

Connecting Effects and Uniqueness with Adoption

John Tang Boyland

University of Wisconsin—Milwaukee

Abstract. In a previous paper, we discussed how the concepts of uniqueness and effects are interdependent. In this paper, we show how “Adoption and Focus,” a proposal for handling linear pointers in shared variables can be extended to connect the two concepts. Our innovations include the ability to define adoption relations between individual fields rather than whole objects, and the ability to “focus” on more than one adoptee at a time. The resulting system uses recursive alias types, “permission closures” and “conditional permissions.” Then we show how previously proposed effect and uniqueness annotations can be represented in the type system.

1 Introduction

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

1.1 Background on Uniqueness

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

However, destructive reads have several problems: (1) they make it difficult to query information about a unique object without losing it; (2) they make it impossible to have sound invariants about the non-nullness of unique variables; (3) they are inappropriate for use in “const” methods, which are supposed to treat their object as read only; (4) they require a language change. Thus several

researchers have independently proposed the use of what we call “borrowing” reads, which do not nullify the field. Unfortunately borrowing reads greatly impact the benefits of uniqueness unless it can be shown that a unique field is not read (or at least not considered unique) during the lifetime of the borrowing. For example, AliasJava permits borrowing without checking for possible reads during the lifetime of the borrow and thus can only guarantee that a “unique” pointer stored in a field will not be also available in another field. In particular, a class cannot prevent “outsiders” from modifying the contents of supposedly unique sub-objects of instances of this class.

Similarly, Leino, Nelson and Stata [7, 8] proposed a pointer property, “virginity,” related to uniqueness that is required of all initialization of “pivot” fields (fields that refer to wholly-owned subsidiary objects or “sub-objects”). The rules have a similar weakness to that of AliasJava: they are (intentionally) not strong enough to prevent a client of an object from having access to its sub-objects.

Leino and others’ recent work [9] and “strong” ownership type systems [10] avoid these aliasing problems by not permitting clients to create objects that will be internal to a container. In essence, the system forbids objects from being transferred from one container to another. External uniqueness extends ownership to cover transferable uniqueness, but does so through destructive reads.

In our earlier work on “alias burying” [11], we proposed using effects annotations to prevent borrowing reads from weakening the semantics of uniqueness. With alias burying, no destructive reads are needed. The paper suggested annotating every method with the list of fields of this or any other object that is read during the dynamic extent of the method call. Clearly, this not only exposes too much of object’s internal structure, but is overly conservative since it does not distinguish different objects. We intended to use our object-oriented effects system [12] instead, but no one has been able to come up with a satisfactory system to combine these two areas. This paper contributes such a type system.

1.2 Contribution

We can check that program accesses meet declared effects by using a *permission* system, in which the context used to check program elements indicates which parts of the state we are allowed to access. (“Fractional permissions” [13] can be used to distinguish reads from writes, but this paper will not discuss this (orthogonal) extension.) Effects are in terms of *state* in the program: variables, and fields of objects on the heap. For encapsulation purposes, we aggregate fields into “data groups” [12, 14]. This is modeled by having the permission to access the field “nested” within the permission to access the data group.

In a permission system, there is only one permission for each state. Thus permissions can also be used to model uniqueness: a unique pointer is one which is packaged along with the permission to access the state pointed to. This conception of uniqueness is weaker than the usual sense of uniqueness in which there are no other pointers to the state in the store. However, in our system, any other potential pointers to the same state come without permission to access it, and thus even the limited sense of uniqueness provided by permissions is sufficient

for analysis purposes. The problem is not the *existence* of aliasing pointers, but rather the *access* of the state through these aliasing pointers.

Of course not all pointers are unique. A *shared* pointer is modeled by having the permission to access the pointed-to state “nested” in a globally accessible permission. Now, we want to make sure that it is impossible for two separate parts of the code to grab the permissions to the same shared state. This could be done using dynamic checks, but in our system we use the type system to prevent the same permission from being removed twice without being replaced in between.

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

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

- It uses permissions to model both effects and uniqueness.
- It uses the concept of permission “nesting” (adoption) to model shared state and aggregation of state in “data groups.”
- Conditional permissions make explicit the use of “null” or other terminating pointers.

In the following section, we describe this type system, and then in Section 3 we showed how one can take some of our earlier effects annotations together with the uniqueness annotations of our work on Alias Burying and interpret them as types in system proposed in this paper. Section 4 compares our system to closely related work.

2 The System

We define our type system over a simple low-level imperative language. Section 3 then adapts the types to a Java-like language.

2.1 Operational Semantics

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

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

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

In this simple language, the only kind of variable is the field of an object (global variables are handled by treating them as fields of the null object). Array elements could be handled with dependent types [15] in a similar manner.

Each field f has a declared default value and type chosen from the following limited set:

$\frac{v_f}{()}$	$\frac{\tau_f}{\text{unit}}$
false	bool
0	int
\$0	ptr(\$0)

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

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

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

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

There is no requirement that adoption is a functional relation.

Figure 1 defines a small-step semantics for evaluating this simple language. The two interesting parts are (1) the evaluation of **new** expressions in which an address is found unused in the memory or adoption information, and (2) nesting which adds an adoption relation fact. Adoption can never be undone.

2.2 Permission Types

The environment $E = (\Delta; \Pi)$ in which code is checked has two parts: a type context Δ that lists address variables ρ ; and a “set” of permissions Π that indicates what state we are permitted to access and the type that the state will have. We treat Π in a somewhat linear fashion, in that permissions cannot be duplicated. However, making use of permissions does not consume them. (The use of “...” here means there are other forms for permissions.)

$$\begin{aligned} (\text{type context}) \quad \Delta &::= \cdot \mid \rho \mid \Delta, \Delta \\ (\text{permissions}) \quad \Pi &::= \cdot \mid \Pi, \Pi \mid \dots \end{aligned}$$

The most important kind of permission is a key k . Now, since any key may have other keys nested in it, and some of those keys may be active, the permission enumerates those nested keys that are currently “carved out.” As mentioned in the introduction, unlike earlier proposals is which a key applies to a whole

$$\begin{array}{c}
\frac{(\mu; a; e_1) \rightarrow_g (\mu'; a'; e'_1)}{(\mu; a; e_1 + e_2) \rightarrow_g (\mu'; a'; e'_1 + e_2)} \qquad \frac{(\mu; a; e_2) \rightarrow_g (\mu'; a'; e'_2)}{(\mu; a; v + e_2) \rightarrow_g (\mu'; a'; v + e'_2)} \\
(\mu; a; n_1 + n_2) \rightarrow_g (\mu; a; n_1 + n_2) \qquad \frac{(\mu; a; e_1) \rightarrow_g (\mu'; a'; e'_1)}{(\mu; a; e_1 = e_2) \rightarrow_g (\mu'; a'; e'_1 = e_2)} \\
\frac{(\mu; a; e_2) \rightarrow_g (\mu'; a'; e'_2)}{(\mu; a; v = e_2) \rightarrow_g (\mu'; a'; v = e'_2)} \qquad (\mu; a; v_1 = v_2) \rightarrow_g (\mu; a; v_1 = v_2) \\
\frac{o \notin \text{Rng}(\mu) \quad \forall f \in F \ o.f \notin \text{Dom}(\mu) \cup \text{Dom}(a) \cup \text{Rng}(a)}{(\mu; a; \text{new}\{f_1, \dots, f_n\}) \rightarrow_g (\mu[o.f_i \rightarrow v_{f_i} \mid 1 \leq i \leq n]; a; o)} \\
\frac{(\mu; a; e) \rightarrow_g (\mu'; a'; e')}{(\mu; a; e.f) \rightarrow_g (\mu'; a'; e'.f)} \qquad \frac{\mu(o.f) = v}{(\mu; a; o.f) \rightarrow_g (\mu; a; v)} \\
\frac{(\mu; a; e_1) \rightarrow_g (\mu'; a'; e'_1)}{(\mu; a; e_1.f := e_2) \rightarrow_g (\mu'; a'; e'_1.f := e_2)} \qquad \frac{(\mu; a; e_2) \rightarrow_g (\mu'; a'; e'_2)}{(\mu; a; v_1.f := e_2) \rightarrow_g (\mu'; a'; v_1.f := e'_2)} \\
\frac{v_2 \sim v_f \quad \mu' = \mu[o.f \rightarrow v_2]}{(\mu; a; o.f := v_2) \rightarrow_g (\mu'; a; ())} \qquad \frac{(\mu; a; e_1) \rightarrow_g (\mu'; a'; e'_1)}{(\mu; a; e_1; e_2) \rightarrow_g (\mu'; a'; e'_1; e_2)} \\
(\mu; a; () ; e_2) \rightarrow_g (\mu; a; e_2) \\
\frac{(\mu; a; e_1) \rightarrow_g (\mu'; a'; e'_1)}{(\mu; a; \text{if } e_1 \text{ then } e_2 \text{ else } e_3) \rightarrow_g (\mu'; a'; \text{if } e'_1 \text{ then } e_2 \text{ else } e_3)} \\
(\mu; a; \text{if true then } e_2 \text{ else } e_3) \rightarrow_g (\mu; a; e_2) \\
(\mu; a; \text{if false then } e_2 \text{ else } e_3) \rightarrow_g (\mu; a; e_3) \qquad (\mu; a; \text{call } p) \rightarrow_g (\mu; a; gp) \\
\frac{(\mu; a; e_1) \rightarrow_g (\mu'; a'; e'_1)}{(\mu; a; \text{nest } e_1.f_1 \text{ in } e_2.f_2) \rightarrow_g (\mu'; a'; \text{nest } e'_1.f_1 \text{ in } e_2.f_2)} \\
\frac{(\mu; a; e_2) \rightarrow_g (\mu'; a'; e'_2)}{(\mu; a; \text{nest } v_1.f_1 \text{ in } e_2.f_2) \rightarrow_g (\mu'; a'; \text{nest } v_1.f_1 \text{ in } e'_2.f_2)} \\
\frac{a' = a \cup \{(o_1.f_1) \prec (o_2.f_2)\}}{(\mu; a; \text{nest } o_1.f_1 \text{ in } o_2.f_2) \rightarrow_g (\mu; a'; ())}
\end{array}$$

Fig. 1. Operational Semantics

object, a key here applies to a *field* of an object. The permission to access a field includes the type stored there. We have four atomic types:

$$\begin{aligned}
& \text{(type)} \quad \tau ::= \text{unit} \mid \text{bool} \mid \text{int} \mid \text{ptr}(l) \mid \dots \\
& \text{(object reference)} \quad l ::= o \mid \rho \\
& \text{(key)} \quad k ::= l.f \\
& \text{(permissions)} \quad \Pi ::= \dots \mid k : \tau \setminus \{k_1 : \tau_1, \dots, k_n : \tau_n\} \mid \dots
\end{aligned}$$

If there are no keys carved out of a key (the permission is of the form $k : \tau \setminus \{\}$), the permission may be abbreviated as $k : \tau$.

Two types τ and τ' are “storage compatible” (written $\tau \sim \tau'$) if space storing a value of type τ can be reused to store a τ' . Of course $\tau \sim \tau$ for all types τ , but we also assume all pointer types are interchangeable: $\text{ptr}(l) \sim \text{ptr}(l')$.

Field of unit type do not store any information, but instead are used to model “data groups” (we reuse the terminology of Leino and others [14, 9]). Aggregation of fields into data groups is accomplished by nesting, for instance $\rho.x : \text{int} \prec \rho.\text{loc}$, where the “x” field of the object is part of the “loc” data group.

The type system described here uses a simple logic that can be represented by boolean formulae over key equalities $k = k'$ and nesting information $k : \tau \prec k'$. In a slight abuse of notation, we conflate syntactic boolean operators (such as \wedge, \vee, \neg) with the underlying semantic boolean operations. The syntax also provides a construct for named recursive formulae t over object references.

$$\text{(formula)} \quad \Gamma ::= \text{true} \mid k = k' \mid k : \tau \prec k' \mid \Gamma \wedge \Gamma' \mid \neg \Gamma \mid t(l_1, \dots, l_n)$$

(We write $k \neq k'$ as shorthand for $\neg(k = k')$.)

Boolean formulae are used in two ways in permissions. First, a formula may be paired with a permissions “set.” Second, we have conditional permissions in which a set of permissions is guarded by a condition. In another slight abuse of notation, we borrow the linear implication “ \multimap ” operator to remind the reader that conditional permissions are consumed when applied:

$$\text{(permissions)} \quad \Pi ::= \dots \mid (\Pi; \Gamma) \mid \Gamma \multimap \Pi$$

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

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

Here “All” is a data group in which all of the fields of the object are nested, perhaps indirectly. Now, we often do not know what the actual pointer value of a variable is and thus need to use some form of quantified types. For this reason a variable may have existential type, for example $\mathbf{x} : \exists \rho.\text{ptr}(\rho)$, but this form is not sufficient, because we need to be able to express permissions about the existentially bound address variable. Thus we have forms of compound types, a pairing of a set of permissions with a type, and an existential:

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

Thus, a variable with an unknown pointer value, but for whose object (if not null) we have permission to access the fields, would be declared as follows:

$$\text{\$0.x} : \exists \rho. \text{ptr}(\rho) \text{ with } \rho \neq \text{\$0} - \circ \rho. \text{All} : \text{unit}$$

This indeed is how we express the concept of a unique pointer. While other places in the system may have the same pointer value, no one else has the permission to access the pointed-to state, since permissions are unique.

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

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

Parameters and results can be passed in global variables or in specially created activation objects. Substitutions are used to transform the permissions into a form accepted by a procedure and to convert the resulting permissions back. Substitutions are typed by their domain and range:

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

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

The type rules corresponding to the syntax are given in Figure 2. Each expression produces a possibly new environment after being checked and thus our relation is $E \vdash_{\omega} e : \tau \dashv E'$ where ω is the program typing.

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

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

The IFTRUE and IFFALSE rules are needed for type preservation.

The rule for **nest** expressions requires that the key being “adopted” be fully available (without anything carved out) and not equal to any key currently carved out of the “adopter.” This rule prevents a key from being twice adopted by the same adopter, and also prevents self or cyclic adoption. These situations do not cause consistency problems, but are probably bugs in the code.

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

Fig. 2. Syntactic Type rules

2.3 Consistency

Types can ensure that programs don't go wrong, but only if the memory is in a state matched by the types, or permissions in our case. Part of the “matching” that must be done is to give the absolute addresses corresponding to each address variable ρ :

$$\text{(mapping)} \quad \psi ::= \{\rho \rightarrow o, \dots\}$$

The consistency relation is written

$$\mu; a; \psi \vdash \Pi \text{ consistent}$$

We have not worked out all the details of consistency at the point of writing, and space constraints would preclude a full description in any case. The basic concepts that need to be verified are that there never be more than one permission for each key (field), and that the value stored in a field be of the type indicated in the permission. The boolean formulae paired with the permissions must also be matched against the actual memory and adoption facts.

Part of the information in any permission are the equality, inequality and adoption facts in it. For instance, if we see that one key has two other keys carved out of it ($k : \tau \setminus \{k_1 : \tau_1, k_2 : \tau_2\}$), then those keys must be distinct and furthermore must each be nested in k . We write $\Pi \models \Gamma$ to mean that the boolean formula Γ is implied by the set of permissions Π . The condition that lets us determine the most information from a set of permissions is to find anything that is consistent with every situation in which the permissions are consistent:

$$\frac{\forall_{\mu; a; \psi} (\mu; a; \psi \vdash \Pi \text{ consistent}) \Rightarrow (\mu; a; \psi \vdash (\cdot; \Gamma) \text{ consistent})}{\Pi \models \Gamma}$$

This definition is not constructive, but it is easy to define special cases such as

$$(\cdot; \Gamma) \models \Gamma \qquad (f \neq f') \Rightarrow (\cdot \models l.f \neq l'.f')$$

$$k : \tau \setminus \{k_1 : \tau_1, \dots, k_2 : \tau_2, \dots\} \models k_1 \neq k_2 \qquad (\Pi_1 \models \Gamma) \Rightarrow ((\Pi_1, \Pi_2) \models \Gamma)$$

$$(\Pi \models \Gamma_1) \wedge (\Pi \models \Gamma_2) \Rightarrow (\Pi \models \Gamma_1 \wedge \Gamma_2)$$

Depending on how consistency is finally formalized, it may be possible to give a fixed number of such structural rules that are complete for the characterization of the \models relation.

We overload the operator to apply to pairs of permissions too:

$$\frac{\forall_{\mu; a; \psi} (\mu; a; \psi \vdash \Pi_1 \text{ consistent}) \Rightarrow (\mu; a; \psi \vdash \Pi_2 \text{ consistent})}{\Pi_1 \models \Pi_2}$$

and then define

$$\frac{\Pi_1 \models \Pi_2 \quad \Pi_2 \models \Pi_1}{\Pi_1 \equiv \Pi_2}$$

As before one can define any number of special cases:

$$\begin{aligned}
\cdot, \Pi &\equiv \Pi & \Pi_1, (\Pi_2, \Pi_3) &\equiv (\Pi_1, \Pi_2), \Pi_3 \\
(\Pi_1; \Gamma_1), (\Pi_2; \Gamma_2) &\equiv (\Pi_1, \Pi_2; \Gamma \wedge \Gamma_2) \\
k : \tau \setminus \{k_1 : \tau_1, k_2 : \tau_2, \dots, k_n : \tau_n\} &\equiv k : \tau \setminus \{k_2 : \tau_2, \dots, k_n : \tau_n, k_1 : \tau_1\} \\
(\Pi \models \Gamma) \Rightarrow (\Pi \equiv (\Pi; \Gamma)) & & (\Gamma \multimap \Pi; \Gamma) &\equiv (\Pi; \Gamma) \\
k' : \tau' \setminus \{k_1 : \tau_1, \dots, k_n : \tau_n\}; k : \tau \prec k' \wedge k \neq k_1 \wedge \dots \wedge k \neq k_n &\equiv \\
& k' : \tau \setminus \{k : \tau, k_1 : \tau_1, \dots, k_n : \tau_n\}, k : \tau
\end{aligned}$$

The last sample rule shows how one can carve out a key from another key, or replace it.

One may substitute a smaller set of permissions before or after any typing:

$$\frac{\text{TRANSFORM} \quad \Pi \models \Pi_1 \quad \Delta; \Pi_1 \vdash_\omega s \Rightarrow \Delta'; \Pi'_1 \quad \Pi'_1 \models \Pi'}{\Delta; \Pi \vdash_\omega s \Rightarrow \Delta'; \Pi'}$$

Dropped keys represent places where work is created for the garbage collector.

2.4 Summary

The novel parts of this type system (as compared to previous work) are:

- The permission keys are fields rather than whole objects, enabling fine-grained protection;
- Multiple (distinct) permissions may be “carved out” of a single nesting location at a time;
- Arbitrary permissions may be packed into existential types.

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

- Defining memory consistency.
- Proving that well-typed programs don’t go wrong.
- Proving that effects are respected.
- Coming up with a usable approximation (if not a complete characterization) of the \models relation. In particular, we need a way to use this relation algorithmically during type checking.
- Making the type rules algorithmic. Assuming we have a (probably conservative) way to apply the \models relation, and thus the TRANSFORM rule, we only need to address the nondeterminism of the IF rules (especially the rule for $E_1 \vee E_2$).

What this paper *does* provide is an interesting way to unify the concepts of effects and uniqueness in a new type system (inspired by Fähndrich and DeLine’s adoption and focus). How these concepts are realized is described next.

3 Realizing Effects and Uniqueness Annotations

As is clear from what precedes this discussion, the types used in this proposal are complex and verbose. One cannot expect programmers to want to use such a system. In this section, we explain how commonly proposed, higher-level annotations can be expressed in the full type system. Of course, these annotations do not give one the full expressive power of the complete type system, but the full system can be provided in unusual situations.

One question that needs to be answered is why one should propose a type system that is too complex to use and then hide it underneath a simpler type system. Why not simply use the simpler type system and be done with it? The problem is that no one has yet come up with a type system that successfully unifies effects and uniqueness annotations. It appears that solving the problem correctly requires more formal machinery than expected. A similar situation applies to object-oriented languages: typing “self” for a language with imperative state and overriding requires a complex combination of existential types, bounded polymorphism and recursive types, even though these features are used in stylized ways [17]. This analogy suggests that there may be a way to type-check uses of the annotations described here without requiring a type checker for the complete type system. The algorithm could be proved sound against the complete type system.

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

data groups A class may declare data groups. A field is tagged with its parent data group; a data group may also have a parent. Unlike our earlier work [12] (where data groups were called “regions”), a field may be nested in two data groups, but the type system will require such a field to be always carved out of all but one of its data groups at any point in the program.

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

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

3.1 Class and Field Annotations

We represent class types by named, potentially recursive, fact-creating functions with one parameter giving the location for the object. The facts include one adoption fact for every field and data group. For simplicity, we assume the existence of a single root data group “All” in which all fields are nested (perhaps

indirectly). The fact for a field includes its type which, for pointers, is an existential whose body is a macro-call that takes the location, the fact-creating function for the type and whether the pointer is “good,” usually $(\rho \neq \$0)$.¹ The macro function to use is named by the reference annotation and is expanded as part of the realization process:

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

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

For a first example, consider the following Java-like class and its fact-creating function:

<pre>class Point { group Loc; int x in Loc; int y in Loc; }</pre>	<pre>Point(ρ) = ρ.Loc : unit \prec ρ.All \wedge ρ.x : int \prec ρ.Loc \wedge ρ.y : int \prec ρ.Loc)</pre>
---	--

Next, a rectangle class that has two unique points and a shared string name:

<pre>class Rectangle { group Looks; group Dims in Looks; unique Point tl in Dims; unique Point br in Dims; shared String n in Looks; }</pre>	<pre>Rectangle(ρ) = ρ.Looks : unit \prec ρ.All \wedge ρ.Dims : unit \prec ρ.Looks \wedge ρ.tl : $\exists \rho$.$\mathit{unique}(\rho, \text{Point}, \rho \neq \\$0) \prec$ ρ.Dims \wedge ρ.br : $\exists \rho$.$\mathit{unique}(\rho, \text{Point}, \rho \neq \\$0) \prec$ ρ.Dims \wedge ρ.n : $\exists \rho$.$\mathit{shared}(\rho, \text{String}, \rho \neq \\$0) \prec$ ρ.Looks</pre>
--	--

3.2 Parameters and Methods Effects

The parameters and return value are packed into an “argument” object pointed to by a global `ap`. We also give general access to a number of global “temporaries.” Permissions to access all these globals, the parameters and the return value are passed to and returned from the procedure. When a procedure is called, the return value is uninitialized (if a pointer, has a pointer type with an address variable for which we have no information). At the end, the parameters are uninitialized. The procedure type is polymorphic using a variable for the argument object, the receiver, each pointer parameter and the result (if a pointer value). The permissions for each parameter are typed on procedure entry using the same macros for the pointer annotations, except this time the macro-calls

¹ If we added non-null annotations (as suggested by Fähndrich and DeLine [18]), the condition can be strengthened to “true” for non-null pointers.

are not wrapped in existentials. The return value is typed using the annotation macro-call upon exit.

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

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

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

Its realized type is

$$\forall \rho_f, \rho_t, \rho_s. \left(\begin{array}{l} (\$0.ap : \text{ptr}(\rho_f), \text{temps}, \rho_f.\text{this} : \text{borrowed}(\rho_t, \text{Rectangle}, \text{true}), \\ \rho_f.n : \text{shared}(\rho_s, \text{String}, \neg(\rho_s = \$0), \rho_t.\text{Looks} : \text{unit}) \rightarrow \\ \exists \rho'_s. (\$0.ap : \text{ptr}(\rho_f), \text{temps}, \rho_f.\text{this} : \text{ptr}(\rho_t), \\ \rho_f.n : \text{ptr}(\rho'_s), \rho_t.\text{Looks} : \text{unit}) \end{array} \right)$$

3.3 Discussion

This realization links parameter annotations and effects more strongly than in our previous work: it does not permit parameters to be aliased; it does not permit shared state to be passed borrowed to a method that affects **other**. Intersection types “solve” this problem at the cost of a number of variants, exponential in the number of parameters. On the other hand, the stricter definition is probably a good default; a parameter whose state could be accessed through another parameter or through a global variable should be declared as such. The stricter rule corresponds closely to the new ANSI C `restrict` qualifier as formalized by Foster and others [19].

The realization described here does not fully handle inheritance and virtual overriding because there is no way to express downcasts. Neither does it handle the problem of partially constructed objects. We hope to use Fähndrich and Leino’s “monotonic heap states” [16] to describe how the “facts” for an object grow from the facts provided by the superclass to the additional ones provided by the subclass.

4 Related Work

This work was conceived as an extension to the work of Manuel Fähndrich and Robert DeLine called “adoption and focus” [20]. This type system permits a linear pointer to be irrevocably “adopted” by another pointer. Then the pointer can be duplicated (i.e. copied nonlinearly) with a “guarded type” $g \triangleright \tau$ where g is the adopter. Should some piece of the code need to use the pointer and it has access to the adopter, it can “focus” on the adoptee and gain the linear pointer

while temporarily giving up the rights on the adopter. When there is no more need to access the linear pointer (and its linearity and type have been restored), the linear pointer can “disappear” back into the adopter which is then restored. Our system extends/changes the ideas of adoption and focus in the following ways:

1. Reads and writes can be distinguished using “fractional permissions” [13]. For reasons of brevity, this issue is not discussed in this paper.
2. Unlike adoption and focus, “carving out” permissions for a nested key (“focusing” in their terminology) does not make the nesting key (the “adopter”) inaccessible. It only forbids the carving out of the *same* nested key again.
3. In our system, adoption (and protection in general) is performed at the level of fields of an object rather than whole objects. This allows finer-grained permissions.

Additionally, “adoption and focus” was described just as a type system, without anything to prove correct. The system described here has made more progress in the direction of defining what correctness means, without reaching the goal. We intend to continue this progress.

Adoption and focus, as well as this work use Alias Types [21], a technique to represent aliasing in the type system. Alias types can be used to precisely describe the shape of recursive data structures [22]. Adoption allows alias types to describe uniqueness as well, and our extensions of adoption allow alias types to describe effects.

Effects systems have been defined for functional languages in order to safely deallocate “regions” of data that are no longer being accessed [23]. A region encapsulates a (possibly heterogeneous) set of objects. This work was extended by Walker and others [24] in a capability calculus that has one permission key for each region. These insights were later used in the “adoption and focus” work.

Boyapati and others [25, 26] have incorporated uniqueness, effects and regions in an ownership type system for Java-like languages. They use a permission system over whole objects (not fields as in this work) to prevent data-races. Uniqueness is supported in a few special situations, such as tree-like structures. For the most part, however, the ownership type system prevents object transfer. It appears that some kinds of transfer could be added to their system using “external uniqueness” [6].

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

5 Conclusions

In this paper, we describe how we can use the ideas of “adoption and focus” to design a type system that takes into account effects and uniqueness in a unified

manner. Conditional permissions permit one to express null pointers without tagged unions. Field adoption allows us to express data groups. We show how high-level annotations (method effects and pointer annotations) can be expressed in this system. Although the work is not fully formalized, especially memory consistency, and we doubt that complete algorithmic type inference is possible, we expect that a practical algorithmic inference system can be defined that will check stylized uses of the type system, such as that used in our realization of annotations.

Acknowledgments

I thank Aaron Greenhouse, Bill Retert and Tim Halloran for their useful comments and corrections on this paper. I also thank Manuel Fähndrich, Dave Clarke and Jan Vitek for conversations that led to clarifying my ideas.

References

1. Boyland, J.: The interdependence of effects and uniqueness. Paper from Workshop on Formal Techniques for Java Programs, 2001 (2001)
2. Hogg, J.: Islands: Aliasing protection in object-oriented languages. In: OOPSLA'91 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications, New York, ACM Press (1991) 271–285
3. Baker, H.G.: ‘Use-once’ variables and linear objects—storage management, reflection and multi-threading. *ACM SIGPLAN Notices* **30** (1995) 45–52
4. Minsky, N.: Towards alias-free pointers. In Cointe, P., ed.: ECOOP'96 — Object-Oriented Programming, 10th European Conference. Volume 1098 of Lecture Notes in Computer Science., Berlin, Heidelberg, New York, Springer (1996) 189–209
5. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA'02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications, New York, ACM Press (2002) 311–330
6. Clarke, D., Wrigstad, T.: External uniqueness. In Pierce, B.C., ed.: Informal Proceedings of International Workshop on Foundations of Object-Oriented Languages 2003 (FOOL 10). (2003)
7. Leino, K.R.M., Stata, R.: Virginty: A contribution to the specification of object-oriented software. *Information Processing Letters* **70** (1999) 99–105
8. Leino, K.R.M., Nelson, G.: Data abstraction and information hiding. Research Report 160, Compaq Systems Research Center, Palo Alto, California, USA (2000)
9. Leino, K.R.M., Poetzsch-Heffter, A., Zhou, Y.: Using data groups to specify and check side effects. In: Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation, New York, ACM Press (2002) 246–257
10. Clarke, D.: Object Ownership and Containment. PhD thesis, University of New South Wales, Sydney, Australia (2001)
11. Boyland, J.: Alias burying: Unique variables without destructive reads. *Software Practice and Experience* **31** (2001) 533–553
12. 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

13. Boyland, J.: Checking interference with fractional permissions. In Cousot, R., ed.: *Static Analysis: 10th International Symposium*. Volume 2694 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, New York, Springer (2003) 55–72
14. Leino, K.R.M.: Data groups: Specifying the modification of extended state. In: *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, New York, ACM Press (1998) 144–153
15. Xi, H.: *Dependent Types in Practical Programming*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA (1998)
16. Fähndrich, M., Leino, K.R.M.: Heap monotonic typestates. In: *Informal Proceedings of “International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)”*, Utrecht University, Netherlands (2003)
17. Bruce, K.C., Cardelli, L., , Pierce, B.C.: Comparing object encodings. In: *Theoretical Aspects of Computer Software*. Volume 1281 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, New York (1997) 415–438
18. Fähndrich, M., DeLine, R.: Declaring and checking non-null types in an object-oriented language. Submitted to *OOPSLA 2003* (2003)
19. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, New York, ACM Press (2002) 1–12
20. Fähndrich, M., DeLine, R.: Adoption and focus: Practical linear types for imperative programming. In: *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, New York, ACM Press (2002) 13–24
21. 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
22. 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
23. Tofte, M., Talpin, J.P.: Implementation of the typed call-by-value λ -calculus using a stack of regions. In: *Conference Record of the Twenty-first Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, New York, ACM Press (1994) 188–201
24. Walker, D., Crary, K., Morrisett, G.: Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems* **22** (2000) 701–771
25. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: *OOPSLA'02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, New York, ACM Press (2002) 211–230
26. Boyapati, C., Salcianu, A., Beebe, W., Rinard, M.: Ownership types for safe region-based memory management in real-time java. In: *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, New York, ACM Press (2003) 324–337