form is not sufficient, because we need to be able to express permissions about the existentially bound address variable. Thus we have a form for compound types which connects a bag of permissions with an existential:

$$\text{(type)} \quad \tau ::= \dots \mid \exists r.\tau \text{ with } \Pi$$

In order to easily maintain precision, the type system described in this paper only accepts $\tau = \text{ptr}(r)$ in this context.

Suppose that \mathbf{x} , if not null, points to some unique object. We can express this situation as follows:

$$\$0.\mathbf{x} : \exists r.\text{ptr}(r) \text{ with } r \neq \$0 \multimap r.\text{All} : \text{unit}$$

This is exactly how we express the concept of a unique pointer. While other fields in the system may have the same pointer value, no one else has the permission to access the pointed-to state, since permissions cannot be duplicated. This formulation is essentially the same as in Vault [7] with

⁴The separation logic implication operator \multimap is closer in semantics.

the addition of conditional permissions which make the possible null-ness of a pointer explicit.

The other form of conditional permissions is used to represent “carvings out” in which permissions of the adopted fields are carved out of the adopter permission. We write

$$k : \tau \setminus \{k_1 : \tau_1, \dots, k_n : \tau_n\}$$

as a shorthand precisely for

$$(k_1 : \tau_1 \prec k \wedge \dots \wedge k_n : \tau_n \prec k), \\ ((k_1 : \tau_1, \dots, k_n : \tau_n) \multimap k : \tau)$$

One use of this shorthand is that the type system allows this compound permission to be sufficient to access the value stored in field k . It is also possible to carve out further adoptees that are known to be distinct from the existing carvings out.

A procedure may be polymorphic over some location variables Δ . It accepts a bag of permissions and returns a (possibly new) bag of permissions, using perhaps some new variables. A program type has a type for each procedure. Substitutions map address variables to other variables or to absolute addresses.

$$\begin{aligned} \text{(procedure type)} \quad \alpha &::= \forall \Delta. (\Pi \rightarrow \exists \Delta. \Pi) \\ \text{(program type)} \quad \omega &::= \{p : \alpha, \dots\} \\ \text{(substitution)} \quad \sigma &::= \{r \rightarrow \rho, \dots\} \end{aligned}$$

For a procedure type $\forall \Delta_1. \Pi_1 \rightarrow \exists \Delta_2. \Pi_2$, the free variables of Π_1 must be included in Δ_1 and the free variables of Π_2 must be included in $\Delta_1 \uplus \Delta_2$. Parameters and results can be passed in global variables or in specially created activation objects. Substitutions are used to transform the permissions into a form accepted by a procedure and to convert the resulting permissions back. Substitutions are typed by their domain and range:

$$\frac{\text{Dom}(\sigma) = \Delta \quad \text{Rng}(\sigma) \subseteq \Delta' \cup N}{\sigma : \Delta \rightarrow \Delta'}$$

Here N is the set of all absolute (object) addresses. Substitutions are lifted to apply to permissions in the normal way, and in particular σ acts as the identity function on address variables outside of its explicit domain.

The type rules corresponding to the syntax are given in Figure 5. Each expression produces a possibly new environment after being checked and thus our relation is $E \vdash_\omega e : \tau \dashv E'$ where ω is the program typing. We will often chain together relations as in $E \vdash_\omega e : \tau \dashv E' \vdash_\omega e' : \tau' \dashv E''$.

The IF rules bear some explanation: as well as a default rule, we have a special case rule for the comparing of pointers. The type system makes the equality or inequality being tested available in the corresponding branches. At the end of an **if**, we need to merge the two environments (see Figure 6), which makes use of substitutions. The IFTRUE and IFFALSE rules are needed for type preservation.

The rule for **nest** expressions requires that the key being “adopted” be fully available (without anything carved out) and not equal to any key currently carved out of the “adopter.” This rule prevents keys from being twice adopted by the same adopter, and also prevents self or cyclic adoption. It does *not* prevent a key from being adopted by two different adopters.

The rules enable the following widening lemma (an analogue of the frame axiom of separation logic):

LEMMA 2.1. *Given a type-checked program g ($\vdash g : \omega$), an expression e that type-checks $(\Delta; \Pi_1 \vdash_\omega e : \tau \dashv \Delta'; \Pi'_1)$ and an environment $\Delta''; \Pi_2$, then e also type-checks with a larger set of permissions $(\Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash_\omega e : \tau \dashv \Delta' \cup \Delta''; \Pi'_1, \Pi_2)$ in which the unused permissions are not changed.*

2.3 Consistency

Types can ensure that programs don’t go wrong, but only if the memory is in a state matched by the types, or permissions in our case. This is represented by the consistency relation which is written

$$\mu; a \vdash \Delta; \Pi \text{ consistent using } \sigma; (A_\prec, A_T)$$

and is defined using the inference rules in Fig. 10.⁵ Connecting the static types to the run-time state has two aspects: object variables must be replaced with absolute addresses, and implied permissions (from existentials or adoption) must be made explicit.

The first witness to consistency is therefore a substitution $\sigma : \Delta \rightarrow \emptyset$ that maps each object variable to an absolute address: the empty set range meaning that the result of the substitution uses no variables.

A second witness to consistency is the pair of assumption sets $A = (A_\prec, A_T)$. The first part of the assumptions is a set of typed adoption facts $A_\prec = \{l : \tau \prec l', \dots\}$ where τ has no free variables. The second set of the assumptions is a set of boolean formulae assumed true $A_T = \{t(\nu_1, \dots, \nu_n), \dots\}$. Figure 7 shows how boolean conditions are evaluated given these assumptions. Adoption facts or applications outside of these sets are *not* assumed to be false; rather they are unknown. Location equality is known to be true or false because all addresses are absolute.

For consistency checking, the substituted permissions $\sigma \Pi$ are flattened into a set $\hat{\Pi}$ of permissions of the form:

$$\begin{aligned} \text{(flat permissions)} \quad \hat{\Pi} &::= \{l : \tau_{\text{atom}}, \dots\} \\ \text{(primitive type)} \quad \tau_{\text{atom}} &::= \text{unit} \mid \text{bool} \mid \text{int} \mid \text{ptr}(\nu) \end{aligned}$$

The flattening operation unpacks all existentials and carves out all adopted permissions so that one is left with a map that gives an atomic type for every location that the original permissions implied. Figure 10 shows how the flattened permissions are checked for consistency against the memory. It is also necessary to ensure that there are not two permissions to the same state. If the permissions involve distinct (atomic) types, the memory consistency check will fail. To prevent two identical permissions from being accepted, the rules only use union of disjoint sets \uplus to combine flattened permissions. The last two requirements of Figure 10 check the adoption assumptions against the actual adoption state, and check the bodies of the assumed predicates.

The bulk of the flattening work is done in Figure 8 which defines the relation $\mu; A; B \vdash \Pi \Downarrow \hat{\Pi}$. The B set consists of permissions that are carved out of Π and thus must be accounted for exactly once while flattening Π (and removed), as can be seen in rules CP-IMP and CP-IDENTITY.

Of special interest here is the CP-FIELD rule which handles adoption. For every typed adopted field $l' : \tau' \prec l$, we flatten this field in turn. The acyclicity of adoption ensures that this rule does not lead to infinite recursion. The requirements B are split between B_1 , which is further split

⁵Consistency serves a similar purpose to that of the forcing relation (\models) in separation logic.

$$\begin{array}{c}
\text{UNIT} \\
E \vdash_{\omega} () : \text{unit} \dashv E \\
\\
\text{PLUS} \\
\frac{E \vdash_{\omega} e_1 : \text{int} \dashv E' \vdash_{\omega} e_2 : \text{int} \dashv E''}{E \vdash_{\omega} e_1 + e_2 : \text{int} \dashv E''} \\
\\
\text{NUM} \\
E \vdash_{\omega} n : \text{int} \dashv E \\
\\
\text{TRUE} \\
E \vdash_{\omega} \text{true} : \text{bool} \dashv E \\
\\
\text{FALSE} \\
E \vdash_{\omega} \text{false} : \text{bool} \dashv E \\
\\
\text{ADDRESS} \\
E \vdash_{\omega} \nu : \text{ptr}(\nu) \dashv E \\
\\
\text{EQUAL} \\
\frac{E \vdash_{\omega} e_1 : \tau_1 \dashv E' \vdash_{\omega} e_2 : \tau_2 \dashv E'' \quad \tau_1 \sim \tau_2}{E \vdash_{\omega} e_1 = e_2 : \text{bool} \dashv E''} \\
\\
\text{NEW} \\
\frac{r \text{ fresh}}{\Delta; \Pi \vdash_{\omega} \text{new}\{f_i \mid 1 \leq i \leq n\} : \text{ptr}(r) \dashv \{r\} \cup \Delta; r.f_i : \tau_{f_i} \mid 1 \leq i \leq n, \Pi} \\
\\
\text{READ} \\
\frac{E \vdash_{\omega} e : \text{ptr}(\rho) \dashv \Delta'; \Pi' \quad \Pi' = \rho.f : \tau \setminus \{\dots\}, \Pi_1 \quad \tau \sim \tau}{E \vdash_{\omega} e.f : \tau \dashv \Delta'; \Pi'} \\
\\
\text{WRITE} \\
\frac{E \vdash_{\omega} e_1 : \text{ptr}(\rho) \dashv E' \vdash_{\omega} e_2 : \tau \dashv \Delta''; \Pi'' \quad \Pi'' = \rho.f : \tau' \setminus \{\dots\}, \Pi_1 \quad \tau \sim \tau' \quad \tau \sim \tau_f}{E \vdash_{\omega} e_1.f := e_2 : \text{unit} \dashv \Delta''; \rho.f : \tau \setminus \{\dots\}, \Pi_1} \\
\\
\text{SEQ} \\
\frac{E \vdash_{\omega} e : \text{unit} \dashv E' \vdash_{\omega} e' : \tau \dashv E''}{E \vdash_{\omega} e; e' : \tau \dashv E''} \quad \text{IF} \\
\frac{E \vdash_{\omega} e_0 : \text{bool} \dashv E' \quad E' \vdash_{\omega} e_1 : \text{unit} \dashv E_1 \quad E' \vdash_{\omega} e_2 : \text{unit} \dashv E_2 \quad E'' = E_1 \vee E_2}{E \vdash_{\omega} \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : \text{unit} \dashv E''} \\
\\
\text{IFEQUAL} \\
\frac{E \vdash_{\omega} e : \text{ptr}(\rho) \dashv E' \vdash_{\omega} e' : \text{ptr}(\rho') \dashv \Delta; \Pi \quad \Delta; (\rho = \rho', \Pi) \vdash_{\omega} e_1 : \text{unit} \dashv E_1 \quad \Delta; (\rho \neq \rho', \Pi) \vdash_{\omega} e_2 : \text{unit} \dashv E_2 \quad E'' = E_1 \vee E_2}{E \vdash_{\omega} \text{if } e = e' \text{ then } e_1 \text{ else } e_2 : \text{unit} \dashv E''} \\
\\
\text{IFTRUE} \\
\frac{E \vdash_{\omega} e_1 : \text{unit} \dashv E'}{E \vdash_{\omega} \text{if true then } e_1 \text{ else } e_2 : \text{unit} \dashv E'} \quad \text{IFFALSE} \\
\frac{E \vdash_{\omega} e_2 : \text{unit} \dashv E'}{E \vdash_{\omega} \text{if false then } e_1 \text{ else } e_2 : \text{unit} \dashv E'} \\
\\
\text{CALL} \\
\frac{\omega(p) = \forall \Delta_1. \Pi_1 \rightarrow \exists \Delta_2. \sigma_2 \Pi_2 \quad \sigma_1 : \Delta_1 \rightarrow \Delta \quad \Delta' \text{ fresh} \quad \sigma_2 : \Delta' \rightarrow \Delta_2}{\Delta; \sigma_1 \Pi_1, \Pi_3 \vdash_{\omega} \text{call } p : \text{unit} \dashv \Delta \cup \Delta'; \sigma_1 \Pi_2, \Pi_3} \\
\\
\text{NEST} \\
\frac{E \vdash_{\omega} e : \text{ptr}(\rho) \dashv E' \vdash_{\omega} e' : \text{ptr}(\rho') \dashv \Delta''; \Pi'' \quad k = \rho.f \quad k' = \rho'.f' \quad \Pi'' = (k : \tau, k \neq k_1 \wedge \dots \wedge k \neq k_n), k' : \tau' \setminus \{k_1 : \tau_1, \dots, k_n : \tau_n\}, \Pi_1}{E \vdash_{\omega} \text{nest } e.f \text{ in } e'.f' : \text{unit} \dashv \Delta''; (k' : \tau' \setminus \{k_1 : \tau_1, \dots, k_n : \tau_n\}; k : \tau \prec k'), \Pi_1} \\
\\
\text{PROC} \\
\frac{\Delta_1; \Pi_1 \vdash_{\omega} e : \text{unit} \dashv \Delta'_1; \sigma \Pi_2 \quad \Delta'_1 \cap \Delta_2 = \emptyset \quad \sigma : \Delta_2 \rightarrow \Delta'_1}{\vdash_{\omega} e : \forall \Delta_1. \Pi_1 \rightarrow \exists \Delta_2. \Pi_2} \quad \text{PROGRAM} \\
\frac{p : \alpha \in \omega \Rightarrow \vdash_{\omega} g(p) : \alpha}{\vdash g : \omega}
\end{array}$$

Figure 5: Syntactic Type Rules

$$\frac{\Delta_1; \Pi_1 = \sigma_1 \Delta'_1; \sigma_1 \Pi'_1 \quad \Delta_2; \Pi_2 = \sigma_2 \Delta'_2; \sigma_2 \Pi'_2 \quad \Delta = \Delta_1 \cap \Delta_2 \quad \Delta' - \Delta \text{ fresh} \quad r \in \Delta \Rightarrow \sigma_i r = r}{(\Delta'; \Pi') = (\Delta_1; \Pi_1) \vee (\Delta_2; \Pi_2)}$$

Figure 6: Environment Merging

into a bag for each adopted field, and B_2 , which is satisfied by τ . All the resulting permissions are combined using union of disjoint sets. Figure 9 handles the possible unpacking of τ using the actual memory value of the location. If the type is already atomic, it does nothing (not even checking that the value has the designated type). For an existential, we use the value to substitute the object variable.

2.4 Transformation

The type rules given above do not have any way to use (or create) conditional permissions, or to carve out or replace permissions from their adopters. Thus we add the concept of transforming permissions. In a typing proof, one may insert permission transformations before or after any expression using rule TRANSFORM in Fig. 11. (This rule is the equivalent of Hoare’s rule of consequence.) The figure also shows that the condition for a valid transformation is that consistency remains unchanged (using the same or larger substitution and adoption assumptions—the other assumptions are verified in consistency).

We prove a lemma that transformations are compositional:

LEMMA 2.5. *If we have two transformations: $\Delta; \Pi_1 \geq \Delta'_1; \Pi'_1$ and $\Delta; \Pi_2 \geq \Delta'_2; \Pi'_2$ where the fresh variables introduced are disjoint $(\Delta'_1 - \Delta) \cap (\Delta'_2 - \Delta) = \emptyset$, then the two transformations can be merged: $\Delta; \Pi_1, \Pi_2 \geq \Delta'; \Pi'_1, \Pi'_2$ where $\Delta' = \Delta'_1 \cup \Delta'_2$.*

The definition of $E \geq E'$ is not constructive, but it is easy to define (and prove soundness) of such special cases as in the following lemma:

LEMMA 2.8. *The following rules hold:*

$$\begin{array}{l}
E \equiv E \quad \Delta; \Pi \geq \Delta; \emptyset \quad \Delta; \emptyset \equiv \Delta; \text{true} \\
\Delta; \Gamma \wedge \Gamma' \equiv \Delta; \Gamma, \Gamma' \quad \Delta; \Gamma \equiv \Delta; \neg\neg\Gamma \\
\Delta; t(\rho_1, \dots, \rho_n) \equiv \Delta; [r_1 \rightarrow \rho_1, \dots, r_n \rightarrow \rho_n]T(t) \\
(\Delta; \Pi \geq \Delta; \Gamma) \Rightarrow (\Delta; \Pi \equiv \Delta; \Pi, \Gamma) \quad \Delta; \neg\Gamma \geq \Delta; \Gamma \multimap \Pi \\
\Delta; \Gamma, \Pi \equiv \Delta; \Gamma, \Gamma \multimap \Pi \quad \Delta; \Pi \equiv \Delta; \emptyset \multimap \Pi \\
\Delta; \emptyset \equiv \Delta; B \multimap B \\
\text{REDUCE} \\
\Delta; B_1 \multimap B_2, (B_2, B_3) \multimap \Pi_4 \geq \Delta; (B_1, B_3) \multimap \Pi_4 \\
\text{CARVE-OUT} \\
\frac{\Gamma = k : \tau \prec k' \wedge k \neq k_1 \wedge \dots \wedge k \neq k_n \quad B = \{\{k_1 : \tau_1, \dots, k_n : \tau_n\}\}}{\Delta; k' : \tau' \setminus \{B\}, \Gamma \geq \Delta; k' : \tau \setminus \{k : \tau, B\}, k : \tau} \\
\text{PACK} \\
(\Delta; k : \text{ptr}(\rho), [r \rightarrow \rho]\Pi) \geq (\Delta; k : \exists r. \text{ptr}(r) \text{ with } \Pi) \\
\text{UNPACK} \\
\frac{r' \text{ is fresh}}{(\Delta; k : \exists r. \text{ptr}(r) \text{ with } \Pi) \geq (\{r'\} \cup \Delta; k : \text{ptr}(r'), [r \rightarrow r']\Pi)}
\end{array}$$

The second sample rule shows that keys can be dropped during a transformation. These represent work created for a garbage collector. Alternatively, if this transformation can be used, it shows where deallocation can be done safely.

Rule REDUCE is a generalization of (linear) Modus Ponens ($B_1 = B_3 = \emptyset$).

Rule CARVE-OUT shows how one can carve out a key from another key. It can be replaced using the earlier more general rules involving linear implication. The last rules show how existentials can be introduced and eliminated.

2.5 Soundness

Consistency enables the progress of evaluation and preservation of types for well-typed expressions in environment with no address variables:

LEMMA 2.9. *Given a type-checked program $g (\vdash g : \omega)$, an expression e that type-checks in a variable-free environment $E = (\emptyset; \Pi)$ ($E \vdash_\omega e : \tau \dashv E''$, $E'' = (\Delta''; \Pi'')$), we do not require $\Delta'' = \emptyset$) and a memory and adoption information consistent with E ($\mu; a \vdash E$ consistent), then either e is a value or there exists an evaluation step $(\mu; a; e) \rightarrow_g (\mu'; a'; e')$ and for any such evaluation step, there exists a substitution (with absolute addresses) on some of the new type variables $\sigma : \Delta \rightarrow \emptyset$ where $\Delta \subseteq \Delta''$ such that the resulting expression type-checks in a new environment $E' = (\emptyset; \Pi')$ with the substituted result ($E' \vdash_\omega e' : \sigma\tau \dashv \Delta'' - \Delta; \sigma\Pi''$). In addition, the witness A' for the new consistency will include everything in the witness of the original consistency A ($A_\prec \subseteq A'_\prec, A_T \subseteq A'_T$), and furthermore the new flattened permissions will only include locations from the old permissions or newly allocated memory: $\text{Dom}(\hat{\Pi}') \cap \text{Dom}(\mu) \subseteq \text{Dom}(\hat{\Pi})$.*

More pertinent to this work than progress and preservation is the fact that if one does not have the permission to some piece of state, one cannot access it (read, write or adopt) as shown in the following lemma:

LEMMA 2.10. *Given a type-checked program $g (\vdash g : \omega)$, an expression e that type-checks in a variable-free environment $\emptyset; \Pi \vdash_\omega e : \tau \dashv E''$) and a memory and adoption consistent in an environment with more permissions ($\mu; a \vdash \emptyset; \Pi, \Pi_+$ consistent), then the evaluation of the expression (if any) will not read or write any field mentioned in the flattening of the extra permissions ($\hat{\Pi}_+$).*

We now can prove a separation theorem:

THEOREM 2.11. *Given a type-checked program $g (\vdash g : \omega)$, two expressions e_1, e_2 each of which type-checks in a variable-free environment $\emptyset; \Pi_i \vdash_\omega e_i : \text{unit} \dashv E'_i$) and a memory and adoption consistent with the combined environments: ($\mu; a \vdash \emptyset; \Pi_1, \Pi_2$ consistent), then when evaluating the expressions in sequence $e_1; e_2$ no state will be accessed by both expressions.*

2.6 Future Work

The work described here is partial and ongoing. Further work that needs to be addressed includes:

- Formalizing the \geq relation as a logic, and coming up with a conservative algorithmic approximation.
- Addressing the nondeterminism of the IF rules (especially the rule for $E_1 \vee E_2$).
- Handling “while” loops. In the system here, a “while” loop must be expressed as a call to an explicitly typed recursive procedure. It would be desirable if the required invariant could be inferred instead.

$$\begin{array}{c}
\text{CB-TRUE} \\
A \vdash \text{true} = \text{true} \\
\hline
\text{CB-AXIOM}\prec \\
(l : \tau \prec l') \in A_{\prec} \\
\hline
A \vdash l : \tau \prec l' = \text{true} \\
\text{CB-EQUAL} \\
A \vdash l=l' = (l = l') \\
\text{CB-NEG} \\
\frac{A \vdash \Gamma = b}{A \vdash \neg \Gamma = \neg b} \\
\text{CB-ANDFALSE1} \\
\frac{A \vdash \Gamma_1 = \text{false}}{A \vdash \Gamma_1 \wedge \Gamma_2 = \text{false}} \\
\text{CB-ANDFALSE2} \\
\frac{A \vdash \Gamma_2 = \text{false}}{A \vdash \Gamma_1 \wedge \Gamma_2 = \text{false}} \\
\text{CB-ANDTRUE} \\
\frac{A \vdash \Gamma_1 = \text{true} \quad A \vdash \Gamma_2 = \text{true}}{A \vdash \Gamma_1 \wedge \Gamma_2 = \text{true}} \\
\text{CB-AXIOM}T \\
\frac{t(\nu_1, \dots, \nu_n) \in A_T}{A \vdash t(\nu_1, \dots, \nu_n) = \text{true}}
\end{array}$$

Figure 7: Consistency rules for boolean formulae: $A \vdash \Gamma = b$

$$\begin{array}{c}
\text{CP-EMPTY} \\
\mu; A; \emptyset \vdash \emptyset \Downarrow \{\} \\
\text{CP-TRUE} \\
\frac{A \vdash \Gamma = \text{true}}{\mu; A; \emptyset \vdash \Gamma \Downarrow \{\}} \\
\text{CP-FALSEIMP} \\
\frac{A \vdash \Gamma = \text{false}}{\mu; A; \emptyset \vdash \Gamma \multimap \Pi \Downarrow \{\}} \\
\text{CP-TRUEIMP} \\
\frac{A \vdash \Gamma = \text{true} \quad \mu; A; B \vdash \Pi \Downarrow \hat{\Pi}}{\mu; A; B \vdash \Gamma \multimap \Pi \Downarrow \hat{\Pi}} \\
\text{CP-IMP} \\
\frac{\mu; A; B_1, B_2 \vdash \Pi \Downarrow \hat{\Pi}}{\mu; A; B_1 \vdash B_2 \multimap \Pi \Downarrow \hat{\Pi}} \\
\text{CP-IDENTITY} \\
\mu; A; l : \tau \vdash l : \tau \Downarrow \{\} \\
\text{CP-UNION} \\
\frac{\mu; A; B_1 \vdash \Pi_1 \Downarrow \hat{\Pi}_1 \quad \mu; A; B_2 \vdash \Pi_2 \Downarrow \hat{\Pi}_2}{\mu; A; B_1, B_2 \vdash \Pi_1, \Pi_2 \Downarrow \hat{\Pi}_1 \uplus \hat{\Pi}_2} \\
\text{CP-FIELD} \\
\frac{B_1 = \sum_{(l': \tau' \prec l) \in A_{\prec}} B_{l': \tau'} \quad \mu; A; B_{l': \tau'} \vdash l' : \tau' \Downarrow \hat{\Pi}_{l': \tau'} \quad \hat{\Pi}_1 = \biguplus_{(l': \tau' \prec l) \in A_{\prec}} \hat{\Pi}_{l': \tau'} \quad \mu; A; B_2 \vdash \mu(l) : \tau \Downarrow \hat{\Pi}_2 : \tau_{\text{atom}}}{\mu; A; B_1, B_2 \vdash l : \tau \Downarrow \hat{\Pi}_1 \uplus \hat{\Pi}_2 \uplus \{l : \tau_{\text{atom}}\}}
\end{array}$$

Figure 8: Consistency rules for permissions: $\mu; A; B \vdash \Pi \Downarrow \hat{\Pi}$

$$\begin{array}{c}
\text{CT-UNIT} \\
\mu; A; \emptyset \vdash v : \text{unit} \Downarrow \{\} : \text{unit} \\
\text{CT-BOOL} \\
\mu; A; \emptyset \vdash v : \text{bool} \Downarrow \{\} : \text{bool} \\
\text{CT-INT} \\
\mu; A; \emptyset \vdash v : \text{int} \Downarrow \{\} : \text{int} \\
\text{CT-PTR} \\
\mu; A; \emptyset \vdash v : \text{ptr}(\nu) \Downarrow \{\} : \text{ptr}(\nu) \\
\text{CT-WITH} \\
\frac{\mu; A; B \vdash [r \rightarrow \nu] \Pi \Downarrow \hat{\Pi}}{\mu; A; B \vdash \nu : \exists r. \text{ptr}(r) \text{ with } \Pi \Downarrow \hat{\Pi} : \text{ptr}(\nu)}
\end{array}$$

Figure 9: Consistency rules for types: $\mu; A; B \vdash v \Downarrow \hat{\Pi} : \tau$

$$\frac{a \text{ is acyclic} \quad \sigma : \Delta \rightarrow \emptyset \quad \mu; A; \emptyset \vdash \sigma \Pi \Downarrow \hat{\Pi} \quad (l : \tau_{\text{atom}}) \in \hat{\Pi} \Rightarrow \vdash \mu(l) : \tau_{\text{atom}} \quad \exists \tau. (l : \tau \prec l') \in A_{\prec} \Leftrightarrow (l \prec l') \in a \quad t(\nu_1, \dots, \nu_n) \in A_T \Rightarrow A \vdash [r_1 \rightarrow \nu_1, \dots, r_n \rightarrow \nu_n] T(t) = \text{true}}{\mu; a \vdash \Delta; \Pi \text{ consistent using } \sigma; (A_{\prec}, A_T)}$$

Figure 10: Consistency rules for memory

$$\frac{\forall \mu; a; \sigma; A_{\prec}, A_T (\mu; a \vdash E \text{ consistent using } \sigma; (A_{\prec}, A_T)) \Rightarrow \exists A'_T, \sigma' \supseteq \sigma (\mu; a \vdash E' \text{ consistent using } \sigma'; (A_{\prec}, A'_T))}{E \geq E'}$$

$$\frac{\text{TRANSFORM} \\
E \geq E_1 \vdash_{\omega} e : \tau \dashv E'_1 \geq E'}{E \vdash_{\omega} e : \tau \dashv E'}$$

$\sigma' \supseteq \sigma$ means that $r \in \Delta \Rightarrow \sigma' r = \sigma r$, and $E_1 \equiv E_2$ means $E_1 \geq E_2 \wedge E_2 \geq E_1$.

Figure 11: Transformation rules

3. REALIZING ANNOTATIONS

Instead of directly using the low-level types formalized in this paper, we can use them to express commonly proposed higher-level annotations. The annotations only use a stylized fragment of the type system, and thus lack its full expressive power. As has been done with complex typings [22] used for object-oriented languages, one may be able to provide a simpler type system to check the annotations; the simpler system could then be proven sound against the full type system.

The following annotations can be realized with our extension of adoption:

data groups A class may declare data groups. A field is tagged with its parent data group; a data group may also have a parent. Unlike our earlier work [15] (where data groups were called “regions”), a field may be nested in two data groups. However, at most one data group may actually contain the field at a time.

reference annotations A field, parameter, receiver or return value is tagged with one of the following annotations: *unique*, *shared* or *borrowed*, where the latter is legal only for parameters and receivers. A weak form of ownership can also be handled as a generalization of *shared*.

effects Any method may be annotated with effects. Since this paper does not use fractional permissions, we do not distinguish between read and write effects. The state accessed is expressed using one of the following forms: *this.f* where *f* may be a data group; *p.f* where *p* is a formal parameter; *other* which means anything accessible from $\$0.All$. We are looking at ways to extend the system to represent effects on state such as *any.f* which represents all *f* fields (or data groups) of any object accessible from $\$0.All$.

3.1 Class and Field Annotations

We represent class types by named, potentially recursive, fact-creating functions with one parameter for the address of the object. The facts include one adoption fact for every field and data group. For simplicity, we assume the existence of a single root data group “All” in which all fields are nested (perhaps indirectly). The fact for a field includes its type which, for pointers, is an existential whose body is a macro-call that takes the reference, the fact-creating function for the type and whether the pointer is “good,” usually ($\rho \neq \$0$).⁷ The macro function to use is named by the reference annotation and is expanded as part of the realization process:

$$\begin{aligned} \text{unique}(\rho, t, c) &= \text{ptr}(\rho) \text{ with } c \multimap (\rho.All : \text{unit}, t(\rho)) \\ \text{shared}(\rho, t, c) &= \text{ptr}(\rho) \text{ with } c \multimap \\ &\quad (\rho.All : \text{unit} \prec \$0.All \wedge t(\rho)) \\ \text{borrowed}(\rho, t, c) &= \text{ptr}(\rho) \text{ with } c \multimap t(\rho) \end{aligned}$$

The “borrowed” annotation is legal only for method parameters (and receivers), and indicates that the parameter does not come with permissions on its own; any permissions are

⁷If we were to add non-null annotations (as Fähndrich and Leino [23] suggest), the condition could be strengthened to “true” for non-null pointers.

provided in the method effects. This formulation of borrowed and unique is similar to that in Vault [7].

Consider the **Point** and **Rectangle** classes in our original example (Figure 1). For these simple examples, we get the following “fact-creating” functions:

```
class Point {
  group Loc;
  int x in Loc;
  int y in Loc;
}
Point(r1) =
  r1.Loc : unit < r1.All &
  r1.x : int < r1.Loc &
  r1.y : int < r1.Loc)
```

```
Rectangle(r1) =
  r1.Looks : unit < r1.All &
  r1.Dims : unit < r1.Looks &
  r1.tl :  $\exists r_2.\text{unique}(r_2, \text{Point}, r_2 \neq \$0) \prec r_1.Dims \wedge$ 
  r1.br :  $\exists r_2.\text{unique}(r_2, \text{Point}, r_2 \neq \$0) \prec r_1.Dims \wedge$ 
  r1.n :  $\exists r_2.\text{shared}(r_2, \text{String}, r_2 \neq \$0) \prec r_1.Looks$ 
```

3.2 Parameters and Methods Effects

The parameters and return value are packed into an “argument” object pointed to by a global *ap*. We also give general access to a number of global “temporaries.” Permissions to access all these globals, the parameters and the return value are passed to and returned from the procedure. When a procedure is called, the return value is uninitialized (if a pointer, has a pointer type with an address variable for which we have no information). At the end, the parameters are uninitialized. The procedure type is polymorphic using a variable for the argument object, the receiver, each pointer parameter and the result (if a pointer value). The permissions for each parameter are typed on procedure entry using the same macros for the pointer annotations, except this time the macro-calls are not wrapped in existentials. (The “with”-bound permissions are simply dropped into the surrounding permission bag.) The return value is typed using the annotation macro-call upon exit.

The effects of the method are realized by permissions that are passed to the procedure and then returned. A data group of a receiver or parameter is represented by a data group of the corresponding location variable. The state *other* is represented by $\$0.All$. If some of the effects’ state are known to be shared (directly or indirectly adopted into $\$0.All$), the permission will be carved out in advance.

For a simple example, consider a method of **Rectangle** that changes the name of a rectangle:

```
void setName(shared String n) borrowed
  accesses this.Looks;
```

Its realized type is

$$\forall r_f, r_t, r_n. \left(\begin{array}{l} (\$0.ap : \text{ptr}(r_f), \text{temps}, r_t.Looks : \text{unit}, \\ r_f.\text{this} : \text{borrowed}(r_t, \text{Rectangle}, \text{true}), \\ r_f.n : \text{shared}(r_n, \text{String}, r_n \neq \$0) \rightarrow \\ \exists r'_n. (\$0.ap : \text{ptr}(r_f), \text{temps}, r_f.\text{this} : \text{ptr}(r_t), \\ r_f.n : \text{ptr}(r'_n), r_t.Looks : \text{unit}) \end{array} \right)$$

3.3 Discussion

This realization links parameter annotations and effects more strongly than in our previous work: it does not permit (read-write) borrowed parameters to be aliased; it does not permit shared state to be passed borrowed to a method that affects *other*. Intersection types “solve” this problem at the cost of a number of variants, exponential in the

number of parameters. On the other hand, the stricter definition is probably a good default; a parameter whose state could be accessed through another parameter or through a global variable should be declared as such. The stricter rule corresponds closely to the new ANSI C `restrict` qualifier as formalized by Foster and others [24].

The realization described here does not fully handle inheritance and virtual overriding because there is no way to express downcasts. Neither does it handle the problem of partially constructed objects. We hope to use Fähndrich and Leino’s “monotonic heap states” [25] to describe how the “facts” for an object grow from the facts provided by the superclass to the additional ones provided by the subclass.

4. EXTENSIONS

Recursive permissions allow the definition of aliased structure in a similar way as in “recursive alias types” [26] with the important distinction that since we use conditional permissions rather than union types, the “end points” (null pointers or back-pointers) don’t need to have a different type: a doubly linked list can be appended to in both directions, and a circular list can be “rotated.” These operations require the whole aliased structure to be viewed from a different angle, which requires transformation steps that can be proved using inductive proofs.

For instance, a doubly-linked list between a header node at address h and a tail node at address t can be defined by the permission $DLLF(h, t)$:

$$\begin{aligned} DLLF(h, t) &= \{h.p : \text{ptr}(0), DLLF1(h, t)\} \\ DLLF1(f, l) &= \left\{ \begin{array}{l} f.n : \exists n. \text{ptr}(n) \text{ with} \\ f \neq l \multimap (n.p : \text{ptr}(f)), \\ DLLF1(n, l), f = l \multimap n = \$0 \end{array} \right\} \end{aligned}$$

This type, which makes it easy to access the front of the list (to add or remove nodes) can be converted into the symmetric type (not shown) that views the list from the tail, without any data changes. One simply uses environment transformation. Thus unlike external uniqueness, one can move individual links between two unrelated lists, and unlike recursive alias types, the structure can be added to at either end without changing the rest of the list.

Similarly, a type for circular lists entered at address h , $CIR(h)$ can be defined:

$$\begin{aligned} CIR(h) &= CIR1(h, h) \\ CIR1(h, p) &= \left\{ \begin{array}{l} p.n : \exists n. \text{ptr}(n) \text{ with} \\ n \neq h \multimap CIR1(h, n) \end{array} \right\} \end{aligned}$$

This type can be “rotated” using environment transformation without changing any stored data. These types are very similar to the heap types written using separation logic [27, 20], and thus these positive aspects are not particularly surprising. The novel twist of this work is the addition of “adoption” which enables permissions to be captured in non-linear (“intuitionistic”) facts. For instance, the header node of the circularly linked list could be described by the following (non-linear) type:

$$\begin{aligned} \text{Circle}(c) &= c.h : (\exists h. \text{ptr}(h) \text{ with} \\ &\quad h \neq \$0 \multimap CIR(h)) \prec c.All \end{aligned}$$

This type, being non-linear, is immutable and can be broadcast everywhere, but wherever/whenever the simple (lin-

ear) permission $c.All : \text{unit}$ is available, the circular aliasing structure can be recovered.

Another extension that we are actively pursuing is the idea of fractional permissions [16]. A write permission is viewed as a whole permission that can be split up into (fractional) read permissions. The write permission can be restored after recovering all the various read permissions and reassembling the whole permission. Stated simply, we add the equivalence:

$$\Pi \equiv \frac{1}{2}\Pi, \frac{1}{2}\Pi$$

Read effects are modeled by read permissions (of some unknown fraction) that are sent to a procedure and then returned. While the procedure is active, the state is temporarily immutable. Full immutability is achieved by hiding a fraction inside an existential. Here x is a global variable that points to an object with an immutable integer field:

$$\$0.x : \exists r. \text{ptr}(r) \text{ with } r \neq \$0 \multimap \exists z. zr.val : \text{int}$$

An immutable reference can be split into two by unpacking the existential, splitting the remaining fraction in two and then repacking two separate existentials. With fractions, one can also model read-only (shared) variables, and unique-write variables.

Fractions complicate the type system, which is one reason why they were not included in the system described in this paper. The thorniest issue is that the equivalence given above is not necessarily sound if Π is not “precise” (the two fractions could refer to incomparable views that cannot be “added” together), but the rule for splitting an immutable reference relies on imprecision in the existential. The dilemma can be solved by disallowing the use of an existentially bound fraction variable in “negative” positions in the permission (on the left-hand side of \multimap).

Fractions are however very powerful. For a flavor of what they can do, consider modeling iterators in a collection system such as JDK 2. While an iterator is running over the container, the container should not be mutated, except through the iterator, and then only if there is only one iterator in action. Thus a container can support multiple read-only iterators, or a single read-write iterator. A read-only iterator is created by splitting off a (read-only) fraction of the whole container, from which is obtained a (read-only) reference to the internal structures which is used for the iterator. When one is done with the iterator, the parts are returned, and the container fraction is reassembled and merged back into the rest of the container. A read-write iterator is constructed using the same process except with the entire (writable) container. Mathematically, the process of creating an iterator uses the equivalence

$$zContainer \equiv 1Iterator(z), 1Iterator(z) \multimap zContainer$$

We run the equivalence forward to construct an iterator, and in reverse when we are done with it. If $z = 1$, we are creating a read-write iterator, otherwise a read-only iterator, or more precisely a mutable iterator over read-only state.

5. RELATED WORK

The work is closely related to the work of separation logic [27] and the logic of bunched implications (BI) [19, 20, 21]. Many individual comparisons are scattered through the paper. The biggest technical addition of this work is

adoption. Our consistency check is syntactic which causes restrictions in the forms of existential and conditional permissions, not present in BI. Overall, this work is working toward automatic type-system-like inference of a proof of aliasing properties, whereas Reynolds, O’Hearn and others are emphasizing a logic used for program verification.

This work was conceived as an extension to the “adoption and focus” work of Fähndrich and DeLine [8]. Their type system permits a linear pointer to be irrevocably “adopted” by another pointer. Then the pointer can be duplicated (i.e. copied nonlinearly) with a “guarded type” $g \triangleright \tau$ where g is the adopter. Should some piece of the code need to use the pointer and it has access to the adopter, it can “focus” on the adoptee and gain the linear pointer while temporarily giving up the rights on the adopter. When there is no more need to access the linear pointer (and its linearity and type have been restored), the linear pointer can “disappear” back into the adopter which is then restored. Our system extends/changes “adoption and focus” in the following ways:

1. Unlike adoption and focus, “carving out” permissions for a nested key (“focusing” in their terminology) does not make the nesting key (the “adopter”) inaccessible. It only forbids carving out the *same* nested key again.
2. In our system, adoption (and protection in general) is performed at the level of *fields* of an object rather than whole objects. This allows finer-grained permissions.
3. We prove the type soundness of our system using an operational semantics of adoption.

We also intend to extend our system to distinguish reads from writes.

Adoption and focus uses Alias Types [28], a technique to represent aliasing in the type system. Alias types can be used to precisely describe the shape of recursive data structures [26].

Effects systems have been defined for functional languages in order to safely deallocate “regions” of data that are no longer being accessed [29]. A region encapsulates a (possibly heterogeneous) set of objects. This work was extended by Walker and others [30] in a capability calculus that has one permission key for each region. These insights were later used in the “adoption and focus” work. An alternate approach is Monnier’s “typed regions” [31] which combines “linear” and “intuitionistic” references in a single system. The system uses a “cast” operation that includes a proof of correctness in the underlying logic (CiC [32]). It also uses an “intended type” which appears similar to how the type under adoption is unchanged in our system.

Boyapati and others [33, 34] have incorporated uniqueness, effects and regions in an ownership type system for Java-like languages. They use a permission system over whole objects (not fields as in this work) to prevent data-races. Uniqueness is supported in a few special situations, such as tree-like structures. For the most part, however, the ownership type system prevents object transfer. It appears that transfer could be added to their system using “external uniqueness” [6].

One of the attractive features of external uniqueness is that uniqueness is defined for *external* pointers and does not prevent aliasing internal to the object. We incorporated this idea into our system so that if a structure with internal aliasing is described using recursive types, a unique reference

could point to it. Using our permission system for effects obviates the need for destructive reads.

6. CONCLUSIONS

In this paper, we describe how we can use the ideas of “adoption and focus” to design a type system that takes into account effects and uniqueness in a unified manner. Conditional permissions permit one to express null pointers without tagged unions. Field adoption allows us to express data groups. We show how high-level annotations (method effects and pointer annotations) can be expressed in this system. We prove the type soundness of the system and show how state separation can be guaranteed. It is unlikely that complete algorithmic type inference is possible, but we expect that a practical algorithmic inference system can be defined that will check stylized uses of the type system, such as that used in our realization of annotations.

Acknowledgments

We thank Aaron Greenhouse, Josh Berdine, and Tim Halloran for their useful comments and corrections on this paper. We also thank Dave Clarke, Manuel Fähndrich, and Jan Vitek for fruitful conversations.

7. REFERENCES

- [1] Boyland, J.: The interdependence of effects and uniqueness. Paper from Workshop on Formal Techniques for Java Programs, 2001 (2001)
- [2] Hogg, J.: Islands: Aliasing protection in object-oriented languages. In: OOPSLA’91 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications, New York, ACM Press (1991) 271–285
- [3] Baker, H.G.: ‘Use-once’ variables and linear objects—storage management, reflection and multi-threading. ACM SIGPLAN Notices **30** (1995) 45–52
- [4] Minsky, N.: Towards alias-free pointers. In Cointe, P., ed.: ECOOP’96 — Object-Oriented Programming, 10th European Conference. Volume 1098 of Lecture Notes in Computer Science., Berlin, Heidelberg, New York, Springer (1996) 189–209
- [5] Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA’02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications, New York, ACM Press (2002) 311–330
- [6] Clarke, D., Wrigstad, T.: External uniqueness. In Pierce, B.C., ed.: Informal Proceedings of International Workshop on Foundations of Object-Oriented Languages 2003 (FOOL 10). (2003)
- [7] DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: Proceedings of the ACM SIGPLAN ’01 Conference on Programming Language Design and Implementation, New York, ACM Press (2001) 59–69
- [8] Fähndrich, M., DeLine, R.: Adoption and focus: Practical linear types for imperative programming. In: Proceedings of the ACM SIGPLAN ’02 Conference on Programming Language Design and Implementation, New York, ACM Press (2002) 13–24

- [9] Leino, K.R.M., Stata, R.: Virginty: A contribution to the specification of object-oriented software. *Information Processing Letters* **70** (1999) 99–105
- [10] Leino, K.R.M., Nelson, G.: Data abstraction and information hiding. Research Report 160, Compaq Systems Research Center, Palo Alto, California, USA (2000)
- [11] Leino, K.R.M., Poetzsch-Heffter, A., Zhou, Y.: Using data groups to specify and check side effects. In: *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, New York, ACM Press (2002) 246–257
- [12] Clarke, D.: *Object Ownership and Containment*. PhD thesis, University of New South Wales, Sydney, Australia (2001)
- [13] Müller, P.: *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, Fernuniversität Hagen (2001)
- [14] Boyland, J.: Alias burying: Unique variables without destructive reads. *Software Practice and Experience* **31** (2001) 533–553
- [15] Greenhouse, A., Boyland, J.: An object-oriented effects system. In Guerraoui, R., ed.: *ECOOP'99 — Object-Oriented Programming*, 13th European Conference. Volume 1628 of *Lecture Notes in Computer Science.*, Berlin, Heidelberg, New York, Springer (1999) 205–229
- [16] Boyland, J.: Checking interference with fractional permissions. In Cousot, R., ed.: *Static Analysis: 10th International Symposium*. Volume 2694 of *Lecture Notes in Computer Science.*, Berlin, Heidelberg, New York, Springer (2003) 55–72
- [17] Leino, K.R.M.: Data groups: Specifying the modification of extended state. In: *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, New York, ACM Press (1998) 144–153
- [18] Xi, H.: *Dependent Types in Practical Programming*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA (1998)
- [19] O'Hearn, P., Pym, D.: The logic of bunched implications. *Bulletin of Symbolic Logic* **5** (1999) 215–244
- [20] Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: *Conference Record of the Twenty-eighth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, New York, ACM Press (2001) 14–26
- [21] O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: *Conference Record of POPL 2004: the 31st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, New York, ACM Press (2004) 268–280
- [22] Bruce, K.C., Cardelli, L., , Pierce, B.C.: Comparing object encodings. In: *Theoretical Aspects of Computer Software*. Volume 1281 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, New York (1997) 415–438
- [23] Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: *OOPSLA'03 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, New York, ACM Press (2003) 302–312
- [24] Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, New York, ACM Press (2002) 1–12
- [25] Fähndrich, M., Leino, K.R.M.: Heap monotonic typestates. In: *Informal Proceedings of “International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)”*, Utrecht University, Netherlands (2003)
- [26] Walker, D., Morrisett, G.: Alias types for recursive data structures. In: *Types in Compilation: Third International Workshop, TIC 2000*. Volume 2071 of *Lecture Notes in Computer Science.*, Berlin, Heidelberg, New York, Springer (2001) 177–206
- [27] Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: *Logic in Computer Science*, Los Alamitos, California, IEEE Computer Society (2002) 55–74
- [28] Smith, F., Walker, D., Morrisett, J.G.: Alias types. In Smolka, G., ed.: *ESOP'00 — Programming Languages and Systems*, 9th European Symposium on Programming. Volume 1782 of *Lecture Notes in Computer Science.*, Berlin, Heidelberg, New York, Springer (2000) 366–381
- [29] Tofte, M., Talpin, J.P.: Implementation of the typed call-by-value λ -calculus using a stack of regions. In: *Conference Record of the Twenty-first Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, New York, ACM Press (1994) 188–201
- [30] Walker, D., Crary, K., Morrisett, G.: Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems* **22** (2000) 701–771
- [31] Monnier, S.: Typed regions. In: *Informal Proceedings of “Second workshop on Semantics, Program Analysis, And Computing Environments For Memory Management”*, University of Copenhagen (2004)
- [32] Werner, B.: *Une Théorie des Constructions Inductives*. PhD thesis, L'Université Paris VII (1994)
- [33] Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: *OOPSLA'02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, New York, ACM Press (2002) 211–230
- [34] Boyapati, C., Salcianu, A., Beebe, W., Rinard, M.: Ownership types for safe region-based memory management in real-time java. In: *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, New York, ACM Press (2003) 324–337

APPENDIX

Most of the lemmas and all of the proofs are available in an electronic appendix available from the page <http://www.cs.uwm.edu/faculty/boyland/papers/connecting2.html>.

A. PROOFS

LEMMA 2.1. *Given a type-checked program g ($\vdash g : \omega$), an expression e that type-checks ($\Delta; \Pi_1 \vdash_\omega e : \tau \dashv \Delta'; \Pi'_1$) and an environment $\Delta''; \Pi_2$, then e also type-checks with a larger set of permissions ($\Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash_\omega e : \tau \dashv \Delta' \cup \Delta''; \Pi'_1, \Pi_2$) in which the unused permissions are not changed.*

PROOF. As e must have been typed using one of typing rules, it suffices to show that, for each rule, if that rule applies to e using $\Delta; \Pi_1$ (and returning $\Delta'; \Pi'_1$), it also applies to e using $\Delta \cup \Delta''; \Pi_1, \Pi_2$ (and returning $\Delta' \cup \Delta''; \Pi'_1, \Pi_2$). Additionally, we inductively assume the lemma applies to subexpressions of e . We now look at each rule as a separate case:

Unit, Num, True, False, Address

As these rules neither depend upon nor alter the environment, they type under any environment and the lemma is trivially true.

Plus $e_1 + e_2$

By hypothesis, we know that

$$\Delta; \Pi_1 \vdash_\omega e_1 + e_2 : \text{int} \dashv \Delta'; \Pi'_1$$

. To have typed this using the PLUS rule, the following must be true:

$$\Delta; \Pi_1 \vdash_\omega e_1 : \text{int} \dashv \Delta_1; \Pi''_1 \vdash_\omega e_2 : \text{int} \dashv \Delta'; \Pi'_1$$

Inductively,

$$\Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash_\omega e_1 : \text{int} \dashv \Delta_1 \cup \Delta''; \Pi''_1, \Pi_2$$

$$\Delta_1 \cup \Delta''; \Pi''_1, \Pi_2 \vdash_\omega e_2 : \text{int} \dashv \Delta' \cup \Delta''; \Pi'_1, \Pi_2$$

So, by applying the rule PLUS, we arrive at

$$\Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash_\omega e_1 + e_2 : \text{int} \dashv \Delta' \cup \Delta''; \Pi'_1, \Pi_2$$

Equal $e_1 = e_2$

The types are different than in PLUS, but the environment is treated in the same manner. Thus the proof is essentially the same.

Read $e.f$

By hypothesis:

$$\Delta; \Pi_1 \vdash_\omega e.f : \tau \dashv \Delta'; \Pi'_1$$

As we assume we applied the READ rule, the following must be true:

$$\Delta; \Pi_1 \vdash_\omega e : \text{ptr}(l) \dashv \Delta'; \Pi'_1$$

$$\Pi_1 = l.f : \tau \setminus \{ \dots \}, \Pi$$

$$\tau \sim \tau$$

By induction,

$$\Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash_\omega e : \text{ptr}(l) \dashv \Delta' \cup \Delta''; \Pi'_1, \Pi_2.$$

From the second fact and from the definition of the comma operator, we can conclude

$$\Pi_1, \Pi_2 = l.f : \tau \setminus \{ \dots \}, \Pi, \Pi_2.$$

This gives us sufficient information to apply the READ rule with the larger environment:

$$\Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash_\omega e.f : \tau \dashv \Delta' \cup \Delta''; \Pi'_1, \Pi_2$$

New $\text{new}\{f_i \mid 1 \leq i \leq n\}$

The only requirement for this rule to have been applied is for r to be fresh (given context Δ . We can safely assume that r is still fresh given context $\Delta \cup \Delta''$ because we can type the expression in the original environment with such a r . Thus, we can successfully apply the NEW rule despite the altered environment.

Write $e_1.f := e_2$

The typing prerequisites follow inductively as with PLUS, one may add extra permissions to both sides of an equation here as in READ and the storage compatibility requirements are unaffected by the permissions. Thus we can prove all that is necessary to apply WRITE in the new environment.

Seq $e; e'$

Again, this is essentially the same as PLUS.

If $\text{if } e_0 \text{ then } e_1 \text{ else } e_2$

If this rule were successfully applied, the following must be true:

$$\Delta; \Pi_1 \vdash_\omega e_0 : \text{bool} \dashv \Delta_0; \Pi''_1$$

$$\Delta_0; \Pi''_1 \vdash_\omega e_1 : \text{unit} \dashv \Delta_1; \Pi'_{1a}$$

$$\Delta_0; \Pi''_1 \vdash_\omega e_2 : \text{unit} \dashv \Delta_2; \Pi'_{1b}$$

$$\Delta'; \Pi'_1 = \Delta_1; \Pi'_{1a} \vee \Delta_2; \Pi'_{1b}$$

By induction,

$$\Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash_\omega e_0 : \text{bool} \dashv \Delta_0 \cup \Delta''; \Pi''_1, \Pi_2$$

$$\Delta_0 \cup \Delta''; \Pi''_1, \Pi_2 \vdash_\omega e_1 : \text{unit} \dashv \Delta_1 \cup \Delta''; \Pi'_{1a}, \Pi_2$$

$$\Delta_0 \cup \Delta''; \Pi''_1, \Pi_2 \vdash_\omega e_2 : \text{unit} \dashv \Delta_2 \cup \Delta''; \Pi'_{1b}, \Pi_2$$

Because $\Delta''; \Pi_2$ is an environment, if r appears in Π_2 then $r \in \Delta''$. From the definition of the \vee operator, we can also conclude that $\Delta_i \cup \Delta''; \Pi_{1x} = \sigma_i(\Delta_0 \cup \Delta''); \sigma_i \Pi''_1$, and therefore $\Delta_i \cup \Delta''; \Pi_{1x}, \Pi_2 = \sigma_i(\Delta_0 \cup \Delta''); (\sigma_i \Pi''_1), \Pi_2$. But because every r that appears in Π_2 must also be in $\Delta'' \subseteq (\Delta_0 \cup \Delta'') \cap (\Delta'_2 \cup \Delta'')$, $\sigma_i \Pi_2 = \Pi_2$. Therefore, $\Delta'_i \cup \Delta''; \Pi_{1x}, \Pi_2 = \sigma_i(\Delta_0 \cup \Delta_0); \sigma_i(\Pi''_1, \Pi_2)$. We further assume the fresh variables in Δ' could be chosen so as not to intersect with Δ'' . If not, simply rename to new fresh variables in the original typing proof. Therefore the variables in $\Delta' - ((\Delta_1 \cap \Delta_2) \cup \Delta'')$ are also fresh, and we may apply the definition of \vee to determine that

$$\Delta' \cup \Delta''; \Pi'_1, \Pi_2 =$$

$$\Delta_1 \cup \Delta''; \Pi'_{1a}, \Pi_2 \vee \Delta_2 \cup \Delta''; \Pi'_{1b}, \Pi_2$$

This, combined with the earlier inductive results is sufficient to apply the IF rule and type check the expression in the new environment.

IfEqual $\text{if } e = e' \text{ then } e_1 \text{ else } e_2$

By hypothesis:

$$\Delta; \Pi_1 \vdash_\omega \text{if } e = e' \text{ then } e_1 \text{ else } e_2 : \text{unit} \dashv \Delta'; \Pi'_1$$

As we are assuming the IFEQUAL rule was applied, we know the following must have been true:

$$\begin{aligned} &\Delta; \Pi_1 \vdash_{\omega} e : \text{ptr}(l) \dashv \Delta_{0_1}; \Pi_{1_1} \\ &\Delta_{0_1}; \Pi_{1_1} \vdash_{\omega} e' : \text{ptr}(l') \dashv \Delta_{0_2}; \Pi_{1_2} \\ &\Delta_{0_2}; (\Pi_{1_2}, l = l') \vdash_{\omega} e_1 : \text{unit} \dashv \Delta_1; \Pi_{1_a} \\ &\Delta_{0_2}; (\Pi_{1_2}; l \neq l') \vdash_{\omega} e_2 : \text{unit} \dashv \Delta_2; \Pi_{1_b} \\ &\Delta'; \Pi_1 = \Delta_1; \Pi_{1_a} \vee \Delta_2; \Pi_{1_b} \end{aligned}$$

By induction:

$$\begin{aligned} &\Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash_{\omega} e : \text{ptr}(l) \dashv \Delta_{0_1} \cup \Delta''; \Pi_{1_1}, \Pi_2 \\ &\Delta_{0_1} \cup \Delta''; \Pi_{1_1}, \Pi_2 \vdash_{\omega} e' : \text{ptr}(l') \dashv \Delta_{0_2} \cup \Delta''; \Pi_{1_2}, \Pi_2 \\ &\Delta_{0_2} \cup \Delta''; (\Pi_{1_2}, l = l'), \Pi_2 \vdash_{\omega} e_1 : \text{unit} \dashv \\ &\quad \Delta_1 \cup \Delta''; \Pi_{1_a}, \Pi_2 \\ &\Delta_{0_2} \cup \Delta''; (\Pi_{1_2}; l \neq l'), \Pi_2 \vdash_{\omega} e_2 : \text{unit} \dashv \\ &\quad \Delta_2 \cup \Delta''; \Pi_{1_b}, \Pi_2 \end{aligned}$$

Further, we can reiterate the argument from the IF case to conclude that

$$\begin{aligned} &\Delta' \cup \Delta''; \Pi'_1, \Pi_2 = \\ &\quad \Delta_1 \cup \Delta''; \Pi'_{1_a}, \Pi_2 \vee \Delta_2 \cup \Delta''; \Pi'_{1_b}, \Pi_2 \end{aligned}$$

Therefore, we can apply the IFEQUAL rule to type the expression in the new environment.

IfTrue if true then e_1 else e_2

Trivially true by induction on e_1 .

IfFalse if false then e_1 else e_2

Trivially true by induction on e_2 .

Call call p

If we assume the CALL rule had been applied to type our expression, then its prerequisites must be true. As before, we may assume that Δ' is fresh with regard to $\Delta \cup \Delta''$ as well as Δ . Thus the only prerequisite for this rule that requires the environment (directly) is

$$\sigma_1 : \Delta_1 \rightarrow \Delta$$

However, it is clear from the definition of substitutions that one may increase the range of σ_1 without altering its definition. In particular, one may define

$$\sigma'_1 : \Delta_1 \rightarrow \Delta \cup \Delta''$$

where σ'_1 and σ_1 are identical maps. By including the Π_2 from the lemma in the Π_3 from the CALL rule, we may directly apply the CALL rule in the environment $\Delta \cup \Delta''; \Pi_1, \Pi_2$ and get back the environment $\Delta \cup \Delta' \cup \Delta''; \Pi'_1, \Pi_2$ (where Π'_1 in the lemma equals $\sigma_1^{(\cdot)} \Pi_2, \Pi_3$ in the CALL rule).

Nest nest $e.f$ in $e'.f'$

By hypothesis:

$$\Delta; \Pi_1 \vdash_{\omega} \text{nest } e.f \text{ in } e'.f' : \text{unit} \dashv \Delta'; (k' :$$

$$\tau' \setminus \{k_1 : \tau_1, \dots, k_n : \tau_n\}; k : \tau \prec k' \wedge k \neq k_n), \Pi''_1$$

Therefore, the following must have been true to apply the NEST rule:

$$\Delta; \Pi_1 \vdash_{\omega} e : \text{ptr}(l) \dashv \Delta_0; \Pi_0$$

$$\Delta_0; \Pi_0 \vdash_{\omega} e' : \text{ptr}(l') \dashv \Delta'; \Pi'_1$$

$$k = l.f$$

$$k' = l'.f'$$

$$\Pi'_1 = (k : \tau \setminus \{k_1 : \tau_1, \dots, k_n : \tau_n\}; k \neq k_1 \wedge \dots \wedge k \neq k_n), k' :$$

$$\tau' \setminus \{k_1 : \tau_1, \dots, k_n : \tau_n\}, \Pi''_1$$

By induction,

$$\Delta \cup \Delta''; \Pi_1, \Pi_2 \vdash_{\omega} e : \text{ptr}(l) \dashv \Delta_0 \cup \Delta''; \Pi_0, \Pi_2$$

$$\Delta_0 \cup \Delta''; \Pi_0, \Pi_2 \vdash_{\omega} e' : \text{ptr}(l') \dashv \Delta' \cup \Delta''; \Pi'_1, \Pi_2$$

And, from the definition of the comma operator, we can deduce that,

$$\Pi'_1, \Pi_2 = (k : \tau \setminus \{k_1 : \tau_1, \dots, k_n : \tau_n\}; k \neq k_1 \wedge \dots \wedge k \neq k_n), k' :$$

$$\tau' \setminus \{k_1 : \tau_1, \dots, k_n : \tau_n\}, \Pi''_1, \Pi_2$$

This is sufficient information to apply the NEST rule and type the expression in the new environment.

□

We also have a substitution lemma:

LEMMA 2.2. *Given a program g as an expression e that type-checks in an environment $E = (\Delta; \Pi)$ ($E \vdash_{\omega} e : \tau \dashv E'$), and given a substitution $\sigma : \Delta_1 \rightarrow \Delta_2$ where $\Delta_1 \uplus \Delta_2$ is a partition of Δ , then e also type checks in the substituted environment $\sigma E = (\Delta_2; \sigma \Pi)$ ($\sigma E \vdash_{\omega} e : \sigma \tau \dashv \sigma E'$).*

PROOF. It is worth mentioning that $\sigma \text{unit} = \text{unit}$, $\sigma \text{int} = \text{int}$, and $\sigma \text{bool} = \text{bool}$ for any substitution σ , as substitutions apply only to location variables. Further, if τ is a pointer type, then $\sigma \tau$ is also, because the range of σ is confined to location variables and literals. Thus, if $\tau_1 \sim \tau_2$, then $\sigma \tau_1 \sim \sigma \tau_2$ for any substitution σ .

Unit, Num, True, False, Address

As these rules neither depend upon nor alter the environment, they type under any environment and the lemma is trivially true.

Plus $e_1 + e_2$

Follows immediately from applying induction twice.

Equal $e_1 = e_2$

By hypothesis, this rule was applied, so

$$E \vdash_{\omega} e_1 : \tau_1 \dashv E' \vdash_{\omega} e_2 : \tau_2 \dashv E''$$

$$\tau_1 \sim \tau_2$$

Then, by induction:

$$\sigma E \vdash_{\omega} e_1 : \sigma\tau_1 \dashv \sigma E' \vdash_{\omega} e_2 : \sigma\tau_2 \dashv \sigma E''$$

As discussed above, $\sigma\tau_1 \sim \sigma\tau_2$. Therefore, we can apply the EQUAL rule to get

$$\sigma E \vdash_{\omega} e_1=e_2 : \sigma\text{bool} \dashv \sigma E''$$

Read $e.f$

We may assume the prerequisites for the READ rule are true. Then, by induction,

$$\sigma E \vdash_{\omega} e : \sigma\text{ptr}(\rho) \dashv \sigma\Delta'; \sigma\Pi'$$

Directly applying the substitution to both sides of the equation:

$$\sigma\Pi' = \sigma\rho.f : \sigma\tau \setminus \{\sigma\dots\}, \sigma\Pi_1$$

And as storage compatibility holds over substitutions,

$$\sigma\tau \sim \sigma\tau$$

Therefore we may apply the READ rule to get the desired result.

New $\text{new}\{f_i \mid 1 \leq i \leq n\}$

As the new reference variable introduced by this rule is fresh, the rule may be applied in the substituted environment. Since the fresh variable is in neither the domain nor the range of the substitution (nor are its concrete field types), the appropriate substituted environment results.

Write $e_1.f := e_2$

The typing preconditions for this rule in the substituted environment follow by double induction, as with PLUS. The equality and storage compatibility preconditions follow as in the READ rule.

Seq $e; e'$

Follows immediately from applying induction twice.

If $\text{if } e_0 \text{ then } e_1 \text{ else } e_2$

By hypothesis,

$$E \vdash_{\omega} \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : \text{unit} \dashv E''$$

The following must have been true to apply this rule:

$$E \vdash_{\omega} e_0 : \text{bool} \dashv E'$$

$$E' \vdash_{\omega} e_1 : \text{unit} \dashv E_1$$

$$E' \vdash_{\omega} e_2 : \text{unit} \dashv E_2$$

$$E'' = E_1 \vee E_2$$

By induction,

$$\sigma E \vdash_{\omega} e_0 : \sigma\text{bool} \dashv \sigma E'$$

$$\sigma E' \vdash_{\omega} e_1 : \sigma\text{unit} \dashv \sigma E_1$$

$$\sigma E' \vdash_{\omega} e_2 : \sigma\text{unit} \dashv \sigma E_2$$

We can apply some of these substitutions:

$$\sigma E \vdash_{\omega} e_0 : \text{bool} \dashv \sigma E'$$

$$\sigma E' \vdash_{\omega} e_1 : \text{unit} \dashv \sigma E_1$$

$$\sigma E' \vdash_{\omega} e_2 : \text{unit} \dashv \sigma E_2$$

If we can show that $\sigma E'' = \sigma E_1 \vee \sigma E_2$, we can then apply the IF rule to prove that

$$\sigma E \vdash_{\omega} \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : \sigma\text{unit} \dashv \sigma E''$$

We therefore argue that if $E'' = E_1 \vee E_2$, $\sigma E'' = \sigma E_1 \vee \sigma E_2$.

If $(\Delta'; \Pi') = (\Delta_1; \Pi_1) \vee (\Delta_2; \Pi_2)$ then by definition, there are substitutions σ_1 and σ_2 such that

$$\Delta_1; \Pi_1 = \sigma_1\Delta'; \sigma_1\Pi'$$

$$\Delta_2; \Pi_2 = \sigma_2\Delta'; \sigma_2\Pi'$$

$$\Delta = \Delta_1 \cap \Delta_2$$

$$\Delta' - \Delta \text{ fresh}$$

$$r \in \Delta \Rightarrow \sigma_i r = r$$

We assume that the domain of σ_1 and σ_2 is restricted to Δ' . If not, we can choose new σ_i which are so restricted, and which are sufficient for the definition of \vee . As such, the only variables in the domains of both σ and σ_i are those in Δ , which σ_1 and σ_2 do not alter.

As $\Delta' - \Delta$ fresh, and as σ 's domain was defined on variables existing before the \vee operation, σ can only affect the variables in Δ . It acts as the identity on the fresh variables in Δ' . Similarly, the range of σ is also confined to Δ . Thus $\sigma\Delta' = (\Delta' - \Delta) \cup \sigma\Delta$ and $\sigma\Delta' - \sigma\Delta = \Delta' - \Delta$ fresh.

As the domain and range of σ are disjoint by definition, σr will be outside the domain of σ for any r . Thus $\sigma = \sigma \circ \sigma$. Additionally, because σ_1 and σ_2 behave as the identity for location variables in Δ and because the range of σ cannot include the fresh variables which make up the (non-identity) domain of σ_1 and σ_2 , $\sigma \circ \sigma_i \circ \sigma = \sigma \circ \sigma \circ \sigma_i = \sigma \circ \sigma_i$ ($i = 1$ or $i = 2$).

We can now define two new substitutions, $\sigma'_1 = \sigma \circ \sigma_1$ and $\sigma'_2 = \sigma \circ \sigma_2$. Clearly, $\sigma\Delta_1; \sigma\Pi_1 = \sigma\sigma_1\Delta'; \sigma\sigma_1\Pi' = \sigma\sigma_1\sigma\Delta'; \sigma\sigma_1\sigma\Pi' = \sigma'_1\sigma\Delta'; \sigma'_1\sigma\Pi'$. Similarly, $\sigma\Delta_2; \sigma\Pi_2 = \sigma'_2\sigma\Delta'; \sigma'_2\sigma\Pi'$.

From $\Delta = \Delta_1 \cap \Delta_2$, we can conclude that $\sigma\Delta = \sigma(\Delta_1 \cap \Delta_2) = \sigma\Delta_1 \cap \sigma\Delta_2$. Then, for all $r \in \sigma\Delta$, there exists a $r' \in \Delta$ where $\sigma r' = r$. Also, $\forall r \in \sigma\Delta$

$$\begin{aligned} \sigma'_1 r &= \sigma\sigma_1 r \\ &= \sigma\sigma_1\sigma r' \quad (\text{definition of } r') \\ &= \sigma\sigma_1 r' \quad (\text{from above}) \\ &= \sigma r' \quad (\sigma_1 r' = r' \quad \forall r' \in \Delta) \\ &= r \end{aligned}$$

An identical argument may be applied to get $\sigma'_2 r = r \quad \forall r \in \sigma\Delta$. These facts can now be used to apply the definition of \vee to determine that if $(\Delta'; \Pi') = (\Delta_1; \Pi_1) \vee (\Delta_2; \Pi_2)$, then $(\sigma\Delta'; \sigma\Pi') = (\sigma\Delta_1; \sigma\Pi_1) \vee (\sigma\Delta_2; \sigma\Pi_2)$.

In particular, for the IF rule, $\sigma E'' = \sigma E_1 \vee \sigma E_2$. This, in turn, is sufficient (combined with the earlier inductive results) to apply the IF rule and conclude that

$$\sigma E \vdash_{\omega} \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : \sigma \text{unit} \dashv \sigma E''$$

IFEQUAL if $e=e'$ then e_1 else e_2

By hypothesis,

$$E \vdash_{\omega} \text{if } e=e' \text{ then } e_1 \text{ else } e_2 : \text{unit} \dashv E''$$

As we are assuming the IFEQUAL rule was applied, the following must have been true:

$$E \vdash_{\omega} e : \text{ptr}(\rho) \dashv E' \vdash_{\omega} e' : \text{ptr}(\rho') \dashv \Delta; \Pi$$

$$\Delta; (\rho = \rho', \Pi) \vdash_{\omega} e_1 : \text{unit} \dashv E_1$$

$$\Delta; (\rho \neq \rho', \Pi) \vdash_{\omega} e_2 : \text{unit} \dashv E_2$$

$$E'' = E_1 \vee E_2$$

After applying induction (and recalling that $\sigma \text{unit} = \text{unit}$ and $\sigma \text{bool} = \text{bool}$)

$$\sigma E \vdash_{\omega} e : \sigma \text{ptr}(\rho) \dashv \sigma E' \vdash_{\omega} e' : \sigma \text{ptr}(\rho') \dashv \sigma \Delta; \sigma \Pi$$

$$\sigma \Delta; (\sigma \rho = \sigma \rho', \sigma \Pi) \vdash_{\omega} e_1 : \sigma \text{unit} \dashv \sigma E_1$$

$$\sigma \Delta; (\sigma \rho \neq \sigma \rho', \sigma \Pi) \vdash_{\omega} e_2 : \sigma \text{unit} \dashv \sigma E_2$$

As $E'' = E_1 \vee E_2$, we can argue again that $\sigma E'' = \sigma E_1 \vee \sigma E_2$. Therefore, we can apply the IFEQUAL rule again to get

$$\sigma E \vdash_{\omega} \text{if } e=e' \text{ then } e_1 \text{ else } e_2 : \sigma \text{unit} \dashv \sigma E''$$

which is what needs to be proven.

IFTRUE if true then e_1 else e_2

Follows immediately from induction on e_1 .

IFFALSE if false then e_1 else e_2

Follows immediately from induction on e_2 .

CALL call p

By hypothesis,

$$\Delta; \sigma_1 \Pi_1, \Pi_3 \vdash_{\omega} \text{call } p : \text{unit} \dashv \Delta \cup \Delta'; \sigma_1 \Pi_2, \Pi_3$$

Therefore, because we checked this using the CALL rule:

$$\omega(p) = \forall \Delta_1. \Pi_1 \rightarrow \exists \Delta_2. \sigma_2 \Pi_2$$

$$\sigma_1 : \Delta_1 \rightarrow \Delta$$

$$\Delta' \text{ fresh}$$

$$\sigma_2 : \Delta' \rightarrow \Delta_2$$

We can safely assume the variables in Δ , Δ_2 , Δ_1 , and Δ' are all disjoint as we may always find a typing using disjoint contexts. Therefore, $\sigma \sigma_1 : \Delta_1 \rightarrow \sigma \Delta$. Also, $\sigma \Delta' = \Delta'$, so $\sigma \Delta$ fresh and $\sigma_2 : \sigma \Delta' \rightarrow \Delta_2$. We thus

have:

$$\omega(p) = \forall \Delta_1. \Pi_1 \rightarrow \exists \Delta_2. \sigma_2 \Pi_2$$

$$\sigma \sigma_1 : \Delta_1 \rightarrow \sigma \Delta$$

$$\sigma \Delta' \text{ fresh}$$

$$\sigma_2 : \sigma \Delta' \rightarrow \Delta_2$$

We can now apply the CALL rule to get:

$$\sigma \Delta; \sigma \sigma_1 \Pi_1, \sigma \Pi_3 \vdash_{\omega} \text{call } p : \text{unit} \dashv$$

$$\sigma \Delta \cup \sigma \Delta'; \sigma \sigma_1 \Pi_2, \sigma \Pi_3$$

Nest nest $e.f$ in $e'.f'$

All the equations still hold true if the substitution is applied equally to both sides, so this rule naturally falls out by inducting twice, then substituting over the equations, similar to WRITE. \square

Next, we prove a narrowing rule for consistency:

LEMMA 2.3. *If we have a memory and adoption information consistent with an environment ($\mu; a \vdash \Delta; \Pi_1, \Pi_2$ consistent), then they are also consistent with an environment with fewer permissions ($\mu; a \vdash \Delta; \Pi_1$).*

PROOF. We first prove that $\mu; A; \emptyset \vdash \pi_1, \dots, \pi_n \Downarrow \hat{\Pi}$ if and only if $\mu; A; \emptyset \vdash \pi_i \Downarrow \hat{\Pi}_i$ and $\hat{\Pi} = \biguplus_i \hat{\Pi}_i$. The result is susceptible to a simple proof by induction using $n \in \{0, 1\}$ as base cases since there is only one rule for show consistency of a set of permissions, and the B sets are empty, the non-determinism of the split of permissions falls away due to the associativity of the union of disjoint sets.

Given this result, it is easy to see that the consistency of the smaller set of permissions can be established using the same substitution and assumption sets; we simply end up with a subset of the requirements on the memory that must be fulfilled. Thus the result follows. \square

An important property of this definition is that the transformed environment will be flattened to a subset of the flat permissions:

LEMMA 2.4. *If we have $\mu; a \vdash \Delta; \Pi$ consistent using $\sigma; (A_{\prec}, A_T)$ such that $\mu; A; \emptyset \vdash \sigma \Pi \Downarrow \hat{\Pi}$ and $(\Delta; \Pi) \geq (\Delta'; \Pi')$ then $\mu; A'; \emptyset \vdash \sigma' \Pi' \Downarrow \hat{\Pi}'$ and $\hat{\Pi} \supseteq \hat{\Pi}'$.*

PROOF. From the definition of the transformation relation, $\mu; a \vdash \Delta'; \Pi'$ consistent using $\sigma'; (A_{\prec}, A'_T)$. Therefore, by the definition of consistency, there must be some $\hat{\Pi}'$ such that $\mu; A; \emptyset \vdash \sigma' \Pi' \Downarrow \hat{\Pi}'$. Now suppose $l : \tau_{\text{atom}} \in \hat{\Pi}'$. Then, $\vdash \mu(l) : \tau_{\text{atom}}$.

Also, $\exists \tau'_{\text{atom}} \ l : \tau'_{\text{atom}} \in \hat{\Pi}$. If not, we can define $\mu' = \mu[l \mapsto v]$ where $\vdash v : \tau''_{\text{atom}}$ and $\tau''_{\text{atom}} \neq \tau_{\text{atom}}$. Then $\mu'; a \vdash \Delta; \Pi$ consistent using $\sigma; (A_{\prec}, A_T)$ because all the preconditions established by μ still hold. But $\mu'; a \vdash \Delta'; \Pi'$ consistent using $\sigma'; (A_{\prec}, A_T)$ is clearly not true, as we will not be able to show that $\vdash \mu'(l) : \tau_{\text{atom}}$ (because, of course, $\vdash \mu'(l) : \tau''_{\text{atom}}$).

But from $\mu; a \vdash \Delta; \Pi$ consistent using $\sigma; (A_{\prec}, A_T)$ such that $\mu; A; \emptyset \vdash \sigma \Pi \Downarrow \hat{\Pi}$, we know that $\vdash \mu(l) : \tau'_{\text{atom}}$. But then τ_{atom} must equal τ'_{atom} , so $l : \tau_{\text{atom}} \in \hat{\Pi}$. \square

LEMMA 2.5. *If we have two transformations: $\Delta; \Pi_1 \geq \Delta'_1; \Pi'_1$ and $\Delta; \Pi_2 \geq \Delta'_2; \Pi'_2$ where the fresh variables introduced are disjoint $(\Delta'_1 - \Delta) \cap (\Delta'_2 - \Delta) = \emptyset$, then the two transformations can be merged: $\Delta; \Pi_1, \Pi_2 \geq \Delta'; \Pi'_1, \Pi'_2$ where $\Delta' = \Delta'_1 \cup \Delta'_2$.*

PROOF. Suppose we have some $\mu, a, \sigma, A_{\prec}, A_T$ such that $\mu; a \vdash \Delta; \Pi_1, \Pi_2$ consistent using $\sigma; (A_{\prec}, A_T)$. Then, from Lemma 2.3 and the definition of transform (\geq), there are A'_T and σ_0 such that $\mu; a \vdash \Delta; \Pi_1$ consistent using $\sigma_0; (A_{\prec}, A'_T)$. Thus, using the definition of transform again, there exists A_{T1} and σ_1 such that $\mu; a \vdash \Delta'_1; \Pi'_1$ consistent using $\sigma_1; (A_{\prec}, A_{T1})$. Similarly, there exists A_{T2} and σ_2 such that $\mu; a \vdash \Delta'_2; \Pi'_2$ consistent using $\sigma_2; (A_{\prec}, A_{T2})$.

From the definition of memory consistency we may conclude that the following facts must be true:

a is acyclic

$\sigma_1 : \Delta'_1 \rightarrow \emptyset$

$\sigma_2 : \Delta'_2 \rightarrow \emptyset$

$\exists \tau. (l : \tau \prec l') \in A_{\prec} \Leftrightarrow (l \prec l') \in a$

$\mu; A_{\prec}, A_{T1}; \emptyset \vdash \sigma_1 \Pi'_1 \Downarrow \hat{\Pi}_1$

$\mu; A_{\prec}, A_{T2}; \emptyset \vdash \sigma_2 \Pi'_2 \Downarrow \hat{\Pi}_2$

$(l : \tau_{\text{atom}}) \in \hat{\Pi}_1 \Rightarrow \vdash \mu(l) : \tau_{\text{atom}}$

$(l : \tau_{\text{atom}}) \in \hat{\Pi}_2 \Rightarrow \vdash \mu(l) : \tau_{\text{atom}}$

$t(\nu_1, \dots, \nu_n) \in A_{T1} \Rightarrow A_{\prec}, A_{T1} \vdash [r_1 \rightarrow \nu_1, \dots, r_n \rightarrow \nu_n]T(t) = \text{true}$

$t(\nu_1, \dots, \nu_n) \in A_{T2} \Rightarrow A_{\prec}, A_{T2} \vdash [r_1 \rightarrow \nu_1, \dots, r_n \rightarrow \nu_n]T(t) = \text{true}$

Let us define $\sigma' : \Delta' \rightarrow \emptyset$ as follows:

$$\sigma'(r) = \begin{cases} \sigma(r) & \text{if } r \in \Delta \cap \Delta' \\ \sigma_1(r) & \text{if } r \in \Delta'_1 - \Delta \\ \sigma_2(r) & \text{if } r \in \Delta'_2 - \Delta \end{cases}$$

This substitution is well-defined, as $\Delta'_1 - \Delta$ and $\Delta'_2 - \Delta$ are disjoint. Because $\sigma_1 \supseteq \sigma_0 \supseteq \sigma$,⁶ $\forall r \in \Delta'_1 \ \sigma'(r) = \sigma_1(r)$ and thus $\sigma'(\Pi'_1) = \sigma_1(\Pi'_1)$. Similarly, $\forall r \in \Delta'_2 \ \sigma'(r) = \sigma_2(r)$ and thus $\sigma'(\Pi'_2) = \sigma_2(\Pi'_2)$. Also, from the definition of σ' , $r \in \Delta \Rightarrow \sigma' r = \sigma r$, so $\sigma' \supseteq \sigma$.

Suppose $A_{\prec}, A_T \vdash \Gamma = b$. Then $A_{\prec}, A_T \cup A'_T \vdash \Gamma = b$. Why? Because the only effect adding assumptions to A_T can have on the consistency of Γ is with the rule CB-AXIOM T which would serve to make something which had been undefined true. But no truth values were undefined in the original proof (or it wouldn't have been a proof). Therefore, the same proof may be reiterated with extra unused assumptions. This in turn implies that if $\mu; A_{\prec}, A_T; \emptyset \vdash \sigma \Pi \Downarrow \hat{\Pi}$, then $\mu; A_{\prec}, A_T \cup A'_T; \emptyset \vdash \sigma \Pi \Downarrow \hat{\Pi}$ because the proof trees for the boolean formulae will not change as above, and the remainder of the permission consistency rules do not refer to A_T and so will not be affected by its contents. Again, the

⁶That the \supseteq relation on substitutions is transitive follows immediately from its definition

same proof may be reused within the larger set of assumptions.

We may therefore modify the above facts:

$\mu; A_{\prec}, A_{T1} \cup A_{T2}; \emptyset \vdash \sigma \Pi' \Downarrow \hat{\Pi}_1$

$\mu; A_{\prec}, A_{T1} \cup A_{T2}; \emptyset \vdash \sigma \Pi'_2 \Downarrow \hat{\Pi}_2$

$(l : \tau_{\text{atom}}) \in \hat{\Pi}_1 \Rightarrow \vdash \mu(l) : \tau_{\text{atom}}$

$(l : \tau_{\text{atom}}) \in \hat{\Pi}_2 \Rightarrow \vdash \mu(l) : \tau_{\text{atom}}$

$t(\nu_1, \dots, \nu_n) \in A_{T1} \Rightarrow A_{\prec}, A_{T1} \cup A_{T2} \vdash [r_1 \rightarrow \nu_1, \dots, r_n \rightarrow \nu_n]T(t) = \text{true}$

$t(\nu_1, \dots, \nu_n) \in A_{T2} \Rightarrow A_{\prec}, A_{T1} \cup A_{T2} \vdash [r_1 \rightarrow \nu_1, \dots, r_n \rightarrow \nu_n]T(t) = \text{true}$

the first two may be combined using the CP-UNION rule to get:

$\mu; A_{\prec}, A_{T1} \cup A_{T2}; \emptyset \vdash \sigma_1 \Pi'_1, \sigma_2 \Pi'_2 \Downarrow \hat{\Pi}_1, \hat{\Pi}_2$

As discussed above, this is equivalent to

$\mu; A_{\prec}, A_{T1} \cup A_{T2}; \emptyset \vdash \sigma'(\Pi'_1, \Pi'_2) \Downarrow \hat{\Pi}_1, \hat{\Pi}_2$

We can further use straightforward logic to combine pairs of implications:

$(l : \tau_{\text{atom}}) \in \hat{\Pi}_1, \hat{\Pi}_2 \Rightarrow \vdash \mu(l) : \tau_{\text{atom}}$

$t(\nu_1, \dots, \nu_n) \in A_{T1} \cup A_{T2} \Rightarrow A_{\prec}, A_{T1} \cup A_{T2} \vdash [r_1 \rightarrow \nu_1, \dots, r_n \rightarrow \nu_n]T(t) = \text{true}$

$t(\nu_1, \dots, \nu_n) \in A_{T2} \Rightarrow A_{\prec}, A_{T1} \cup A_{T2} \vdash [r_1 \rightarrow \nu_1, \dots, r_n \rightarrow \nu_n]T(t) = \text{true}$

We now have enough facts to directly apply the definition of memory consistency and get:

$\mu; a \vdash \Delta'; \Pi'_1, \Pi'_2$ consistent using $\sigma; (A_{\prec}, A_{T1} \cup A_{T2})$

Thus, for any $\mu, a, \sigma, A_{\prec}, A_T$ such that $\mu; a \vdash \Delta; \Pi_1, \Pi_2$ consistent using we can produce an $A_{T1} \cup A_{T2}$ such that $\mu; a \vdash \Delta'; \Pi'_1, \Pi'_2$ consistent using $\sigma; (A_{\prec}, A_{T1} \cup A_{T2})$. Therefore, by definition, $\Delta; \Pi_1, \Pi_2 \geq \Delta'; \Pi'_1, \Pi'_2$. \square

After adding TRANSFORM, we must re-prove Lemmas 2.1 and 2.2.

LEMMA 2.6 (2.1). *Given a transformation $(\Delta; \Pi \geq \Delta'; \Pi')$ and an environment $(\Delta_1; \Pi_1)$, the corresponding larger sets also form a transformation $(\Delta \cup \Delta_1; \Pi, \Pi_1 \geq \Delta' \cup \Delta_1; \Pi', \Pi_1)$. Therefore, given a type-checked program $g (\vdash g : \omega)$, an expression e that type-checks $(\Delta; \Pi_1 \geq \Delta''; \Pi''_1 \vdash_\omega e : \tau \dashv \Delta'''; \Pi'''_1 \geq \Delta'; \Pi'_1)$ under TRANSFORM, and an environment $\Delta_2; \Pi_2$, then e also type-checks with a larger set of permissions $(\Delta \cup \Delta_2; \Pi_1, \Pi_2 \geq \Delta'' \cup \Delta_2; \Pi''_1, \Pi_2 \vdash_\omega e : \tau \dashv \Delta'''' \cup \Delta_2; \Pi''_1, \Pi'_2 \geq \Delta' \cup \Delta_2; \Pi'_1, \Pi_2)$ in which the unused permissions are not changed. That is, Lemma 2.1 holds even with the TRANSFORM rule added.*

PROOF. It should be clear that adding extra, unused variables will not affect consistency. Thus we may conclude that $(\Delta \cup \Delta_1; \Pi \geq \Delta' \cup \Delta_1; \Pi')$. Also, trivially $\Delta_1; \Pi_1 \geq \Delta_1; \Pi_1$ as whenever $\Delta_1; \Pi_1$ is consistent, $\Delta_1; \Pi_1$ is consistent. This second transformation introduces no fresh variables. Therefore, we may apply Lemma 2.5 to get $(\Delta \cup \Delta_1; \Pi, \Pi_1 \geq \Delta' \cup \Delta_1; \Pi', \Pi_1)$.

This fact may immediately be used with Lemma 2.1 to prove the second statement in this lemma. \square

LEMMA 2.7 (2.2). *Given a transformation $(\Delta; \Pi \geq \Delta'; \Pi')$ and a substitution $(\sigma : \Delta \cup \Delta' \rightarrow \Delta'')$, the substituted environments also form a transformation $(\sigma\Delta; \sigma\Pi \geq \sigma\Delta'; \sigma\Pi')$. Therefore, given a program g as an expression e that type-checks under TRANSFORM in an environment $E = (\Delta; \Pi)$ ($E \geq E_1 \vdash_\omega e : \tau \dashv E'_1 \geq E'$), and given a substitution $\sigma : \Delta_1 \rightarrow \Delta_2$ where $\Delta_1 \uplus \Delta_2$ is a partition of Δ , then e also type checks in the substituted environment $\sigma E = (\Delta_2; \sigma\Pi)$ ($\sigma E \geq \sigma E_1 \vdash_\omega e : \sigma\tau \dashv \sigma E'_1 \geq \sigma E'$).*

PROOF. Let $E = \Delta; \Pi$ and $E' = \Delta'; \Pi'$. By definition, $\sigma\Delta; \sigma\Pi \geq \sigma\Delta'; \sigma\Pi'$ if and only if

$$\forall \mu; a; \sigma_1; A_{\prec}, A_T (\mu; a \vdash$$

$$\sigma\Delta; \sigma\Pi \text{ consistent using } \sigma_1; (A_{\prec}, A_T)) \Rightarrow$$

$$\exists A'_T (\mu; a \vdash \sigma\Delta'; \sigma\Pi' \text{ consistent using } \sigma_1; (A_{\prec}, A'_T))$$

So let us select arbitrary $\mu; a; \sigma_1; A_{\prec}, A_T$ such that $\mu; a \vdash \sigma E$ consistent using $\sigma_1; (A_{\prec}, A_T)$. Then by definition:

a is acyclic

$$\sigma_1 : \sigma\Delta \rightarrow \emptyset$$

$$\mu; A; \emptyset \vdash \sigma_1\sigma\Pi \Downarrow \hat{\Pi}$$

$$(l : \tau_{\text{atom}}) \in \hat{\Pi} \Rightarrow \vdash \mu(l) : \tau_{\text{atom}}$$

$$(l : \tau \prec l') \in A_{\prec} \Rightarrow (l \prec l') \in a$$

$$t(\nu_1, \dots, \nu_n) \in A_T \Rightarrow A \vdash [r_1 \rightarrow \nu_1, \dots, r_n \rightarrow \nu_n]T(t) =$$

true

If we now define $\sigma_2 = \sigma_1\sigma$, we can conclude that

$$\sigma_2 : \Delta \rightarrow \emptyset$$

$$\mu; A; \emptyset \vdash \sigma_2\Pi \Downarrow \hat{\Pi}$$

And therefore,

$$\mu; a \vdash \Delta; \Pi \text{ consistent using } \sigma_2; (A_{\prec}, A_T)$$

Because $\Delta; \Pi \geq \Delta'; \Pi'$, this in turn implies that

$$\mu; a \vdash \Delta'; \Pi' \text{ consistent using } \sigma_2; (A_{\prec}, A_T)$$

which is to say:

a is acyclic

$$\sigma_2 : \Delta' \rightarrow \emptyset$$

$$\mu; A; \emptyset \vdash \sigma_2\Pi' \Downarrow \hat{\Pi}'$$

$$(l : \tau_{\text{atom}}) \in \hat{\Pi}' \Rightarrow \vdash \mu(l) : \tau_{\text{atom}}$$

$$(l : \tau \prec l') \in A_{\prec} \Rightarrow (l \prec l') \in a$$

$$t(\nu_1, \dots, \nu_n) \in A_T \Rightarrow A \vdash [r_1 \rightarrow \nu_1, \dots, r_n \rightarrow \nu_n]T(t) =$$

true

Substituting again with $\sigma_2 = \sigma_1\sigma$ gives:

$$\sigma_1 : \sigma\Delta' \rightarrow \emptyset$$

$$\mu; A; \emptyset \vdash \sigma_1\sigma\Pi \Downarrow \hat{\Pi}'$$

Therefore, by definition,

$$\exists A'_T (\mu; a \vdash \sigma\Delta'; \sigma\Pi' \text{ consistent using } \sigma_1; (A_{\prec}, A'_T))$$

which is what needed to be proven.

Now let $E = (\Delta; \Pi)$. ($E \geq E_1 \vdash_\omega e : \tau \dashv E'_1 \geq E'$), and $\sigma : \Delta_1 \rightarrow \Delta_2$ where $\Delta_1 \uplus \Delta_2$ is a partition of Δ . As above, $\sigma E \geq \sigma E_1$ and $\sigma E'_1 \geq \sigma E'$. Let $E_1 = (\Delta''; \Pi'')$. Then define $\sigma' : \Delta_1 \cap \Delta'' \rightarrow (\Delta'' - (\Delta_1 \cap \Delta''))$ as $\sigma' = \sigma|_{\Delta_1 \cap \Delta''}$. By Lemma 2.1, $\sigma' E_1 \vdash_\omega e : \tau \dashv \sigma' E'_1$. But $\sigma' E_1 = \sigma E_1$ from its definition, and, as the typing can only introduce fresh variables, $\sigma' E'_1 = \sigma E'_1$. Thus, $\sigma E \geq \sigma E_1 \vdash_\omega e : \tau \dashv \sigma E'_1 \geq \sigma E'$ \square

LEMMA 2.8. *The following rules hold:*

$$E \equiv E \quad \Delta; \Pi \geq \Delta; \emptyset \quad \Delta; \emptyset \equiv \Delta; \text{true}$$

$$\Delta; \Gamma \wedge \Gamma' \equiv \Delta; \Gamma, \Gamma' \quad \Delta; \Gamma \equiv \Delta; \neg \neg \Gamma$$

$$\Delta; t(\rho_1, \dots, \rho_n) \equiv \Delta; [r_1 \rightarrow \rho_1, \dots, r_n \rightarrow \rho_n]T(t)$$

$$(\Delta; \Pi \geq \Delta; \Gamma) \Rightarrow (\Delta; \Pi \equiv \Delta; \Pi, \Gamma) \quad \Delta; \neg \Gamma \geq \Delta; \Gamma \rightarrow \Pi$$

$$\Delta; \Gamma, \Pi \equiv \Delta; \Gamma, \Gamma \rightarrow \Pi \quad \Delta; \Pi \equiv \Delta; \emptyset \rightarrow \Pi$$

$$\Delta; \emptyset \equiv \Delta; B \rightarrow B$$

REDUCE

$$\Delta; B_1 \rightarrow B_2, (B_2, B_3) \rightarrow \Pi_4 \geq \Delta; (B_1, B_3) \rightarrow \Pi_4$$

CARVE-OUT

$$\Gamma = k : \tau \prec k' \wedge k \neq k_1 \wedge \dots \wedge k \neq k_n$$

$$B = \{\{k_1 : \tau_1, \dots, k_n : \tau_n\}\}$$

$$\frac{}{\Delta; k' : \tau' \setminus \{B\}, \Gamma \geq \Delta; k' : \tau \setminus \{k : \tau, B\}, k : \tau}$$

PACK

$$(\Delta; k : \text{ptr}(\rho), [r \rightarrow \rho]\Pi) \geq (\Delta; k : \exists r. \text{ptr}(r) \text{ with } \Pi)$$

UNPACK

r' is fresh

$$\frac{}{(\Delta; k : \exists r. \text{ptr}(r) \text{ with } \Pi) \geq (\{r'\} \cup \Delta; k : \text{ptr}(r'), [r \rightarrow r']\Pi)}$$

PROOF. In the proof, we use $A'_T = A_T, \sigma' = \sigma$ except when otherwise stated.

$E \equiv E$

Trivial.

$\Delta; \Pi \geq \Delta; \emptyset$

This case follows immediately since $\mu; A; \emptyset \vdash \sigma\emptyset \Downarrow \{\}$ is an axiom.

$\Delta; \emptyset \equiv \Delta; \text{true}$

The \leq direction is already established by the previous case. The \geq direction is established by the permission consistency rule for $\Gamma = \text{true}$.

$\Delta; \Gamma \wedge \Gamma' \equiv \Delta; \Gamma, \Gamma'$

This case works because both sides require $A \vdash \Gamma = \text{true}$ and $A \vdash \Gamma' = \text{true}$.

$\Delta; \Gamma \equiv \Delta; \neg\neg\Gamma$

This case works because both sides require $A \vdash \Gamma = \text{true}$.

$\Delta; t(\rho_1, \dots, \rho_n) \equiv \Delta; [r_1 \rightarrow \rho_1, \dots, r_n \rightarrow \rho_n]T(t)$

For the \geq direction: the left side can achieve consistency only if $t(\sigma\rho_1, \dots, \sigma\rho_n) \in A_T$, which requires in turn that $A \vdash [r_1 \rightarrow \sigma\rho_1, \dots, r_n \rightarrow \sigma\rho_n]T(t) = \text{true}$ which (since we assume the correct number of parameters to t) is equivalent to $A \vdash \sigma[r_1 \rightarrow \rho_1, \dots, r_n \rightarrow \rho_n]T(t) = \text{true}$ which is precisely what is needed to prove consistency of the right-hand side.

For the \leq direction, let $A'_T = A_T \cup \{t(\sigma\rho_1, \dots, \sigma\rho_n)\}$, which is no obstacle to consistency because the right-hand side requires this fact. With this addition, proving consistency of the left-hand side is straight-forward.

$(\Delta; \Pi \geq \Delta; \Gamma) \Rightarrow (\Delta; \Pi \equiv \Delta; \Pi, \Gamma)$

The \leq direction of the equivalence to prove follows immediately since removing Γ imposes no obstacle to consistency. For the \geq direction, the antecedent shows that Γ can be found consistent, possibly with a new A_T that we shall call A'_T . Now let $A'_T = A_T \cup A_T^\Gamma$. The new permissions Π, Γ are consistent as before and the expanded A'_T will still be checkable since it is composed of parts that were checkable previously.

$\Delta; \neg\Gamma \geq \Delta; \Gamma \multimap \Pi$

This case follows immediately because the left-hand side requires $A \vdash \Gamma = \text{false}$ which enables the right-hand-side to be proved.

$\Delta; \Gamma, \Pi \equiv \Delta; \Gamma, \Gamma \multimap \Pi$

The \geq direction is similar to the last case, the result follows since the left-hand side requires $A \vdash \Gamma = \text{true}$. The \leq direction is also simple: the right side requires $A \vdash \Gamma = \text{true}$ and thus the proof of $\Gamma \multimap \Pi$ requires that we have Π , and thus the consistency of the left-hand side is easily established.

$\Delta; \Pi \equiv \Delta; \emptyset \multimap \Pi$

This case follows immediately from the consistency axiom for $B_2 \multimap \Pi$ specialized for the case $B_2 = \emptyset$:

$$\frac{\mu; A; B_1 \vdash \Pi \Downarrow \hat{\Pi}}{\mu; A; B_1 \vdash \emptyset \multimap \Pi \Downarrow \hat{\Pi}}$$

$\Delta; \emptyset \equiv \Delta; B \multimap B$

The \leq direction is a special case of the second case. The \geq direction is handled by constructing a proof for consistency with one copy of CP-IMP (with $B_1 = \emptyset, B_2 = \sigma B$) and multiple copies of CP-UNION until we have split the permissions to single keys and apply CP-IDENTITY to cancel each $\sigma\beta$ against itself.

$\Delta; B_1 \multimap B_2, (B_2, B_3) \multimap \Pi_4 \geq \Delta; (B_1, B_3) \multimap \Pi_4$

This rule is a generalization of the linear *modus ponens* rule, which is achieved when $B_1 = B_3 = \emptyset$.

When proving this rule, we may without loss of generality assume all variables have been substituted away ($\Delta = \emptyset$ and σ is the empty substitution) because if there are variables, the consistency rule will immediately substitute them away.

Now let $B_2 = \{\{\beta_1, \dots, \beta_n\}\}$, and thus from consistency on the left we have the following facts:

$$\mu; A; B_{1i} \vdash \beta_i \Downarrow \hat{\Pi}_2 \quad B_1 = \sum_{1 \leq i \leq n} B_{1i}$$

$$\mu; A; \beta_1, \dots, \beta_n, B_3 \vdash \Pi_4 \Downarrow \hat{\Pi}_4 \quad \hat{\Pi} = \hat{\Pi}_2 \uplus \hat{\Pi}_4$$

If B_2 is empty ($n = 0$), then so must be B_1 and the result is trivial. The case for $n > 1$ can be handled by repeatedly applying the $n = 1$ case. Thus without loss of generality, we may assume $n = 1$ and thus $B_2 = \{\{\beta\}\}$. So to summarize, we have the situation:

$$\mu; A; B_1 \vdash \beta \Downarrow \hat{\Pi}_2 \quad \mu; A; \beta, B_3 \vdash \Pi_4 \Downarrow \hat{\Pi}_4$$

$$\hat{\Pi} = \hat{\Pi}_2 \uplus \hat{\Pi}_4$$

Now we prove by induction over the derivation of the second rule that whenever we have these three rules, we also have

$$\mu; A; (B_1, B_3) \vdash \Pi_4 \Downarrow \hat{\Pi}$$

Proving this “mini-lemma” requires that we examine the following cases for the last step in the derivation of $\mu; A; \beta_1, B_3 \vdash \Pi_4 \Downarrow \hat{\Pi}_4$:

CP-Empty $\Pi_4 = \emptyset$ (Impossible)

CP-True $\Pi_4 = \Gamma$ (Impossible)

CP-FalseImp $\Pi_4 = \Gamma \multimap \Pi'$ where $A \vdash \Gamma = \text{false}$ (Impossible)

CP-ImpTrue $\Pi_4 = \Gamma \multimap \Pi'$ (Follows immediately by induction)

CP-Imp $\Pi_4 = B_3 \multimap \Pi'$ (By induction)

CP-Union $\Pi_4 = \Pi_1, \Pi_2$

Here β must appear on the left-hand side of one of the two facts above the line; we use induction on that one, and the other remains as before, and the desired result follows.

CP-Identity $\beta, B_3 = \Pi_4 = \{\{l : \tau\}\}$

In this case, $\hat{\Pi}_4 = \emptyset$. Also, we must have $\beta = (l : \tau), B_3 = \emptyset$, which means the required fact is already established.

CP-Field Here we have

$$B'_1 = \sum_{(l': \tau' \prec l) \in A_{\prec}} B_{l': \tau'}$$

$$\mu; A; B_{l': \tau'} \vdash l' : \tau' \Downarrow \hat{\Pi}'_{l': \tau'}$$

$$\hat{\Pi}'_1 = \biguplus_{(l': \tau' \prec l) \in A_{\prec}} \hat{\Pi}'_{l': \tau'}$$

$$\frac{\mu; A; B'_2 \vdash \mu(l) : \tau \Downarrow \hat{\Pi}'_2 : \tau_{\text{atom}}}{\mu; A; B'_1, B'_2 \vdash l : \tau \Downarrow \hat{\Pi}'_1 \uplus \hat{\Pi}'_2 \uplus \{\{l : \tau_{\text{atom}}\}\}}$$

we have two cases, either $\beta \in B'_1$ or $\beta \in B'_2$ (or both). In the former case, that means that $\beta \in$

$B_{l':\tau'}$ for some $l' : \tau' \prec l \in A_{\prec}$. Then since $\hat{\Pi}_1$ is disjoint with $\hat{\Pi}_4 = \hat{\Pi}'_1 \uplus \hat{\Pi}'_2 \uplus \{l : \tau_{\text{atom}}\}$ then it is certainly disjoint with all the $\hat{\Pi}'_{l':\tau'}$ that partition $\hat{\Pi}'_1$. Then by induction for the $l' : \tau' \prec l$ case, and the associativity of \uplus , we achieve the desired result.

If on the other hand $\beta \in B'_2$, then if τ is atomic, we are done since $\hat{\Pi}'_2 = \{\}$. Otherwise if τ is an existential, then we can use induction on the permissions packed into the existential and again achieve our result.

Thus with this mini-lemma (which is also used in the following case), we can now state

$$\mu; A; \emptyset \vdash (B_1, B_3) \multimap \Pi_4 \Downarrow \hat{\Pi} \quad .$$

which enables us to prove consistency of the right-hand side.

Carve-Out As before, we ignore variables. Now the expansion of the $\cdot : \cdot \setminus \{\dots\}$ adds adoption facts to a linear implication.

Thus $k' : \tau' \setminus \{k_1 : \tau_1, \dots, k_n : \tau_n\}, k : \tau \prec k', k \neq k_1, \dots, k \neq k_n$ where each k has no variables (is a l) in a consistent state $\mu; a$ using (A_{\prec}, A_T) requires

$$\begin{aligned} & \mu; A; l_1 : \tau_1, \dots, l_n : \tau_n \vdash k' : \tau' \Downarrow \hat{\Pi} \\ (l_i : \tau_i \prec l') \in A_{\prec} \quad & (l : \tau \prec l') \in A_{\prec} \quad l \neq l_i \end{aligned}$$

Now let $\{l'' : \tau'' \prec l \in A_{\prec}\}$ be enumerated in the following order for simplicity (where $(l_{n+1} : \tau_{n+1}) = (l : \tau)$):

$$\{l_1 : \tau_1, \dots, l_n : \tau_n, l_{n+1} : \tau_{n+1}, \dots, l_m : \tau_m\}$$

Now since adoption is acyclic $l' \neq l_i$ for any $1 \leq i \leq m$, and so the last consistency rule for permissions must be the one to prove the consistency of $k' : \tau$, which means we have the following facts:

$$\begin{aligned} & \mu; A; B_i \vdash l_i : \tau_i \Downarrow \hat{\Pi}_i \\ (l_1 : \tau_1, \dots, l_n : \tau_n) = B_0 + \sum_{1 \leq i \leq m} B_i \\ & \mu; A; B_0 \vdash \mu(l') : \tau' \Downarrow \hat{\Pi}_0 : \tau'_{\text{atom}} \\ \hat{\Pi} = \{l' : \tau'\} \uplus \hat{\Pi}_0 \uplus \biguplus_{1 \leq i \leq m} \hat{\Pi}_i \end{aligned}$$

Of interest here is B_{n+1} . If it is empty, we can create $B'_i = B_i$ except for $B'_{n+1} = \{l : \tau\}$. Now we can easily prove $\mu; A; l : \tau \vdash l : \tau \Downarrow \{\} = \hat{\Pi}'_i$, and keeping all

other $\hat{\Pi}'_i = \hat{\Pi}_i$, we can prove

$$\begin{array}{c} \mu; A; B'_i \vdash l_i : \tau_i \Downarrow \hat{\Pi}'_i \\ (l_1 : \tau_1, \dots, l_n : \tau_n, l_{n+1} : \tau_{n+1}) = B_0 + \sum_{1 \leq i \leq m} B'_i \\ \mu; A; B_0 \vdash \mu(l') : \tau' \Downarrow \hat{\Pi}_0 : \tau'_{\text{atom}} \\ \hat{\Pi}' = \{l' : \tau'\} \uplus \hat{\Pi}_0 \uplus \biguplus_{1 \leq i \leq m} \hat{\Pi}'_i \\ \hline \mu; A; l : \tau, l_1 : \tau_1, \dots, l_n : \tau_n \vdash l' : \tau' \Downarrow \hat{\Pi}' \\ \hline \mu; A \vdash (l : \tau, l_1 : \tau_1, \dots, l_n : \tau_n) \multimap l' : \tau' \Downarrow \hat{\Pi}' \\ \hline \vdots \\ \hline \mu; A; l_i : \tau_i \Downarrow \hat{\Pi}_i \\ \hline \mu; A; (l : \tau, l_1 : \tau_1, \dots, l_n : \tau_n) \multimap l' : \tau', l : \tau \Downarrow \hat{\Pi} = \hat{\Pi} \end{array}$$

which permits us to prove consistency of the right-hand side.

But suppose B_{n+1} is not empty: $B_{n+1} = l_j : \tau_j, B''$. In that case, we use the mini-lemma from the previous case to convert

$$\mu; A; B_j \vdash l_j : \tau_j \Downarrow \hat{\Pi}_j$$

$\mu; A; l_j : \tau_j, B'_{n+1} \vdash l : \tau \Downarrow \hat{\Pi}_{n+1} \quad \hat{\Pi}'_{n+1} = \hat{\Pi}_j \uplus \hat{\Pi}'_{n+1}$
into

$$\mu; A; B_j, B'' \vdash l : \tau \Downarrow \hat{\Pi}'_{n+1}$$

to which we add the axiom

$$\mu; A; l_j : \tau_j \vdash l_j : \tau_j \Downarrow \{\} = \hat{\Pi}'_j$$

and produce a new proof of our original fact using $B'_{n+1} = B_j, B'', B'_j = \{\{l_j : \tau_j\}\}$, $\hat{\Pi}'_j = \{\}$ and all the other $B'_i = B_i$, $\hat{\Pi}'_i = \hat{\Pi}_i$. If this new B'_{n+1} is still not empty we can repeat this process. Eventually because of linearity (we have a partition) and finiteness, we end up with a B^*_{n+1} which is empty and can proceed.

Pack $(\Delta; k : \text{ptr}(\rho), [r \rightarrow \rho]\Pi) \geq (\Delta; k : \exists r. \text{ptr}(r)$ with $\Pi)$

Let $l = \sigma k$. Associativity of *uplus* and consistency on the left requires that we have the following facts:

$$\begin{aligned} & \mu; A; \emptyset \vdash l' : \tau' \Downarrow \hat{\Pi}'_{l':\tau'} \quad \hat{\Pi}_1 = \biguplus_{l':\tau' \prec l} \hat{\Pi}'_{l':\tau'} \\ & \mu; A; \emptyset \vdash \mu(l) : \text{ptr}(\sigma\rho) \Downarrow \{\} : \text{ptr}(\sigma\rho) \\ & \mu; A; \emptyset \vdash \sigma[r \rightarrow \rho]\Pi \Downarrow \hat{\Pi}_3 \\ & \hat{\Pi} = \hat{\Pi}_1 \uplus \{l : \text{ptr}(\sigma\rho)\} \uplus \hat{\Pi}_3 \end{aligned}$$

Furthermore, this element of the flattened permissions $l : \text{ptr}(\sigma\rho)$ ensures that $\mu(l) = \sigma\rho$, some object reference, we call ν .

Now since r is fresh, $\sigma[r \rightarrow \rho]\Pi = [r \rightarrow \sigma\rho]\sigma_r\Pi = [r \rightarrow \nu]\sigma_r\Pi$, where $\sigma_r r = r, r' \neq r \Rightarrow \sigma_r r' = \sigma_r'$, and thus

$$\begin{array}{c} \vdots \\ \hline \mu; A; \emptyset \vdash [r \rightarrow \nu]\sigma_r\Pi \Downarrow \hat{\Pi}_3 \\ \hline \mu; A; \emptyset \vdash \nu : \sigma(\exists r. \text{ptr}(\rho)) \text{ with } \Pi \Downarrow \hat{\Pi}_3 \end{array}$$

which permits us to prove the consistency of the right-hand side.

Unpack

$$\frac{r' \text{ is fresh}}{(\Delta; k : \exists r. \text{ptr}(r) \text{ with } \Pi) \geq (\{r'\} \cup \Delta; k : \text{ptr}(r'), [r \rightarrow r']\Pi)}$$

Let $l = \sigma k$. For consistency on the left we must have used CP=FIELD and thus have the following facts:

$$\begin{aligned} \mu; A; \emptyset \vdash l' : \tau' \Downarrow \hat{\Pi}_{l':\tau'} \\ \hat{\Pi}_1 = \biguplus_{l':\tau' \prec l} \hat{\Pi}_{l':\tau'} \\ \mu; A; \emptyset \vdash \mu(l) : \sigma(\exists r. \text{ptr}(r) \text{ with } \Pi) \Downarrow \hat{\Pi}_2 : \text{ptr}(\sigma\rho) \\ \hat{\Pi} = \hat{\Pi}_1 \uplus \hat{\Pi}_2 \uplus \{l : \text{ptr}(\sigma\rho)\} \end{aligned}$$

Furthermore, this element of the flattened permissions $l : \text{ptr}(\sigma\rho)$ ensures that $\mu(l) = \sigma\rho$, and both equal some object reference, ν .

Let $\sigma' = \sigma[r' \rightarrow \nu]$. Trivially $\sigma' \supseteq \sigma$. Then, $\mu; A; \emptyset \vdash \mu(\sigma'k) : \text{ptr}(\sigma'r') \Downarrow \{\} : \text{ptr}(\nu)$. We can use this with the first two facts above to get:

$$\mu; A; \emptyset \vdash \sigma'(k : \text{ptr}(r')) \Downarrow \hat{\Pi}_1 \uplus \{l : \text{ptr}(\sigma'\rho)\} \quad (*)$$

(As σ and σ' differ only on r' (fresh) $\sigma\rho = \sigma'\rho$.)

The third fact, in turn, requires that

$$\mu; A; \emptyset \vdash [r \rightarrow \nu]\sigma\Pi \Downarrow \hat{\Pi}_2$$

But this is identical to

$$\mu; A; \emptyset \vdash \sigma'([r \rightarrow r']\Pi) \Downarrow \hat{\Pi}_2 \quad (*')$$

The facts $*$ and $*'$ can be combined using CP-UNION to get:

$$\begin{aligned} \mu; A; \emptyset \vdash \sigma'(k : \text{ptr}(r'), [r \rightarrow r']\Pi) \Downarrow \\ \hat{\Pi}_1 \uplus \hat{\Pi}_2 \uplus \{l : \text{ptr}(\sigma'\rho)\} \end{aligned}$$

The union of disjoint sets is well-defined as it duplicates that used to define $\hat{\Pi}$ above.

□

LEMMA 2.9. *Given a type-checked program g ($\vdash g : \omega$), an expression e that type-checks in a variable-free environment $E = (\emptyset; \Pi)$ ($E \vdash_\omega e : \tau \dashv E'', E'' = (\Delta''; \Pi'')$, we do not require $\Delta'' = \emptyset$) and a memory and adoption information consistent with E ($\mu; a \vdash E$ consistent), then either e is a value or there exists an evaluation step $(\mu; a; e) \rightarrow_g (\mu'; a'; e')$ and for any such evaluation step, there exists a substitution (with absolute addresses) on some of the new type variables $\sigma : \Delta \rightarrow \emptyset$ where $\Delta \subseteq \Delta''$ such that the resulting expression type-checks in a new environment $E' = (\emptyset; \Pi')$ with the substituted result ($E' \vdash_\omega e' : \sigma\tau \dashv \Delta'' - \Delta; \sigma\Pi''$). In addition, the witness A' for the new consistency will include everything in the witness of the original consistency A ($A_\prec \subseteq A'_\prec, A_T \subseteq A'_T$), and furthermore the new flattened permissions will only include locations from the old permissions or newly allocated memory: $\text{Dom}(\hat{\Pi}') \cap \text{Dom}(\mu) \subseteq \text{Dom}(\hat{\Pi})$.*

PROOF. Let σ and A be the (original) witnesses of consistency. We prove the result by induction over the typing derivation. with the following case analysis:

Unit, Num, True, False, Address

For all these cases, e is a value, and thus the result is vacuously satisfied.

Plus $e_1 + e_2$

If both e_1 and e_2 are values, then they must be integer constants and evaluation to another integer constant is immediate. The memory hasn't changed and typing is assured and thus we choose the same environment $E' = E$, an empty substitution $\sigma = []$ and use the same witnesses for consistency.

If e_1 is a value but e_2 is not, then we get an evaluation and the new environment and witnesses by induction. Since e_1 is a value, its typing is assured and we can form a typing of all of e' with the new environment.

If e_1 is not a value, then by induction, we get a new environment E' and substitution σ with witnesses for the consistency of the memory after the evaluation of e_1 . Thus we have evaluation of e and using the substitution lemma can type e_2 in the (substituted) output environment from e' .

Equal $e_1 = e_2$

Analogous to the case for PLUS. Here we use the fact that substitutions on atomic types always yield atomic types, in fact types that are storage compatible, thus enabling us to preserve the typing $\tau_1 \sim \tau_2$.

Read $e_1 . f$

If e_1 is a value, then it must be an object ν , and $\tau_1 = \text{ptr}(\nu)$. We must have therefore $\Pi = \nu.f : \tau \setminus \{B\}, \Pi_1$ where τ is an atomic type. By consistency, we know that $\mu; A; B \vdash \nu.f : \tau \Downarrow \hat{\Pi}_1$ where $\hat{\Pi}_1 \subseteq \hat{\Pi}$ used check μ for consistency, and where every $\beta \in B$, we have $(\beta_i \prec \nu.f) \in A_\prec$ and thus for every $\beta_i = l_i : \tau_i$ we have $(l_i \prec \nu.f) \in a$. By the acyclicity of a , this means $l_i \neq \nu.f$, and thus the only rule for consistency is the final one, which (since τ is already atomic) requires that $\hat{\Pi}_1 \ni \nu.f : \tau$. Thus the rule for consistency requires that $\vdash \mu(\nu.f) : \tau$. Given these facts, we can determine that evaluation to $(\mu; a; e')$ is assured where $e' = \mu(\nu.f)$. Now let $E' = E, \sigma = []$, and so $E \vdash_\omega e' : \tau \dashv E''$ is true. Since we have $\mu' = \mu, a' = a$, consistency is checking the same values.

Otherwise if e_1 is not a value, then the result follows by induction.

New $\text{new}\{f_i \mid 1 \leq i \leq n\}$

In this case, we have $\tau = \text{ptr}(r)$ for a fresh variable r . Since we assume μ is finite while the set of addresses is infinite, evaluation is assured to a new state $(\mu'; a'; e')$ where $a' = a, e' = \nu$. Now let $\sigma = [r \rightarrow \nu]$ be a substitution that replaces the fresh variable with ν , and thus $\sigma\tau = \text{ptr}(\nu)$. Now let $E' = \sigma E''$; it is clear that $E' \vdash_\omega e' : \tau' \dashv E'$, and thus all we need to prove is the preservation of consistency. The new set of permissions Π' has additions for the newly allocated fields: $\{\nu.f_1 : \tau_{f_1}, \dots, \nu.f_n : \tau_{f_n}\}$. Now the evaluation ensures us that $\nu.f_i \notin \text{Rng}(a = a')$, and thus

$\mu'; a'; \emptyset \vdash \nu.f_i : \tau_{f_i} \Downarrow \{\nu.f_i : \tau_{f_i}\}$. Next we check that these flattened sets are disjoint with the flattened permissions from the original Π . Since these permissions were consistent with μ , and because that means μ must be defined on the locations in these flattened sets, and because the evaluation rule ensures that $\mu\nu.f$ was not defined for any f , we get the desired result. We note that this meets the requirements on the domain of the new flattened permissions. Finally, we need to check the full flattened permission set against μ' . It is only changed for the new locations, so it remains consistent with the original flattened permissions, and the changes of evaluation gives it exactly the correct values that correspond to the fields' initial types and thus we are done.

Write $e_1.f := e_2$

For this type rule, the type of the pointer e_1 is $\text{ptr}(\rho)$ and the type τ of new value is storage compatible with τ' the type recorded for the field in the permission. If e_1 and e_2 are both values, then e_1 must be an object reference ν . As with the rule for **READ**, consistency requires that $\vdash \mu(\nu.f) : \tau'$ and τ' is atomic. We also have that $\vdash e_2 : \tau$, the value is of the required (atomic) type. Since the type rule requires $\tau \sim \tau_f$, we have $e_2 \sim \nu.f$ and thus we have progress to $(\mu; a; ())$. Let $E' = E''$, and thus $E' \vdash_{\omega} () : \text{unit} \dashv E''$ follows immediately. Now, because of the union of disjoint sets computing $\hat{\Pi}$, Π_1 must not flatten to include a requirement that $\mu(o.f)$ has type τ' , and thus changing the type of this field will not cause problems, and indeed the new type now matches what is in the store, and so we have consistency. The domain of the flattened permissions remains the same as before.

If e_1 is a value, but not e_2 , the desired result follows immediately by induction. If e_1 is not a value, again we use induction, and then also the substitution lemma to achieve the required result.

Seq $e_1; e_2$

If e_1 is a value, then since it must have unit type, it can only be $()$ and thus we have one step of evaluation to $(\mu; a; e_2)$. The type rule permits us to write $E \vdash_{\omega} e_1 : \text{unit} \dashv E_1 \vdash_{\omega} e_2 : \tau \dashv E_2$ and since $e_1 = ()$, we have $E_1 = E$. Let $E' = E = E_1$ and then typing is preserved trivially. Consistency is also preserved since no part of the relation is affected by evaluation.

If e_1 is not a value, then the result follows through application of induction and the substitution lemma.

If $\text{if } e_0 \text{ then } e_1 \text{ else } e_2$

If e_0 is a value, then since the type is boolean, it must be **true** or **false**. In the former case, we have evaluation immediately to $(\mu; a; e_2)$. Now we know that $E \vdash_{\omega} e_0 : \text{bool} \dashv E' \vdash_{\omega} e_1 : \text{unit} \dashv E_1$ and $E' = E$, and also that $E'' = E_1 \vee E_2$, which means $E_1 = \sigma_1 E''$, and thus the substitution we use is $\sigma = \sigma_1$. Thus we have the typing result needed, and since memory and environment are unchanged, consistency is also as needed. The case for $e_0 = \text{false}$ is analogous.

Now if e_0 is not a value, we can use induction and the substitution lemma (including the part that says

that substitution carries over \vee) to achieve the desired result.

IfEqual $\text{if } e_0 = e_1 \text{ then } e_2 \text{ else } e_3$

If both e_0 and e_1 are values, then given their types, they must be ν_1 and ν_2 respectively, and their types must be $\text{ptr}(\nu_1)$ and $\text{ptr}(\nu_2)$. Now the type rule ensures that e_2 type checks in an environment in which the equality is assumed and e_3 in the environment where they are assumed not equal:

$$\Delta; \nu_1 = \nu_2, \Pi \vdash_{\omega} e_2 : \text{unit} \dashv E_2 \Delta; \nu_1 \neq \nu_2, \Pi \vdash_{\omega} e_3 :$$

$$\text{unit} \dashv E_3 E'' = E_2 \vee E_3$$

If the objects are indeed equal, we have immediate progress to **if true then** e_2 **else** e_3 . Furthermore, (it can be easily shown), the equality fact adds no information and can be dropped, and thus we have $E \vdash_{\omega} e_2 : \text{unit} \dashv E_2$. By the definition of \vee , we have $E_2 = \sigma_2 E''$ and thus $E \vdash_{\omega} e_2 : \text{unit} \dashv \sigma E''$ which is precisely what we need to use **IFTRUE** to type-check the new program. Since the store and environment are unchanged, consistency also follows. Thus we have achieved the desired result. The case for inequality is completely analogous.

If either e_0 or e_1 is not a value, then we can use the inductive hypothesis and straightforward reasoning afterwards.

IfTrue $\text{if true then } e_1 \text{ else } e_2$

Trivial.

IfFalse $\text{if false then } e_1 \text{ else } e_2$

Trivial.

Call $\text{call } p$

The type rule, **PROC** and **PROGRAM**, ensure we have the following facts:

$$E = \emptyset; \sigma_1 \Pi_1, \Pi_3$$

$$E'' = \Delta_+; \sigma_1 \Pi_2, \Pi_3$$

$$\omega(p) = \forall \Delta_1. \Pi_1 \rightarrow \exists \Delta_2. \sigma_2 \Pi_2$$

$$\sigma_1 : \Delta_1 \rightarrow \emptyset$$

$$\Delta_+ \text{ fresh}$$

$$\sigma_2 : \Delta_+ \rightarrow \Delta_2$$

$$\Delta_1; \Pi_1 \vdash_{\omega} g(p) : \text{unit} \dashv \Delta'_1; \sigma_3 \sigma_2 \Pi_2$$

$$\Delta'_1 \cap \Delta_2 = \emptyset$$

$$\sigma_3 : \Delta_2 \rightarrow \Delta'_1$$

In particular, we know that $g(p)$ is defined and thus we have immediate progress to $(\mu; a; g(p))$. By application of the substitution lemma (for σ_1) and the widening lemma (adding Π_3 to both sides), we achieve

$$E \vdash_{\omega} g(p) : \text{unit} \dashv \sigma_1 \Delta'_1; \sigma_1 \sigma_3 \sigma_2 \Pi_2, \Pi_3$$

Since σ_1 is idempotent and neither σ_2 nor σ_3 have any effect on variables in Δ_1 (Δ_1 must be disjoint with

both Δ_2 and with fresh variables Δ_+ , $\sigma_1\sigma_3\sigma_2\Pi_2 = \sigma_1\sigma_3\sigma_2(\sigma_1\Pi_2)$. Let $\sigma = \sigma_1\sigma_3\sigma_2$. Now $\sigma\Delta_+ = \sigma_1\sigma_3\sigma_2\Delta_+ = \sigma_1\sigma_3\Delta_2 = \sigma_1\Delta'_1$ and thus since Π_3 has no free variables, the previous typing can be written $E \vdash_\omega g(p) : \text{unit} \dashv \sigma E''$ which is the required type preservation. Consistency is preserved trivially since the input environment $E' = E$, memory and adoption are all unchanged.

Nest nest $e_0.f_0$ in $e_1.f_1$

If both e_0 and e_1 are values, then the typing rules require (using the unstated canonical forms lemma) that they both be objects ν and ν' respectively, and thus we have immediate progress to $(\mu; a'; \emptyset)$ where $a' = a \cup \{(o, f) \prec (o', f')\}$. Typing is preserved too using $E' = E''$ and UNIT: $E' \vdash_\omega () : \text{unit} \dashv E''$. Preservation of consistency is more interesting. The starting permissions are

$$\Pi = (l : \tau, l \neq l_1 \wedge \dots \wedge l \neq l_n), \\ l' : \tau' \setminus \{l_1 : \tau_1, \dots, l_n : \tau_n\}, \Pi_1$$

Consistency in the original state gives inequalities (which we do not need for this lemma) and also the following (we may need to use the associativity and commutativity of $+$ and \uplus to rearrange the consistency proof to have this shape):

$$\mu; A; \emptyset \vdash l : \tau \Downarrow \hat{\Pi}_2 \\ \mu; A; l_1 : \tau_1, \dots, l_n : \tau_n \vdash l' : \tau' \Downarrow \hat{\Pi}_3 \\ \mu; A; \emptyset \vdash \Pi_1 \Downarrow \hat{\Pi}_1 \quad \hat{\Pi} = \hat{\Pi}_1 \uplus \hat{\Pi}_2 \uplus \hat{\Pi}_3$$

Furthermore, we have $a \ni (l_i \prec l)$ for all $1 \leq i \leq n$. Since the adoption relation cannot be cyclic the consistency rule for the second consistency fact above cannot be the self-canceling one, and must instead use adoption. Now if $l' \prec^* l$ using our original adoption relation, then the consistency proof for the first consistency fact above would include a subrule $\mu; A; \emptyset \vdash l' : \tau'' \Downarrow \hat{\Pi}_n$ but then $\hat{\Pi}_n$ would include $l' : \tau''_{\text{atom}}$ whereas $\hat{\Pi}_3 \ni l' : \tau''_{\text{atom}}$. If these atomic types are the same, then the union of disjoint sets operator will fail at some point. If they are different, then $\mu(l)$ will be required to match two different atomic types, which is not possible. Thus the new addition to a cannot cause a cycle. Now define $A'_\prec = A_\prec \cup \{l : \tau \prec l'\}$ and thus matches a' as required for consistency. It remains only to show that we can form the new set of flattened permissions and that this set has no more requirements upon μ than did the previous set.

We consider two cases: whether or not the location was already adopted with this type: $l : \tau \prec l' \in A_\prec$. If it was already adopted, we have $A'_\prec = A_\prec, a' = a$ and thus there is no change in the consistency proof and thus we have $\mu; A'; l_1 : \tau_1, \dots, l_n : \tau_n \vdash l' : \tau' \Downarrow \hat{\Pi}_3$ and thus, the consistency proof simply produces a smaller set than it did before, and thus consistency is preserved.

Otherwise, first we note that $\mu; A; \emptyset \vdash \Pi_1 \Downarrow \hat{\Pi}_1$ cannot depend on the absence of the new adoption fact. This is due to the ways in which adoption facts are used. In particular, a fact permission $\pi = \Gamma$ can never depend

on the *absence* of an adoption fact. Similarly for the case $\pi = \Gamma \dashv \Pi$. And the only other place adoption is used is when we use the consistency rule CP-FIELD. If the consistency proof for Π_1 uses this rule and depends on the presence or absence of $l : \tau \prec l'$ then the resulting flattened permission set must include an element of the form $l' : \tau''_{\text{atom}}$ which would have caused consistency problems as described previously. Thus we conclude $\mu; A'; \emptyset \vdash \Pi_1 \Downarrow \hat{\Pi}_1$.

Next, the new application of CP-FIELD for $l' : \tau'$ is completed by adding the subgoal $\mu; A'; \emptyset \vdash l : \tau \Downarrow \hat{\Pi}_2$, which means that we have:

$$\mu; A'; l_1 : \tau_1, \dots, l_n : \tau_n \vdash l' : \tau' \Downarrow \hat{\Pi}_3 \uplus \hat{\Pi}_2 \\ \mu; A'; \emptyset \vdash \Pi_1 \Downarrow \hat{\Pi}_1 \quad \hat{\Pi}' = \hat{\Pi}_1 \uplus \hat{\Pi}_3 \uplus \hat{\Pi}_2$$

This set is the same as before and thus consistency is preserved.

Transform $E \geq E_1 \vdash_\omega e : \tau \dashv E_2 \geq E''$

If e is a value, we are done. Otherwise, we have progress by induction $(\mu; a; e) \rightarrow_g (\mu'; a'; e')$, and also have E'_1 and σ where $E'_1 \vdash_\omega e' : \tau \dashv \sigma E_2$ and $\mu'; a' \vdash E'_1$ consistent. Now by the substitution lemma, $\sigma E_2 \geq \sigma E''$ and thus we can apply TRANSFORM again to achieve:

$$E'_1 \geq E'_1 \vdash_\omega e' : \tau \dashv \sigma E_2 \geq \sigma E''$$

which is the required result. The set of flattened permissions is always a subset of the original ones (Lemma 2.4) and thus the domain requirement is met. \square

LEMMA 2.10. *Given a type-checked program g ($\vdash g : \omega$), an expression e that type-checks in a variable-free environment $\emptyset; \Pi \vdash_\omega e : \tau \dashv E''$ and a memory and adoption consistent in an environment with more permissions ($\mu; a \vdash \emptyset; \Pi, \Pi_+$ consistent), then the evaluation of the expression (if any) will not read or write any field mentioned in the flattening of the extra permissions ($\hat{\Pi}_+$).*

PROOF. First we note that consistency requires first that $\text{Dom}(\hat{\Pi}) \cup \text{Dom}(\hat{\Pi}_+) \subseteq \text{Dom}(\mu)$, and second that the two sets of (flattened) permissions refer to different segments of memory: $\text{Dom}(\hat{\Pi}) \cap \text{Dom}(\hat{\Pi}_+) = \emptyset$. Otherwise, either they would have overlap (and have the union of disjoint sets be undefined) or they would have conflicting requirements on the store which would make consistency impossible.

Now the statement is trivial if evaluation is already complete (e is a value). Otherwise, we proceed with induction over the length of the evaluation sequence, by first showing the result for one step of evaluation and then showing that we can re-establish the conditions of the lemma.

First, however, we observe that if if we type a value v ($E \vdash_\omega v : \tau \dashv E'$) then the resulting flattened set of permissions for any consistent memory will be a (possible improper) subset of the original flattened permissions $\hat{\Pi}' \subseteq \hat{\Pi}$. This follows because the five rules specifically for values UNIT/NUM/TRUE/FALSE/ADDRESS make no change in the environment at all, and the only other one TRANSFORM (which may be applied an arbitrary number of times) can only produce a subset of the flattened permissions (by Lemma 2.4).

If evaluation proceeds for one step $Eval\mu; ae\mu'; a'e'$ let $\hat{\Pi}$ be the flattened permissions before evaluation. We make a simple proof by induction over the typing of e that any location l read or written in the evaluation will be present in the flattened permissions: $l \in \text{Dom}(\hat{\Pi})$:

Unit/Num/True/False/Address No evaluation possible.

Plus e_1+e_2

If e_1 is not a value, then the next step of evaluation will evaluate e_1 . The environment used in the typing of e_1 is same as the environment for e and thus the result follows immediately by induction. Otherwise, if it is a value, but e_2 is not, then we see that the flattened permissions for the environment used to check e_2 are a subset, and thus the result follows by induction. If both e_1 and e_2 are values, then they must be integer constants, and thus the evaluation does not access memory at all.

Equal $e_1=e_2$

Analogous to PLUS

Read $e_1.f$

If e_1 is not a value, then the result follows by induction (there is no change in environment). Otherwise, e_1 must be a value of type $\text{ptr}(\nu)$ (given that the environment is empty). Now E_1 (the environment after typing the object) must either be the same as E , or must be a transformed version of E . In either case, it must have an empty Δ , and its permissions Π_1 must include $\nu.f : \tau \setminus \{\dots\}$. Using a similar process as was used for the proof of progress, we can show that the flattened permissions $\hat{\Pi}_1$ must include $\nu.f : \tau$. Thus it must be in the flattened permissions for the original environment $\hat{\Pi}$.

New $\text{new } \{ \dots \}$

This construct does not access any existing memory locations when evaluated.

Write $e_1.f:=e_2$

Analogous to READ

Seq $e_1;e_2$

If e_1 is a value, then it must be $()$, and thus evaluation goes forward without any read or write effects. Otherwise, the result follows by induction.

If/IfEqual $\text{if } e_0 \text{ then } e_1 \text{ else } e_2$

If e_0 is a value, then it must be a boolean constant and evaluation proceeds without affecting memory. Otherwise the result holds by induction.

IfTrue/IfFalse/Call Evaluation doesn't use memory.

Nest $\text{nest } e_0.f_0 \text{ in } e_1.f_1$

If e_0 is not a value, the result follows by induction. Otherwise, it e_1 is not a value, the result follows by induction once we take into account possible application of TRANSFORM, which as we have seen allow us to achieve our result still.

The interesting case is the one in which both are values. Now, the permission required (possibly after transformation that only reduce the flattened permissions) ensures that the flattened permissions will include the two locations of the nesting expression, and thus the lemma is proved.

Transform $E \geq E_1 \vdash_\omega e : \tau \dashv E_2 \geq E''$

By induction the result is true for E_1 and then because of Lemma 2.4, must be true for E as well.

Now we use the preservation aspect of Lemma 2.9 to get $\emptyset; \Pi' \vdash_\omega e' : \tau \dashv \sigma E''$, where $\text{Dom}(\hat{\Pi}') \cap \text{Dom}(\mu) \subseteq \text{Dom}(\hat{\Pi})$. Now since $\text{Dom}(\hat{\Pi}_+) \subseteq \text{Dom}(\mu)$ and as shown above $\text{Dom}(\hat{\Pi}) \uplus \text{Dom}(\hat{\Pi}_+) \subseteq \text{Dom}(\mu)$. As a result, we get that $\hat{\Pi}' \uplus \hat{\Pi}_+$ is well defined. Finally this union of disjoint sets is consistent with μ' because the first part is already consistent (by preservation), and the second part (which was consistent with μ) only puts requirements on locations that are not allowed to be modified in evaluation (by this lemma). Furthermore, evaluation only adds things to a , never removes them, and because consistency of permissions never depends on the absence of adoption information, we have $\mu'; a' \vdash \Pi', \Pi_+$ consistent, which preserves the conditions of this lemma for another evaluation. \square

THEOREM 2.11. *Given a type-checked program g ($\vdash g : \omega$), two expressions e_1, e_2 each of which type-checks in a variable-free environment $\emptyset; \Pi_i \vdash_\omega e_i : \text{unit} \dashv E'_i$) and a memory and adoption consistent with the combined environments: $(\mu; a \vdash \emptyset; \Pi_1, \Pi_2 \text{ consistent})$, then when evaluating the expressions in sequence $e_1; e_2$ no state will be accessed by both expressions.*

PROOF. Preservation and widening ensures that when we finish evaluating e_1 , we have a substitution $\sigma_1 : \Delta'_1 \rightarrow \emptyset$ for which $\emptyset; \sigma_1 \Pi'_1, \Pi_2 \vdash_\omega e_2 : \text{unit} \dashv \Delta'_2; \sigma_1 \Pi'_1, \Pi_2$. Then we apply Lemma 2.10 to ensure that evaluating e_2 does not access anything in $\hat{\Pi}'_1$ (the flattening of $\sigma \Pi'_1$). Furthermore, this set does not include any values for locations in $\text{Dom}(\hat{\Pi}_2)$. On the other hand, if we follow preservation step by step, we see that the evaluation e_1 only accessed locations in $\text{Dom}(\hat{\Pi}'_1)$. Thus the two expressions are separate. \square