Checking Interference with Fractional Permissions*

John Boyland**

University of Wisconsin-Milwaukee, USA boyland@cs.uwm.edu

Abstract. We describe a type system for checking interference using the concept of linear capabilities (which we call "permissions"). Our innovations include the concept of "fractional" permissions: reads can be permitted with fractional permissions whereas writes require complete permissions. This distinction expresses the fact that reads on the same state do not conflict with each other. One may give shared read access at one point while still retaining write permission afterwards. We give an operational semantics of a simple imperative language with structured parallelism and prove that the permission system enables parallelism to proceed with deterministic results.

1 Introduction

In this paper we describe a new way to check effects on mutable state (reads and writes) in imperative code for the purpose of determining when two segments of code are non-interfering. This information can be used by a compiler for scheduling purposes, or by a refactoring tool when reordering code. Analysis is made modular by having an effects specification for each procedure. Thus two tasks must be performed: checking that a procedure meets its effect specification; and the original task—checking interference for two statements.

Previous work suggests two different models:

effect-based In this model, one infers effects for statements. Effects inference has been studied extensively for functional languages [1, 2]. For a modular analysis, inferred effects for a procedure body are then checked against the declared effects. Interference is checked by comparing inferred effects. Each statement is type-checked in context Γ and produces a set of effects φ . For interference checking, a side condition $\varphi_1 \# \varphi_2$ is used to check that if there is a write in one set of effects, the other set includes no reads or writes on the same state:

$$\frac{\Gamma \vdash s_1 ! \varphi_1 \qquad \Gamma \vdash s_2 ! \varphi_2 \qquad \varphi_1 \# \varphi_2}{\Gamma \vdash s_1 \text{ does not interfere with } s_2}$$

 $^{^{\}star}$ This material is based upon work supported by the National Science Foundation under Grant No. 9984681

^{**} The author wishes to acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

See for example Reynolds syntactic control of interference [3], our earlier work [4] or Clarke and Drossopoulou's JOE [5].

permission-based In this model, one checks a statement to see if it can be executed under a given set of permissions. (For example, consider the lock checking of Flanagan and Abadi [6], or Boyapati *et al* [7,8].) A procedure is checked by determining whether the body can be typed using the declared effects (viewed as permissions). To determine if two statements can be executed in interleaved fashion, we see if the set of permissions can be partitioned into two sets, one for each statement:

$$\frac{\varPi_1 \vdash s_1 \qquad \varPi_2 \vdash s_2}{\varPi_1, \varPi_2 \vdash s_1 \text{ does not interfere with } s_2}$$

The permissions are treated as *linear* keys: they cannot be duplicated, or discarded. See for example Walker, Crary and Morrisett's capability language (CL) [9], Ishtiaq and O'Hearn's use of "bunched implication" (BI) logic [10], or Reynolds' "separation logic" [11, 12].

These two models are almost duals of each other (especially when the typing rules given above are seen simply as relations) but the practical difference between checking effect conflict $\varphi_1 \# \varphi_2$ and splitting of permissions Π_1, Π_2 causes the two approaches to be incomparable.

1.1 Problems with Effects and Permissions

Our earlier work [4] used an effect-based system: effects inferred for a method body were checked against the method's declared effects and two statement effects were compared to see if they conflict or not. When effects were compared, we needed the answer to an aliasing question to determine if there was overlap and hence conflict. Consider the following two compound statements:

$$\{ \ldots; *x = 10; \ldots \}$$
 $\{ \ldots; *y = 42; \ldots \}$

In order to determine whether these statements interfere, we need to know whether \mathbf{x} could be the same as \mathbf{y} . More precisely, we need to know if the set of cells that \mathbf{x} could point to at the point that the assignment $*\mathbf{x} = 10$ occurs, could overlap the set of cells that \mathbf{y} could point to at the second assignment. We call this kind of question a "MayEqual" question [13]. One simple way to answer the question conservatively is to use the fact that objects of different type cannot be aliases of each other. This approach is used by Clarke and Drossopoulou where the addition of ownership parameterization allows for fine type distinctions. But any less conservative analysis, such as a Steensgaard's "Points-To" analysis [14], will need to examine the code. In fact, to do a good job at MayEqual, one needs to know about data dependencies, in particular, about effects. Thus we are left with the unsatisfying result that the inferred effects do not in themselves include enough information to perform the interference checking; we must combine the effects analysis with an alias analysis.

An alternate technique is to use permissions. Each individual permission applies to a single part of the store and thus the mere existence of two separate (write) permissions ensures that they do not refer to the same area of storage. In order to handle allocation and deallocation which manufacture and consume permissions respectively, we check statements with an input and output set of permissions: $\Pi \vdash s \Rightarrow \Pi'$ means that s can be executed given permissions Π after which permissions Π' are available.

A problem that arises is how to distinguish reads (which do not conflict with each other) from writes (which conflict with each other and with reads). For example, Reynolds' separation logic [12] (unlike his earlier work [3]) does not permit one to separate two side-effecting computations that read the same state. Two different solutions have been used in previous work:

- Permit a linear key (giving write permission) to be coerced into a nonlinear key, which then permits only reads from this point forward. (This approach is possible using subtyping in CL.)
- Permit a linear key to be treated non-linearly in a bounded context. Wadler's let! construct [15] permits a linear variable to be used nonlinearly by code that only needs read access. SCIR [16] permits a linear key to be moved into a non-linear section while type checking a statement that only needs read permission. CL uses bounded quantification to pass a unique region to a function that can use any kind of region. (Thus CL supports both approaches.)

In the first case, we irrevocably lose the permission to write. In the second case, it is restored after the section needing read-only access is done. This "amplification" is sound as long as the context needing read permissions is not able to retain this permission for later use. For instance let! forbids the code using the variable nonlinearly from (among other things) returning a function, since a reference could be hidden in the closure. CL does not permit a closure to hold capabilities at all. Neither does it permit a capability variable to be used wherever its bound can be used. Assuming r^+ is a duplicable capability, CL permits the contraction rule on the left, but not the right:

$$\frac{r^+, r^+, \varPi \vdash s \Rightarrow \varPi'}{r^+, \varPi \vdash s \Rightarrow \varPi'} \text{ Ок} \qquad \frac{\epsilon \leq r^+ \qquad \epsilon, r^+, \varPi \vdash s \Rightarrow \varPi'}{\epsilon, \varPi \vdash s \Rightarrow \varPi'} \text{ NotOk}$$

A system that permits a linear capability to be treated non-linearly will need to use some rules that are at the surface-level, unmotivated. The problem is (speaking roughly) that if a permission is arbitrarily duplicable, then there is no way to determine if one has all the copies.

One is left with the unpalatable choice between conservatively surrendering write permission irrevocably, and conservatively restricting the type system.

1.2 Fractional Permissions

Our solution to this dilemma is to avoid non-linearity: read permissions are not duplicable. The major innovation of this paper is to show how one can

manufacture arbitrarily many read permissions without copying a permission: we *split* permissions. Each piece has a definite fraction and thus we can determine if all the pieces have been recovered and reconstruct the whole permission. This property is enabled by adding a single substructural rule:

$$\pi \equiv \varepsilon \pi, (1 - \varepsilon) \pi$$

where ε is some fraction between zero and one (exclusive) and π is a permission.

This solution gives a simple explanation of why writes conflict with reads and writes, but that reads do not conflict with each other: two pieces can co-exist but one cannot have the whole thing at the same time as another piece of it.

Consider a procedure requiring read permission to a cell and returning this read permission as well as read permission to some unknown cell. We don't have a "contraction" rule in this system and thus cannot take a read permission and convert it into two read permissions that are identical to the first. Instead if one wants to get two read permissions from one, one needs to split it into smaller permissions. Thus if the second result returned by the procedure is an alias of the parameter, it will be unable to return the entire read permission of the parameter; it will have to return a smaller fraction (which will at least appear as a different fraction than was delivered to the procedure). Even if the caller had write permission to the parameter before the call, afterwards, it will not be possible to reassemble an (unsafe) write permission.

On the other hand, if the second returned permission is *not* an alias of the parameter, it will be possible to return the same read permission that was passed, and the caller will be able to reassemble a write permission.

1.3 Contributions

The contribution of this work are as follows:

- We provide a way to check read-write effects with permissions—there is no need for a MayEqual analysis.
- We provide a new substructural rule for permissions-like systems to enable sharing of read-only state without needing to include non-linear permissions.
- We provide a way for a writable key to be temporarily made read-only while still being able to track all the copies, thus preventing unsoundness if a read permission is retained in some way.
- We present the idea in a simple language with aliasing, procedures and parallel computations. We give an operational semantics and define a permission type system (including simple existential return values) and prove soundness.
- We prove that checkable parallel constructs do not interfere: execution leads to deterministic results.

¹ In this paper, we define a fraction to be a real number. The proofs would also all work with rational fractions, or fractions with powers of two as denominators. Other non-numeric encodings are possible, with suitable changes to definitions and proofs.

Section 2 describes the simple language, gives an operational semantics and a permission type system. Then we prove the main result: that the type system ensures non-interference. The following section describes a variety of extensions made possible using fractional permissions. Section 4 reviews related work.

2 Types and Permissions

This section first describes the operational semantics of a simple language with pointers to cells containing integers. Then it describes the permission type system that can be used to check non-interference. We prove that this check of non-interference permits the execution of two pieces of code to be interleaved.²

2.1 Operational Semantics

The language used to demonstrate the permissions system is a simple language where source level global variables may point to allocated cells which hold integers. We have a (finite) set of variables V, an infinite set of (cell) locations³ L and a set of memories (stores) M which map variables to locations and some (finite subset of) locations to integers (\mathbb{Z}):

source vars
$$v \in V$$
 locations $l \in L$ memory $(\mu_V, \mu_L) = \mu \in M = (V \to L) \times (L \to \mathbb{Z})$ where $\operatorname{Dom} \mu_L \supseteq \mu_V(V)$

where \rightharpoonup denotes a partial finite function. The side condition ensures that a memory does not have dangling pointers (here $\mu_V(V) = \{\mu_V v \mid v \in V\}$). We permit μ to apply directly to variables and locations. Thus if $\mu = (\mu_V, \mu_L)$ then $\mu(\mu v)$ is short for $\mu_L(\mu_V v)$. The notation $\mu[v \mapsto l]$ updates the pointer stored in a variable and $\mu[l \mapsto i]$ updates the integer stored in a cell. We write $\mu_1 \sim \mu_2$ to mean that two memories are isomorphic.⁴

A program consists of a (finite) set of procedures. Each procedure has a statement for its body, and thus a program is represented by a map from each procedure to a statement. (For simplicity, there are no local variables.)

$$\begin{array}{ll} \text{procedures } p \in P \\ \text{programs} & g \in G = P \rightarrow S \end{array}$$

One can allocate a cell, copy pointers or update cell contents. We have sequential (;) and parallel (||) composition, as well as conditionals and procedure calls:

$$\begin{array}{c} \mathrm{statements} \; S \ni s ::= v \colon = \mathrm{new} \; | \; v \colon = v' \; | \; *v \colon = e \; | \; \mathrm{skip} \\ | \; \; s \; ; \; s' \; | \; s \; | \; | \; \mathrm{if} \; b \; \mathrm{then} \, s \; \mathrm{else} \, s' \; | \; \mathrm{call} \; p \\ \end{array}$$

² This paper has a separate appendix which contains the lemmas and proofs. URL: http://www.cs.uwm.edu/faculty/boyland/papers/permissions-appendix.ps

 $^{^{3}}$ In this paper, we do not model the possibility of running out of heap storage.

⁴ The precise definition of isomorphicity is given in the appendix.

Integer expressions include literals, additions and dereferencing of variables. Boolean expressions permit pointer comparison or comparison with zero.

```
integer expressions e := n \mid e+e \mid *v boolean expressions b := \texttt{true} \mid \texttt{false} \mid v==v' \mid e!=0
```

Figure 1 gives a small-step semantics for statements and expressions. A pair $\langle \mu, x \rangle$ where μ is a memory and x is a statement, an integer expression or a boolean expression is rewritten as a new pair for one step of evaluation. The rewriting for statements is subscripted with a program g because statements may include procedure calls and a program maps procedure names to bodies.

The evaluation of **new** statements stores 0 into the cell to prevent a dangling pointer. None of the other rules can introduce dangling pointers either. The evaluation of parallel compositions is nondeterministic: either branch may be evaluated one step further. The parallel composition can be eliminated once both branches are done. The lack of dangling pointers means that evaluation cannot get stuck.

Example Two different runs of the same parallel composition may yield different results. Consider the example in Fig. 2. If we evaluate it in a memory where v4 points to the same cell as v1 or v2, nondeterminism could lead to different results. For example, consider $\mu(v1) = \mu(v4) = l_1$, $\mu(v2) = \mu(v3) = l_2$, $\mu(l_1) = \mu(l_2) = 1$. If the left part is fully evaluated before the right, then at the end *v1 will be 4. If the right part is fully evaluated before the left, the end result of *v1 will be 1. However, in other memories, nondeterminism in execution leads to the same result. For instance, if all of the variables point to different cells, then the execution of the two parts can be interleaved arbitrarily without affecting the final result.

If we use an effect system to check interference, it first notices that neither part writes a variable the other reads (a simple matter of matching names), and thus the two parts only interfere if $MayEqual(v1_1, v3_2) \vee MayEqual(v1_1, v3_2) \vee MayEqual(v2_1, v3_2)$ where the subscripts refer to the occurrences of the variables. A precise answer will require an alias analysis smart enough to determine that $v3_2 = v4_1$ and such. An effect system simply does not provide sufficient information on its own to determine interference. Reynolds' original syntactic control of interference has the same difficulty with this example.

On the other hand, BI-logic or separation logic will fail to determine noninterference since both parts need to access the cell pointed to by v2. These logics do not distinguish shared reading from shared writing, and thus cannot determine that the sharing in this case is safe. (The sharing of v4 is non-problematic since only the heap is partitioned.)

O'Hearn et al's SCIR *can* handle this example by temporarily making v4 and *v2 read-only, but this solution is not sound in the presense of existentials as seen in the discussion of Fig. 5.

If one were to define noninterference using the Walker et al's capability language (not its intended purpose), one encounters a problem with *v2. If the

$$\frac{l \notin \operatorname{Dom} \mu_L}{\langle \mu, v \colon = \operatorname{new} \rangle \to_g \langle \mu[v \mapsto l, l \mapsto 0], \operatorname{skip} \rangle} \qquad \langle \mu, v \colon = v' \rangle \to_g \langle \mu[v \mapsto \mu v'], \operatorname{skip} \rangle$$

$$\frac{\langle \mu, e \rangle \to \langle \mu, e' \rangle}{\langle \mu, *v \colon = e \rangle \to_g \langle \mu, *v \colon = e' \rangle} \qquad \langle \mu, *v \colon = i \rangle \to_g \langle \mu[\mu v \mapsto i], \operatorname{skip} \rangle$$

$$\frac{\langle \mu, s_1 \rangle \to_g \langle \mu', s_1' \rangle}{\langle \mu, s_1 \colon s_2 \rangle \to_g \langle \mu', s_1' \rangle} \qquad \langle \mu, \operatorname{skip} \colon s \rangle \to_g \langle \mu, s \rangle \qquad \frac{\langle \mu, s_1 \rangle \to_g \langle \mu', s_1' \rangle}{\langle \mu, s_1 \mid s_2 \rangle \to_g \langle \mu', s_1' \mid s_2 \rangle}$$

$$\frac{\langle \mu, s_2 \rangle \to_g \langle \mu', s_2' \rangle}{\langle \mu, s_1 \mid s_2 \rangle \to_g \langle \mu', s_1 \mid s_2' \rangle} \qquad \langle \mu, \operatorname{skip} \mid \operatorname{lskip} \rangle \to_g \langle \mu, \operatorname{skip} \rangle$$

$$\langle \mu, \operatorname{if true then } s_1 \operatorname{else } s_2 \rangle \to_g \langle \mu, s_1 \rangle$$

$$\langle \mu, \operatorname{if false then } s_1 \operatorname{else } s_2 \rangle \to_g \langle \mu, s_2 \rangle$$

$$\frac{\langle \mu, b \rangle \to \langle \mu, b' \rangle}{\langle \mu, \operatorname{if } b \operatorname{then } s_1 \operatorname{else } s_2 \rangle \to_g \langle \mu, \operatorname{if } b' \operatorname{then } s_1 \operatorname{else } s_2 \rangle}$$

$$\langle \mu, \operatorname{call } p \rangle \to_g \langle \mu, g p \rangle \qquad \frac{\langle \mu, e_1 \rangle \to \langle \mu, e_1' \rangle}{\langle \mu, e_1 + e_2 \rangle \to \langle \mu, e_1' + e_2 \rangle} \qquad \frac{\langle \mu, e_2 \rangle \to \langle \mu, e_2' \rangle}{\langle \mu, n_1 + e_2 \rangle \to \langle \mu, n_1 + e_2' \rangle}$$

$$\langle \mu, \operatorname{n1+n2} \rangle \to \langle \mu, n_1 + n_2 \rangle \qquad \langle \mu, *v \rangle \to \langle \mu, \mu(\mu v) \rangle \qquad \langle \mu, v \coloneqq v' \rangle \to \langle \mu, \mu v = \mu v' \rangle$$

$$\frac{\langle \mu, e \rangle \to \langle \mu, e' \rangle}{\langle \mu, e_1 \Vdash 0 \rangle \to \langle \mu, e' \vdash 0 \rangle} \qquad \langle \mu, i \vdash 0 \rangle \to \langle \mu, i \neq 0 \rangle$$

Fig. 1. Evaluation

Fig. 2. Example program.

permission to read this value is represented by a duplicable capability r^+ , then there is no difficulty, but then write permission to *v2 can never be recovered. On the other hand, if we have write permission in the form of a unique capability r^1 , it does not seem to be possible to check noninterference in the example here without irreversibly downgrading this permission. One can prove $r^1 < \{r^+, r^+\}$ and use this with bounded quantification, but a capability such as ϵ cannot be split into two pieces, even if we know $\epsilon < \{r^+, r^+\}$, without destroying it. A split (similar to what occurs in SCIR) would result in unsoundness because the capability language has existential return types (in the guise of polymorphic continuations).

In the following section, we define a permission type system that can handle such examples. Well-typed statements always have deterministic results.

2.2 Permission Types

We follow Smith, Walker and Morrisett [17] in using a singleton type $ptr(\rho)$ to type pointer variables containing the pointer to location ρ . (The permission type system uses location variables in all places rather than actual locations.)

location var
$$\rho \in R$$

We have two kinds of base permissions: one to permit reading/writing of a source-level variable v (and also to give its type) and one to permit reading/writing the integer in a cell:

base permission
$$\beta := v : ptr(\rho) \mid \rho$$

We do not use fraction constants, but make use of fraction variables which represent some fraction between zero and one, exclusive:

fraction var
$$z \in Z$$

Base permissions can be "multiplied" by fractions. Syntactically we distinguish between fractions that may be complete (ξ) from ones that are strictly between zero and one (ε) .

$$\begin{array}{ll} \text{permission} & \pi & ::= \xi \beta \\ \text{fraction} & \xi & ::= 1 \mid \varepsilon \\ \text{partial fraction} \ \mathcal{E} \ni \varepsilon ::= z \mid 1 - \varepsilon \mid \varepsilon \varepsilon \end{array}$$

A complete fraction permits writing (as well as reading), but a partial fraction permits only reading.

A statement is permission-checked in an environment E consisting of a set Δ of free location and fraction variables, and a "set" of permissions Π :

environment
$$E ::= \Delta; \Pi$$

context $\Delta \subseteq R \cup Z$
permissions $\Pi ::= \cdot \mid \pi \mid \Pi, \Pi$

We have three simple sub-structural rules on permission "sets":

$$\cdot, \Pi \equiv \Pi$$
 $\Pi_1, \Pi_2 \equiv \Pi_2, \Pi_1$
 $\Pi_1, (\Pi_2, \Pi_3) \equiv (\Pi_1, \Pi_2), \Pi_3$

We have the permission splitting operation on a complete environment only to ensure we don't split a permission using an unbound variable:

$$\frac{\Delta \vdash \varepsilon \; \text{frac}}{\Delta; \varepsilon \pi, (1 - \varepsilon) \pi, \Pi \equiv \Delta; \pi, \Pi}$$

where we define $\varepsilon(\xi\beta) = (\varepsilon\xi)\beta$ with $\varepsilon 1 = \varepsilon$. We also have rules for fractions:

$$\varepsilon\varepsilon' \equiv \varepsilon'\varepsilon$$
$$\varepsilon(\varepsilon'\varepsilon'') \equiv (\varepsilon\varepsilon')\varepsilon''$$
$$(1 - (1 - \varepsilon)) \equiv \varepsilon$$

A procedure accepts a "set" of permissions and returns a "set" of permissions. It is polymorphic in a type context (the \forall scopes over the whole type) and returns existentially bound permissions. The program type maps procedures to types:

procedure type
$$A \ni \alpha ::= \forall \Delta.\Pi \to \exists \Delta.\Pi$$

program type $\omega \in \Omega = P \to A$

When we perform a call, we need to substitute actual partial fractions for fraction variables and actual location variables for location variables:

substitution
$$\sigma \in \Sigma = (R \to R) \times (Z \to \mathcal{E})$$

As with μ , we permit the pair to apply to either kind of variable, and we say $\text{Dom}(\sigma_R, \sigma_Z) = \text{Dom} \, \sigma_R \cup \text{Dom} \, \sigma_Z$. Application is extended to permissions and fractions, and to variables not in the domain:

$$\sigma \rho = \rho \text{ if } \rho \notin \text{Dom } \sigma_R \qquad \qquad \sigma z = z \text{ if } z \notin \text{Dom } \sigma_Z \qquad \qquad \sigma 1 = 1$$

$$\sigma(1 - \varepsilon) = 1 - \sigma \varepsilon \qquad \qquad \sigma(\varepsilon \varepsilon') = (\sigma \varepsilon)(\sigma \varepsilon') \qquad \qquad \sigma \cdot = \cdot$$

$$\sigma(v : \text{ptr}(\rho)) = v : \text{ptr}(\sigma \rho) \qquad \qquad \sigma \xi \beta = (\sigma \xi)(\sigma \beta) \qquad \sigma(\Pi, \Pi') = \sigma \Pi, \sigma \Pi'$$

Fig. 3 gives the rules for well-formedness of the various syntactic entities with respect to a set of location and fraction variables. Essentially well-formedness merely checks whether all variables are bound. Well-formedness of a substitution checks that the variables in the domain context are mapped to well-formed entities in the range context. Using this definition we extend substitution to complete environments: if $\vdash \sigma : \Delta \to \Delta'$ then $\sigma(\Delta''; \Pi'') = (\Delta' \cup (\Delta'' - \Delta); \sigma \Pi'')$.

Figure 4 gives the rules for permission-checking a program. Allocating a cell (New) requires write permission on the variable and gets write permission on

Fig. 3. Well-formedness rules

the new cell. The singleton types for variables permit the system to keep track of aliasing in the Copy rule. The permissions are "threaded" through both parts of a sequential composition but are split into two for a parallel composition and then recombined.

For if statements, the environment is sent to both branches and the resulting permissions may be different. Linearity prevents discarding permissions and thus there must exist two substitutions σ_1 and σ_2 that can be used to represent each branch's result as an instance of the unified result. At a call, we need two substitutions: one to determine what the actual locations and fractions will be and another to rename the existentially bound resulting variables. Permissions that are not needed in a procedure around the call are preserved. The corresponding rule Proc checks that it is possible to witness the existential variables.

Examples The example code previously shown in Fig. 2 can be permission-checked using

$$\Pi = 1$$
v1: ptr(ρ), zv2: ptr(ρ'), 1v3: ptr(ρ'), z'v4: ptr(ρ''), 1 ρ , 1 ρ' , 1 ρ''

(here the key ρ' needs to be divided between the two parallel parts because each needs read access) but not using

$$\Pi' = 1$$
v1 : ptr(ρ), zv2 : ptr(ρ'), 1v3 : ptr(ρ''), z'v4 : ptr(ρ'), 1 ρ , 1 ρ' , 1 ρ''

because the left part needs some fraction of ρ' but the right needs the whole key. This shows that the permissions system can check noninterference more precisely than BI-logic or separation logic, at least for examples such as this.

$$\frac{\rho \text{ fresh}}{\Delta; 1v : \text{ptr}(\rho'), \Pi \vdash_{\omega} v := \text{new} \Rightarrow \{\rho\} \cup \Delta; 1\rho, 1v : \text{ptr}(\rho), \Pi} \text{ New}}{\Delta; 1v : \text{ptr}(\rho), \xi v' : \text{ptr}(\rho'), \Pi \vdash_{\omega} v := v' \Rightarrow \Delta; 1v : \text{ptr}(\rho'), \xi v' : \text{ptr}(\rho'), \Pi} \text{ Copy}}$$

$$\frac{E = (\Delta; \xi v : \text{ptr}(\rho), 1\rho, \Pi') \qquad E \vdash e : \text{Int}}{E \vdash_{\omega} * v := e \Rightarrow E} \qquad \text{Update} \qquad \frac{E \vdash_{\omega} \text{ skip} \Rightarrow E}{E \vdash_{\omega} \text{ ship} \Rightarrow E} \text{ Skip}}$$

$$\frac{E \vdash_{\omega} s_1 \Rightarrow E' \qquad E' \vdash_{\omega} s_2 \Rightarrow E''}{E \vdash_{\omega} s_1; \ s_2 \Rightarrow E''} \text{ Seq}}$$

$$\frac{\Delta; \Pi_1 \vdash_{\omega} s_1 \Rightarrow \Delta'_1; \Pi'_1 \qquad \Delta; \Pi_2 \vdash_{\omega} s_2 \Rightarrow \Delta'_2; \Pi'_2}{\Delta; \Pi_1, \Pi_2 \vdash_{\omega} s_1 \mid |s_2 \Rightarrow \Delta'_1 \cup \Delta'_2; \Pi'_1, \Pi'_2} \text{ Park}}$$

$$\frac{\Delta; \Pi \vdash_{b} : \text{Bool}}{\Delta; \Pi \vdash_{\omega} s_1 \mid |s_2 \Rightarrow \Delta'_1 \cup \Delta'_2; \Pi'_1, \Pi'_2} \text{ Park}}$$

$$\frac{\Delta; \Pi \vdash_{b} : \text{Bool}}{\Delta; \Pi \vdash_{\omega} s_1 \mid |s_2 \Rightarrow \Delta'_1 \cup \Delta'_2; \Pi'_1, \Pi'_2} \text{ Park}}$$

$$\frac{\Delta_3 \text{ fresh}}{\Delta \cup \Delta_3 \vdash_{\omega} s_1 \mid |s_3 \Rightarrow \Delta_1; \sigma_1 \Pi_3} \qquad \Delta; \Pi \vdash_{\omega} s_2 \Rightarrow \Delta_2; \sigma_2 \Pi_3}{\Delta; \Pi \vdash_{\omega} s_1 \mid |s_1 \Rightarrow \Delta_1; \sigma_1 \Pi_3} \qquad \Delta; \Pi \vdash_{\omega} s_2 \Rightarrow \Delta_2; \sigma_2 \Pi_3} \text{ If}}$$

$$\frac{\Delta_3 \text{ fresh}}{\Delta; \Pi \vdash_{\omega} \text{ if } b \text{ then } s_1 \text{ else } s_2 \Rightarrow \Delta \cup \Delta_3; \Pi_3} \text{ If}}{\Delta; \Pi \vdash_{\omega} \text{ if } b \text{ then } s_1 \text{ else } s_2 \Rightarrow \Delta \cup \Delta_3; \Pi_3} \text{ If}} \text{ If}}$$

$$\frac{\omega p = \forall \Delta_1. \Pi_1 \rightarrow \exists \Delta_2. \sigma_2 \Pi_3}{\Delta; \sigma_1 \Pi_1, \Pi \vdash_{\omega} \text{ call } p \Rightarrow \Delta \cup \Delta_3; \sigma_1 \Pi_3, \Pi} \text{ Call}}{\Delta; \sigma_1 \Pi_1, \Pi \vdash_{\omega} \text{ call } p \Rightarrow \Delta \cup \Delta_3; \sigma_1 \Pi_3, \Pi}} \text{ Call}}$$

$$\frac{\omega p = \forall \Delta_1. \Pi_1 \rightarrow \exists \Delta_2. \sigma_2 \Pi_3}{\Delta; \sigma_1 \Pi_1, \Pi \vdash_{\omega} \text{ call } p \Rightarrow \Delta \cup \Delta_3; \sigma_1 \Pi_3, \Pi} \text{ Coll}}{E \vdash_{e} : \text{loc}} \text{ Bool}} \text{ Bool}}$$

$$\frac{E \vdash_{e} : \text{loc}}{\Delta; \Pi \vdash_{v} : \text{posite}} \text{ Plus}}{\Delta; \Pi \vdash_{v} : \text{loc}} \text{ Plus}} \frac{E \vdash_{e} : \text{Int}}{E \vdash_{e} : \text{loc}} \text{ Plus}}{E \vdash_{e} : \text{loc}} \text{ Plus}} \frac{H = \xi v : \text{ptr}(\rho), \xi'\rho, \Pi'}{\Delta; \Pi \vdash_{w} : \text{s} \Rightarrow \Delta'_1; \sigma \Pi_2} \text{ Plus}}$$

$$\frac{\Delta_1; \Pi_1 \vdash_{\omega} : \text{s} \Rightarrow \Delta'_1; \sigma \Pi_2}{\Delta; \Pi \vdash_{w} : \text{loc}} \text{ Plus}} \frac{H = \xi v : \text{ptr}(\rho), \xi'\rho, \Pi'}{\Delta; \Pi \vdash_{w} : \text{loc}} \text{ Plus}}$$

$$\frac{\Delta_1; \Pi_1 \vdash_{\omega} : \text{s} \Rightarrow \Delta'_1; \sigma \Pi_2}{\Delta; \Pi \vdash_{w} : \text{loc}} \text{ Plus}} \frac{H = \xi v : \text{ptr}(\rho), \xi'\rho, \Pi'}{\Delta; \Pi \vdash_{w} : \text{loc}} \text{ Plus}}$$

$$\frac{E \vdash_{\omega} : \text{loc}}{\Delta; \Pi \vdash_{w} : \text{loc}} \text{ Plus}} \frac{H = \xi v : \text{ptr}(\rho), \xi'\rho, \Pi'}{\Delta; \Pi \vdash_{w} : \text{loc}} \text{ Plus}} \frac{H = \xi v : \text{ptr}(\rho), \xi'\rho, \Pi$$

Fig. 4. Permission-Checking a Program

```
\begin{split} g \, \texttt{alias} &= \texttt{v2} \, := \, \texttt{v1} \\ g \, \texttt{noalias} &= \texttt{v2} \, := \, \texttt{new} \\ \\ \alpha_1 &= \, \forall \big\{ \rho, \rho', z, z' \big\}.z \\ \texttt{v1} : \operatorname{ptr}(\rho), 1 \\ \texttt{v2} : \operatorname{ptr}(\rho'), z' \rho \\ &\rightarrow \exists \big\{ \rho'', y, y', y'' \big\}.z \\ \texttt{v1} : \operatorname{ptr}(\rho), 1 \\ \texttt{v2} : \operatorname{ptr}(\rho''), y \rho, y' \rho'', y'' \rho'' \\ \alpha_2 &= \, \forall \big\{ \rho, \rho', z, z' \big\}.z \\ \texttt{v1} : \operatorname{ptr}(\rho), 1 \\ \texttt{v2} : \operatorname{ptr}(\rho'), z' \rho \\ &\rightarrow \exists \big\{ \rho'', y, y', y'' \big\}.z \\ \texttt{v1} : \operatorname{ptr}(\rho), 1 \\ \texttt{v2} : \operatorname{ptr}(\rho''), z' \rho, y' \rho'', y'' \rho'' \end{split}
```

Fig. 5. Two simple procedures and their types

The next example, Fig. 5, gives the code for two procedures: one that returns (v2) an alias of its parameter (v1); and one that returns a new cell. The procedure alias has the first type α_1 , but the second procedure noalias has both types α_1 and α_2 . Both procedure types α_1 and α_2 require permission to read v1 and write v2 and to read the cell that v1 points to. Both procedure types also say that the read permission for v1 and the write permission for v2 are returned to the caller as is read permission for the cell pointed to by (the presumably changed) pointer in v2.⁵ The two procedure types differ only in what fraction is returned of the read permission for the cell pointed to by v1. The first procedure type does not specify how "much" is returned; the new fraction is bound existentially. The second procedure type specifies that the same fraction coming in is returned. If one has write permission to the cell pointed to by v1 and calls a procedure with type α_2 , one can recover write permission after the procedure is complete, but if the callee has type α_1 , one cannot.

In Walker et al's CL, one can formulate corresponding procedure types that either permit or forbid aliasing in the result region/value. Polymorphic continuations gives roughly the same power as existential return types.

In an effects system, the procedures would not need to have read permission on the cell pointed to by v1 because it is not accessed; the types would be much simpler, but then this information must be recovered by the alias analysis used to answer MayEqual questions. A similar situation occurs with separation logic or BI-logic: information about the heap is not needed to type either procedure but any potential aliasing after the procedure is finished is not described.

Recall that in SCIR, a writeable variable may be given a read-only type temporarily. This rule is not sound if we add the ability to pack a copy of a read-only permission in an existential (as in procedure type α_1), and to unpack it later. After the write permission is recovered, the read-only permission could be mistakenly seen as not interfered with. In our system, in contrast, even read-

⁵ The reason why the procedures need to return two permissions to ρ'' is because a fraction variable can never refer to 1. If fraction variables could be one, then a fraction 1-z could be zero and render the permission type system unsound.

only permissions cannot be duplicated; if read-only permission is retained, the procedure is unable to return as "much" of the permission as it received from the caller, making it impossible to recover the write permission.

2.3 Consistency

In order to prove correctness of the type system we need to use a typing invariant with regard to a memory. A memory μ includes the values of pointer variables, but the type system introduces new variables: key variables and fraction variables. Let Ψ be mappings (partial function) from location and fraction variables to locations and numbers between zero and 1, respectively:

type variable map
$$\psi \in \Psi = (R \rightharpoonup L) \times (Z \rightharpoonup (0,1))$$

As with memories, we treat the pair of mappings as a single mapping with both types. We extend a mapping ψ to run on fraction expressions ($\psi \xi \in (0,1]$):

$$\psi 1 = 1$$

$$\psi(\varepsilon \varepsilon') = (\psi \varepsilon)(\psi \varepsilon')$$

$$\psi(1 - \varepsilon) = 1 - \psi \varepsilon$$

These rules ensure that ψ works the same on equivalent fractions: $\xi \equiv \xi' \Rightarrow \psi(\xi) = \psi(\xi')$. We further extend a mapping to apply to permissions. Now instead of getting a single value, we get a value for each variable or cell. Thus the result is a function from variables and locations to real numbers: $\psi \Pi : ((V \cup L) \to \mathbb{R})$. The function is made total by mapping all other $x \in V \cup L$ to zero:

$$\psi . = []$$

$$\psi(\xi v : ptr(\rho)) = [v \mapsto \psi \, \xi]$$

$$\psi(\xi \rho) = [\psi \, \rho \mapsto \psi \, \xi]$$

$$\psi(\Pi_1, \Pi_2) = (\psi \, \Pi_1) + (\psi \, \Pi_2)$$
where
$$(\psi \, \Pi_1 + \psi \, \Pi_2) \, x = (\psi \, \Pi_1) \, x + (\psi \, \Pi_2) \, x$$

The range of the result is syntactically only guaranteed to be nonnegative. A memory is not considered consistent with the environment unless the range includes only numbers between zero and one, inclusive. The ψ is also used to check that variables indeed have the location represented in their type, and that there are no dangling pointers.

$$\begin{array}{ccc} \underline{\mathrm{Dom}\,\psi=\Delta} & \mathrm{Rng}\,(\psi\,\varPi)\subseteq[0,1] & \psi;\mu\vdash\varPi \; \mathrm{consistent} \\ & \Delta;\varPi\vdash\mu\; \mathrm{ok} \\ \\ \psi;\mu\vdash\cdot\mathrm{consistent} & \frac{\psi;\mu\vdash\varPi_1\; \mathrm{consistent} & \psi;\mu\vdash\varPi_2\; \mathrm{consistent} \\ & \frac{\psi;\mu\vdash\varPi_1,\varPi_2\; \mathrm{consistent}}{\psi;\mu\vdash\varPi_1,\varPi_2\; \mathrm{consistent}} \\ \\ & \frac{\psi(\rho)=\mu(v)}{\psi;\mu\vdash\xi v: \mathrm{ptr}(\rho)\; \mathrm{consistent}} & \frac{\psi\;\rho\in\mathrm{Dom}\,\mu}{\psi;\mu\vdash\xi\rho\; \mathrm{consistent}} \end{array}$$

2.4 Non-interference

Non-interference in the checking of parallel composition permits us to prove a strong result: terminating evaluation always leads to an isomorphic store. In other words, the nondeterminism cannot affect the final result.

Theorem 1. If we have a well-typed program $g \ (\vdash g : \omega)$ and a statement s that permission-checks in an environment $E \ (E \vdash_{\omega} s \Rightarrow E')$ and a memory μ_1 that is consistent with the environment $(E \vdash \mu_1 \text{ ok})$, and s can be fully evaluated in this memory in k steps $(\langle \mu_1, s \rangle \xrightarrow{k}_g \langle \mu_1^*, \text{skip} \rangle)$ then for any isomorphic memory $\mu_2 \sim \mu_1$, any other evaluation sequence $\langle \mu_2, s \rangle \xrightarrow{g} \langle \mu_2', s' \rangle \xrightarrow{g} \dots$ terminates in exactly k steps and has an isomorphic result $\mu_2^* \sim \mu_1^*$.

2.5 Summary

We have taken a simple language with aliasing and explicit parallelism and have shown that fractional permissions give us a way to ensure determinism of execution through non-interference. The following section considers how this basic system can be lifted to more complex situations.

3 Extensions

This section explores further work made possible with fractional permissions.

Algorithmic Checking The permission checking system described in this paper is not algorithmic since the splitting required for a parallel composition is not deterministic. The solution is to permission-check the first branch with all the permissions, but keep track of which ones are actually needed. When only a fraction of a permission is needed, split it before recording the use of the fraction. After checking the first branch, check the second branch using the permissions that were not needed during the first branch.

Aliasing information The system here does not make use of pointer equality checking in if conditions. Equality is useful in order to connect a variable with an unknown pointer with a permission on an unknown cell. Inequality is useful in asserting uniqueness. To handle both situations, one can add a separate aliasing context that expresses known equalities and inequalities between location variables, and even logical connectives between these facts. For example if one knows that a variable z is equal to one of x and y, then if we have write permission for the cells pointed to by both x and y, then we also implicitly have permission for the cell pointed to by z. In general, one can use any three-valued logic [18] to hold this information. This information represents "facts" and not permissions and thus can be copied to both sides of a parallel composition.

Memory Management Adding garbage collection could be accomplished using a formulation similar to that of Morrisett, Felleisen and Harper [19]. Instead of the isomorphicity constraint, one would ensure that a final integer expression would have a value unchanged by garbage collection. We would also need a way to remove permissions to unreachable cells.

Adding explicit memory management (dispose) can be handled. Deallocation removes a key just as allocation introduces it. The usual semantics of dispose leaves dangling pointers in place, and the proof of determinism fails when dangling pointers exist: the sequence v1 := v2; dispose v2; new v2 may reallocate the same memory location for v1 or not, leading to non-isomorphic memories. There are at least two possible ways around this difficulty: (1) relax isomorphicity and then prevent dangling pointers from being compared to other pointers; or (2) change the semantics of dispose to work only in ways that do not leave dangling pointers. In the first solution, one needs permission to compare pointers, which corresponds to the I access right of BNR capabilities [20].

The permission system described here cannot check recursive procedures that allocate cells on their recursive path, since each allocation produces a new key which cannot be forgotten. This brings us to the next extension topic.

Records and recursive data types When we have singleton types for pointers, then record types and especially recursive data types require the use of existentials [21]. The existentials include not just bindings, but also permissions. This permits us to represent an unknown unique pointer: the complete permission stored with the pointer. Immutable pointers are represented by storing an existential fraction of the permission with the pointer.

Since the packed existential includes (linear) witness permissions, a variable with existential type cannot be read or written (that is, copying or destroying the value) until the existential is unpacked. Therefore a permission system needs to distinguish "open variables" (variables with singleton type) from "closed variables" (variables with existential type). Closed variables can be fractionally opened in which case only a fraction of the witness permissions are usable.

Adoption and Ownership Adoption involves logically storing a key inside another one. Adoption cannot be undone. In this way, it is similar to ownership. In adoption and focus, the adoptee can only be made accessible by temporarily making the adopter inaccessible. With fractions, one could access a fraction of the adoptee (and thus have read-only access) given only a fraction of the adopter.

A shared variable is modeled by adopting its complete permission into a globally accessible key. "Fractional adoption" permits the modeling of unique-write variables, variables that are globally read-only with write access at a single point. For such a variable, a known fraction is adopted by a globally accessible key and the remainder is kept at the write-access point. The two fractions can be put together to gain write access.

4 Related Work

Reynolds' "syntactic control of interference" [3] checked that call-by-name would not cause "covert interference" where a parameter and a procedure each observe the same changing state. This work was revisited by O'Hearn and others [16] (SCIR). SCIR split the context into two parts: an active part (writable); and a passive part (read only). The passive part can be duplicated in two branches of the proof (unlike the active part) enabling the sharing of read-only state. An interesting rule called "passification" enabled a write of a variable to be ignored if the result was a passive type. A monadic-like structure ensures that (visible) state mutations cannot be hidden in a passively-type result.

Reynolds and O'Hearn have continued analysis of mutable data structures using the logic of "bunched implications" [10] and "separation logic" [12]. A spatial conjunction operator in the logics allows parts of the heap to be analyzed separately. Allocation and deallocation add and remove spatial conjuncts. However, the spatial conjunction operator strictly separates heap access: it does not distinguish reads from writes. It appears that fractions could be applied, so that one could have $P \models \varepsilon P * (1 - \varepsilon)P$ and get the ability to share read-only heaps.

Walker, Crary and Morrisett's static capability system [9] inspired DeLine and Fähndrich's alias typing system for Vault [22], from which adoption and focus [23] grew. The capabilities or guards can be seen as permissions. With the "focus" operation, one temporarily gives up a guard in order to get unrestricted access to a unique variable. Once uniqueness is re-established the guard can be returned. This process is handled with a linear implication $h \to g$.

Effects systems have been used to check non-interference [4,5] but need to be augmented with MayEqual information to check for conflict. One simple (but conservative) analysis is to assume any two references with the same type (or compatible types) may be aliased.

In the area of compilers, non-interference is traditionally checked through using a data-flow graph (or some superset thereof). From early on, the interdependence between aliasing and data dependencies has been recognized. Traditionally, the alias information is presented in terms of may-alias (and must-alias) facts, pairs of aliases at program points. MayEqual, on the other hand, compares pointer expressions at disparate program points [13]. Ross and Sagiv [24] have show how data dependencies can be recovered from may-alias information by instrumenting the program (in a global transformation).

Rugina and Rinard give an algorithm for doing flow-sensitive pointer analysis in programs with structured parallelism [25]. It models interference by assuming that any mutation performed in one parallel branch may be visible at any time in other parallel branches. The analysis described here is simpler since interference between parallel branches is forbidden.

5 Conclusions

We define a permission type system which enables us to solve the interdependent problems of uniqueness and effects in a single formalism. We extend earlier

work on permissions to distinguish reads from writes using fractional permissions, rather than non-linearity. We define a simple language with aliasing and parallelism and show that well-typed programs have deterministic results.

Acknowledgments I thank Dave Clarke, Manuel Fähndrich and Bill Retert for helping me frame this idea and reading innumerable drafts. All remaining errors are strictly my own.

References

- Jouvelot, P., Gifford, D.K.: Algebraic reconstruction of types and effects. In: Conference Record of the Eighteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages. ACM Press, New York (1991) 303–310
- Talpin, J.P., Jouvelot, P.: Polymorphic type, region and effect inference. Journal of Functional Programming 2 (1992) 245–271
- Reynolds, J.C.: Syntactic control of interference. In: Conference Record of the Fifth ACM Symposium on Principles of Programming Languages, New York, ACM Press (1978) 39–46
- 4. 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
- Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA'02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications. Volume 37., New York, ACM Press (2002) 292–310
- Flanagan, C., Abadi, M.: Types for safe locking. In Swierstra, S.D., ed.: ESOP'99
 — Programming Languages and Systems, 8th European Symposium on Programming. Volume 1576 of Lecture Notes in Computer Science., Berlin, Heidelberg, New York, Springer (1999) 91–108
- Boyapati, C., Rinard, M.: A parameterized type system for race-free Java programs. In: OOPSLA'01 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications. Volume 36., New York, ACM Press (2001) 56–69
- 8. 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. Volume 37., New York, ACM Press (2002) 211–230
- Walker, D., Crary, K., Morrisett, G.: Typed memory management via static capabilities. ACM Transactions on Programming Languages and Systems 22 (2000) 701–771
- 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
- 11. Reynolds, J.C.: Intuitionistic reasoning about shared mutable data structure. In: Millenial Perspectives in Computer Science, Palgrave (to appear) Draft dated July 28, 2000.

- 12. 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
- 13. Boyland, J., Greenhouse, A.: MayEqual: A new alias question. Presented at IWAOOS '99: Intercontinental Workshop on Aliasing in Object-Oriented Systems. http://cuiwww.unige.ch/ecoopws/iwaoos/papers/papers/greenhouse.ps.gz (1999)
- Steensgaard, B.: Points-to analysis in almost linear time. In: Conference Record of the Twenty-third Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, New York, ACM Press (1996) 32–41
- 15. Wadler, P.: Linear types can change the world! In Broy, M., Jones, C.B., eds.: Programming Concepts and Methods. Elsevier, North-Holland (1990)
- O'Hearn, P.W., Takeyama, M., Power, A.J., Tennent, R.D.: Syntactic control of interference revisited. In: MFPS XI, conference on Mathematical Foundations of Program Semantics. Volume 1., Elsevier (1995)
- 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
- Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Conference Record of the Twenty-sixth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, New York, ACM Press (1999) 105–118
- Morrisett, G., Felleisen, M., Harper, R.: Abstract models of memory management.
 In: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95), New York, ACM Press (1995) 66–77
- Boyland, J., Noble, J., Retert, W.: Capabilities for sharing: A generalization of uniqueness and read-only. In Knudsen, J.L., ed.: ECOOP'01 — Object-Oriented Programming, 15th European Conference. Volume 2072 of Lecture Notes in Computer Science., Berlin, Heidelberg, New York, Springer (2001) 2–27
- 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
- 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. Volume 36., New York, ACM Press (2001) 59–69
- 23. Fähndrich, M., DeLine, R.: Adoption and focus: Practial linear types for imperative programming. In: Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation. Volume 37., New York, ACM Press (2002) 13–24
- 24. Ross, J.L., Sagiv, M.: Building a birdge between pointer aliases and program dependencies. In Hankin, C., ed.: ESOP'98 Programming Languages and Systems, 7th European Symposium on Programming. Volume 1381 of Lecture Notes in Computer Science., Berlin, Heidelberg, New York, Springer (1998) 221–235
- 25. Rugina, R., Rinard, M.C.: Pointer analysis for structured parallel programs. ACM Transactions on Programming Languages and Systems **25** (2003) 70–116

Technical matter for "Checking Interference with Fractional Permissions" (SAS 2003)

Definition 1. Two memories μ and μ' are isomorphic (written $\mu \sim \mu'$) if and only if a 1-1 and onto function φ exists on the set of locations ($\varphi l = \varphi l' \iff$ l = l') such that the following conditions hold:

- 1. For all $v \in V$, $\mu' v = \varphi(\mu v)$;
- 2. $\operatorname{Dom} \mu_L' = \varphi(\operatorname{Dom} \mu_L);$ 3. For all $l \in \operatorname{Dom} \mu_L, \mu_L'(\varphi l) = \mu_L l$

These conditions in particular imply that the stores have the same integers for variables: $\mu(\mu v) = \mu'(\mu' v)$.

Lemma 1. The relation \sim is an equivalence relation.

Proof. **reflexivity** Obvious (φ is the identity function).

symmetricity Given $\mu_1 \sim \mu_2$ with 1-1 function φ , the relation $\mu_2 \sim \mu_1$ can be proved using $\varphi' = \varphi^{-1}$:

- 1. $\mu_1 v = \varphi^{-1}(\varphi(\mu_1 v)) = \varphi^{-1}(\mu_2 v)$.
- 2. Dom $\mu_{1L} = \varphi^{-1}(\text{Dom }\mu_{2L})$.
- 3. Let $l \in \text{Dom } \mu_2$, then by the second condition, $l = \varphi l'$ for some $l' \in$ Dom μ_1 . Thus $\mu_1(\varphi^{-1}l) = \mu_1(l') = \mu_2(\varphi l') = \mu_2 l$.

transitivity Given $\mu_1 \sim \mu_2 \sim \mu_3$ with 1-1 functions φ_{12} and φ_{23} , $\mu_1 \sim \mu_3$ can be proven using $\varphi = \varphi_{12} \circ \varphi_{23}$, that is, $\varphi l = \varphi_{23}(\varphi_{12} l)$.

- 1. $\mu_3 v = \varphi_{23}(\mu_2 v) = \varphi_{23}(\varphi_{12}(\mu_1 v)) = \varphi v$.
- 2. $\operatorname{Dom} \mu_{3L} = \varphi_{23}(\operatorname{Dom} mu_{2L}) = \varphi_{23}(\varphi_{12}(\operatorname{Dom} \mu_{1L})) = \varphi(\operatorname{Dom} \mu_{1L}).$
- 3. $\mu_3 l = \mu_2(\varphi_{23} v) = \mu_1(\varphi_{12}(\varphi_{23} v)) = \mu_1(\varphi v)$.

Lemma 2. For any statement or expression x not skip, true, false or an integer constant, evaluation can proceed one step in any memory $\mu: \langle \mu, x \rangle \to_q$ $\langle \mu', x' \rangle$.

Proof. We prove by induction on x:

- v:=new Progress is immediate since μ_L is a finite map and L is infinite: x' =
- v:=v' Progress to skip is immediate since all memories are complete functions on V.
- *v := e If e isn't an integer constant, the result follows from the inductive hypothesis. Otherwise the result follows since μ_V is a complete function.
- $s_1; s_2$ If $s_1 = \text{skip}$, the result follows immediately, otherwise it follows by induction.
- $s_1 | | s_2$ If both $s_1 = s_2 = \text{skip}$, progress to skip is immediate. Otherwise the result follows by induction.
- if b then s_1 else s_2 If b is a boolean constant, progress to s_1 or s_2 is immediate. Otherwise the result follows by induction.
- call p Immediate, since evaluation is unconditional.

- e_1+e_2 If both e_1 and e_2 are integer constants, progress to an integer constant is immediate. Otherwise it follows by induction.
- *e Progress to an integer constant is immediate because memories have no dangling pointers: $\mu(\mu v)$ will be defined.

Progress irrespective of typing depends upon the lack of dangling pointers. If an extension wishes to permit dangling pointers, progress will only be possible in typed (permission-checked) programs.

Expression evaluation yields the same result in isomorphic memories:

Lemma 3. Suppose we have two isomorphic memories $\mu_1 \sim \mu_2$, then if $\langle \mu_1, e \rangle \rightarrow \langle \mu_1, e' \rangle$ then $\langle \mu_2, e \rangle \rightarrow \langle \mu_2, e'' \rangle$ and e' = e''. Similarly, if $\langle \mu_1, b \rangle \rightarrow \langle \mu_1, b' \rangle$, then $\langle \mu_2, b \rangle \rightarrow \langle \mu_2, b'' \rangle$, and b' = b''.

Proof. We prove by structural induction over the form of the expression. We prove for the following cases:

```
v==v' Here b'=(\mu_1\,v=\mu_1\,v'), and thus b''=(\mu_2\,v=\mu_2\,v')=(\varphi(\mu_1\,v)=\varphi(\mu_1\,v'))=(\mu_1\,v=\mu_1\,v')=b'.
```

i!=0 (Trivial since memory is not involved.)

e!=0 (Follows immediately using the inductive hypothesis.)

b (Boolean constants are trivial since the antecedent cannot be met.)

 n_1+n_2 (Trivial since memory is not used.)

 e_1+e_2 (Follows immediately using the inductive hypothesis.)

- *v Here $e' = \mu_1(\mu_1 v)$. Now $\mu_2 v = \varphi(\mu_1 v)$ and even if dangling pointers were permitted, we would still have $\mu_2 v \in \text{Dom } \mu_2$ since $\mu_1(\mu_1 v)$ is defined and thus $\mu_1 v \in \text{Dom } \mu_1$ and thus by the second condition for isomorphicity $\text{Dom } \mu_2 = \varphi(\text{Dom } \mu_1) \ni \varphi(\mu_1 v)$. Thus $\mu_2(\mu_2 v)$ is defined and by the third condition of isomorphicity $\mu_2(\mu_2 v) = \mu_2(\varphi(\mu_1 v)) = \mu_1(\mu_1 v) = e'$, and so $\langle \mu_2, e \rangle \to \langle \mu_2, e' \rangle$.
- n_1 (Trivial since the antecedent cannot be satisfied.)

If we use the same sequence of derivations when evaluating statements, isomorphicity is preserved:

Lemma 4. Suppose we have two isomorphic memories $\mu_1 \sim \mu_2$ and a statement s that can be evaluated in the first memory $(\langle \mu_1, s \rangle \to_g \langle \mu'_1, s' \rangle)$, then there exists an isomorphic resulting memory $(\mu'_2 \sim \mu'_1)$ for evaluation in the second memory $(\langle \mu_2, s \rangle \to_g \langle \mu'_2, s' \rangle)$.

Proof. We prove the result by structural induction over s. We perform a case analysis on the form of s:

New v The fact that evaluation can proceed in μ_2 is immediate since new statements evaluate to skip in one step. let l_1 and l_2 be the two new locations added in the respective evaluations. Then we must prove $(\mu'_1 = \mu_1[v \mapsto l_1, l_1 \mapsto 0]) \sim (\mu'_2 = \mu_2[v \mapsto l_2, l_2 \mapsto 0])$. If φ is the map that shows $\mu_1 \sim \mu_2$, then if $\varphi l_1 = l_2$, then let $\varphi' = \varphi$. Otherwise it must be $\varphi^{-1} l_2 \not\in \text{Dom } \mu_1$ (by the second condition of isomorphicity). Then let $\varphi' = \varphi[l_1 \mapsto l_2, \varphi^{-1} l_2 \mapsto \varphi l_1$. Checking the isomorphicity conditions:

- 1. $\mu'_2 v = l_2 = \varphi' l_1 = \varphi'(\mu'_1 v)$. If $v' \neq v$, then $\mu'_2 v' = \mu_2 v' = \varphi(\mu_1 v') = \varphi(\mu'_1 v')$ and since memories are not allowed to have dangling pointers $\mu'_1 v'$ must be in the domain of μ'_L and thus cannot be l_1 or $\varphi^{-1} l_2$ and so $\varphi(\mu'_1 v') = \varphi'(\mu'_1 v')$ satisfying the first isomorphicity condition.
- 2. $\operatorname{Dom} \mu_2' = \{l_2\} \cup \operatorname{Dom} \mu_2 = \{\varphi \, l_1\} \cup \varphi(\operatorname{Dom} \mu_1) = \varphi(\operatorname{Dom} \mu_1')$
- 3. Let $l \in \text{Dom } \mu_1'$. If $l = l_1$, then $\mu_2'(\varphi l) = \mu_2' l_2 = 0 = \mu_1' l$. Otherwise, $l \neq l_1$, and thus $\mu_2'(\varphi l) = \mu_2(\varphi l) = \mu_1 l = \mu_1' l$.
- v:=v' We need only verify that $\mu_1'=\mu_1[v\mapsto \mu_1\,v']\sim \mu_2'=\mu_2[v\mapsto \mu_2\,v']$. If v=v', this is trivial. Otherwise, we check the three isomorphicity conditions:
 - 1. For v itself, $\mu'_2 v = \mu_2 v' = \varphi(\mu_1 v') = \varphi(\mu'_1 v')$. For other $v'' \neq v$, $\mu'_2 v'' = \mu_2 v'' = \varphi(\mu_1 v'') = \varphi(\mu'_1 v'')$.
 - 2. The domains are unchanged so this condition is satisfied trivially.
 - 3. The mappings for locations are unchanged by evaluation of v:=v', and thus this condition is still met.
- *v:=e If e is an integer constant i, then s evaluates immediately to skip and we need only check that $\mu_1' = \mu_1[(\mu_1 \, v) \mapsto i] \sim \mu_2' = \mu_2[(\mu_2 \, v) \mapsto i]$
 - 1. The first condition depends only on the memory for variables and thus is unchanged.
 - 2. The second condition depends only on the domains for locations, and this is unaffected since $\mu_{?} v$ is not permitted to be a dangling pointer.
 - 3. Let $l = \mu_1 v$. Since $\mu_1 \sim \mu_2$, we have $\mu_2 v = \varphi l$. Thus $\mu'_2(\varphi l) = i = \mu'_1(l)$. For other Dom $\mu'_1 \ni l' \neq l$, $\mu'_2(\varphi l') = \mu_2(\varphi l) = \mu_1(l) = \mu'_1(l)$.

Otherwise, the result follows using Lemma 3.

- skip Trivial since the antecedent cannot be met.
- $s_1; s_2$ If $s_1 = \mathtt{skip}$, then s' must be s_2 and the memory is unchanged and thus the result is established immediately. Otherwise, we use the inductive hypothesis on s_1 to get s'_1 and $\mu'_1 \sim \mu'_2$. This gives us $\langle \mu_i, s_1; s_2 \rangle \to_g \langle \mu'_i, s'_1; s_2 \rangle$ and we are done.
- skip||skip In this case $\langle \mu_i, s \rangle \to_q \langle \mu_i, \text{skip} \rangle$ and we are done.
- $s_1 \mid \mid s_2$ Suppose $s' = s'_1 \mid \mid s_2$ where $\langle \mu_1, s_1 \rangle \to_g \langle \mu'_1, s'_1 \rangle$. Then by the inductive hypothesis, we have $\langle \mu_2, s_1 \rangle \to_g \langle \mu'_2, s'_1 \rangle$ with $\mu'_1 \sim \mu'_2$. From this follows immediately $\langle \mu_2, s_1 \mid \mid s_2 \rangle \to_g \langle \mu'_2, s'_1 \mid \mid s_2 \rangle$ and we are done. Otherwise if the right part is evaluated first, analogous reasoning applies.
- if true then s_1 else s_2 The result follows immediately since s' must be s_1 and $\mu'_1 = \mu_1$. And similarly for code
- if b then s_1 else s_2 where $\langle \mu_1, b \rangle \to \langle \mu_1, b' \rangle$. The result follows immediately using Lemma 3.
- call p The result follows immediately since $\langle \mu_i, \text{call } p \rangle \to_g \langle \mu_i, g p \rangle$ with unchanged memories.

The lack of dangling pointers is crucial to the proof of this lemma: if in the memories some variable v has a dangling pointer: $\mu_i v \notin \text{Dom } \mu_i$, then the statement v':=new for some other variable v' could perhaps result in v and v' being aliases, or perhaps not, breaking isomorphicity.

Definition 2. The parallel composition (written $\sigma \uplus \sigma'$) of two substitutions on disjoint domains, and the sequential composition (written $\sigma \circ \sigma'$) are defined as follows:

$$(\sigma \uplus \sigma') x = \begin{cases} \sigma x & x \in \text{Dom } \sigma \\ \sigma' x & x \in \text{Dom } \sigma' \end{cases}$$
$$(\sigma \circ \sigma') x = \sigma'(\sigma x)$$

Lemma 5. Given $\Delta' \supseteq \Delta$, then

$$\Delta \vdash \rho \text{ loc-var} \Rightarrow \Delta' \vdash \rho \text{ loc-var}$$

$$\Delta \vdash \beta \text{ base-perm} \Rightarrow \Delta' \vdash \beta \text{ base-perm}$$

$$\Delta \vdash \xi \text{ frac} \Rightarrow \Delta' \vdash \xi \text{ frac}$$

$$\Delta \vdash \Pi \text{ perms} \Rightarrow \Delta' \vdash \Pi \text{ perms}$$

Proof. Straightforward structural induction.

Well-formedness is preserved under application by well-formed substitutions:

Lemma 6. Given $\vdash \sigma : \Delta \to \Delta'$ and $\Delta \cup \Delta'' \vdash \Pi$ perms then $\Delta \cup \Delta' \vdash \sigma \Pi$ perms.

Proof. Straightforward structural induction.

Composition preserves well-formedness under straightforward conditions:

Lemma 7. Let
$$\vdash \sigma_1 : \Delta_1 \to \Delta_1'$$
 and $\vdash \sigma_2 : \Delta_2 \to \Delta_2'$. Then
$$- \text{ If } \Delta_1 \cap \Delta_2 = \emptyset \text{ then } \vdash \sigma_1 \uplus \sigma_2 : (\Delta_1 \cup \Delta_2) \to (\Delta_1' \cup \Delta_2')$$

$$- \text{ If } \Delta_1' \subseteq \Delta_2 \text{ then } \vdash \sigma_1 \circ \sigma_2 : \Delta_1 \to \Delta_2'$$

Proof. Straightforward proof by induction.

We now prove several technical lemmas that apply to proof trees using the rules in Figure 4. First one that ensures we have we don't lose bindings and thus maintain a well-formed environment:

Lemma 8. If we can permission-check s in environment Δ ; Π yielding a resulting environment Δ' ; Π' (Δ ; $\Pi \vdash_{\omega} s \Rightarrow \Delta'$; Π'), and the first environment was well-formed ($\Delta \vdash \Pi$ perms) then the new bindings $\Delta' - \Delta$ are all fresh, no bindings are lost ($\Delta \subseteq \Delta'$) and the resulting environment is well-formed ($\Delta' \vdash \Pi'$ perms).

Proof. Straightforward structural induction.

We also have a weakening lemma:

Lemma 9. If we can permission-check s in environment Δ ; Π yielding a resulting environment Δ' ; Π' (Δ ; $\Pi \vdash_{\omega} s \Rightarrow \Delta'$; Π'), then we can permission-check s in an environment with more variables and more permissions ($\Delta \cup \Delta_e$; Π , $\Pi_e \vdash_{\omega} s \Rightarrow \Delta' \cup \Delta_e$; Π' , Π_e).

Proof. We prove the transformation using induction over the height of the proof tree. We perform a case analysis on the rule at the case of the tree:

New, Copy, Update, Skip, Call The extra variables and permissions pass through instances of any of these rules unchanged.

SEQ Assume the proof tree has the form:

$$\frac{\vdots}{\Delta; \Pi \vdash_{\omega} s_{1} \Rightarrow \Delta'; \Pi'} \stackrel{?_{1}}{\longrightarrow} \frac{\vdots}{\Delta'; \Pi' \vdash_{\omega} s_{2} \Rightarrow \Delta''; \Pi''} \stackrel{?_{2}}{\longrightarrow} \frac{\Delta}{\Im} \stackrel{?_{2}}{\longrightarrow} \frac{\Delta}{\Im} \stackrel{?_{1}}{\longrightarrow} \frac{\Delta}{\Im} \stackrel{?_{2}}{\longrightarrow} \frac{\Delta}{\Im} \stackrel{?_$$

By induction, we can push extra variables $\Delta_{\rm e}$ and permissions $\Pi_{\rm e}$ into both branches, and form the tree:

$$\vdots$$

$$\overline{\Delta \cup \Delta_{\mathbf{e}}; \Pi, \Pi_{\mathbf{e}} \vdash_{\omega} s_{1} \Rightarrow \Delta' \cup \Delta_{\mathbf{e}}; \Pi', \Pi_{\mathbf{e}}} ?_{1}$$

$$\vdots$$

$$\overline{\Delta' \cup \Delta_{\mathbf{e}}; \Pi', \Pi_{\mathbf{e}} \vdash_{\omega} s_{2} \Rightarrow \Delta'' \cup \Delta_{\mathbf{e}}; \Pi'', \Pi_{\mathbf{e}}} ?_{2}$$

$$\overline{\Delta \cup \Delta_{\mathbf{e}}; \Pi, \Pi_{\mathbf{e}} \vdash_{\omega} s_{1}; s_{2} \Rightarrow \Delta'' \cup \Delta_{\mathbf{e}}; \Pi'', \Pi_{\mathbf{e}}}$$
SEQ

PAR Assume the proof tree has the form:

$$\frac{\vdots}{\Delta; \Pi_1 \vdash_{\omega} s_1 \Rightarrow \Delta'_1; \Pi'_1} \stackrel{?_1}{\longrightarrow} \frac{\vdots}{\Delta; \Pi_2 \vdash_{\omega} s_2 \Rightarrow \Delta'_2; \Pi'_2} \stackrel{?_2}{\longrightarrow} \frac{\Delta; \Pi_1 \vdash_{\omega} s_1 \mid |s_2 \Rightarrow \Delta'_1 \cup \Delta'_2; \Pi'_1, \Pi'_2}$$
PAR

We perform a similar transformation but only into the one branch, forming the tree:

$$\frac{\vdots}{ \Delta; \Pi_1 \vdash_{\omega} s_1 \Rightarrow \Delta_1'; \Pi_1'} \stackrel{?_1}{} \frac{\vdots}{ \Delta \cup \Delta_{\mathbf{e}}; \Pi_2, \Pi_{\mathbf{e}} \vdash_{\omega} s_2 \Rightarrow \Delta_2' \cup \Delta_{\mathbf{e}}; \Pi_2', \Pi_{\mathbf{e}}} \stackrel{?_2}{} \frac{}{}_{\mathrm{PAR}} \frac{}{ \Delta \cup \Delta_{\mathbf{e}}; \Pi_1, \Pi_2, \Pi_{\mathbf{e}} \vdash_{\omega} s_1 \mid s_2 \Rightarrow \Delta_1' \cup \Delta_2' \cup \Delta_{\mathbf{e}}; \Pi_1', \Pi_2', \Pi_{\mathbf{e}}} \frac{}{}_{\mathrm{PAR}}$$

IF The condition can be checked with extra variables or permissions, and by induction, the extra variables and permissions can be passed through each branch. Finally the substitutions $\vdash \sigma_i : \Delta_3 \to \Delta$ can be trivially considered to be of type $\Delta_3 \to (\Delta \cup \Delta_e)$. Since $Delta_3$ are fresh, we have $(\sigma_i \Pi_3, \Pi_e) = \sigma_i(\Pi_3, \Pi_e)$.

We also have a substitution lemma:

Lemma 10. If we have a well-typed program $\vdash g : \omega$ and a well-typed statement Δ ; $\Pi \vdash_{\omega} s \Rightarrow \Delta'$; Π' and a substitution $\vdash \sigma : \Delta_1 \to \Delta_2$, and $\Delta_1 \subseteq \Delta$, then the statement can be typed in the substituted environment: $\Delta_2 \cup (\Delta - \Delta_1)$; $\sigma \Pi \vdash_{\omega} s \Rightarrow \Delta_2 \cup (\Delta' - \Delta_1)$; $\sigma \Pi'$, that is, $\sigma(\Delta; \Pi) \vdash_{\omega} s \Rightarrow \sigma(\Delta'; \Pi')$

Proof. The type context part of the result is clear because of Lemma 6. We prove the remainder by induction over all kinds of rules in the proof tree (including those for boolean and integer expressions). We use a case analysis on the rule applied at the root:

New

$$\frac{\rho \text{ fresh}}{\Delta; 1v: \operatorname{ptr}(\rho'), \Pi_1 \vdash_{\omega} v\colon =\operatorname{\mathtt{new}} \Rightarrow \{\rho\} \cup \Delta; 1\rho, 1v: \operatorname{ptr}(\rho), \Pi_1} \text{ New}$$

Here $\sigma \Pi = 1v : \operatorname{ptr}(\sigma \rho'), \sigma \Pi_1$. These permissions permit us to use NEW to type the statement with output permissions $1\rho, 1v : \operatorname{ptr}(\rho), \sigma \Pi_1 = \sigma \Pi'$ since ρ cannot be in the domain of σ since it is fresh. COPY

$$\frac{\Delta; 1v : \operatorname{ptr}(\rho), \xi v' : \operatorname{ptr}(\rho'), \Pi_1 \vdash_{\omega} v : = v' \Rightarrow \Delta; 1v : \operatorname{ptr}(\rho'), \xi v' : \operatorname{ptr}(\rho'), \Pi_1}{\text{Copy}}$$

Here $\sigma \Pi = 1v : \operatorname{ptr}(\sigma \rho), (\sigma \xi)v' : \operatorname{ptr}(\sigma \rho'), \sigma \Pi_1$ which permits us to apply Copy to get output permissions $1v : \operatorname{ptr}(\sigma \rho'), (\sigma \xi)v' : \operatorname{ptr}(\sigma \rho'), \sigma \Pi_1$ that is $\sigma \Pi'$. UPDATE

$$\frac{\Delta; \xi v : \operatorname{ptr}(\rho), 1\rho, \Pi_1 \vdash e : \operatorname{Int}}{\Delta; \xi v : \operatorname{ptr}(\rho), 1\rho, \Pi_1 \vdash_{\omega} *v : =e \Rightarrow \Delta; \xi v : \operatorname{ptr}(\rho), 1\rho, \Pi_1} \text{ Update}$$

Here $\sigma \Pi = (\sigma frac)v : ptr(\sigma \rho), 1\sigma rho$ where permits us to apply UPDATE to get the same permissions out.

Skip Trivial.

SEQ Follows by induction.

PAR By induction.

IF

$$\frac{\varDelta; \varPi \vdash b : \mathrm{Bool} \quad \varDelta; \varPi \vdash_{\omega} s_{1} \Rightarrow \varDelta'_{1}; \sigma_{1}\,\varPi_{3} \quad \varDelta; \varPi \vdash_{\omega} s_{2} \Rightarrow \varDelta'_{2}; \sigma_{2}\,\varPi_{3}}{\Delta_{3} \text{ fresh } \quad \varDelta \cup \varDelta_{3} \vdash \varPi_{3} \text{ perms} \quad \vdash \sigma_{1} : \varDelta_{3} \to \varDelta'_{1} \quad \vdash \sigma_{2} : \varDelta_{3} \to \varDelta'_{2}}{E \vdash_{\omega} \text{ if } b \text{ then } s_{1} \text{ else } s_{2} \Rightarrow \varDelta \cup \varDelta_{3}; \varPi_{3}} \text{ If}$$

By the definition of substitution Δ_i ; $\sigma_i \Pi_3 = \sigma_i(\Delta \cup \Delta_3; \Pi_3) = \sigma_i E'$. By induction, we obtain the following:

$$\sigma E \vdash b : \text{Bool}$$

$$\sigma E \vdash_{\omega} s_1 \Rightarrow \sigma(\sigma_1 E')$$

$$\sigma E \vdash_{\omega} s_2 \Rightarrow \sigma(\sigma_2 E')$$

Each σ_i makes substitutions only for $x \in \Delta_3$, and thus we can define $sigma'_i x = \sigma(\sigma_i x)$ and have $\sigma(\sigma_i E') = \sigma'_i(\sigma E')$ with a well-types σ_i , and thus we can apply IF to achieve the desired result.

Call

$$\begin{array}{c|c} \omega \ p = \forall \Delta_1'.\Pi_1 \to \exists \Delta_2'.\sigma_2 \ \Pi_3 \\ \underline{\Delta_3 \ \text{fresh}} & \vdash \sigma_1 : \Delta_1' \to \Delta & \vdash \sigma_2 : \Delta_3 \to \Delta_2' \\ \underline{\Delta_3 \ \sigma_1 \ \Pi_1, \Pi_4 \vdash_\omega \ \text{call} \ p \Rightarrow \Delta \cup \Delta_3; \sigma_1 \ \Pi_3, \Pi_4} \ \text{Call} \end{array}$$

Let $\Pi_2 = \sigma_2 \Pi_3$ be the output permissions for the procedure. Now $\sigma \Pi = \sigma(\sigma_1 \Pi_1), \sigma \Pi_4 = (\sigma_1 \circ \sigma) \Pi_1, \sigma \Pi_4$ has the form needed to apply CALL getting output permissions $(\sigma_1 \circ \sigma) \Pi_3, \sigma \Pi_4 = \sigma(\sigma_1 Pi_3), \sigma \Pi_4 = \sigma(\sigma_1 \Pi_3, \Pi_4) = \sigma \Pi'$ which was to be proved.

ΕQ

$$\frac{1}{\Delta; \xi_1 v_1 : \operatorname{ptr}(\rho_1), \xi_2 v_2 : \operatorname{ptr}(\rho_2), \Pi_1 \vdash v_1 == v_2 : \operatorname{Bool}}$$
EQ

Here $\sigma \Pi = (\sigma \xi_1)v_1 : \operatorname{ptr}(\sigma \rho_1), (\sigma \xi_2)v_2 : \operatorname{ptr}(\sigma \rho_2), \sigma \Pi_1$ permitting us to apply EQ with the same result.

NOTEQ Direct from the inductive hypothesis.

BOOL Trivial.

PLUS Direct from the inductive hypothesis. DEREF

$$\frac{\Pi = \xi v : \operatorname{ptr}(\rho), \xi' \rho, \Pi_1}{\Delta : \Pi \vdash *v : \operatorname{Int}} \operatorname{DEREF}$$

Here $\sigma \Pi = (\sigma \xi)v$: $ptr(\sigma \rho), (\sigma \xi')(\sigma \rho), \sigma \Pi_1$ permits us to apply DEREF to achieve the desired result.

Int Trivial.

Lemma 11. If $\psi \varepsilon$ is defined, it is in the range (0,1). If $\psi \xi$ is defined, it is in the range (0,1].

Proof. Straightforward proof by induction using mathematical properties.

Environment and permission equivalence have no effect on the definition of $\psi \Pi$:

Lemma 12. If we have Δ ; $\Pi \equiv \Delta$; Π' and $\operatorname{Dom} \psi = \Delta$ then $\psi \Pi = \psi \Pi'$.

 ${\it Proof.}$ We consider each equivalence rule in turn and sketch why it has no effect on the result:

 $\cdot, \Pi \equiv \Pi$ The constant function $[\cdot \mapsto 0]$ is the identity for functional addition.

 $\Pi_1, \Pi_2 \equiv \Pi_2, \Pi_1$ Functional addition is commutative.

 $\Pi_1, (\Pi_2, \Pi_3) \equiv (\Pi_1, \Pi_2), \Pi_3$ Functional addition is associative

 $\{z\} \cup \Delta; z\pi, (1-z)\pi, \Pi \equiv \{z\} \cup \Delta; \pi, \Pi$ Since z is in the type context, it will be in the domain of ψ , and thus we can use the distributive law of real multiplication over real addition.

 $\varepsilon \varepsilon' \equiv \varepsilon' \varepsilon$ Real multiplication is commutative.

 $\varepsilon(\varepsilon'\varepsilon'') \equiv (\varepsilon\varepsilon')\varepsilon''$ Real multiplication is associative.

Consistency with the concatenation of permissions implies the consistency of each part:

Lemma 13. If Δ ; $\Pi_1, \Pi_2 \vdash \mu$ ok then Δ ; $\Pi_1 \vdash \mu$ ok

Proof. Obvious.

The following lemma formalizes the idea that writes cannot occur in parallel with reads or other writes:

Lemma 14. If we have an environment $E = (\Delta; \Pi_1, \Pi_2)$ and a consistent memory $E \vdash \mu$ ok with type variable mapping ψ , then if $(\psi \Pi_1) x = 1$ for some variable or location x, then $(\psi \Pi_2) x = 0$. In other words, it is not possible to rewrite Π_2 to include permission to observe $x \colon \Pi_2 \not\equiv \xi x \dots, \Pi'_2$.

Proof. Suppose we have such a $(\psi \Pi_1) x = 1$. Then $(\psi \Pi) x = (\psi \Pi_1) x + (\psi \Pi_2) x = 1 + (\psi \Pi_2)$. But the function $\psi \Pi_2$ returns a non-negative number (as can be seen by its construction), and thus the only way that we could have $\operatorname{Rng} \psi \Pi \subseteq [0,1]$ is to have $(\psi \Pi_2) x = 0$. Furthermore, it could not be possible to rewrite Π_2 to include a permission $\xi x \dots$ because $(\psi \Pi_2) x \geq \psi \xi > 0$ where the final inequality is from Lemma 11.

We have preservation for expressions as a separate lemma:

Lemma 15. If we have an expression $x \ E \vdash x : \tau$ (where $\tau \in \{Int, Bool\}$) which can be evaluated in some memory $\mu \langle \mu, x \rangle \rightarrow_g \langle \mu, x' \rangle$, then types will be preserved $E \vdash x' : \tau$.

Proof. Obvious case analysis.

Lemma 16. If we have a well-typed program $g (\vdash g : \omega)$ and a statement s that permission-checks in an well-formed environment $E = (\Delta; \Pi)$ ($\Delta \vdash \Pi$ perms and $E \vdash_{\omega} s \Rightarrow E''$) (where $E'' = (\Delta''; \Pi'')$) and a memory μ that is consistent with the environment $(E \vdash \mu)$ ok), then for any evaluation $(\langle \mu, s \rangle \to_g \langle \mu', s' \rangle)$, the resulting memory μ' is consistent in an environment E' ($E' \vdash_{\mu'}$ ok) suitable for permission-checking s' ($E' \vdash_{\omega} s' \Rightarrow \sigma E''$) for some well-formed substitution $\vdash \sigma : (\Delta'' - \Delta) \to \Delta'''$ for some Δ''' . (In other words, some of the newly introduced type variables may be substituted, but none of the existing ones.)

Proof. We strengthen the statement to prove by adding three extra results as follows:

```
Then \exists E' = (\Delta'; \Pi'), \psi', \sigma where
For all g,\omega,E\!=\!(\Delta;\Pi),s,E^{\prime\prime},\mu,\psi,\mu^\prime,s^\prime
where
                                                                               1'. \vdash \sigma : (\Delta'' - \Delta) \to \Delta'''
                                                                               2'. E' \vdash_{\omega} s' \Rightarrow \sigma E''
 1. \vdash g : \omega
                                                                               3'. \Delta' \supseteq \Delta
 2. \Delta \vdash \Pi perms
                                                                               4'. E' \vdash \mu' ok with \psi', that is
 3. E \vdash_{\omega} s \Rightarrow E''
                                                                                     (a') \Delta' = \operatorname{Dom} \psi'
 4. E \vdash \mu ok with \psi, that is
                                                                                    (b') Rng \psi' \Pi' \subseteq [0, 1]
      (a) \Delta = \text{Dom } \psi
                                                                                     (c') \psi'; \mu' \vdash \Pi' consistent
      (b) \operatorname{Rng} \psi \Pi \subseteq [0, 1]
                                                                               5'. \psi' \mid_{\Delta} = \psi
       (c) \psi; \mu \vdash \Pi consistent
                                                                               6'. (\psi' \Pi')_{\text{Dom }\mu} = \psi \Pi
 5. \langle \mu, s \rangle \rightarrow_q \langle \mu', s' \rangle
                                                                               7'. (\psi \Pi) x < 1 \Rightarrow \mu' x = \mu x
```

We prove by structural induction over the permission-check of s. We perform a case analysis on the root of the proof tree:

New s = v := new

$$\frac{\rho \text{ fresh}}{\Delta; 1v: \operatorname{ptr}(\rho'), \Pi_1 \vdash_\omega v\colon =\operatorname{\mathtt{new}} \Rightarrow \{\rho\} \cup \Delta; 1\rho, 1v: \operatorname{ptr}(\rho), \Pi_1} \text{ New}$$

Let l be the newly allocated location ($l \notin \text{Dom } \mu_L$). We have then $\mu' = \mu[v \mapsto l, l \mapsto 0]$. We prove the results with $E' = E'', \psi' = \psi[\rho \mapsto l], \sigma = [\rho \mapsto \rho]$.

- 1'. Straightforward with $\Delta''' = \Delta''$.
- 2'. Immediate since s' = skip.
- 3'. Immediate since $\Delta' = \Delta \cup \{\rho\}$.
- $4'.(a') \operatorname{Dom} \psi' = \operatorname{Dom} \psi \cup \{\rho\} = \Delta \cup \{\rho\} = \Delta'$
 - (b') Now ρ is fresh so it cannot appear in Π_1 and then $\psi' \Pi_1 = \psi \Pi_1$, and so we can compute $\operatorname{Rng} \psi' \Pi' = \operatorname{Rng} [l \mapsto 1] + [v \mapsto 1] + \psi' \Pi_1 = \operatorname{Rng} [l \mapsto 1] + \psi \Pi$, and thus the range is fine as long as $(\psi \Pi) l = 0$. Suppose $\psi \Pi l > 0$, then we must have $\xi \rho'' \in \Pi$ where $\psi \rho'' = l$. But then by , we would have $\psi \rho'' \in \operatorname{Dom} \mu$ which is a contradiction.
 - (c') Since $\psi \rho = l \in \text{Dom } \mu'$, we have $\psi'; \mu' \vdash 1\rho$ consistent. Also $\mu' v = l$ and thus $\psi'; \mu' \vdash 1v : \text{ptr}(\rho)$ consistent. Now by 4 and Lemma 14, we get that Π_1 cannot contain any permission $\xi v : \text{ptr}(\rho'')$ and the changes in μ' have no effect on checking consistency with Π_1 . Similarly ρ cannot appear in Π_1 because ρ is fresh, and thus we achieve $\psi'; \mu' \vdash \Pi_1$ consistent and so fulfill
- 5'. Follows immediately from definition of ψ' since ρ is fresh.
- 6'. As shown above $\psi' \Pi = (\psi \Pi)[l \mapsto 1]$ and so we have the result.
- 7'. Immediate since μ' changes only for l (not in the domain of $\psi \Pi$) and v (for which we have $\psi v = 1$.

Copy s = v := v'

$$\overline{\Delta}$$
; $1v : ptr(\rho), \xi v' : ptr(\rho'), \Pi_1 \vdash_{\omega} v := v' \Rightarrow \Delta$; $1v : ptr(\rho'), \xi v' : ptr(\rho'), \Pi_1$ Copy

Let $l = \mu v$. We have then $\mu' = \mu[v \mapsto l], E'' = E$. We prove the results with $E' = E'', \psi' = \psi, \sigma = []$ (the identity substitution).

- 1'. Trivial since $\Delta'' \Delta = \emptyset$, $\sigma = []$.
- 2'. Immediate since s' = skip.
- 3'. Trivial since $\Delta' = \Delta$.
- 4'. Since the permission fractions are unchanged in E', we need only prove $\psi; \mu' \vdash \Pi'$ consistent. Here μ' differs from μ only for v. Now by Lemma 14 Π_1 must not include any permissions involving v, and thus $\psi; \mu' \vdash \Pi$ consistent follows immediately from $\psi; \mu \vdash \Pi$ consistent. Similarly, we get $\psi; \mu' \vdash \xi v'$: $\operatorname{ptr}(\rho')$ consistent and $\psi \rho' = \mu v' = l$ from $\psi; \mu \vdash \xi v'$: $\operatorname{ptr}(\rho')$ consistent. The final condition $\psi; \mu' \vdash 1v$: $\operatorname{ptr}(\rho')$ consistent is satisfied since $\psi \rho' = l = \mu' v$.
- 5'. Trivial since $\psi' = \psi$.
- 6'. Immediate since no fractions are changed in Π' .

7'. Immediate since the only change is for v. UPDATE s = *v := e

$$\frac{E = (\Delta; \xi v : \operatorname{ptr}(\rho), 1\rho, \Pi_1) \qquad E \vdash e : \operatorname{Int}}{E \vdash_{\omega} *v := e \Rightarrow E} \text{ UPDATE}$$

Here we choose $E' = E, \psi' = \psi, \sigma = []$. If e is an addition or dereference expression, then we have preservation immediately by Lemma 15 and the additional items are trivially satisfied since $\mu' = \mu$. Otherwise if e is an integer constant $i \mu' = \mu[l \mapsto i]$ where $l = \mu v$.

- 1'. Trivial
- 2'. Trivial since s' = skip.
- 3'. Trivial.
- 4'. The consistency of μ' depends only on μ'_V and the domain of μ'_L , neither of which are changed in evaluation, preservation also follows easily.
- 5'. Trivial since $\psi' = \psi$.
- 6'. Trivial since $\Pi' = \Pi'' = \Pi$.
- 7'. Straightforward since $(\psi \Pi) v = 1$, and v is the only place where μ' differs from μ .

SKIP This case follows trivially since it is not satisfiable.

SEQ $s = s_1$; s_2

$$\frac{E \vdash_{\omega} s_1 \Rightarrow E_1'' \qquad E_1'' \vdash_{\omega} s_2 \Rightarrow E''}{E \vdash_{\omega} s_1 \colon s_2 \Rightarrow E''} \operatorname{SEQ}$$

If $s_1 = \mathtt{skip}$, then $\langle \mu, s \rangle \to_g \langle \mu, s_2 \rangle$ and we have $E_1 = E$ and thus can choose $E' = E, \psi' = \psi, \sigma = []$ and achieve preservation immediately.

Otherwise, by the inductive hypothesis applied to s_1 , we have E'_1, ψ'_1, σ_1 that meet conditions (1'-7') for s_1 . We choose the same variables $E' = E'_1, \psi' = \psi'_1, \sigma = \sigma_1$ and thus the only remaining result to prove is which follows immediately from the substitution lemma around the permission-check of s_1 , that is, by the substitution lemma, we have $\sigma E''_1 \vdash_{\omega} s_2 \Rightarrow \sigma E''$, and thus

$$E' \vdash_{\omega} s'_1; s_2 \Rightarrow \sigma E''$$

which was to be proved.

Par

$$\frac{\Delta; \Pi_1 \vdash_{\omega} s_1 \Rightarrow \Delta_1''; \Pi_1'' \qquad \Delta; \Pi_2 \vdash_{\omega} s_2 \Rightarrow \Delta_2''; \Pi_2''}{\Delta; \Pi_1, \Pi_2 \vdash_{\omega} s_1 \mid \mid s_2 \Rightarrow \Delta_1'' \cup \Delta_2''; \Pi_1'', \Pi_2''} \text{ PAR}$$

If $s_1 = s_2 = \text{skip}$, then preservation is immediate with $E' = E = E'', \psi' = \psi, \sigma = []$.

Otherwise, suppose we have the evaluation $\langle \mu, s_1 | | s_2 \rangle \rightarrow_g \langle \mu', s_1' | | s_2 \rangle$ where $\langle \mu, s_1 \rangle \rightarrow_g \langle \mu', s_1' \rangle$.

By Lemma 2.3, we have Δ ; $\Pi_1 \vdash \mu$ ok using the same ψ and thus we can apply for inductive hypothesis for s_1 using $E_1 = (\Delta; \Pi_1)$ to obtain $E'_1 = (\Delta'_1; \Pi'_1), \psi'_1, \sigma_1$ that satisfy conditions (1'-7'). Then we choose $E' = (\Delta'_1; \Pi'_1, \Pi_2), \psi' = \psi'_1 \sigma = \sigma_1$:

- 1'. Trivial from the inductive result.
- 2'. By Lemma 8, the variables in $\Delta_2'' \Delta$ are fresh, and thus σ must have no effect on variables in Δ_2'' and thus $\sigma \Pi_2'' = \Pi_2''$. This equality plus the weakening lemma(Lemma 9) enable us to construct the proof:

$$\frac{\Delta'; \Pi'_1 \vdash_{\omega} s_1 \Rightarrow \sigma(\Delta''_1; \Pi''_1) \qquad \Delta'; \Pi_2 \vdash_{\omega} s_2 \Rightarrow \sigma(Delta''_2; \Pi''_2)}{\Delta'; \Pi'_1, \Pi_2 \vdash_{\omega} s_1 \mid |s_2 \Rightarrow \sigma(\Delta''_1 \cup \Delta''_2; \Pi''_1, \Pi''_2)} \text{ PAR}$$

- 3'. Trivial since $\Delta' = \Delta'_1 \supseteq \Delta$.
- 4'. (a') Dom $\psi' = \text{Dom } \psi'_1 = Delta'_1$
 - (b') Now $\psi' \Pi' = \psi' \Pi'_1 + \psi' \Pi_2 = \psi' \Pi'_1 + \psi \Pi_2$ where the latter equality is due to condition 5' from the inductive use and the fact that Π_2 is well-formed. Now consider $(\psi'; \Pi')x$ for some $x \in \text{Dom } \mu$. By condition 6' on the inductive use, we get $(\psi' \Pi'_1)x = (\psi \Pi_1)x$ and thus by our original condition 44b, the result must be in the range 0 to 1. For some other $x \notin \text{Dom } \mu'$, we must have $(\psi \Pi_2)x = 0$ and thus the inductive condition 4'b'b' suffices to show $(\psi'; \Pi')x \in [0, 1]$.
 - (c') Here we need to check if $\Pi_2 \equiv \xi v : \operatorname{ptr}(\rho), \ldots$ we have $\psi' \rho = \mu' v$. Now by well-formedness and inductive condition 5', we get $\psi' \rho = \psi \rho$. Also $(\psi \Pi_2) v = \psi \xi > 0$ and thus by Lemma 14, $(\psi \Pi_1) v < 1$ and so by inductive condition 7' we have $\mu' v = \mu v$, and thus the consistency condition converts to $\psi \rho = \mu v$ which follows from requirements for preservation.
- 5'. Follows by induction.
- 6'. Follows by induction.
- 7'. Follows by induction.

The case for s_2 being evaluated is analogous.

If $s = \text{if } b \text{ then } s_1 \text{ else } s_2$

$$\frac{\Delta;\Pi\vdash b: \text{Bool} \quad \Delta;\Pi\vdash_{\omega}s_{1}\Rightarrow \Delta_{1};\sigma_{1}\Pi_{3} \quad \Delta;\Pi\vdash_{\omega}s_{2}\Rightarrow \Delta_{2};\sigma_{2}\Pi_{3}}{\Delta_{3}\text{ fresh} \quad \Delta\cup\Delta_{3}\vdash\Pi_{3}\text{ perms} \quad \vdash\sigma_{1}:\Delta_{3}\to\Delta_{1} \quad \vdash\sigma_{2}:\Delta_{3}\to\Delta_{2}}{E\vdash_{\omega}\text{ if }b\text{ then }s_{1}\text{ else }s_{2}\Rightarrow\Delta\cup\Delta_{3};\Pi_{3}}\text{ IF}$$

If b is not a boolean constant, then we have preservation directly by using Lemma 15. Otherwise since the memory is unchanged, we can use E' = E to keep consistency. If the "true" branch is taken, note that since $\sigma_1(\Delta \cup \Delta_3; \Pi_3) = (\Delta_1; \sigma_1 \Pi_3)$, we have preservation immediately with $\sigma = \sigma_1$. If the "false" branch is taken, the same reasoning applies.

Call s = call p

$$\begin{array}{c|c} \omega \ p = \forall \Delta_1.\Pi_1 \to \exists \Delta_2.\sigma_2 \ \Pi_3 \\ \underline{\Delta_3 \ \text{fresh}} & \vdash \sigma_1 : \Delta_1 \to \Delta & \vdash \sigma_2 : \Delta_3 \to \Delta_2 \\ \underline{\Delta_7 \sigma_1 \ \Pi_1, \Pi \vdash_{\omega} \ \text{call} \ \ p \Rightarrow \Delta \cup \Delta_3; \sigma_1 \ \Pi_3, \Pi} \end{array} \ \text{Call}$$

Here we can choose E' = E and preserve memory consistency since the memory is unchanged. For preservation of permission-checking, the fact that procedure p is well-typed yields the following facts:

$$\Delta_1; \Pi_1 \vdash_{\omega} g \, p \Rightarrow \Delta_1'; \sigma \, \Pi_2 \qquad \Delta_1' \cap \Delta_2 = \emptyset \qquad \vdash \sigma : \Delta_2 \to \Delta_1'$$

where $\Pi_2 = \sigma_2 \Pi_3$. We can use the first fact and the substitution lemma, and the weakening lemma to prove

$$\Delta$$
; $\sigma_1 \Pi_1$, $\Pi \vdash_{\omega} q p \Rightarrow \Delta \cup (\Delta'_1 - \Delta_1)$; $(\sigma_2 \circ \sigma \circ \sigma_1)\Pi_3$, Π

Now define σ_3 with domain Δ_3 as $\sigma_3 x_3 = (\sigma_2 \circ \sigma \circ \sigma_1) x_3$. By its construction, it is clear $\vdash \sigma_3 : \Delta_3 \to (\Delta \cup (\Delta_1' - \Delta_1))$. Now for $x \notin \Delta_3$, $(\sigma_2 \circ \sigma \circ \sigma_1) x = \sigma_1 x$. Thus $(\sigma_2 \circ \sigma \circ \sigma_1) \Pi_3 = (\sigma_1 \uplus \sigma_3) \Pi_3$. Moreover since $\vdash \sigma_1 : \Delta_1 \to \Delta$ and $\Delta \cap \Delta_3 = \emptyset$ (on account of freshness) we can apply σ_1 first and then σ_3 and $\sigma_3 \Pi = \Pi$ yielding finally:

$$\Delta; \sigma_1 \Pi_1, \Pi \vdash_{\omega} g p \Rightarrow \sigma_3(\Delta \cup \Delta_3; \sigma_1 \Pi_3, \Pi)$$

as required.

Definition 3. We say that two statements s_1 and s_2 are non-interfering in an environment E (written $E \vdash_{\omega} s_1 \# s_2$) if their parallel composition can be permission-checked in the environment $(E \vdash_{\omega} s_1 | | s_2 \Rightarrow E')$. Non-interference is extended to a boolean expression b using S(b) = if b then skip else skip and to an integer expression e using S(e) = if e! = 0 then skip else skip. For any statement s, let S(s) = s.

The kernel of the proof of non-interference is that one can re-order adjacent pairs of derivations:

Lemma 17. If in a well-typed program ($\vdash g : \omega$), we have two non-interfering statements or expressions $E \vdash_{\omega} x_1 \# x_2$ and a consistent memory μ ($E \vdash \mu$ ok) and we can evaluate either one step ($\langle \mu, x_1 \rangle \to_g \langle \mu_1, x_1' \rangle$ and $\langle \mu, x_2 \rangle \to_g \langle \mu_2, x_2' \rangle$), then we can reduce each in the other's output memory with the same new form ($\langle \mu_1, x_2 \rangle \to_g \langle \mu_{12}, x_2' \rangle$ and $\langle \mu_2, x_1 \rangle \to_g \langle \mu_{21}, x_1' \rangle$) and the resulting memories are isomorphic ($\mu_{12} \sim \mu_{21}$).

Proof. We prove the result inductively over x_1 and x_2 together. Now if both evaluations have no effect on memory, $\mu_1 = \mu_2 = \mu$, the result follows immediately. Moreover, if the evaluation of x_1 not only leaves the memory unchanged, but does not even depend on memory, that is x_1 is $pure\ (\forall_{\mu'}\langle\mu',x_1\rangle\to_g\langle\mu',x_1'\rangle)$, then we have $\mu_1 = \mu$ and thus $\mu_{12} = \mu_2$ and $\mu_{21} = \mu_2$, and so the result follows. Similarly if x_2 is pure, we are done. Furthermore if the memory effects and dependencies of x_1 or x_2 are indirect, because of subexpressions or substatements, the result follows using the inductive hypothesis. Finally, because the result is symmetric, we need only consider one direction of reordering. Thus we only have twelve cases to consider: the three primitive statements that update memory (allocation, copying and update) against each other and against the two other primitives that depend on memory (pointer equality and dereference). In the following case analysis, assume $E = (\Delta; \Pi_1, \Pi_2), \Delta; \Pi_i \vdash_{\omega} S(x_i) \Rightarrow \Delta_i'; \Pi_i',$ and ψ is the witness to $E \vdash \mu$ ok:

 v_1 :=new# v_2 :=new Let l_1 and l_2 be the two new locations (possibly the same location). Here $\mu_1 = \mu[v_1 \mapsto l_1, l_1 \mapsto 0], \mu_2 = \mu[v_2 \mapsto l_2, l_2 \mapsto 0]$. Now since

 $\Pi_1 = 1v_1 : \operatorname{ptr}(\rho_1'), \ldots, (\psi \Pi_1) v_1 = 1, \text{ by Lemma 14 } \Pi_2 \text{ cannot include any permissions to modify } v_1 \text{ and thus } v_2 \neq v_1. \text{ Now it is trivial to evaluate } x_1 = v_1 := \operatorname{new in } \mu_2' \text{ and vice versa to } x_1' = x_2' = \operatorname{skip}. \text{ We can choose the locations to allocate: if } l_1 = l_2, \text{ then let } l_1' = l_2' \notin \{l_1 = l_2\} \cup \operatorname{Dom} \mu_L, \text{ otherwise let } l_1' = l_1, \text{ and } l_2' = l_2. \text{ We end up with the following two memories: }$

$$\mu_{12} = \mu[v_1 \mapsto l_1, v_2 \mapsto l_2', l_1 \mapsto 0, l_2' \mapsto 0]$$

$$\mu_{21} = \mu[v_1 \mapsto l_1', v_2 \mapsto l_2, l_1' \mapsto 0, l_2 \mapsto 0]$$

If $l'_1 = l_1 \neq l_2 = l'_2$ then the two memories are not just isomorphic, but indeed equal. Otherwise $l'_1 = l'_2 \neq l_1 = l_2$ and the two memories are isomorphic using the one-to-one mapping that maps l_1 to l'_1 and vice versa, but keeps all other locations the same.

 $v_1:=\text{new}\#v_2:=v_2'$ Similar as the previous case, by Lemma 14, we determine $v_1\neq v_2, v_1\neq v_2'$ and thus by a similar argument the statements can be evaluated in either order.

```
\begin{array}{l} v_1 := \text{new} \# * v_2 := n_2 \text{ (Similar.)} \\ v_1 := \text{new} \# v_2 == v_2' \text{ (Similar.)} \\ v_1 := \text{new} \# * v_2 \text{ (Similar.)} \\ v_1 := \text{new} \# * v_2 \text{ (Similar.)} \\ v_1 := v_1' \# v_2 := v_2' \text{ Similarly, we find by Lemma 14 that } v_1 \neq v_2, v_1 \neq v_2', v_1' \neq v_2 \\ \text{ (although } v_1' = v_2' \text{ is possible—these variables are only read).} \\ v_1 := v_1' \# * v_2 := n_2 \text{ (Similar, as all following cases:)} \\ v_1 := v_1' \# v_2 == v_2' \\ v_1 := v_1' \# * v_2 \\ * v_1 := n_1 \# * v_2 := n_2 \\ * v_1 := n_1 \# v_2 == v_2' \\ * v_1 := n_1 \# * v_2 \end{aligned}
```

The reordering lemma allows us to prove a one-step non-determinism lemma:

Lemma 18. Given a well-typed program ($\vdash g : \omega$) and a permission-checked statement $E \vdash_{\omega} s \Rightarrow E'$ and a consistent memory μ ($E \vdash \mu$ ok) and we have two different evaluation steps ($\langle \mu, s \rangle \to_g \langle \mu', s' \rangle$ and $\langle \mu, s \rangle \to_g \langle \mu'', s'' \rangle$, where $s' \neq s''$) then there exists an evaluation step to unify the two different results ($\langle \mu', s' \rangle \to_g \langle \mu_{12}, s^* \rangle$ and $\langle \mu'', s'' \rangle \to_g \langle \mu_{21}, s^* \rangle$) with isomorphic results ($\mu_{12} \sim \mu_{21}$).

Proof. NB: There is no need to prove this lemma for expressions since expression evaluation is deterministic.

We prove by induction on the structure of s. Now the only constructs that permit non-determinism are sequential and parallel composition. In the former case $(s=s_1;s_2)$, the nondeterminism must come in the left branch $(s'=s'_1;s_2,s''=s''_1;s_2)$, and thus we can apply the inductive hypothesis to achieve a single result on the left $(\langle \mu',s'_1\rangle \to_g \langle \mu_{12},s_1^*\rangle$ and $\langle \mu'',s''_1\rangle \to_g \langle \mu_{21},s_1^*\rangle$ with $\mu_{12}\sim \mu_{21}$. From this we can see $s^*=s_1^*;s_2$ unifies the two evaluations for s.

In the parallel composition case, either the two different evaluations each operate on the same branch (in which case we can apply the inductive hypothesis

in the same way as we just did for sequential composition) or else (assuming without loss of generality that the first evaluation reduces the left branch) we have the following situation:

$$\langle \mu, s_1 \mid \mid s_2 \rangle \rightarrow_q \langle \mu', s_1' \mid \mid s_2 \rangle \quad \langle \mu, s_1 \mid \mid s_2 \rangle \rightarrow_q \langle \mu'', s_1 \mid \mid s_2' \rangle$$

Since $E \vdash_{\omega} s_1 \mid \mid s_2 \Rightarrow E'$, we have $E \vdash_{\omega} s_1 \# s_2$ and thus we can apply Lemma 17 to get $\langle \mu'', s_1 \rangle \to_g \langle \mu_{21}, s_1' \rangle$ and $\langle \mu', s_2 \rangle \to_g \langle \mu_{12}, s_2' \rangle$ where $\mu_{12} \sim \mu_{21}$ and then apply PAR to get

$$\langle \mu', s_1' \mid \mid s_2 \rangle \rightarrow_g \langle \mu_{12}, s_1' \mid \mid s_2' \rangle \quad \langle \mu'', s_1 \mid \mid s_2' \rangle \rightarrow_g \langle \mu_{21}, s_1' \mid \mid s_2' \rangle$$

and we have the result with $s^* = s_1' \mid \mid s_2'$.

Finally we have our theorem of deterministic results for permission-checked programs:

Theorem 1. If we have a well-typed program g ($\vdash g:\omega$) and a statement s that permission-checks in an environment E ($E \vdash_{\omega} s \Rightarrow E'$) and a memory μ_1 that is consistent with the environment ($E \vdash \mu_1$ ok), and s can be fully evaluated in this memory in k steps ($\langle \mu_1, s \rangle \stackrel{k}{\to}_g \langle \mu_1^*, \mathtt{skip} \rangle$) then for any isomorphic memory $\mu_2 \sim \mu_1$, any other evaluation sequence $\langle \mu_2, s \rangle \xrightarrow{g} \langle \mu_2', s' \rangle \xrightarrow{g} \dots$ terminates in exactly k steps and has an isomorphic result $\mu_2^* \sim \mu_1^*$.

Proof. By Lemma 4, we can redo the first statement evaluation with μ_2 and achieve an isomorphic result μ_{12}^* . Thus since isomorphicity is transitive, we can assume a single starting memory $\mu = \mu_1 = \mu_2$ without loss of generality.

We prove the result by induction over k. If k=0, we must have $s=\mathtt{skip}$ and the result follows immediately. Otherwise, if k=1, then if the second sequence starts with the same reduction, it terminates too in one step. If it were to start with a different reduction $(\langle \mu, s \rangle \to_g \langle \mu_2', s' \rangle)$ where $s' \neq \mathtt{skip}$, then by Lemma 18, we would be able to form an evaluation $\langle \mu_1^*, \mathtt{skip} \rangle \to_g \langle \mu_2'', s'' \rangle$ which is impossible. It is similarly impossible for the second sequence to terminate in one step if the first does not.

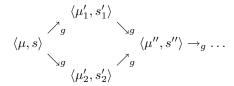
Otherwise assume k > 1. The two evaluation sequences are

$$\langle \mu, s \rangle \rightarrow_q \langle \mu_1', s_1' \rangle \rightarrow_q \ldots \rightarrow_q \langle \mu_1^*, \text{skip} \rangle$$

and

$$\langle \mu, s \rangle \rightarrow_a \langle \mu_2', s_2' \rangle \rightarrow_a \dots$$

If $s'_1 = s'_2$, we can use the inductive hypothesis on the tail of each evaluation to achieve the result. Otherwise, we can apply Lemma 18 to form the following two new evaluation sequences that share the same tail:



Now we apply the inductive hypothesis on the tail of the original evaluation sequence and the upper evaluation sequence shown here to determine that the upper evaluation sequence must terminate in exactly k steps with an isomorphic result, and thus the bottom evaluation sequence must also terminate in k steps. Next Lemmas 18 and 4 ensure the two evaluation sequences in the diagram have isomorphic results. Finally, we apply the inductive hypothesis again to the tail of the bottom evaluation sequence with the tail of the second sequence to achieve the desired result.