A CONSERVATIVE TYPE SYSTEM BASED ON FRACTIONAL PERMISSIONS

by

Chao Sun

A Dissertation Submitted in
Partial Fulfillment of the
Requirements for the degree of

Doctor of Philosophy
in Engineering

at
The University of Wisconsin-Milwaukee
May 2016

ABSTRACT

A CONSERVATIVE TYPE SYSTEM BASED ON FRACTIONAL PERMISSIONS

by

Chao Sun

The University of Wisconsin-Milwaukee, 2016

Under the Supervision of Professor John Boyland

The system of fractional permissions is a useful tool for giving semantics to various annotations for uniqueness, data groups, method effect, nullness, etc. However, due to its complexity, the current implementation for fractional permissions has various performance issues, and is not suitable for real world applications.

This thesis presents a conservative type system on top of the existing fractional permission type system. The system is designed with high-level types, and is more restrictive. The benefit is that it can run much faster. With this system, we propose a multi-tiered approach for type checking: the conservative type system is first applied, and only those that it cannot handle will then be processed by the more powerful fractional permission system.

A crucial property about a type system is its soundness. In this thesis we also present a mechanized proof, written in Twelf, for the conservative type system. A mechanized proof is checked by computer, and offers much more confidence about its correctness. Moreover, we proved the soundness property with a novel approach: instead of defining the semantics of the language and proving progress and preservation directly, we delegate it to the soundness proof of the fractional permission system.

The novel technical features in this thesis include: 1) a multi-tiered approach for type checking and a conservative type system build on top of fractional permissions; 2) a mechanized proof for the type system, and 3) a novel way of proving soundness property for a type system.

Preface

To my parents and my dear wife Ruijiao

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank everyone who helped me to go through this journey.

Chapter 1

Introduction

1.1 Aliasing

One common safety issue in imperative programming languages is *aliasing*, which happens when a data location in memory is accessed through different symbolic names (or *variable*s) in program. With the presence of aliasing, it is difficult to reason about a program's behavior, because write action on a memory location through one variable may affect read action by others.

Even worse, the write action could happen at a totally unrelated point in the program, which could make reasoning even harder. Also, in a language without garbage collection, where objects need to be deallocated explicitly, uncontrollable aliasing can cause two problems: *dangling reference* and *memory leaks*. The former is caused by deallocating a memory block too early while some pointers to it still alive in the system; the latter is caused by deallocating a memory block too late which depletes the memory available to the program.

In addition, aliasing can have serious affect on *information hiding* and *encapsulation*,

which are essential elements in object-oriented programming. For instance, a private member of an object may have aliases outside the scope, therefore even though the intention is to disallow access to this member from outside the class, it can still be done so through the aliases. Modern software is often required to offer *implementation transparency*, which means the ability to change internal implementation without affecting the rest of the system.

1.2 Annotation

To resolve the issue of aliasing, many researchers have suggested using *annotations* [Eva96, LLP+00, FLL+02, BLS05, HLL+12]. Unlike program types, which are "hardcoded" in the language, and mainly concern low-level semantics, annotations are more about the high-level program behaviors, like the fields that a method may modify, or whether a variable is not null. Annotations usually will not change the runtime behavior, and for this they can serve two purposes: implementors can attach their *design intent* to the program, for better understanding, and maintainers can use them to extract more semantic information, for better analysis of the program. In general, annotations enable a component supplier to offer *contracts* in which certain *demands* are described, and clients are supposed to follow these requirements, to guarantee the result of execution meets the expectation.

Another reason of using annotations is their flexibility. Rather than design a new language and add all the desired features, people can deploy annotations as an *optional type system* [FFA99, Bra04]. Existing languages can be improved in this way without their essential elements altered. Compiler and run-time system also do not need to be modified.

Fractional Permissions [Boy10b, Boy03, BR05, BRZ09], originating from separation logic [Rey02, OYR04], provide a general tool for managing access to mutable state. Under this framework,

each piece of mutable state is associated with a whole permission, which is required when one needs to modify the state. The whole permission, nevertheless, can be split into multiple fractions, and each fraction can be used for reading the state. The fractions can also be recollected and joined back, to restore the whole permission for writing the state.

Because of its expressiveness, Fractional Permissions can be used to give precise semantic meaning to most annotations concerning mutable state, such as `unique`, `readonly`, effects and data groups [BRZ09]. Not only that it can help us to better understand these annotations, but it also provide a foundation for the implementation to be built upon.

Although Fractional Permissions possess great expressive power, the tradeoff is its complexity. To fully implement it on a practical programming language like Java is rather difficult. The current implementation of Fractional Permissions [Ret09], which is built on Fluid project [GHS03], is very complex, and has various performance issues, even though it only handles part of the system. For instance, to model a base permission:

$$\xi(o.f \to o')$$

it needs lattices for both location and fraction, and has two separate maps for them. To model Java evaluation at a low-level, the transfer function needs to simulate stack operations, and therefore a stack lattice in which elements are of some other base lattices is used. One side-effect for this is all the primitive types in Java, such as `int` and `string` need to have their lattice representations too. Besides, along the control flow, the analysis also needs to collect various "facts", like the equality (inequality) between object locations, as well as nesting situations.

Because of the massive information that needs to be represented, and the complexity of operations (especially the join operation upon control flow merge), the analysis has high

runtime overhead and large memory footprint. This makes it not so practical to use on real world programs [Ret09].

```
class Node {
  @InRegion("Instance")
  @Unique Node next;
}

class List {
  @Unique Node head;

  @RegionEffect("writes head")
  void prepend(@Unique Node n) {
    n.next = head;
    head = n;
  }
}
```

Figure 1.1: Sample Annotated Code in Fluid

1.3 A Multi-Tiered Approach

To make annotation checking more efficient, instead of applying the heavyweight checker on the input program directly, one solution is to first apply a more "conservative" type system. The conservative type system and the corresponding implementation should run much faster, albeit with less precision. Instead of encoding fractional permission directly, it uses much higher-level types. The fractional permission type system, can then serve as a foundation

for the new type system. With this approach, it gives us two benefits: we can have better understanding of semantics of annotations, using fractional permission, and therefore derive better type system to check them; we can build the soundness proof of the conservative type system directly on that of fractional permissions.

The conservative type system identifies and accepts those "obviously correct" cases. Inside a program, some methods may contain obvious errors that should be easy to detect. However, these methods will fail both conservative checker and permission checker, as both of them only allow correct methods to be passed. Therefore, this situation generates even worse performance compared to the old way. To solve this, one approach is to use some "liberal checker", which detect those "obviously wrong" cases. For instance, storing a `borrowed` method parameter into a `unique` field, or not returning the effects passed to a method, etc.

By using adopting this "multi-layered" approach, type checking can be much more efficient; the majority of an input program should be checked rather quickly by using the liberal checker and conservative checker, and one should rarely need to use the heavyweight permission checker.

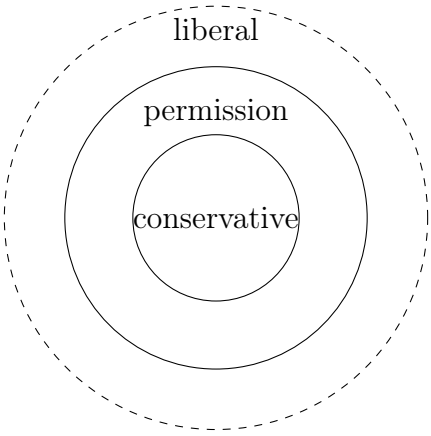

Figure 1.2: Multi-layered Type Systems

1.4 Proof of Correctness

For any type system, soundness is one of its important properties. For the conservative type system, we provide mechanized proofs for its soundness. Compare to hand-written proofs, mechanized proof takes much more effort to write, since the proof is checked by machine, which requires precise definitions and theorems, and checks every possible cases. On the flip side, a machine-checked proof gives one much greater confidence about its correctness. Also, a mechanized proof is easier to maintain and update [Chl10]. When some change needs to be done on an existing proof, the proof system usually tells one about all the relavent changes that need to be made.

The proof for the conservative type system described in this dissertation is written in Twelf [PS99, PS02], which is an implementation of LF logical framework [HHP93] that is especially useful for proving properties of programming languages and logics.

In addition, we adopted a novel approach for proving the soundness. Instead of following a tranditional approach, which is to prove the preservation and progress of the system, we reduce the proof for the system to the soundness proof for fractional permissions [Boy10a, BS11], which is also written in Twelf. To achieve this, for each term in the language, we show that if it is well-typed under the conservative type system, then it is also well-typed under the permission system, and since the latter is already proven sound, this shows the former is sound too. With this piggy-packing approach, we avoid the need to prove progress and preservation directly, and the dynamic semantics of the language is also separated from the type system.

In this thesis we present the conservative type system that is built upon the fractional permission logic. Also, we demonstrate how the sematics of the conservative type system

can be converted to the corresponding pieces under fractional permission system, and how to prove the correctness of the former by reduction to that on the latter.

1.5 Outline

The remainder of this thesis is organized in the following sections. In Chap. 2, we introduce related work in the area, and compare them with the approach that we are going to use. In Chap. 3, we introduce some basic concepts of fractional permission system. In Chap. 4, we introduce a pilot study that we have done, that is, proving the soundness of a simple non-null type system by reducing it to fractional permission system. In Chap. 5, we formalize the conservative type system, although without the proof. In Chap. 6, we show how the components in the conservative type system can be transformed to those under fractional permission system, and how the soundness theorem for the former can be proved by using that of the latter. In Chap. 7, we discuss some of the difficulties we encountered while writing the proof, and possible directions for future work. In Chap. 8, we conclude the thesis.

Chapter 2

Related Work

In this section we introduce some related work in the literature.

2.1 Linear Types

The notion of uniqueness build its foundation on linear types [Wad90], which in turn come from one branch of substructural logic called *linear logic* [Gir87]. Linearity provides a powerful tool for reasoning about aliasing, since a linear reference is effectively the sole reference to the object it points to, and thus one is assured that there would be no aliasing for the object. However, it also means restrictions on how meaning can be represented in a program. First, a linear value can only be used once, and therefore to preserve semantic meaning, a program's structure has to be changed dramatically. Second, the pure distinction between linear and non-linear type makes them hard to co-exist in a same data structure, which makes linear type system impractical to use in large software systems.

Hogg's "islands" and Minsky's "unique"[Hog91, Min96] are earliest attempts to incorporate uniqueness into an object-oriented language. These works use *destructive read* to handle

usage of unique variable. That is, a unique variable is set to `null` immediately after it is used, also called *consumed*. This approach guarantees every unique object can only have one reference at any time, with the cost of making programming awkward and more complex [Boy01a]. Also, for formal parameters, it's not desirable to always consume them. In many situations one simply wants to use a unique variable without breaking the uniqueness. For this case, Hogg and Minsky use the notion of *borrowed* for those parameters. Inside a procedure body, a borrowed parameter is not allowed to be stored in a field.

Within the above work, borrowing essentially weakens the uniqueness invariant, which then makes preserving the invariant harder. This is especially true when there are complex data dependencies. Alias burying [Boy01a, BNR01] suggests the nullification of the unique variable can be delayed as long as no alias is read. And, whenever a unique field is read, all other aliases to this field are then become *undefined*, and not usable anymore. Also, since aliasing is a global effect, to enable intra-procedural analysis, borrowed annotation is used to grant temporary access on a unique variable. Similar to the previous work, inside a procedure body, a borrowed parameter cannot be stored in any field and thus cannot be further aliased. The advantage of alias-burying is that instead of modifying the compiler, a separate static check can be used to check that uniqueness is preserved. A static object-oriented effect system [GB99] may be used to verify the effects each method imposed on unique variables. Boyland also described the inter-dependence between uniqueness and effects [Boy01b], which reasons about the need to unify both of them inside one system, as was done later in the fractional permissions system [Boy03, BR05].

Other attempts [CWM99, WCM00, SWM00, WM01] have also been made to apply linear type on low-level programming languages. Different from unique annotations, which use either a set of rules or static analysis to *prevent* alias, their approach is to define a type

system to *track* alias. In their proposal, linear type is represented through two parts: a singleton type $ptr(\ell)$ is given to a pointer to the location ℓ, and a set of *aliasing constraint* on the heap. A type system has been defined and proven sound.

The above approaches have been introduced into high-level languages by Deline and Fähndrich, in their Vault system [DF01], in which they use *tracked types* and *guards* to track the life span of allocated objects. Specifically, when an object is newly allocated, a fresh key is bound to it, and when the object is removed from memory, the key is released. All the currently available keys are put into a *holder-key set*, which is in turn looked up when a operation is to be performed on a certain object. In this way, it can model the run-time behavior through compile-time entities. As in alias types from Smith, Walker and Morrisett [SWM00], aliasing is tracked through type system. To address the problem that a linear type structure cannot exist inside a non-linear structure, Fähndrich uses "adoption and focus" [FD02] which allows linear types to be nested inside a non-linear structure. A linear value can be adopted by another linear value, and after which it is assigned a guarded type, indicating that is can be shared via multiple references. Whenever one needs the linear fact about certain variable, a focus operation is performed, and during a certain lexical scope, the value can be treated as linear.

Boyland's fractional permission system [Boy03, BR05, BRZ09, Boy10b] is another capability-based approach. The system incorporates the idea of "adoption and focus" into a logic structure which closely resembles separation logic [IO01, Rey02]. Separation logic enables one to reason of a heap through reasoning about partial heaps. Fractional permission offers power semantics, and can be used to interpret the semantic meanings of various common annotations, like `unique`, `owned`, `borrowed`, etc. Since this thesis is closely related to fractional permission system, we shall introduce it in Chap. 3.

There are more work based on the idea of permissions. Bierhoff and Aldrich developed a modular typestate checking tool [BA07] based on a concept called *access permissions*, which is high-level abstraction of fractional permissions and is used to capture different patterns of them. Later, Naden, Bocchino, et al. presented a language and type system [NBAB12] based on the idea of *borrowing*. Their type system covers different types of permissions such as unique, none, immutable, local immutable, etc. A particular kind of permission called "local permission" is supported by this system, which is useful for permission splitting and combining. The system also supports "changing permissions", which is part of a method's contract, and is similar to the input and output permissions specified for a method type in the fractional permission system.

Based on separation logic, Parkinson and Bierman built formalism for programmers to write specifications for object-oriented languages such as Java [PB05, PB08, PB13], using a notion called abstract predicate. These can be used for properties about abstract datatypes (ADTs), class hierarchies with inheritance and method overriding, and so on. However, verification for these properties still needs to be done by hand. Also based on separation logic, Smallfoot [BCO06] is a automatic verification tool that checks specifications for both sequential and concurrent programs that manipulate recursive data structures. jStar [DPJ08] is another automatic verification tool based on the notion of abstract predicate and can be used to check specifications for Java-like languages.

2.2 Ownership Types

Another important technique of handling aliasing is *ownership types* [CPN98] , which, instead of either forbidding or tracking alias, tries to *confine* [BV99] the aliases to a certain scope.

This is especially useful when coping with large-scale software system, since it allowes one to reason about one module at a time, independent of the others. In fact, the idea of restricting the scope of alias is similar to adoption in Adoption and Focus [FD02], in which the adopter of a linear value can be seen as its owner. This idea is further made explicit in fractional permissions [BR05], where ownership relation is established by *nesting* a linear value to a special `Owned` region in its owner.

The main works on ownership types can be divided into two categories, namely "owners-as-dominators" and "owners-as-modifiers". In below, we shall introduce them in order.

Clarke, Potter and Noble first propose to use ownership type [CPN98] as a way to localize aliasing. In their work, each object owns a *context*, and is itself residing in some other object's context. A context of object o can be thought of a set of objects that are nested inside o. This approach establishes a partition on the run-time store, and enable one to speak of an object's interior and exterior.

When an object is first created, it is initialized with a owner object, this is implemented by augmenting class declaration with *ownership parameters*. The default owner is `world`, which is also the outermost context. With `world` as root, all objects in the system form a tree structure, based on which reference access is restricted. In specific, an object can only access all objects in its own context (that is, all the objects nested in its context), alongside with its peer objects (that is, other objects owned by its owner). In other words, an object cannot be accessed by any object outside its owner.

The above condition is also referred as "owners-as-dominators". Basically, each owner can be seen as the *dominator* for all the object in its context, and the ownership structure guarantees that every access path from root to an object must contain its owner as one of the node [Cla01].

Clarke further extend the above work by combining a ownership type system with an effect system [CD02]. Similar to the object-oriented effect system from Greenhouse and Boyland [GB99], it enables modular reasoning about object-oriented programs, by checking effects generated by individual method against its annotation.

However, this "owners-as-dominators" approach also has its downside. One particular problem occurs when one wants to implement design patterns such as iterator [BRZ07]. Basically, an iterator needs to be both inside and outside of the collection it represent. However, this is hard to implement with the "owners-as-dominators" approach, as it has to be *either* inside or outside, to preserve the ownership structure. One proposal is to use an inner class to provide special privilege for the objects in the same module [BSBR03], even though the object of inner class may not be owned by outer object. The idea of using inner class has been further developed into Tribal Ownership [CNW10], where *families* of classes are used solely to define ownership structure. The advantage of this is that the burden on programmer to add ownership annotations is totally eliminated. The author also described how to implement the traditional "owners-as-dominators" and "owners-as-modifiers" mechanism in this system.

Different from "owners-as-dominators", one may allow an object be referenced by another object, as long as the other object doesn't *modify* it. This approach is called "owners-as-modifiers" [MPH00]. A special `any` annotation is used in this approach, which is similar to `readonly` [MPH00, KT01]. It is especially useful in the iterator design pattern, in which the iterator can refer to a collection's internal elements as long as it doesn't modify them. Universes [DM05] extends the above system with Java-like generics [BML97]. Müller and Rudich further extend their work by enabling ownership transfer between different objects [MR07].

Ownership Domains [AC04] is a further attempt on acquiring a balance between safety

and expressiveness. Unlike the work described above, which impose a fixed structure on the objects, Ownership Domain separates the aliasing policy from ownership mechanism. Specifically, inside an object, multiple *domains* are declared to represent different encapsulation level on the objects. A *link* may be established between domains to grant one access to the other.

2.3 Others

Clarke and Wrigstad combines both uniqueness and ownership type with a concept called *external uniqueness* [CW03]. The key idea is that, to ensure an object's uniqueness, it is enough to only guarantee one reference from outside to it, while multiple references from inside the object to itself are allowed. This is useful, for example, when one wants to assign `this` to some other internal structure. With destructive reads, `this` is unique, and therefore will be consumed after use. With external uniqueness, multiple references to `this` can co-exist since they are all internal to the object. A unique reference may be moved to another scope, or be *borrowed*. In the former case, a *movement bound* is specified to ensure the ownership structure is not broken after moving. In the latter case, destructive read is applied first, and then the final contents of the borrowed variable are restored after the borrowing is done. Haller and Odersky later designed and implemented a simple type system which uses capabilities to model uniqueness and borrowing [HO10]. Their system offers a slightly more strict version of external uniqueness, that is more suitable in a setting of message-based concurrency. Atomic operations are offered for transfering unique values and merging unique values.

External uniqueness offers a good unification between uniqueness and ownership types.

However, it is still suffers from restriction on the *owner-as-dominator* structure. For instance, a collection may have multiple iterators linked to it. This is hard to represent in external uniqueness as only one reference from outside is allowed.

An effect system [Luc87] also provides another important tool for handling aliasing [GB99]. Boyland discussed the inter-dependence between uniqueness and effects, and reasoned about the necessity of unifying both inside one system, as was done in his later works [Boy03, BR05]. In a different direction, Clarke and others combine ownership type and effects [CD02], which is similar to the work before, with the addition of annotating effects on owner targets.

Lu and Potter designed a type system [LP06] based on *effect encapsulation*, instead of object encapsulation. In addition of ownership parameters on fields, methods are annotated with *effective owners*, which describe what fields they are allowed to generate effects for. Compared with fractional permission discussed above, it is more restrictive, with the advantage that less annotation is needed.

2.4 Difference of Our Work

Our work, as shall be introduced later, is very different from all the above work, in the following aspects:

First, while all the type systems proposed above are independent, and standalone, ours is directly built on top of another more powerful one, i.e., the fractional permissions system. The base system provides a foundation for us to explore the semantic meanings of various annotations, and use this information to design more efficient type system. Second, most of the previous works use natural language proofs. Instead, we provide mechanized proofs for our type system, which can provide much more confidence for correctness. Moreover, one

distinguishable feature is the way we used to prove the soundness: we reduce the proof of the simpler system to that of a more powerful system. This approach, as far as we know, is new for proving type soundness of an effect system.

Third, all the previous work only use a single type system for checking the program. Instead, we use multiple type systems for this purpose. By making the process layered, we believe the type checking can be much more efficient, as most parts of the program should be handled rather quickly by the lightweight checker, and only a small part of the program needs to utilize the heavyweight permission checker.

Chapter 3

Fractional Permissions

In this section we introduce the Fractional Permission system. For a more formal presentation, readers are encouraged to [Boy03, BR05, BRZ09].

In Sec 3.1, we introduce permission logic with nesting. In Sec 3.2, we introduce a Java-like kernel language for reasoning about concurrent imperative programs. In Sec 3.3, we introduce a permission type system for the kernel language, and how the type system is proved using mechanized proofs. In Sec 3.4, we discuss the existing implementation based on permission type system, and its limitations.

3.1 Permission Logic

Fractional permissions are used to manage access on mutable states. Generally speaking, each piece of mutable state is associated with a fraction number in the range of $(0, 1]$, with 1 representing the full access, including both read and write, to the state. Any other fraction, no matter how small it is, represents only read access to the state.

The most basic permission is a field permission:

$$o.f \to o'$$

This represents the full access to field f of object o. Additionally, it gives information that f currently refers to object o'. Permissions (in contrary to *formulae*, which will be introduced below) are *linear*: one can only scale permissions, but not duplicate them. Otherwise, a change of field value on one permission would invalidate the other.

To obtain "fractional permission", one can scale any permission by a positive number. For instance, from the above we have:

$$o.f \to o' \equiv \frac{1}{2}(o.f \to o') + \frac{1}{2}(o.f \to o')$$

This can roughly be read: a whole permission for field f in object o is equivalent to two half permissions for it. In general, a read permission has the form $\xi(o.f \to o')$, while $0 < \xi \leq 1$. Sometimes, the object and the fraction may not yet be known, in which case we shall use variable r and z to represent them respectively. For instance, a permission $zr.f \to r'$ represents a read permission for the field $r.f$, although the object r and r', as well as the fraction z, remain unknown.

Notice the "+" operator above: permissions can be combined together, and a compound permission give one both accesses through its component permissions.

In many cases we may need to pack a base permission to obtain its existential form (for instance, as part of a class invariant). This is written:

$$\exists r \cdot (\rho.f \to r + \Pi) \quad \text{where } \rho \neq r$$

where Π is a permission may or may not use r. One example for the above general form is $\exists r \cdot (o.f \to r + r.g \to o')$. This gives write access to $o.f$, where the field points to an object r for which we have access to its field g, which currently points to another object o'.

$\rho ::= o \mid x$	*literal reference, variable*	$\Gamma ::=$			*formula:*
$\xi ::= q \mid z$	*literal fraction, variable*		$\top$		*true*
$k ::= \rho.f$	*field f of object at ρ*		$\neg\Gamma$		*negation*
			$\Gamma \wedge \Gamma$		*conjunction*
$\Pi, \Psi ::=$	*permission:*		$\rho = \rho$		*comparison*
v	*variable*		$\Psi \prec k$		*nesting*
$k \to \rho$	*field*		$p(\overline{X})$		*predicate call*
$\emptyset$	*empty*		$\exists x \cdot \Gamma$		*existential*
$\Pi + \Pi$	*combined*				
$\xi\Pi$	*scaled*				
Γ	*formula*	$x ::= r \mid z \mid v$			*any variable*
$\exists r \cdot (k \to r + \Pi)$	*existential*	$X ::= \rho \mid \xi \mid \Pi$			*any variable value*
$\Gamma \,?\, \Pi : \Pi$	*conditional*	$P ::= \left\{ \overline{p(\overline{x}) = \Gamma} \right\}$			*predicate definitions*
$\Psi \multimap \Pi$	*implication*	$\sigma ::= \left[\overline{x \mapsto X} \right]$			*substitution*

Figure 3.1: Syntax of Permissions and Formulae.

A "null" pointer exists in many systems (albeit refereed to as a "billion-dollar mistake" by C.A.R Hoare, its inventor) as a pointer to no object. In permission logic, we represent a null object using the special symbol 0, and use formula $r = 0$ to represent the fact that object r is null. Since accessibility only makes sense to those non-null references, we use *conditional* permission to represent this:

$$r = 0 \,?\, \emptyset : r.f \to o$$

If r is null, we have no permission ($\emptyset$). Otherwise we have the write permission for field f

of r. In general, the conditional part here can be any formula, which is defined by grammar in Fig. 3.1.

Before introducing the last form of permission, we shall take a break and introduce the various forms of formula first. In the permission logic, formula represents "facts" that are known to be true. Most of the formulae in the system are standard, except nesting relation, which is written:

$$\Pi \prec \rho.f$$

This means the permission Π is available whenever one has permission for accessing field $\rho.f$. Nesting is useful for modeling ownership and abstraction: a permission such as $o.\mathrm{All} \rightarrow 0$ expresses the direct access to the "All" field, which is a default data group [Lei98, GB99, LPHZ02], of object o. Also, it (indirectly) grants permissions which are nested inside "All" field. Hence, all the other permissions are treated as if they are "owned" by the permission for "All" field, which is similar to the idea of "adoption and focus", proposed by Fähndrich and Deline [FD02]. Another default data group is "Owned", which is for express the idea of object ownership [CPN98]. A permission $o.\mathrm{Owned} \rightarrow 0$ gives access to the data group which contains all objects that o owns. The "Owned" data group is also nested inside "All" data group.

The last form of permission is permission implication $\Pi_1 \multimap \Pi_2$, which, if given Π_1, can be combined and produce permission Π_2. For this case, we call Π_1 is *encumbered* in permission Π_2. The permission itself is not useful, except recording the information that Π_1 is "carved" out Π_2, and need to be restored by applying the linear modus-ponens rule before Π_2 can be reused.

$$
\begin{array}{rcl}
e & ::= & o \mid x \mid \texttt{new } (\overline{f}) \mid e\texttt{.}f \mid e\texttt{.}f\texttt{:=}e \mid \texttt{let } x\texttt{=}e \texttt{ in } e \mid \\
 & & \texttt{if } c \texttt{ then } e \texttt{ else } e \mid \texttt{while } c \texttt{ do } e \mid m(\overline{e}) \\
c & ::= & \texttt{true} \mid \texttt{not } c \mid c \texttt{ and } c \mid e \texttt{ == } e
\end{array}
$$

$d ::= m(\overline{x}) \texttt{ = } e$ *procedure definition*

$g ::= d\texttt{; . . .;}d$ *program*

$\mu ::= \left[\overline{o \mapsto [\overline{f \mapsto o}]}\right]$ *memory*

Figure 3.2: Syntax of Kernel Language (omitting concurrency).

3.2 Kernel Language

In this section we describe a Java-like kernel language for reasoning about concurrent imperative programs.

The kernel language is first described in [Boy09]; Fig. 3.2 shows the single-threaded subset of this language. We also intend to handle the full multi-threaded feature of this language, but this still remains a future work at this time.

The expressions, in order, are object literals, variables, allocation, field reads and writes, local bindings, procedure calls, conditionals and loops. The language has separate syntactic kinds for expressions and conditionals, although since equality tests include expressions, side-effect are possible in conditionals as well as in expressions. The language does not include primitive arithmetics and dynamic dispatch, as neither of these affect aliasing or threading.

Evaluation is of the form:

$$(e; \mu) \rightarrow_g (e'; \mu')$$

which reads: a expression e, in context of memroy μ, can be evaluated to expression e', with a (perhaps) changed memory μ', under "program" g, which is a set of procedure definitions.

$\alpha ::= \forall_{\overline{r}} \forall_{\Delta}\ \Pi \rightarrow \exists_{r_0} \exists_{\Delta'}\ \Pi'$ *procedure type*

$\omega ::= \{\overline{m \mapsto \alpha}\}$ *program type*

Figure 3.3: Procedure and Program Types

Ignoring concurrency, evaluation of a thread can only get stuck if one of the following cases happens:

1. Calling a non-existent procedure;
2. Calling a procedure with the wrong number of parameters;
3. Reading or writing a field on an object not (yet) allocated;
4. Reading or writing a field on an object that does not have it;

3.3 Permission Type

In this section we briefly introduce a permission type system [BS11] built on the kernel language.

Procedures in programs are assigned procedure types (see Fig. 3.3); these are used to check if a program is well-typed. A procedure type has universally quantified variables for its input variables (including a distinguished series of object variables for parameters) in the input permission Π, and existentially quantified variables for the output variables (including one distinguished variable for result value) that may additionally appear in the output permission Π'.

The typing rules are shown in Fig. 3.4 and Fig. 3.5. The relation $(\Pi \vdash_{\omega} e \Downarrow \rho \dashv \Delta'; \Pi')$ reads: in the environment $E = (\emptyset; \Pi)$, the expression e will (if it terminates) evaluate to

$$\textsc{Literal}\quad \Pi \vdash_\omega o \Downarrow o \dashv \emptyset; \Pi \qquad \textsc{Alloc}\quad \frac{}{\Pi \vdash_\omega \texttt{new}\ \ (\overline{f}) \Downarrow r \dashv r; \overline{r.f \to 0} + \Pi}$$

$$\textsc{Let}\quad \frac{\Pi_1 \vdash_\omega e_1 \Downarrow \rho_1 \dashv \Delta_2; \Pi_2 \quad \forall_{\Delta_2}\ \Pi_2 \vdash_\omega [x \mapsto \rho_1]e_2 \Downarrow \rho' \dashv \Delta'; \Pi'}{\Pi_1 \vdash_\omega \texttt{let}\ \ x{=}e_1\ \ \texttt{in}\ \ e_2 \Downarrow \rho' \dashv \Delta_2, \Delta'; \Pi'} \qquad \textsc{Read}\quad \frac{\Pi \vdash_\omega e \Downarrow \rho \dashv \Delta'; \xi\rho.f \to \rho' + \Pi'}{\Pi \vdash_\omega e\,\texttt{.}\,f \Downarrow \rho' \dashv \Delta'; \xi\rho.f \to \rho' + \Pi'}$$

$$\textsc{Write}\quad \frac{\Pi_1 \vdash_\omega e_1 \Downarrow \rho_1 \dashv \Delta_2; \Pi_2 \quad \forall_{\Delta_2}\ \Pi_2 \vdash_\omega e_2 \Downarrow \rho_2 \dashv \Delta'; \rho_1.f \to \rho' + \Pi'}{\Pi_1 \vdash_\omega e_1\,\texttt{.}\,f{=}e_2 \Downarrow \rho_2 \dashv \Delta_2, \Delta'; \rho_1.f \to \rho_2 + \Pi'}$$

$$\textsc{Cond}\quad \frac{\Pi \vdash_\omega c \Downarrow \Gamma \dashv \Delta_0; \Pi_0 \quad \forall_{\Delta_0}\ \Gamma + \Pi \vdash_\omega e_1 \Downarrow \rho' \dashv \Delta'; \Pi' \qquad \forall_{\Delta_0}\ \neg\Gamma + \Pi \vdash_\omega e_2 \Downarrow \rho' \dashv \Delta'; \Pi'}{\Pi \vdash_\omega \texttt{if}\ \ c\ \ \texttt{then}\ \ e_1\ \ \texttt{else}\ \ e_2 \Downarrow \rho' \dashv \Delta_0, \Delta'; \Pi'}$$

Figure 3.4: Permission Typing Relations Part 1

an object reference or variable ρ in the new environment $E' = (\Delta'; \Pi')$, where Δ is a set of variables.

In a slight abuse of notation, we write $\forall_\Delta D$ as a short-hand for $\forall_{\sigma:\Delta\to\emptyset}\sigma D$: it means that D should be true no matter what binding the variables in Δ have.

In Fig. 3.4, the LITERAL rule types an object (o) as itself , and returns the same set of permission (Π) with no new variables ($\emptyset$).

The ALLOC rule allocates an object with the given set of fields and in the result adds a unit permission for each field of the new object which is assigned the name r in the context.

The rule LET first types e_1, and then use the result to type e_2. When typing e_2, all occurrences of variable x are replaced with the result value from typing e_1. The result variables are the union of output variables from both e_1 and e_2.

Rule READ specifies how reading a field is typed. In this case, the expression e needs to be well-typed, and the result permissions must contain the permission for reading the field ($\xi\rho.f \to \rho' + \Pi'$).

Next, rule WRITE specifies how writing a field is well-typed. In this case, both expression e_1 and e_2 need to be typed, consecutively. Unlike READ, the result permissions for checking e_2 must contain the whole permission ($\rho.f \to \rho' + \Pi'$) for the field, since it is a write operation. The result value is the same value after typing e_2.

In rule COND, for a conditional expression to be typed, first the conditional part (c) needs to be typed. Then, to check both branches, the "then" branch can assume the condition is true, while the "else" branch can assume it is false. Both branches are required to produce the same result value, set of variables and set of permissions. This can be done through applying the TRANSFORM rule. Also, notice that here conditional expressions can include side-effects as well, so they also need to be evaluated to formulae, which can then be used

for checking the branches.

The second part of the rules are in Fig. 3.5. The rule LOOP types a while loop expression. It uses permission transformation to establish an invariant $(\Delta; \Pi)$, and also re-establish it after the body is evaluated in each iteration. Similar to COND, the conditional part is typed and the result formula is included in the input permissions for typing the loop body.

In rule CALL, to type a procedure call, first each of its actual parameters must be typed with the input permissions. Then, the procedure m is required to be well-typed. The permissions after typing all the arguments (Π_n) should be able to be splitted into the required permissions for typing the method body ($\sigma\Pi$) plus the rest of the permissions (Π''). The result permissions are the declared output permissions for the method ($\sigma\Pi'$) plus the permissions Π''. Note that here we need to substitute each actual parameter with the corresponding formal parameter, using σ.

The rule TRANSFORM specifies how to change the input and out permissions for typing an expression. This is useful for other rules such as COND and LOOP.

Lastly, there are rules for conditional expressions. Rule TRUE types the literal `true` to be the formula true. No change on permissions for this case. In rule NOT, the conditional expression c must be typed, and the result is the negation of its output formula. For AND, both sides of the expression must be typed consecutively with the input permissions, and the result formula is the conjunction of formulae after typing each of them. For EQ, expressions from both sides need to be typed consecutively and the result formula is the object equal relation between the result values for both sides.

Figure 3.6 defines a well-typed program: the body of each procedure can be typed using its procedure type.

Type soundness says that programs that permission check execute without errors. Execution depends on memory, but permission checking depends on permissions. The connection between the two is called "consistency" and depends on a "fractional heap" [Boy10b] to connect the two.

3.4 Implementation

In this section we briefly describe the current implementation of fractional permission on Java; a detailed introduction is in Retert's PhD thesis[Ret09]. The implementation is built upon Fluid project [GHS03] (see Fig. 1.1 for sample annotated code), which provides a framework for various program analyses. The tool does not let user to specify fractional permissions directly. Instead, annotations for uniqueness, data groups and method effects are supported on input programs. These are then translated to permission semantics and checked by the tool.

The permission analysis is implemented as a control-flow analysis. At bottom level, there are two basic lattices: one for locations and one for abstract fractions. A base permission $(\xi k \to \rho)$ is therefore implemented with two distinct mappings, one for each lattice, and a pair of map lattice represents the collection of all base permissions. Linear implication can also be implemented in a similar way, as a mapping from consequent to the set of all keys for permission carved out of it.

To simulate Java evaluation at low level, the transfer function also needs to handle stack operation. This is done by a stack lattice, whose elements are location lattices. The join operation is defined only between stacks of same height. Otherwise, the analysis will throw an exception.

LOOP

$$\frac{\emptyset; \Pi_1 \rightsquigarrow \Delta; \Pi \quad \forall_\Delta\ \Pi \vdash_\omega c \Downarrow \Gamma \dashv \Delta'; \Pi' \qquad \forall_{\Delta,\Delta'}\ \Gamma + \Pi' \vdash_\omega e \Downarrow \rho \dashv \Delta''; \Pi'' \qquad \Delta, \Delta', \Delta''; \Pi'' \rightsquigarrow \Delta; \Pi}{\Pi_1 \vdash_\omega \texttt{while}\ \ c\ \ \texttt{do}\ \ e \Downarrow 0 \dashv \Delta, \Delta'; \neg\Gamma + \Pi'}$$

CALL

$$\frac{\begin{array}{c}\Pi_0 \vdash_\omega e_1 \Downarrow \rho_1 \dashv \Delta_1; \Pi_1 \\ \forall_{\Delta_1}\ \Pi_1 \vdash_\omega e_2 \Downarrow \rho_2 \dashv \Delta_2; \Pi_2 \\ \vdots \\ \forall_{\Delta_1,\ldots,\Delta_{n-1}}\ \Pi_{n-1} \vdash_\omega e_n \Downarrow \rho_n \dashv \Delta_n; \Pi_n \\ \omega m = \forall_{r_1,\ldots,r_n} \forall_\Delta \Pi \to \exists_{r_0} \exists_{\Delta'} \Pi' \qquad \sigma : \{r_1, \ldots, r_n\} \cup \Delta \mapsto \Delta_1, \ldots, \Delta_n \qquad \forall_i\ \sigma r_i = \rho_i \\ \Pi_n = \sigma\Pi + \Pi'' \qquad (\Delta_1, \ldots, \Delta_n) \cap (r_0, \Delta') = \emptyset\end{array}}{\Pi_0 \vdash_\omega m\,(e_1, \ldots, e_n) \Downarrow r_0 \dashv \Delta_1, \ldots, \Delta_n, r_0, \Delta'; \sigma\Pi' + \Pi''}$$

TRANSFORM

$$\frac{\begin{array}{c}\emptyset; \Pi_1 \rightsquigarrow \Delta_2; \Pi_2 \\ \forall_{\Delta_2}\ \Pi_2 \vdash_\omega e \Downarrow \rho \dashv \Delta_3; \Pi_3 \\ \forall_o\ \Delta_2, \Delta_3; \rho = o + \Pi_3 \rightsquigarrow \Delta_4; \rho' = o + \Pi_4\end{array}}{\Pi_1 \vdash_\omega e \Downarrow \rho' \dashv \Delta_4; \Pi_4}$$

TRUE

$$\Pi \vdash_\omega \texttt{true} \Downarrow \top \dashv \emptyset; \Pi$$

NOT

$$\frac{\Pi \vdash_\omega c \Downarrow \Gamma \dashv \Delta'; \Pi'}{\Pi \vdash_\omega \texttt{not}\ \ c \Downarrow \neg\Gamma \dashv \Delta'; \Pi'}$$

AND

$$\frac{\Pi_0 \vdash_\omega c_1 \Downarrow \Gamma_1 \dashv \Delta_1; \Pi_1 \qquad \forall_{\Delta_1}\ \Pi_1 \vdash_\omega c_2 \Downarrow \Gamma_2 \dashv \Delta_2; \Pi_2}{\Pi_0 \vdash_\omega c_1\ \ \texttt{and}\ \ c_2 \Downarrow \Gamma_1 \wedge \Gamma_2 \dashv \Delta_1, \Delta_2; \Pi_2}$$

EQ

$$\frac{\Pi_0 \vdash_\omega e_1 \Downarrow \rho_1 \dashv \Delta_1; \Pi_1 \qquad \forall_{\Delta_1}\ \Pi_1 \vdash_\omega e_2 \Downarrow \rho_2 \dashv \Delta_2; \Pi_2}{\Pi_0 \vdash_\omega e_1\ \ \texttt{==}\ \ e_2 \Downarrow \rho_1 = \rho_2 \dashv \Delta_1, \Delta_2; \Pi_2}$$

Figure 3.5: Permission Typing Relations Part 2

$$\frac{\{\overline{r}\}, \Delta, \{r_0\}, \Delta' \text{ pairwise disjoint} \qquad \forall_{\Delta,\overline{r}}\ (\Pi \vdash_\omega [\overline{x \mapsto r}]e \Downarrow r_0 \dashv r_0, \Delta'; \Pi')}{\vdash_\omega \mathtt{m}\,(\overline{x})\ \ =\ \ e : \forall_{\overline{r}} \forall_\Delta\ \Pi \to \exists_{r_0} \exists_{\Delta'}\ \Pi'}$$

$$\frac{\overline{\vdash_\omega \mathtt{m}\,(\overline{x})\ \ =\ \ e : \omega\, m}}{\vdash_\omega \overline{\mathtt{m}\,(\overline{x})\ \ =\ \ e}\ \mathrm{OK}}$$

Figure 3.6: Well-Typed Procedures and Programs

Other than base permissions, formulae need to be represented too. In the implementation, there are three types of facts: object equality, object inequality, and nesting relation (some key is nested in another key). All these are kept in a set representing the conjunction. The join operation is intersection. For a `if` expression, the condition can be kept as a fact in the true branch, and its negation in the false branch. Both are dropped when the branches join. Disjunction is also implemented too, to represent facts like a location for a local variable may equal to multiple other locations. The downside of this, however, is algorithm efficiency; the number of disjoined elements is linear to the number of merges. Therefore, if program has complex control flow, the analysis sometimes will take 10-30 minutes to finish [Ret09]. Existential and conditional permissions are left out by the implementation.

Chapter 4

A Non-Null Type System

In this chapter, we will describe the type system and show how its soundness can be proved by reduction to fractional permissions. In Sec. 4.1, we discuss the motivation for design a nonnull type system based on fractional permissions. In Sec. 4.2, we introduce the nonnull type system. In Sec. 4.3, we show how the soundness of the nonnull type system can be proved.

4.1 Motivation

To show that we can indeed piggy-pack the proof of one type system onto that of another (in this case, the fractional permission type system), and also to get familiar of writing proof in the Twelf language. As a pilot study, I first designed a simple non-null type system, and proved its correctness using the piggy-packing approach [BS11]. The proof are all done in Twelf and can be checked under version 1.5R3. This is used as a basic foundation for our later proof for the more complex conservative type system.

4.2 The Type System

Our target language uses the same syntax as the kernel language defined in Chap. 3. However, since the latter is sequential, and we need constructor to establish the non-null invariant, several extra structures are defined, which shall be explained below.

To simplify the proof, in our non-null system, each class is encoded as a collection of fields, with exactly one constructor for initializing them. All the procedures are global. For constructor, the first parameter is the newly allocated object. We can also easily model methods as procedures by marking the first parameter implicitly as the receiver.

The typing rules for the non-null type system are shown in Fig. 4.1. As can be seen, each reference type in the system is augmented with additional information of whether the reference is *not null* or *possibly null*. Following Fähndrich and Leino [FL03], we denote the former with c^- and latter with c^+, while c is the class identifier. We also use the notation c^ε to denote a type where the nullness is represented by the variable ε. In addition, we use a special `Null` type for the reference whose value is exactly `null` (which we will use 0 to represent).

There are several environments used in the typing rules. A class map (C) is a map from class identifiers to their field maps, a procedure map (M) is a map from procedure identifiers to their types, a field map (F) is a map from field identifiers to their types, a (E) is a map from local variables to their types.

$$C ::= \epsilon \mid C, c : F \text{ (c is the class identifier)}$$

$$M ::= \epsilon \mid M, m : (c_1^{\varepsilon_1}, \ldots, c_n^{\varepsilon_1}) \rightarrow c^\varepsilon$$

$$F ::= \epsilon \mid F, f : c^\varepsilon$$

$$E ::= \epsilon \mid E, x : c^\varepsilon$$

We denote $E(x) = c^{\varepsilon}$ for the same meaning as $x : c^{\varepsilon} \in E$. This also applies to C, M, and F. The lookup for E always start from the rightmost element. Also, in a slight abuse of notation, we use $C(c)(f) = c_f^{\varepsilon}$ to mean $C(c) = F$ and $F(f) = c_f^{\varepsilon}$.

Before going further, there are some details need mentioning. First, in fractional permission system, invariants are established through nesting, In our non-null system, we simply assume every field is `shared` (that is, in terms of ownership, belongs to the world). In permission syntax, suppose a field f is inside a object o, and is pointing to another object o', it can be written as follow:

$$(\exists r' \cdot r.f \rightarrow r' + p(r')) \prec 0.\text{Owned}$$

where p is the class predicate for field f. Here we use 0 to encode the "world" object. As mentioned before, the Owned region is used to model object ownership. Note that, here for simplicity, we nest each field permission inside the Owned region directly, instead of in the All region first. This would have no affect on the proof since everything is shared in the system.

Second, for every procedure, the input permission is always $0.\text{Owned} \rightarrow 0$, and the output permission is the same plus the permission for return value. In terms of effect notation [BRZ09], the input permission can be expressed as "`writes shared`", which grants the annotated method privilege to write field that is shared.

Since the system does not have inheritance, we do not need to consider the "rawness" problem [FL03]. To avoid leaking a partially constructed object, constructor in the system is syntactically restricted; the body of the constructor must be a sequence of assignments to

$$\textsc{Var}\ \frac{E(x) = c^{\varepsilon}}{C; M; E \vdash x : c^{\varepsilon}} \qquad \textsc{Null}\ \frac{}{C; M; E \vdash 0 : \texttt{Null}} \qquad \textsc{Read}\ \frac{C; M; E \vdash e : c^{-} \qquad C(c)(f) = c_f^{\varepsilon}}{C; M; E \vdash e\,\texttt{.}\,f : c_f^{\varepsilon}}$$

$$\textsc{Write}\ \frac{C; M; E \vdash e_1 : c_1^{-} \qquad C; M; E \vdash e_2 : c_2^{\varepsilon} \qquad C(c_1)(f) = c_2^{\varepsilon}}{C; M; E \vdash e_1\,\texttt{.}\,f\texttt{=}e_2 : c_2^{\varepsilon}} \qquad \textsc{Let}\ \frac{C; M; E \vdash e_1 : c_1^{\varepsilon} \qquad C; M; E, x : c_1 \vdash e_2 : c^{\varepsilon}}{C; M; E \vdash \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 : c^{\varepsilon}}$$

$$\textsc{Cond}\ \frac{C; M; E \vdash_b b \qquad C; M; E \vdash e_1 : c^{\varepsilon} \qquad C; M; E \vdash e_2 : c^{\varepsilon}}{C; M; E \vdash \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : c^{\varepsilon}}$$

$$\textsc{Loop}\ \frac{C; M; E \vdash_b b \qquad C; M; E \vdash e : c^{\varepsilon}}{C; M; E \vdash \texttt{while}\ b\ \texttt{do}\ e : \texttt{Null}}$$

$$\textsc{NotNull}\ \frac{E(x) = c_0^{+} \qquad C; M; E, x : c_0^{-} \vdash e_1 : c_1^{\varepsilon} \qquad C; M; E \vdash e_2 : c_1^{\varepsilon}}{C; M; E \vdash \texttt{if not}\ x\texttt{==0}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : c_1^{\varepsilon}}$$

$$\textsc{Sub}\ \frac{C; M; E \vdash e : c_0^{\varepsilon_0} \qquad c_0^{\varepsilon_0} <: c_1^{\varepsilon_1}}{C; M; E \vdash e : c_1^{\varepsilon_1}} \qquad \textsc{Call}\ \frac{M(m) = (c_1^{\varepsilon_1}, \ldots, c_n^{\varepsilon_n}) \to c^{\varepsilon} \qquad C; M; E \vdash e_i : c_i^{\varepsilon_i}}{C; M; E \vdash m\,\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} : c^{\varepsilon}}$$

$$\textsc{Eq}\ \frac{C; M; E \vdash e_1 : c_0^{\varepsilon_0} \qquad C; M; E \vdash e_2 : c_1^{\varepsilon_1}}{C; M; E \vdash_b e_1\ \texttt{==}\ e_2} \qquad \textsc{Not}\ \frac{C; M; E \vdash e : c^{\varepsilon}}{C; M; E \vdash \texttt{not}\ e : c^{\varepsilon}}$$

$$\textsc{And}\ \frac{C; M; E \vdash_b b_1 \qquad C; M; E \vdash_b b_2}{C; M; E \vdash_b b_1\ \texttt{and}\ b_2}$$

Figure 4.1: Non-null Typing Rules

the fields of this class, followed by returning the special `this` (the first parameter) reference. The special variable `this` can only appear on the left-hand-side of each assignment, thus the object will not be used until its invariant is fully established. We also restrict through typing rules that each non-null field has to be assigned inside constructor body. Therefore we must treat constructor as a special syntactical construct, and prove extra theorems about it.

The most interesting clause in the type system is NotNull; the typing rule has added assumption that x is *not null* in the `else` part. In this case, the syntactic construct offers a *narrowing* operation on a type.

In our Twelf realization for the non-null system, we use the `map` signature defined in previous work [Boy10a] for all the environments. Each class identifier is represented by an unique natural number, as are methods and fields. In addition, inside M, the constructor shares the same identifier as the class. We also need to ensure each map is consistent with the others (for instance, each class identifier used in some type should also exist as an entry in C). These restrictions are enforced by a series of consistency relations.

4.3 Converting to Fractional Permissions and Soundness

Having the class structures defined, we need to convert them to fractional permissions. The most important part of this process is to construct a *predicate* for each class in C. The predicate for object o of class C has the form:

$$p(o) \stackrel{\text{def}}{=} (\exists \rho_1 \cdot (o.f_1 \to \rho_1 + \Pi_1) +$$

$$\ldots + \exists \rho_n \cdot (o.f_n \to \rho_n + \Pi_n)) \prec 0.\text{Owned}$$

where Π_i for reference ρ_i has the form:

$$\Pi_i = \begin{cases} \rho_i \neq 0 + p_i(\rho_i) & \rho_i \text{ is not null} \\ \rho_i \neq 0 \,?\, p_i(\rho_i) : \emptyset & \rho_i \text{ is possibly null} \end{cases}$$

While realizing this in Twelf, we first tried to construct predicate for each class on ad-hoc. However, as we found out later, this approach did not work, since each class predicate is not equivalent to its unfolded version in permission logic (in other words, it is *iso-recursive*). In our second approach, we construct class predicates all at once, and store them inside a special *predicate map*, which is defined as follow:

$$P ::= \epsilon \mid P, c : p(x) \text{ (c is the class identifier)}$$

where p is the predicate for class c. Specifically, when constructing the predicate for a particular class, we first bind the class with a variable, and then use it to construct permissions for its field map. The algorithm is similar to a depth-first-traversal on class structure. After all the classes of the enclosing fields are seen, it replaces the predicate variable with the actual one, then the algorithm moves on to the next unseen class in C.

Once the predicate map is constructed, the conversion for type environment E is straightforward. The conversion of procedure map M to program type ω is a simple iteration; for each procedure type $(c_1, \ldots, c_n) \to c$, we convert it to a procedure type α, where α is:

$$\forall x_1, \ldots, x_n \; (\Pi_1 + \ldots + \Pi_n + 0.\text{Owned} \to 0)$$
$$\to \exists x_t \; (\Pi_t + 0.\text{Owned} \to 0)$$

where Π_i is the converted permission for type c_i.

The soundness of the non-null type system depends on the following result: for each expression e in the kernel language, if e is well-typed and has type t in non-null type system,

with consistent environments C, M, and E, then after converting these environments to P, ω, and Π, respectively, e is also well-typed under permission type system, with ω and input permission Π. In addition, the output permission is $\Pi + \Pi'$, where Π' is the permission for type t, converted from t by using P. This is proved by case analysis on all the possibly forms of expression e. For a conditional b, a similar property is proved via mutual induction.

Take the WRITE rule as a example, to show $e_1.f{=}e_2$ is well-typed under the permission type system, we first need to use induction to get the assumption that e_1 and e_2 are well-typed. For e_1, since it is not null, the output permission Π_2 is

$$\rho_1 \neq 0 + p_1(\rho_1) + \Pi_1$$

where p_1 is the predicate for class c_1. For e_2, the above permission is the input of permission typing rule, therefore we need to use the frame rule first. Depending on e_2's type, the permission varies. Here, assuming e_2 is not null, then the output permission is:

$$\rho_1 \neq 0 + p_1(\rho_1) + \rho_2 \neq 0 + p_2(\rho_2) + \Pi_1$$

where p_2 is the predicate for class c_2.

To convert this output to something like $\rho_1.f \rightarrow \rho' + \Pi'$, we need to have the field permission *carved out* from 0.Owned $\rightarrow$ 0 in Π_1. Then, the permission will be transformed to:

$$\exists \rho' \cdot (\rho_1.f \rightarrow \rho' + p_f(\rho')) +$$
$$\exists \rho' \cdot (\rho_1.f \rightarrow \rho' + p_f(\rho')) \multimap 0.\text{Owned} +$$
$$\rho_1 \neq 0 + p_1(\rho_1) + \rho_2 \neq 0 + p_2(\rho_2) + \Pi_1'$$

where Π_1' is Π_1 without 0.Owned $\rightarrow$ 0. We can then get the form by unpacking the existential.

The final step is to transform the output permission of consequence to the right form. Since we have $\rho_1.f \rightarrow \rho_2$ and $p_2(\rho_2)$, we can pack them again to existential form, and use linear modus ponens to get the permission 0.Owned $\rightarrow$ 0 back. Also, extra formulae, like $\rho_1 \neq 0$ and $p_1(\rho_1)$, are discarded. For the case that e_2 is possibly null, the proof is similar.

Given the above result, the well-typedness of methods is straightforward. However, constructors need extra attention; for each constructor, the input field permission is raw, and needs to be packed depending on its non-null type. Also, since not-null fields are guaranteed to be assigned in constructor body, we need to filter them out and construct the permission accordingly. This is done by classifying fields into disjoint sets, with separate relations describing properties about them. As "output" for the theorem, all fields are packed and the class invariant is established.

With the above works done, the soundness theorem is as follow:

soundness For every program g in the kernel language, if g is well-typed under consistent environments C and M, then with the converted program type ω, g can also be type checked under the fractional permission system.

the proof of this theorem is basically iterating over all procedures in the program, and use the above proof of well-typedness for either method or constructor. Being able to prove it, we showed not only that non-null types can be re-expressed using fractional permissions, but also that an entire type system can be reduced to fractional permissions.

Chapter 5

The Conservative Type System

In this section we formalize the conservative type system. Unlike the nonnull type system in Chap. 4, this system also handles annotations for uniqueness, borrowed, and shared. Unlike the system defined in previous work [GB99, Ret09], the method effects are declared through the borrowed annotation. Also, this system does not distinguish between read and write effects. Handling the distinction is left to future work since on the permissions level, fractions need to be treated separately, and is a complication that obscures the interesting issues that arise even without the distinction.

In the following, Sec. 5.1 describes the syntax for the language we used for this type system, as well as some constructs used in the type rules; Sec. 5.2 describes the concepts of targets, sources and capabilities; Sec. 5.3 describes the main type rules for the system, and explain some of the design decisions; Sec. 5.4 demonstrates some simple examples that are either rejected or accepted by the conservative type system.

$$
\begin{array}{rcl}
P \in \textbf{Program} & ::= & \overline{defn} \\
defn \in \textbf{Class} & ::= & \texttt{class}\ cn\ \{\ \overline{field}\ constr\ \overline{meth}\ \} \\
field \in \textbf{Field} & ::= & \alpha\ \tau\ f \\
meth \in \textbf{Method} & ::= & \alpha\ \tau\ mn(\overline{\alpha\ \tau\ x})\ \alpha\ \{\ e\ \} \\
constr \in \textbf{Constructor} & ::= & cn(\overline{\alpha\ \tau\ x})\ \{\ e\ \} \\
e \in \textbf{Expr} & ::= & x \mid \texttt{null} \mid e.f \mid e.f = e \\
 & & \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid (e\texttt{;}e) \\
 & & \mid \texttt{if}(b)\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{while}\ (b)\ \texttt{do}\ e \\
 & & \mid \texttt{new}\ c(\overline{e}) \mid e.mn(\overline{e}) \\
b \in \textbf{BoolExpr} & ::= & \texttt{true} \mid \texttt{not}\ b \mid e\texttt{==}e \mid b\ \texttt{and}\ b \\
\alpha \in \textbf{Annotation} & ::= & \texttt{unique} \mid \texttt{shared} \mid \texttt{borrowed}(\overline{f}) \\
\varepsilon \in \textbf{Nullness} & ::= & \texttt{notnull} \mid \texttt{nullable} \\
\tau \in \textbf{Type} & ::= & \varepsilon\ c
\end{array}
$$

Figure 5.1: Syntax

5.1 Syntax

The syntax for the kernel language used for the conservative type system is shown in Fig. 5.1. For the purpose of better explanation, a slightly different language is used here from the one in previous chapters. The differences are mainly:

- The language is more high-level, and has explicit class, field and method declarations.
- Beside annotations for nullness, the language also has annotations for uniqueness and method effects. Note that Retert's thesis does not handle nullness [Ret09]. We included it here because 1) it is relatively easy to do because of the pilot study work described in Chap. 4; 2) conversion to fractional permissions requires the knowledge of nullness for a variable.

In the Twelf realization, the language is modeled using the same kernel language defined in Chap. 3 and Chap. 4. For instance, methods are modeled with procedures, of which the first parameter is reserved for `this`. Classes, fields and methods are modeled using various environments, such as class map, field map and method map. This is similar to what we did in Chap. 4.

On the top-most level, a program P is simply a collection of class declarations. Each class is represented by a collection of field, a constructor declaration, followed by method declarations. Since inheritance is orthogonal to the aliasing problem, for simplicity, it is not included in this language.

A field declaration includes an annotation as well as the type and name of the field. The annotation consists of two parts: the uniqueness and the nullness information. Uniqueness is specified as either `unique` or `shared`. As stated before, `unique` means the field should be the *only* reference to the object it refers to, while `shared` imposes no restriction on the

number of aliases. Similar to the type system in Chap. 4, the nullness information is also attached to a field declaration. A field can be declared as either `notnull`, which indicates the field should never be null, or `nullable`, which indicates it could possibly be null. For a class, all of its not-null fields are required to be initialized in the constructor. For a possibly-null field, it may or may not be initialized. In the latter case, it defaults to null.

In this thesis, we use τ to represent a type variable, which is a nullness variable ε followed by a class identifier c. We also call $\alpha\ \tau$ an *annotated type*.

Methods are defined in a similar way as in Java, except that each method parameter is prefixed with an annotation α, as well as a nullness ε. Unlike Java, a method receiver also has an annotation, which appears after the method parameters. Before checking the method body, according to the specific annotation, a set of input and output capabilities (which shall be described in the next section) are generated for each method parameter and the receiver. Note that, unlike the type system defined in Retert's thesis [Ret09], method effects are declared as part of the borrowed annotation in this type system. We shall discuss this further in the next section.

A constructor is treated as a special kind of method. First, its name should be the same as the class name that it belongs to. Also, its body must be a sequence of field assignments, followed by the `this` expression. In the field assignments, the special variable `this` cannot occur on the right hand side of any of the assignments, to avoid leaking a partially constructed object. A constructor, unlike a method declaration, does not have return value and annotation for receiver; inside a constructor body, one is able to write any field for the newly allocated object. The type rule for constructor also guarantees that all not-null fields are properly initialized.

The following is a simple example for a constructor in the kernel language:

```
class A {
  unique notnull B b;
  unique nullable C c;

  A(unique notnull B b1) {
    this.b = b1;
    this
  }
}
```

Here the field `b` is initialized in the constructor, using the unique parameter `b1`; field `c` defaults to null.

Expressions are, in order, local variable access, `null`, field reads and writes, "let" expression, sequence, "if" expression, while loop expression, method call and object creation. A sequence expression (e_0`;`e_1) is essentially a syntax suger for `let _ =` e_0 `in` e_1, but we list it here separetely just for convenience. A sequence expression is also treated in a special way by the type system, which will be described in Sec. 5.3.

The language also has boolean expressions, which are used in conditionals. The boolean expressions are, in order, `true`, negation, expression comparison and conjunction.

Annotations for a type can be `unique`, `shared` or `borrowed`($\overline{f}$). For `unique` and `shared`, they have the same meaning as stated before. Also, `borrowed`($\overline{f}$) is restricted to be used on method parameters (including method call receivers).

5.2 Targets, Sources and Capabilities

This section introduces the concepts of targets, sources and capabilities, which are essential components to the type system. Their definitions are shown in Fig. 5.2. In the type system, sources represent where a value comes from, while targets represent all the sources for a unique value. Capabilities represent the ability to access a value.

$$
\begin{array}{rcl}
\rho \in \textbf{Target} & ::= & x \mid x.f \mid \textbf{s-tgt} \mid \textbf{f-tgt} \\
\psi \in \textbf{Targets} & ::= & \cdot \mid \psi, \rho \\
\kappa_f \in \textbf{FieldCapabilities} & ::= & \cdot \mid \kappa_f, x.f^{\bullet} \mid \kappa_f, \textbf{s-tgt}^{\bullet} \\
\kappa_o \in \textbf{ObjectCapabilities} & ::= & \cdot \mid \kappa_o, x \\
\kappa \in \textbf{Capabilities} & ::= & (\kappa_f, \kappa_o) \\
\beta \in \textbf{Sources} & ::= & \textbf{shared} \mid \psi
\end{array}
$$

Figure 5.2: Targets, Sources and Capabilities

The main type rule is of the format:

$$P; E; \kappa_1 \vdash e : (\tau, \beta) \dashv \kappa_2$$

This is read: in program P, given environment E and input capabilities κ_1, expression e has type τ, and sources β. The type checking also produces output capabilities κ_2. Capabilities appear both as input and output. In the former case it means all the capabilities that can be used to check the expression e, and in the latter case it means all the remaining capabilities after checking e.

Because capabilities appear both as input and output in the type rule, the system is flow-sensitive. I initially thought about a flow-insensitive effect system, which may be simpler.

```
class Node {
  unique nullable Node next;
  shared nullable Object data;

  Node(shared nullable Object d, unique nullable Node n) {
    this.next = n;
    this.data = d;
  }

  void append(unique notnull Node m, unique notnull Node n)
      borrowed(next) {
    let x = if (...) then m else n in
      x.next = this.next;
      this.next = x;
    null
  }
}
```

Figure 5.3: A Motivating Example

However, the linear nature of the problem means that the effect system is very restrictive. For instance, we need to know whether a unique value is used before or after it is consumed, or whether a unique field is consumed before or after the field's uniqueness is restored.

When compared to the fractional permission system, even with the flow-sensitive nature, the current type system is still much more lightweight. This is because operations on capabilities are much simpler than those on fractional permissions. In fractional permissions, there are different forms of permissions, such as conditional permissions, field permissions with fractions, encumbered permissions, formulae, etc., and operations are complex to transform these permissions from one format to another, for instance, carving out one permission from another, or split one read or write permission into several read permissions, and so on. In contrast, the operations on capabilities and targets are very limited: operations on them are essentially set operations such as add and remove.

Let us consider example:

In the method `append`, depending on the result of evaluating the condition part for the

"if" expression, let-bound variable `x` could either equal to `m` or `n`. In the let body, `x.next` is first assigned the unique value of `this.next`, and then the unique value of `x` is consumed as effect of being assigned to `this.next`. The ordering of the two expressions in the let body is important: if we swap the two expressions, then it would no longer be valid, since after consuming `x`, its `next` field can no longer be written.

In order to check this method, the type system needs to track information about the variable `x`. Specifically, it needs to know that `x` could be either `m` or `n`, and that when executing the first expression in the let body, the uniqueness of `x` is not compromised. Also, the type system should record that the variable is no longer unique after the second expression, and therefore guarantees that it is not accessed as a unique variable afterwards.

In the conservative type system, we use two key concepts to track the above information. A *target* is used to track the information of where a value comes from, and a *capability* is used to track whether one can access a particular value.

A *target*, denoted with ρ, is used to track where a unique value could come from. By saying that a value comes from a target ρ, it means that the only access path to that value is through the object represented by the target. And, in order to access the unique value, one needs to hold the capability on ρ. The concept of capability shall be introduced shortly after.

In the type system, there are four different kinds of unique targets:

- x, referred as *object target*, represents that the unique value it is associated to *could* be an alias to the local variable x.

- $x.f$, referred as a *field target*, represents that the unique value could come from $x.f$, where x is a local variable, and f is a field identifier.

- **s-tgt**, referred as a *shared target*, indicates that the variable comes from a shared object. That is, the variable refers to a unique field of a shared object.

- **f-tgt**, which is referred as a *fresh target*, indicates that the value could come from a fresh object, which is either a newly allocated object or a returned value from a method call. For fresh target, there is no corresponding capability.

In the case of shared target, it means the value could be accessed from some shared object, and hence the accesses could potentially come from different places. In the case of fresh target, it means the access comes from some local object. A value from a fresh target is guaranteed to be only used once (e.g., appear on the right hand side of an assignment expression). This is enforced by the way the let expression is treated in the type rules. The type rules also enforce that in either object or field target, both x and f are unique.

Readers may wonder why only the first field (f in $x.f$) is tracked, instead of the *whole path*. For instance, suppose `a`, `f`, `g` and `h` are all unique, it would be more precise to represent the value:

```
a.f.g.h
```

using a target `a.f.g.h`. This is possible, with the cost of extra complexities in the type system and conversion to fractional permissions. In addition, in a while loop, an unbounded path could be formed because there could be an infinite number of iterations, and therefore approximation is necessary at certain point.

The result value of an expression could come from multiple targets. For instance, in the example shown in Fig. 5.3, the targets for variable `x` are object targets `m` and `n`.

As another example, consider:

```
if (..) then a.f
```

```
else if (..) then b.g else new C()
```

Assuming `a` is unique, `b` is shared, and both field `f` and `g` are unique, targets associated with the expression are `a.f`, **s-tgt** and **f-tgt**, in the order of the "if" branches.

Capabilities are represented by a pair of κ_f, which is a set of *field capabilities*, and κ_o, which is a set of *object capabilities*.

A field capability gives access to a particular target. There are two different kinds of field capabilities: a field capability $x.f^\bullet$, which gives access to target $x.f$, and the capability **s-tgt**$^\bullet$, which gives access to the shared target **s-tgt**.

For instance, to read field `f` of a variable `a`:

```
a.f
```

one needs to have capability `a.f`$^\bullet$ if `a` is unique. In case `a` is shared, the field capability **s-tgt**$^\bullet$ is needed.

A unique value is said to be *consumed*, when it is assigned to a unique field or method parameter. When consumed, capabilities associated with the targets of this value must be taken away from the input and can no longer be used, unless the uniqueness of this value is restored by another unique value. Similarly, we also say the capabilities are consumed or restored in this case.

If a unique value has targets $x.f$ or **s-tgt**, then the corresponding capabilities need to be removed in order to consume the variable. However, if the variable has an object target x, then an corresponding object capability x is required, *as well as field capabilities for all fields of the object referred by* x. An object capability represents the ability to consume an object as a whole.

For instance, back to the example shown in Fig. 5.3, to type check the second expression

in the let body:

```
this.next = x;
```

One needs to have

- field capabilities $\texttt{m.next}^\bullet$, $\texttt{m.data}^\bullet$, $\texttt{n.next}^\bullet$ and $\texttt{n.data}^\bullet$.
- object capability m and n.

Therefore, including the capability to access this.next, the input capabilities need to be:

$$(\{\texttt{this.next}^\bullet, \texttt{m.next}^\bullet, \texttt{m.data}^\bullet, \texttt{n.next}^\bullet, \texttt{n.data}^\bullet\}, \{\texttt{m}, \texttt{n}\})$$

In the type system, input and output capabilities are generated for each parameter in a method declaration. For a method parameter a of class c:

- if it is unique, then it generates input field capabilities on all fields of class c, as well as object capability a. There is no output capability.
- if it is $\texttt{borrowed}(\overline{f})$, then it generates input and output field capabilities on all fields in $\overline{f}$.
- if it is shared, then no capability is generated. For accessing shared values, capability $\textbf{s-tgt}^\bullet$ is always included in both input and output capabilities.

Therefore, in the example shown in Fig. 5.3, the input capabilities for method append are:

$$(\{\texttt{this.next}^\bullet, \texttt{m.next}^\bullet, \texttt{m.data}^\bullet, \texttt{n.next}^\bullet, \texttt{n.data}^\bullet, \textbf{s-tgt}^\bullet\}, \{\texttt{m}, \texttt{n}\})$$

and output capabilities are:

$$(\{\texttt{this.next}^\bullet, \textbf{s-tgt}^\bullet\}, \emptyset)$$

In this thesis, we use κ to represent a pair of field capability set κ_f and object capability set κ_o. We also refer κ as *capabilities*. In several occasions we need to union two pairs of capabilities. For brevity, we use the following notation:

$$(\kappa_{f_0}, \kappa_{o_0}) \cup (\kappa_{f_0}, \kappa_{o_0}) \stackrel{\text{def}}{=} (\kappa_{f_0} \cup \kappa_{f_1}, \kappa_{o_0} \cup \kappa_{o_1})$$

to represent the union of two such pairs. Similarly, we define:

$$(\kappa_{f_0}, \kappa_{o_0}) \subseteq (\kappa_{f_0}, \kappa_{o_0}) \stackrel{\text{def}}{=} (\kappa_{f_0} \subseteq \kappa_{f_1}, \kappa_{o_0} \subseteq \kappa_{o_1})$$

and

$$(\kappa_{f_0}, \kappa_{o_0}) \setminus (\kappa_{f_0}, \kappa_{o_0}) \stackrel{\text{def}}{=} (\kappa_{f_0} \setminus \kappa_{f_1}, \kappa_{o_0} \setminus \kappa_{o_1})$$

Finally, in the type rules, β represents the *source* for the result value. A source could be a shared source **shared**, or for a unique value, a set of targets ψ. **shared** indicates that the value is shared, and therefore there could be other aliases that are pointing to the object that this value is pointing to. For accessing a field inside a value with shared source, one needs to have $\textbf{s-tgt}^\bullet$.

Given an expression $a.f_0.f_1 \ldots f_n$, to infer what are the result source for this expression, there are three cases to consider:

1. f_n is shared, then the result source is **shared**.

2. f_n is unique, and either a or at least one field in $f_0 \ldots f_{n-1}$ is shared, then the source is the set $\{\textbf{s-tgt}\}$. This means the result is a unique value that can only be accessed through some shared reference.

3. both a and all fields in $f_0 \ldots f_n$ are unique. In this case, the source for this expression is the set $\{a.f_0\}$.

On the other hand, for an expressions such as `new` $c(\overline{e})$, the source is the set {**f-tgt**}, since there is no capability to track for a new object.

It is important to know the difference between a shared target **s-tgt** and a shared source **shared**. The former indicates that the result value is unique, but comes from a shared object, while the latter indicates that the value is shared. For instance, for an expression `a.f`, if `a` is shared, while `f` is unique, then it has source {**s-tgt**}. However, if `f` is shared, the expression will have source **shared** regardless of the type of `a`.

When an expression e has targets ψ, it is guaranteed that the field capabilities for the field targets and shared target in ψ are removed from the input until the e is no longer in use, at which time they will be returned back to the available capabilities. In the following of this thesis, we shall say these field capabilities are *pinned* by the e, and are *unpinned* when the targets are released.

For instance:

```
if (b == null) then a.u else b.u
```

In this example, suppose `a` and `u` are unique, and `b` is shared. In order to type check this expression, one needs to have capabilities $(\{\texttt{a.u}^\bullet, \textbf{s-tgt}^\bullet\}, \emptyset)$. After checking the expression, the result source is $\{\texttt{a.u}, \textbf{s-tgt}\}$, and result capabilities are $(\emptyset, \emptyset)$. The field capabilities $\texttt{a.u}^\bullet, \textbf{s-tgt}^\bullet\}$ are pinned by the "if" expression, and are no longer available until the "if" expression is not used anymore.

When converting to fractional permissions, removing field capabilities for ψ of e means the permissions associated with ψ are *encumbered* inside permissions for e. The first set of permissions cannot be used until e is not needed anymore, at which time linear *modus-ponens* rule can then be applied to restore the pinned field capabilities.

5.3 The Type System

In this section, we describe the conservative type system built on the language introduced in Sec. 5.1. The main type judgments are shown in Fig. 5.4.

5.3.1 Type Rules for Expressions

judgment	meaning
$P \vdash \diamond$	well-formed program P
$P \vdash \mathit{defn}$	well-formed class definition *defn*
$P, c \vdash \mathit{meth}$	method *meth* is well-formed in class c
$P, c \vdash \mathit{field}$	well-formed *field* in class c
$P; E; \kappa_1 \vdash e : (\tau, \beta) \dashv \kappa_2$	Under environment E and with input capabilities κ_1, e has type τ and sources β, and output capabilities κ_2.
$P; E; \kappa_1 \vdash b \dashv \kappa_2$	Under environment E and with input capabilities κ_1, boolean expression b is well-typed, and has output capabilities κ_2.
$\kappa_1 \vdash (\tau_1, \beta) <: \alpha_2\ \tau_2 \dashv \kappa_2$	Given input capabilities κ_1, type τ_1 and source β is a subtype of declared annotated type $\alpha_2\ \tau_2$. The subtyping also produces output capabilities κ_2.

Figure 5.4: Main Type Judgments

The definition of type environment E is given by the following rule:

$$E ::= \cdot \mid E,\ x : \alpha\ \tau$$

Here, the variable x is bound to an annotated type $\alpha\ \tau$. In the rest of this thesis, when it simplifies the presentation, we will treat E as a partial function from variable names to their

type and targets. Therefore, $E(x) = \alpha\ \tau$ is equivalent to $E = E\,', x : \alpha\ \tau$. For a well-formed environment E, we require that each variable x can appear at most once, and that there is exactly one associated type for the variable. In addition, all the type variables τ used in a well-formed environment must also be well-formed in the program P (e.g., $P \vdash \tau$). A well-formed environment has the exchange property so that when we have $E(x) = \alpha\ \tau$, there exist an environment E' such that $E = E', x : \alpha\ \tau$.

In the following paragraphs, for the purpose of explanation, we present type rules as a set of groups, with a description following each.

The first group of the type rules is shown in Fig. 5.6. Rule NULL specifies how a `null` expression can be checked. No capability is required in this case, and therefore the output capability set is the same as input. A `null` can have any class c, and is clearly possibly-null (`nullable`).

For local variable access (VAR) to be type checked, the variable x need to be in the domain of E, and has annotated type $\alpha\ \tau$. The result source is generated through the auxiliary function **annot-to-source** defined in Fig. 5.5. There are two cases: if the annotation is `shared`, then the result source is simply the shared source **shared**; if the annotation is `unique`, then the result source is a singleton set with the object target x. Accessing a local variable does not cost any capability, so the input and output capabilities are the same.

For reading a field (READ), $e.f$. First, the expression e must be well-typed, and the result value must not be null (`notnull`). In addition, f should be an valid field inside class c of expression e, which is specified by the relation **ftype**.

Reading a field also requires capabilities to targets of e. This is specified by the **read-field** function defined in Fig. 5.5. Given input sources β, input capabilities (κ_f, κ_o), and a field identifier f, **read-field** computes output targets and capabilities. There are two cases

$$\textbf{annot-to-source}(x,\,\alpha) = \begin{cases} \textbf{s-tgt} & \alpha = \texttt{shared} \\ \{x\} & \alpha = \texttt{unique} \\ \{x\} & \alpha = \texttt{borrowed}(\overline{f}) \end{cases}$$

$$\textbf{read-field}(\beta,\,(\kappa_f, \kappa_o),\,f) = \begin{cases} (\{\textbf{s-tgt}\}, (\kappa_f \setminus \{\textbf{s-tgt}^\bullet\}, \kappa_o)) & \beta = \textbf{shared} \wedge \textbf{s-tgt}^\bullet \in \kappa_f \\ (\psi_2 \cup \textbf{non-obj-tgts}(\psi), (\kappa_f \setminus \psi_2^\bullet, \kappa_o)) & \beta = \psi \wedge \psi_2 = \textbf{extend-fld}(\psi, f) \wedge \psi_2^\bullet \subseteq \kappa_f \end{cases}$$

$$\textbf{obj-tgts}(\psi) = \{\ \rho \mid \rho = x \in \psi\ \}$$

$$\textbf{non-obj-tgts}(\psi) = \psi \setminus \textbf{obj-tgts}(\psi)$$

$$\textbf{non-obj-fresh-tgts}(\psi) = \psi \setminus (\textbf{obj-tgts}(\psi) \cup \{\textbf{f-tgt}\})$$

$$\textbf{extend-fld}(\psi,\,f) = \{\ x.f \mid x \in \textbf{obj-tgts}(\psi)\ \}$$

$$\textbf{extend-flds}(\psi,\,\overline{f}) = \{\ x.f \mid x \in \textbf{obj-tgts}(\psi), f \in \overline{f}\ \}$$

$$\textbf{restore-shared}(\psi,\,\kappa,\,\alpha) = \begin{cases} (\psi, \kappa) & \alpha = \texttt{unique} \\ (\textbf{shared}, \kappa \cup (\textbf{non-obj-fresh-tgts}(\psi)^\bullet, \emptyset)) & \alpha = \texttt{shared} \end{cases}$$

$$\textbf{pinned-capabilities}(\beta) = \begin{cases} \emptyset & \beta = \textbf{shared} \\ \textbf{non-obj-fresh-tgts}(\psi)^\bullet & \beta = \psi \end{cases}$$

$$\textbf{annot-to-result-source}(\alpha) = \begin{cases} \{\textbf{f-tgt}\} & \alpha = \texttt{unique} \\ \textbf{shared} & \alpha = \texttt{shared} \end{cases}$$

$$\textbf{reclaim-borrowed}(\beta,\,\alpha) = \begin{cases} \textbf{extend-flds}(\psi, \overline{f})^\bullet \cup \textbf{non-obj-fresh-tgts}(\psi)^\bullet & \beta = \psi \wedge \alpha = \texttt{borrowed}(\overline{f}) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\textbf{not-in-source}(x,\,\beta) = \begin{cases} \bot & \beta = \textbf{shared} \\ \forall \rho \in \psi : \textbf{not-in-target}(x, \rho) & \beta = \psi \end{cases}$$

$$\textbf{not-in-target}(x,\,\rho) = \begin{cases} x \neq x' & \rho = x' \\ x \neq x' & \rho = x'.f \\ \bot & \text{otherwise} \end{cases}$$

Figure 5.5: Auxiliary Rules For Type Checking

NULL

$$\frac{}{P; E; \kappa \vdash \texttt{null} : (\texttt{nullable}\ c, \{\textbf{f-tgt}\}) \dashv \kappa}$$

VAR

$$\frac{E(x) = \alpha\ \tau \qquad \textbf{annot-to-source}(x, \alpha) = \beta}{P; E; \kappa \vdash x : (\tau, \beta) \dashv \kappa}$$

READ

$$\frac{\begin{array}{c} P; E; \kappa_0 \vdash e : (\texttt{notnull}\ c, \beta_1) \dashv \kappa_1 \qquad \textbf{ftype}(P, c, f) = \alpha\ \tau \\ \textbf{read-field}(\beta_1, \kappa_1, f) = \psi, \kappa_2 \qquad \textbf{restore-shared}(\psi, \kappa_2, \alpha) = \beta_3, \kappa_3 \end{array}}{P; E; \kappa_0 \vdash e.f : (\alpha\ \tau, \beta_3) \dashv \kappa_3}$$

WRITE

$$\frac{\begin{array}{c} P; E; \kappa_0 \vdash e_1 : (\texttt{notnull}\ c_1, \beta_1) \dashv \kappa_1 \\ \textbf{ftype}(P, c_1, f) = \alpha\ \tau \qquad \textbf{read-field}(\beta_1, \kappa_1, f) = \psi, \kappa_2 \qquad P; E; \kappa_2 \vdash e_2 : (\tau_2, \beta_2) \dashv \kappa_3 \\ \kappa_3 \vdash (\tau_2, \beta_2) <: \alpha\ \tau \dashv \kappa_4 \qquad \textbf{restore-shared}(\psi, \kappa_4, \alpha) = \beta_3, \kappa_5 \end{array}}{P; E; \kappa_0 \vdash e_1.f = e_2 : (\tau, \beta_3) \dashv \kappa_5}$$

Figure 5.6: Type Rules for the Conservative Type System (Part 1)

depending on the input β.

1. β is **shared**: this is reading a field from a shared value, which requires the capability to access the target **s-tgt**. Hence, the input κ_f needs to contain **s-tgt**$^\bullet$, which then is removed. Correspondingly, the result target is a singleton set containing **s-tgt**.

2. β is a target set ψ: this is reading a field from a unique value that comes from targets in ψ. In this case, we need to look at each target ρ in ψ, and, depending on the type of ρ, we may need to *extend* ρ with the field f. There are four sub-cases:

   (a) if ρ is a field target $x.g$, then it means the expression e comes from $x.g$. Reading a field for this expression ($x.g.f$) should also comes from the same target. Hence, the result is still the field target $x.g$.

   (b) if ρ is the shared target, it means the unique value of e comes from the shared target. Similar to the first case, the result target should still be the same.

   (c) if ρ is an object target x, it means the expression e comes from x. Reading the field f should generate a field target $x.f$.

   (d) if ρ is the fresh target **f-tgt**, then it is simply skipped because there is no capability to track for a fresh object.

The "extend" operation is defined by the function **extend-flds** in Fig. 5.5, which takes all object targets in the input ψ, and extend them with f. Also, given a target set ψ, function **non-obj-tgts** returns all non-object targets in the set. Therefore, the result of extending a target set ψ is equal to:

$$\textbf{extend-fld}(\psi, f) \cup \textbf{non-obj-tgts}(\psi)$$

For the function **read-field**, the output targets are the extended targets on input ρ. The output capabilities are the input capabilities with all the pinned field capabilities (the capabilities on targets **extend-fld**(ψ, f)) removed.

Lastly for the READ rule, the function **restore-shared** takes the output of **read-field** as input, and produces a new pair of target set and capability set, considering the annotation for field f. Note that the input targets for this function can only be field targets.

There are two cases for **read-field**, depending on the annotation:

1. α is `shared`: this means the result is shared, and therefore any further access on fields in this value must come through shared instead. In this case, all targets (if there are any) associated with e can be released, as well as all the pinned capabilities.

2. α is `unique`: this means the result is unique. In this case, any further field access for the result value shall still go through the same targets, and therefore the output targets and capabilities should remain the same.

Next is the rule for writing a field $e_1.f = e_2$ (WRITE). This rule shares some of the common premises as the READ rule. First, like the rule READ, the expression e_1 needs to be well-typed and the output should be not null. The field f should also be a well-formed field, which is defined by the **ftype** function. In addition, reading the field f will cost some capabilities, perhaps temporarily, which is described using **read-field**.

After removing the required capabilities for reading field f, κ_2 is the remaining set of capabilities. This is then used as input for checking e_2.

For writing a field we need to check whether the actual result type from checking e_2 is a subtype of the declared type $\alpha\ \tau$ for field f. This is defined by the subtyping rule in

Sec. 5.3.3. A subtyping relation in the type system is of the format:

$$\kappa_1 \vdash (\tau_1, \beta_1) <: \alpha_2\ \tau_2 \dashv \kappa_2$$

This is read: given input capabilities κ_1, type τ_1 and source β_1 is a subtype of the annotated type $\alpha_2\ \tau_2$. The subtyping relation also produces output capabilities κ_2.

For the WRITE rule, there are two cases:

1. β_2 (which corresponds to the β_1 in the subtyping rule above) is a unique source, and the annotated type for f is either unique or shared (it cannot be borrowed since it is a field type), all the associated capabilities are consumed. For any target ρ associated with e_2, if ρ is an object target x, then the whole object is consumed. For this, field capabilities for all fields of the x are removed, and the x itself is also removed from the object capabilities. On the other hand, if ρ is a field target, then nothing needs to be done, since the capability for ρ is already pinned, they are already removed from the available capabilities. This computation is defined in the SUBUNIQUE rule.

2. In the case that the β_2 is **shared**, the declared field type is required to be shared as well. No change happens for the capabilities when both sides of the subtyping rule are shared. This is defined by the SUBSHARED rule.

The subtyping rules defined in Fig. 5.13 captures the above cases and change the input capabilities accordingly.

Finally, similar to READ, the pinned field capabilities for the expression e_1 can be recovered if the field is shared. This is done through **restore-shared**. Together with the output capabilities κ_4, and target set ψ, **restore-shared** generates β_3 and κ_5, which are final the output sources and capabilities for the whole expression.

Now look at a simple example:

```
x.f = y.g
```

To type check this expression, assuming after checking the expression `x` the result source is a unique target set $\{\texttt{a}, \texttt{b.k}\}$, and the expression `y` produces unique target set $\{\texttt{c}\}$. Also, suppose both variable `a` and `b`, and field `f` and `g` are unique, and the input capabilities are $(\{\texttt{a.f}^\bullet, \texttt{b.k}^\bullet, \texttt{c.g}^\bullet\}, \emptyset)$. Let us analyze step by step to see how this can be type checked.

1. Type checking expression `x`. This produces target set $\{\texttt{a}, \texttt{b.k}\}$. The output capabilities are $(\{\texttt{a.f}^\bullet, \texttt{c.g}^\bullet\}, \emptyset)$. The capability $\texttt{b.k}^\bullet$ is pinned.

2. Checking whether field `f` is well-formed, using function **ftype**.

3. Generating new capabilities using function **read-field**. Since the field is unique, the result targets are $\{\texttt{a.f}, \texttt{b.k}\}$, and the result capabilities are $(\{\texttt{c.g}^\bullet\}, \emptyset)$.

4. Type checking expression `y.g`, with input capabilities $(\{\texttt{c.g}^\bullet\}, \emptyset)$. The output capabilities are $(\emptyset, \emptyset)$, and the field capability $\texttt{c.g}^\bullet$ is pinned for reading `g`.

5. Applying the subtyping rule on the declared field type and the actual result type from checking `y.g`. Since the field type is unique, the field capability $\texttt{c.g}^\bullet$ associated with `c.g` is consumed. But, since it is already pinned, the input and out capabilities are still the same $((\emptyset, \emptyset))$.

6. Finally, together with the field annotation, $\{\texttt{a.f}, \texttt{b.k}\}$ and $(\emptyset, \emptyset)$ are passed to **restore-shared** as input, and the outputs are $\{\texttt{a.f}, \texttt{b.k}\}$ and $(\emptyset, \emptyset)$. Therefore, the result target set is $\{\texttt{a.f}, \texttt{b.k}\}$, and result capabilities are $(\emptyset, \emptyset)$.

For the case when both field `f` and `g` are shared, or when `f` is shared and `g` is unique, the process is also the same.

Notice that, for the case that both `a` and `b` are shared, and `f` and `g` are also shared, the expression cannot be type checked under the type system. This is because after reading `a.f`, the capability **s-tgt** is removed from the input capabilities, and is not available when checking `b.g`. To solve this issue, it is possible to postpone removing **s-tgt** in **read-field**, until after e_2 is checked. Checking e_2 should never cost the capability **s-tgt** or move it to the target set, so therefore it can be used to read the field f right before writing it. To implement this, it requires changes to both the type system and the proof. This is a future work.

The second part of type rules is shown in Fig. 5.7. Given the fact that an expression can be checked, the rule Sub defines how to weaken it and obtain different type checking results. This is useful, for instance, in checking "if" expressions, when we need to obtain the same results from both branches.

$$\textsc{Sub}\quad \frac{P; E; \kappa_0 \vdash e_0 : (\tau_0, \beta_0) \dashv \kappa_1 \qquad P; E; \kappa_1 \vdash (\tau_0, \beta_0) <: (\tau_1, \beta_1) \dashv \kappa_2}{P; E; \kappa_0 \vdash e_0 : (\tau_1, \beta_1) \dashv \kappa_2}$$

$$\textsc{If}\quad \frac{P; E; \kappa_0 \vdash b \dashv \kappa_1 \qquad P; E; \kappa_1 \vdash e_0 : (\tau, \beta) \dashv \kappa_2 \qquad P; E; \kappa_1 \vdash e_1 : (\tau, \beta) \dashv \kappa_2}{P; E; \kappa_0 \vdash \texttt{if}(b)\ \texttt{then}\ e_0\ \texttt{else}\ e_1 : (\tau, \beta) \dashv \kappa_2}$$

$$\textsc{IfNull}\quad \frac{E_0 = E', x : \alpha\ \varepsilon\ c \qquad E_1 = E', x : \alpha\ \texttt{notnull}\ c \qquad P; E_0; \kappa_1 \vdash e_0 : (\tau, \beta) \dashv \kappa_2 \qquad P; E_1; \kappa_1 \vdash e_1 : (\tau, \beta) \dashv \kappa_2}{P; E_0; \kappa_1 \vdash \texttt{if}\ (x{=}{=}0)\ \texttt{then}\ e_0\ \texttt{else}\ e_1 : (\tau, \beta) \dashv \kappa_2}$$

$$\textsc{Loop}\quad \frac{P \vdash c \qquad P; E; \kappa_0 \vdash b \dashv \kappa_1 \qquad P; E; \kappa_1 \vdash e : (\tau, \beta) \dashv \kappa_2 \qquad \kappa_2 \cup \textbf{pinned-capabilities}(\beta) = \kappa_0}{P; E; \kappa_0 \vdash \texttt{while}\ (b)\ \texttt{do}\ e : (\texttt{nullable}\ c, \{\textbf{f-tgt}\}) \dashv \kappa_1}$$

Figure 5.7: Type Rules for the Conservative Type System (Part 2)

There are three places that can be weakened, *the type, the sources and the capabilities.* The format for the sub-result relation is as follow:

$$P; E; \kappa_0 \vdash (\tau_0, \beta_0) <: (\tau_1, \beta_1) \dashv \kappa_1$$

This is read: under program P and environment E, with input capabilities κ_0, the result type τ_0 and source β_0 can be relaxed to τ_1 and source β_1, with output capabilities κ_1.

First, rule SHARED2SHARED defines the case when both sides are of shared source. For this, it is simple: we can relax the nullness as well as capabilities, by removing parts of the input capabilities.

Rule UNIQUE2SHARED defines the case when relaxing a unique source to a shared source. This is restrictive: all capabilities associated with the unique source are consumed (that is, in terms of fractional permissions, nested into the permission for the shared object). The reason for this restriction is because in the type system, a result type is either unique or shared, but never both. A possible future work is to use a hybrid type for a value that could be either unique or shared. This would make the system more flexible, but potentially more complex.

For the case of relaxing one unique source to another, it is more complex. The rule UNIQUE2UNIQUE defines how this is done. Suppose we want to expand (relax) the original target set ψ, by adding another target set ψ'. There are several restrictions on the latter:

- For any object target x in ψ', x must be in the environment E, and the nonnull type of x must be a subtype of ε_1.
- For any field target $x.f$ in ψ', the variable x must be in the environment E and must be not null. Also, f must be a valid field inside the class of x, and its type must be a subtype of the final type.

$$\textsc{Shared2Shared}\quad \frac{\tau_0 <: \tau_1 \qquad \kappa' \subseteq \kappa}{P; E; \kappa \vdash (\tau_0, \mathbf{shared}) <: (\tau_1, \mathbf{shared}) \dashv \kappa'}$$

$$\textsc{Unique2Shared}\quad \frac{\varepsilon_0\ c_0 <: \tau_1 \qquad \kappa' \subseteq \kappa \qquad \kappa'' = (\mathbf{extend\text{-}flds}(\psi, \mathbf{fields}(c_0))^\bullet, \mathbf{obj\text{-}tgts}(\psi)) \qquad \kappa'' \subseteq \kappa'}{P; E; \kappa \vdash (\varepsilon_0\ c_0, \psi) <: (\tau_1, \mathbf{shared}) \dashv \kappa' \setminus \kappa''}$$

$$\textsc{Unique2Unique}\quad \frac{\begin{array}{c}(\kappa_f, \kappa_o) \subseteq \kappa \qquad \tau_0 <: \tau_1 \\ \forall x \in \psi' : E(x) = \alpha'\ \tau' \wedge \tau' <: \tau_1 \\ \forall x.f \in \psi' : x.f^\bullet \in \kappa_f \wedge E(x) = \alpha\ \texttt{notnull}\ c_x \wedge \mathbf{ftype}(P, c_x, f) = \alpha_f\ \tau_f \wedge \tau_f <: \tau_1 \\ \mathbf{non\text{-}obj\text{-}tgts}(\psi) = \emptyset \Rightarrow \mathbf{f\text{-}tgt} \notin \psi' \ \wedge\ \mathbf{s\text{-}tgt} \notin \psi' \\ \psi_1 = \mathbf{non\text{-}obj\text{-}fresh\text{-}tgts}(\psi') \qquad \psi_1^\bullet \subseteq \kappa_f\end{array}}{P; E; \kappa \vdash (\tau_0, \psi) <: (\tau_1, \psi \cup \psi') \dashv (\kappa_f \setminus \psi_1^\bullet, \kappa_o)}$$

Figure 5.8: Sub Rules for Sources

- If the original target set ψ' only contains object target, then the target set cannot contain either **s-tgt** or **f-tgt**.

The first case is valid since each object target x in ψ represents that the value could be an alias to x. It is sound to be more conservative by adding more aliases (object targets) that the value may equal to.

In the second case, for a field capability $x.f^\bullet$, if the type of f can be relaxed to the final type τ_1, then we can also weaken the original target set by adding $x.f$ to be an alias of the original value. This effectively cost the capability and thus it needs to be removed from the input.

For the third case, **s-tgt** cannot appear in ψ'. To include it in ψ', we need to prove that there exists a path: $a.f_0.f_1 \ldots .f_n$, such that:

- a is a valid variable in E

- f_n is a unique field of type τ, and ε is a subtype of ε_1.
- At least one ancestor of f_n in the path is shared.

This is complicated to define and transform to fractional permissions. At the mean time, we choose to be more conservative and disallow this.

The third case also requires that if the input target set only contains object target, then the added target set ψ' can not contain fresh target **f-tgt**. This is because of the way we transform the system into fractional permissions, and shall be discussed more in Chap. 6.

Therefore, in the current system, expressions such as

```
if (..) then a else new C()
```

(assuming `a` is unique) cannot be checked, because on the left hand side the result target set only contains an object target `a`, while the result target set for the right hand side contains **f-tgt**.

Also:

```
if (..) then a else b.f.g
```

can not be type checked, if the type of `b.f.g` is not the same as that of `a`. The type system only tracks the first field, and to know that `b.f.g` has the right type, we need to either do a search on all the direct or indirect fields of `a.f`, or track more information in the type system. At the moment, we choose to disallow this.

Next is the IF rule. For an "if" expression to be well-typed, both of its branches need to be well-typed, and have identical outputs (type, source, and capabilities). This can be achieved by applying the SUB rule. The result of the "if" rule is simply the result of either of its branches.

If a variable is of type `nullable` c, then it stays possibly-null forever, and none of its fields can ever be read or written. This is very restrictive. A IfNull rule can be used to address this issue. Similar to the one we defined in Chap. 4. In this rule, from the result of checking the condition part of the "if" expression, it is guaranteed that the variable x is not null when checking the "else" branch, and therefore the input environment can be updated with that information.

The rule Loop types a while loop expression. After checking e, the result capabilities κ_2, combined with the pinned capabilities in the result targets of e, should be equal to the input capabilities κ_0. This is similar to the Loop rule defined in Fig. 3.4 of Chap. 3. The requirement is needed because we need to re-establish the loop invariant after each iteration of the body expression. Similarly to the rule Null, the result type for this rule can have any well-formed class, and is possibly-null.

Fig. 5.9 defines a few type rules on a "let" expression.

LetUnique

$$\frac{\begin{array}{c} P; E; \kappa_0 \vdash e_0 : (\varepsilon_0\ c_0, \psi) \dashv \kappa_1 \\ \kappa_2 = (\mathbf{extend\text{-}flds}(\psi, \mathbf{fields}(c_0))^\bullet, \mathbf{obj\text{-}tgts}(\psi)) \qquad \kappa_2 \subseteq \kappa_1 \\ \kappa_3 = (\{\ x.f^\bullet \mid f \in \mathbf{fields}(c_0)\ \}, \{x\}) \\ x \notin E \qquad P; E, x : \texttt{unique}\ \varepsilon_0\ c_0; \kappa_1 \setminus \kappa_2 \cup \kappa_3 \vdash e_1 : (\tau_1, \beta) \dashv \kappa_4 \\ \kappa_3 \subseteq \kappa_4 \qquad \mathbf{not\text{-}in\text{-}source}(x, \beta) \end{array}}{P; E; \kappa_0 \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : (\tau_1, \beta) \dashv \kappa_4 \setminus \kappa_3 \cup \kappa_2}$$

LetShared

$$\frac{\begin{array}{c} P; E; \kappa_0 \vdash e_0 : (\tau_0, \mathbf{shared}) \dashv \kappa_1 \\ x \notin E \qquad P; E, x : \texttt{shared}\ \tau_0; \kappa_1 \vdash e_1 : (\tau_1, \beta) \dashv \kappa_2 \end{array}}{P; E; \kappa_0 \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : (\tau_1, \beta) \dashv \kappa_2}$$

Seq

$$\frac{\begin{array}{c} P; E; \kappa_0 \vdash e_0 : (\tau_0, \beta_0) \dashv \kappa_1 \\ \mathbf{pinned\text{-}capabilities}(\beta_0) = \kappa \\ P; E; \kappa_1 \cup \kappa \vdash e_1 : (\tau_1, \beta_1) \dashv \kappa_2 \end{array}}{P; E; \kappa_0 \vdash (e_0\texttt{;}\ e_1) : (\tau_1, \beta_1) \dashv \kappa_2}$$

Figure 5.9: Type Rules for the Conservative Type System (Part 3)

In LetUnique, the result value from checking e_0 is unique, and has targets ψ. To check e_1, we need to add a fresh variable x into the environment, and also *all field capabilities for* x, for checking e_1.

To ensure that all the necessary capabilities for x are available when checking e_1, the type system requires that the capabilities of all the targets associated with the reference of e_0 are available. This is defined in a restricted way: all the field capabilities for the unique value of e_0 are removed from the input, and replaced by field capabilities on x. Also, all object capabilities for object targets (**obj-tgts**(ψ)) in ψ are removed from input.

Also, when checking the let body e_1, it is not allowed to consume any capability for the let-bound variable x; all of these are restored after the checking.

Finally, the variable x is not allowed to appear in the result sources after checking the let body. In other words, the variable is not allowed to "leak" out of the "let" body. This is defined by the function **not-in-sources** in Fig. 5.5.

For instance, expression such as:

```
let x = ... in ...; x
```

is not permitted.

Also, some obvious correct expression such as:

```
let x = new C() in a.u = x;
```

is also not permitted.

These constraints make the let bound expression for the unique case particularly restrictive, especially for the second example above, which seems obviously correct. In order to remove these constraints, we need to track the sources for the variable x, and in result, for every capability that x occurs, substitute it with the actual capabilities it comes from. This is complex due to the difficulties in transforming to fractional permissions, and we choose

to left this out as a future work. We will discuss some enhancements that can be done in Chap. 7.

In comparison, the type rule LetShared is much simpler. If the type for e_0 is shared, a fresh variable x with shared type is added to the environment, which is used to check the "let" body.

Seq defines how a sequence expression can be checked. After type checking the first expression e_0, we can claim back the pinned field capabilities and put them back to the available capabilities. They can then be reused to check e_1.

Now let us take a look at a simple example:

```
let x =
  if (..) then a.k else b.l
in
  (x.f = c.g; null)
```

Assuming `a`, `b` and `c` are unique references, and `f`, `g`, `l` and `k` are unique fields, and, for simplicity, that class of `a` only contains one field `f`. The input capabilities required to check this expression are $(\{\texttt{a.k}^\bullet, \texttt{b.l}^\bullet, \texttt{c.g}^\bullet\}, \emptyset)$.

The type checking process for this expression is performed as following:

1. Type checking the "if" expression, which in turn requires checking `a.k` and `b.l`. After checking the former, the result capabilities are $(\{\texttt{b.l}^\bullet, \texttt{c.g}^\bullet\}, \emptyset)$, and for the latter, are $(\{\texttt{a.k}^\bullet, \texttt{c.g}^\bullet\}, \emptyset)$. After applying the Sub rule, we can derive the same source $\{\texttt{a.k}, \texttt{b.l}\}$, and same capabilities $(\{\texttt{c.g}^\bullet\}, \emptyset)$ for both branches.

2. Calculating capabilities κ_2 and κ_3 in the rule. For κ_2, it is $(\emptyset, \emptyset)$ since there is no object target in the result target set after checking the "if" expression. For κ_3, it is $(\{\texttt{x.f}^\bullet\}, \{\texttt{x}\})$. Therefore, after checking the initialization expression for the let-bound

variable, the capabilities are $(\{\texttt{c.g}^\bullet, \texttt{x.f}^\bullet\}, \{\texttt{x}\})$, which is κ_3 combined with the result capabilities after checking the "if" expression.

3. Type checking the expression `x.f = c.g; null`. This first checks the expression `x.f = c.g`. Since from the previous step we have obtained $\texttt{x.f}^\bullet$, and also capability for `c.g` is in the output capabilities after checking the "if" expression. This can be type checked. For the whole expression, the result source is {**f-tgt**} from `null`, and result capabilities are $(\{\texttt{x.f}^\bullet\}, \{x\})$, as capability $\texttt{c.g}^\bullet$ is consumed.

4. Checking that `x` is not in the result of `x.f = c.g; null`, using **not-in-source**. This is obviously true.

5. Removing $\texttt{x.f}^\bullet$ from the remaining field capabilities, and `x` from the object capabilities. Also we need to claim back capabilities for the targets in the result of the "if" expression. Therefore, the final result capabilities are $(\{\texttt{a.k}^\bullet, \texttt{b.l}^\bullet\}, \emptyset)$.

Note that in this example, we need to put a `null` at the end. For expression:

```
let x =
  if (..) then a.k else b.l
in
  x.f = y.g
```

cannot be typed under the type system, because variable `x` occurs in the result targets.

The second part of the rules for "let" expression are defined in Fig. 5.10.

Rule LetRestore defines how a consumed field capability can be restored through another field assignment. This rule is useful when one wants to consume some unique field and then restore it using another unique value. A motivating example is shown in Sec. 5.4.3 at the end of this chapter.

In this rule, it first consumes the field capability for the target $x'.f$, whose type must be unique. It then restores the uniqueness of $x'.f$ by assigning the result of another unique expression e to it. During type checking the expression e, all capabilities for the let-bound variable x are added into the input capabilities, and can be freely used. However, after finishing type checking e, all remaining capabilities for x are discarded.

LetRestore

$$\frac{\begin{array}{c} E(x') = \alpha\ \texttt{notnull}\ c \qquad \textbf{annot-to-source}(x', \alpha) = \beta \\ \textbf{ftype}(P, c, f) = \texttt{unique}\ \varepsilon_f\ c_f \qquad \textbf{read-field}(\beta, \kappa_0, f) = \psi, \kappa_1 \\ \kappa = (\{\ x.f \mid f \in \textbf{fields}(c_f)\ \}^{\bullet}, \{x\}) \qquad \kappa_1 \cap \kappa = \emptyset \\ P; E, x : \texttt{unique}\ \varepsilon_f\ c_f; \kappa_1 \cup \kappa \vdash e : (\tau_1, \psi_1) \dashv \kappa_2 \\ \kappa_2 \vdash (\tau_1, \psi_1) <: \texttt{unique}\ \varepsilon_f\ c_f \dashv \kappa_3 \end{array}}{P; E; \kappa_0 \vdash \texttt{let}\ x = x'.f\ \texttt{in}\ x'.f = e : (\varepsilon_f\ c_f, \psi) \dashv \kappa_3 \setminus \kappa}$$

LetSwap

$$\frac{\begin{array}{c} E(x') = \alpha\ \texttt{notnull}\ c \qquad \textbf{annot-to-source}(x', \alpha) = \beta \\ \textbf{ftype}(P, c, f) = \texttt{unique}\ \tau_f \qquad \textbf{read-field}(\beta, \kappa_0, f) = \psi, \kappa_1 \\ x \text{ does not occur in } e \qquad P; E; \kappa_1 \vdash e : (\tau_1, \psi_1) \dashv \kappa_2 \qquad \kappa_2 \vdash (\tau_1, \psi_1) <: \texttt{unique}\ \tau_f \dashv \kappa_3 \\ \textbf{pinned-capabilities}(\psi) = \kappa_4 \end{array}}{P; E; \kappa_0 \vdash \texttt{let}\ x = x'.f\ \texttt{in}\ (x'.f = e; x) : (\tau_f, \{\textbf{f-tgt}\}) \dashv \kappa_3 \cup \kappa_4}$$

Figure 5.10: Type Rules for the Conservative Type System (Part 4)

For the LetRestore rule, the let-bound variable x is not allowed to leak from the body. But, sometimes we do want to pass the result to other places. Consider the following example:

```
class A {
  unique nullable B item;

  ...

  unique notnull B getItem() borrowed(item) {
```

```
    B ret = this.item;
    this.item = null;
    return ret;
  }
}
```

In the method `getItem`, it first saves the unique value of field `item` using a local variable. It then nullifies the field, and return its old value using the local variable. This essentially transfers the unique value that `item` points to to another place.

To permit cases like this, we can use an extra rule, which we call LetSwap, defined in Fig. 5.10. It is similar to the `swap` function defined in the capability based type system proposed by Haller and Odersky [HO10]. In the rule, we must first be able to lookup x' and it should not be null. The field f should also be a valid field and is unique. It then requires that the let-bound variable x cannot appear in e, and therefore no capability for x will be consumed. Lastly, the result from checking e is used to restore the uniqueness for $x'.f$, and the pinned capabilities for this expression are restored in the output capabilities. The final target is a singleton set containing the fresh target.

In Fig. 5.11, rule Call defines how a method call is checked. Firstly, the method call receiver e_0 needs to be type checked, and is not null. Secondly, the method should be well-typed, which is specified by the function **mtype**.

A method type is of the following format:

$$(\alpha_1\ \tau_1), \ldots, (\alpha_n\ \tau_n) \longrightarrow \alpha\ \tau$$

where the $\alpha\ \tau$ on the right hand side of the arrow is the return type for the method. It is restricted to be either `unique` or `shared`. Also, for a non-constructor method type, the first parameter type is always the type for the method call receiver.

The next step for checking a method call is to type check every argument. We also need to make sure that the type of the method call receiver and each actual parameter is a subtype of that of the declared parameter. This is done using the subtyping rule defined in Fig. 5.13. The type rule also ensures that the input capabilities contain the necessary capabilities for each method parameter. Those are then removed as side-effect of the subtyping rules. For borrowed parameters, their capabilities come from the fields specified in the annotation. These are also removed, but will be recovered after the type checking of method body is done.

CALL

$$\frac{\begin{array}{c} P; E; \kappa_0 \vdash e_0 : (\alpha\ \texttt{notnull}\ c, \beta_0) \dashv \kappa_1 \\ \mathbf{mtype}(c, mn) = (\alpha_0\ \tau_0), (\alpha_1\ \tau_1), \ldots, (\alpha_n\ \tau_n) \longrightarrow \alpha_r\ \tau_r \\ \kappa_1 \vdash (\texttt{notnull}\ c, \beta_0) <: \alpha_0\ \tau_0 \dashv \kappa_2 \qquad \forall i \in [0, n) : P; E; \kappa_{2i+2} \vdash e_{i+1} : (\tau'_{i+1}, \beta_{i+1}) \dashv \kappa_{2i+3} \\ \kappa_{2i+3} \vdash (\tau'_{i+1}, \beta_{i+1}) <: \alpha_{i+1}\ \tau_{i+1} \dashv \kappa_{2i+4} \qquad \kappa = \kappa_{2n+2} \cup \bigcup_{i=1}^{n} \mathbf{reclaim\text{-}borrowed}(\beta_i, \alpha_i) \\ \mathbf{annot\text{-}to\text{-}result\text{-}source}(\alpha) = \beta \end{array}}{P; E; \kappa_0 \vdash e_0.mn(e_1 \ldots e_n) : (\tau_r, \beta) \dashv \kappa}$$

NEW

$$\frac{\begin{array}{c} \mathbf{mtype}(c, c) = (\alpha_1\ \tau_1), \ldots, (\alpha_n\ \tau_n) \longrightarrow \texttt{unique notnull}\ c \\ \forall i \in [0, n) : P; E; \kappa_{2i+1} \vdash e_{i+1} : (\tau'_{i+1}, \beta_{i+1}) \dashv \kappa_{2i+2} \\ \kappa_{2i+2} \vdash (\tau'_{i+1}, \beta_{i+1}) <: \alpha_{i+1}\ \tau_{i+1} \dashv \kappa_{2i+3} \qquad \kappa = \kappa_{2n+1} \cup \bigcup_{i=1}^{n} \mathbf{reclaim\text{-}borrowed}(\beta_i, \alpha_i) \end{array}}{P; E; \kappa_1 \vdash \texttt{new}\ c(e_1 \ldots e_n) : (\texttt{unique notnull}\ c, \{\mathbf{f\text{-}tgt}\}) \dashv \kappa}$$

Figure 5.11: Type Rules for the Conservative Type System (Part 5)

For checking method arguments, the input capabilities are the output from checking the receiver, and these capabilities are passed along for checking every parameter. At each step, parts of the capabilities may be removed.

After the method body is checked, we need to reclaim capabilities for those borrowed

method parameters. This is defined by the **reclaim-borrowed** function in Fig. 5.5. For each argument, the pinned field capabilities as well as the capabilities for the declared fields are restored after the method call is checked.

Lastly, the result source for a method call is determined by the declared method return type: if it is `unique`, then the result source is a singleton target set containing the **f-tgt**; if it is `shared`, then the result source is **shared**. A method return type is not allowed to be borrowed. In Chap. 7, we will discuss the possibility of adding a "from" return type as a future work. This is useful for scenarios such as external iterators [BRZ07].

Rule NEW defines how a new object is allocated and instantiated. This is a special case of a method call, in that the return type for the constructor is required to be unique and not null. The result target is a singleton set containing **f-tgt**.

Now consider the following example for method call:

```
unique nullable A
foo(unique notnull B a) borrowed(f,g) {
  ..
}

this.foo(new B());
```

In this example, the method type for `foo` is:

$$((\texttt{borrowed}(\texttt{f},\texttt{g})\ \texttt{notnull}\ \texttt{C}), (\texttt{unique}\ \texttt{notnull}\ \texttt{B})) \longrightarrow \texttt{unique}\ \texttt{nullable}\ \texttt{A}$$

To type check the expression `this.foo(new B())`, assuming the input capabilities are $(\{\texttt{this.f}^\bullet, \texttt{this.g}^\bullet\}, \emptyset)$, the following steps are performed:

1. Type checking the method call receiver, which in this case is simply the `this` variable. The rule VAR is applied and no change on the capabilities.

2. Checking whether the method `foo` has a well-formed method type. This is also checked and the method type is the one we listed above.

3. Type checking each arguments. In this case, there are two arguments, `this` and `new B()`. For the argument `this`, again the VAR is applied first, and then followed by the subtyping rule SUBBORROWED. Since this method requires field capabilities $\{\texttt{this.f}^\bullet, \texttt{this.g}^\bullet\}$, they are removed from the input. For the second argument `new B()`, it is well-typed and has result type `unique notnull B`, and the result source is a singleton target set containing **f-tgt**. The subtyping rule is then applied for this argument, and since in both cases only fresh objects are involved, no change happens in input capabilities. Therefore, after checking all the arguments, the final capabilities are $(\emptyset, \emptyset)$.

4. Reclaiming the borrowed capabilities for each method argument. For the borrowed method argument `this`, the associated capabilities $\{\texttt{this.f}^\bullet, \texttt{this.g}^\bullet\}$ are reclaimed. There is no pinned field capability.

5. For the return type, since the declared method return type is `unique`, the result type is `notnull A` and result source is {**f-tgt**}.

5.3.2 Type Rules for Bool Expressions

Type checking rules for bool expressions are defined in Fig. 5.12.

First, for the type rule TRUE, it is simple. The input and output capabilities are the same in this case.

For the type rule NOT, in order to type check the expression `not` b, the sub-expression b needs to type check. The final capabilities are the same as the one generated from checking

b.

For the rule EQ, first, the sub-expressions e_0 and e_1 need to be separately type checked. After the checking is done, for the case that the result is unique, all pinned capabilities for both expressions are no longer used, and therefore should be reclaimed. This is described by **pinned-capabilities** defined in Fig. 5.5.

For the rule AND, again, both sub-expression b_0 and b_1 must be type checked. However, for b_1, the output capabilities are required to be the same as the input capabilities. This is because fractional permissions allow short-circuiting for an "and" expression, and therefore when converting to permissions, both branches need to have the same set of permissions.

$$\textsc{True}\ \frac{}{P; E; \kappa \vdash \texttt{true} \dashv \kappa} \qquad \textsc{Not}\ \frac{P; E; \kappa_0 \vdash b \dashv \kappa_1}{P; E; \kappa_0 \vdash \texttt{not}\ b \dashv \kappa_1} \qquad \textsc{Eq}\ \frac{\begin{array}{l} P; E; \kappa_0 \vdash e_0 : (\tau_0, \beta_0) \dashv \kappa_1 \\ P; E; \kappa_1 \vdash e_1 : (\tau_1, \beta_1) \dashv \kappa_2 \\ \textbf{pinned-capabilities}(\beta_0) = \kappa_3 \\ \textbf{pinned-capabilities}(\beta_1) = \kappa_4 \end{array}}{P; E; \kappa_0 \vdash e_0 == e_1 \dashv \kappa_2 \cup \kappa_3 \cup \kappa_4}$$

$$\textsc{And}\ \frac{P; E; \kappa_0 \vdash b_0 \dashv \kappa_1 \qquad P; E; \kappa_1 \vdash b_1 \dashv \kappa_1}{P; E; \kappa_0 \vdash b_0\ \texttt{and}\ b_1 \dashv \kappa_1}$$

Figure 5.12: Rules for Bool Expressions

5.3.3 SubTyping Rules

The subtyping rules are defined in Fig. 5.13. A subtyping rule is of format:

$$\kappa_1 \vdash (\tau_1, \beta_1) <: \alpha_2\ \tau_2 \dashv \kappa_2$$

This is read: under input capabilities κ_1, type τ_1 and source β_1 is subtype of declared type $\alpha_2\ \tau_2$. The subtyping relation also generates output capabilities κ_2.

SubUnique

$$\frac{\alpha \neq \texttt{borrowed}(\overline{f}) \qquad \varepsilon_1\ c_1 <: \tau_2 \qquad \kappa' = (\textbf{extend-flds}(\psi, \textbf{fields}(c_1))^\bullet, \textbf{obj-tgts}(\psi)) \qquad \kappa' \subseteq \kappa}{\kappa \vdash (\varepsilon_1\ c_1, \psi) <: \alpha\ \tau_2 \dashv \kappa \setminus \kappa'}$$

SubBorrowed

$$\frac{\tau_1 <: \tau_2 \qquad \kappa' = (\textbf{extend-flds}(\psi, \overline{f})^\bullet, \emptyset) \qquad \kappa' \subseteq \kappa}{\kappa \vdash (\tau_1, \psi) <: \texttt{borrowed}(\overline{f})\ \tau_2 \dashv \kappa \setminus \kappa'}$$

SubShared

$$\frac{\tau_1 <: \tau_2}{\kappa \vdash (\tau_1, \textbf{shared}) <: \texttt{shared}\ \tau_2 \dashv \kappa}$$

Figure 5.13: The Main Subtyping Rules

Nonnull

$$\frac{}{\texttt{notnull}\ c <: \texttt{nullable}\ c}$$

Ref

$$\frac{}{\tau <: \tau}$$

Figure 5.14: Sub Nonnull and Class Rules

Fig. 5.14 defines the sub relation for types. It is quite straightforward, except that we require the class identifiers to be the same on both sides, since there is no inheritance in our kernel language.

The most interesting rule is SubUnique: when passing a unique value to a unique or shared field, all associated capabilities for the expression are considered as consumed, and be removed from the input.

For the SubBorrowed rule, it is defined in a similar way as SubUnique. The difference is that the set $\overline{f}$ is used instead of all the fields of class c. Also, object capabilities are unchanged, since a borrowed parameter is never allowed to be consumed as a whole object.

For a borrowed parameter, the associated field capabilities are taken away from the input,

just like SubUnique. However, it is required that they should be restored after the method call is checked. It is illegal to consume any of these capabilities when checking the method body.

Rule SubShared is straightforward. When passing a shared expression to a shared field, capabilities are unchanged.

5.3.4 Well-formedness

In Fig. 5.3.4, rules for well-formedness of various constructs of a program are defined. In the following we shall explain each of them in order.

First, the rule Field describes what is a well-formed field declaration. This requires that field f must be a valid field identifier for the class c, and the field annotation is not borrowed.

The rule Method defines the well-formedness for a method declaration. First, the method return type must not be borrowed. Each method parameter is also accompanied by a set of fields that the method body may read or write. The function **CheckFields** defined in Fig. 5.16 checks whether this set of fields are valid, i.e., whether it is a subset of the fields of the declared class for the method parameter.

Next, for checking the method body e, we need to construct an input environment and ainput/output capabilities. For the input environment, it simply contains a mapping from each method parameter to its declared type. For the input and output capabilities, they are derived by using the function **InputCaps** and **OutputCaps** respectively, which are defined in Fig. 5.16. The only interesting case is for borrowed method parameter, for which the associated capabilities need to appear in both input and output. The input and output capabilities should also include the **s-tgt**$^\bullet$ for accessing anything that is of type shared.

With the above constructed input environment E, input capabilities κ_{in} and output

FIELD

$$\frac{f \in \mathbf{fields}(c) \qquad \alpha \neq \texttt{borrowed}(\overline{f})}{P, c \vdash \alpha\ \tau\ f}$$

METHOD

$$\frac{\begin{array}{c}\alpha_r \neq \texttt{borrowed}(\overline{f}) \qquad \tau_0 = \texttt{notnull}\ c \qquad E = \texttt{this} : \alpha_0\ \tau_0, x_i : \alpha_i\ \tau_i \\ \forall i \in [0,n] : \mathbf{CheckFields}(\alpha_i, \tau_i) \\ \kappa_{in} = \bigcup \mathbf{InputCaps}(\alpha_i, \tau_i, x_i) \cup (\{\textbf{s-tgt}^\bullet\}, \emptyset) \\ \kappa_{out} = \bigcup \mathbf{OutputCaps}(\alpha_i, x_i) \cup (\{\textbf{s-tgt}^\bullet\}, \emptyset) \\ P; E; \kappa_{in} \vdash e : (\tau_r', \beta_r) \dashv \kappa \qquad \kappa \vdash (\tau_r', \beta_r) <: \alpha_r\ \tau_r \dashv \kappa_{out}\end{array}}{P, c \vdash \alpha_r\ \tau_r\ mn(\alpha_1\ \tau_1\ x_1, \ldots, \alpha_n\ \tau_n\ x_n)\ \alpha_0\ \{\ e\ \}}$$

CONSTRUCTOR

$$\frac{\begin{array}{c}\forall i \in [1,n], j \in [1,m] : E = x_i : \alpha_i\ \tau_i \\ \mathbf{CheckFields}(\alpha_i, \tau_i) \qquad \kappa_1 = \bigcup \mathbf{InputCaps}(\alpha_i, \tau_i, x_i) \cup (\{\textbf{s-tgt}^\bullet\}, \emptyset) \\ \kappa_{2m+1} = \bigcup \mathbf{OutputCaps}(\alpha_i, x_i) \cup (\{\textbf{s-tgt}^\bullet\}, \emptyset) \\ \text{all not-null fields are in } f_1 \ldots f_m \qquad \mathbf{ftype}(P, c, f_j) = \alpha_{f_j}\ \tau_{f_j} \\ P; E; \kappa_{2j-1} \vdash e_j : (\tau_j, \beta_j) \dashv \kappa_{2j} \qquad \kappa_{2j} \vdash (\tau_j, \beta_j) <: \alpha_{f_j}\ \tau_{f_j} \dashv \kappa_{2j+1}\end{array}}{P, c \vdash cn(\alpha_1\ \tau_1\ x_1, \ldots, \alpha_n\ \tau_n\ x_n)\ \{\ \texttt{this}.f_1 = e_1\texttt{;}\ \ldots\ \texttt{;this}.f_m = e_n\texttt{;this}\ \}}$$

CLASS

$$\frac{cn \in \mathbf{classes}(P) \qquad P, cn \vdash \mathit{field}_i \qquad P, cn \vdash \mathit{constr} \qquad P, cn \vdash \mathit{meth}_i}{P \vdash \texttt{class}\ cn\ \{\ \mathit{field}_1, \ldots, \mathit{field}_n\ \mathit{constr}\ \mathit{meth}_1, \ldots, \mathit{meth}_n\ \}}$$

PROG

$$\frac{P = \mathit{defn}_1, \ldots, \mathit{defn}_n \qquad P \vdash \mathit{defn}_i}{P \vdash \diamond}$$

Figure 5.15: Rules for Well-formed Classes and Methods

$$\textbf{CheckFields}(\alpha,\tau) = \text{if } \alpha = \texttt{borrowed}(\overline{f}) \text{ then } \overline{f} \subseteq \textbf{fields}(c)$$

$$\textbf{InputCaps}(\alpha,\, \tau,\, x) = \begin{cases} (\{\ x.f \mid f \in \textbf{fields}(c)\ \}, \{x\}) & \alpha = \texttt{unique} \\ (\emptyset, \emptyset) & \alpha = \texttt{shared} \\ (\{\ x.f \mid f \in \overline{f}\ \}, \emptyset) & \alpha = \texttt{borrowed}(\overline{f}) \end{cases}$$

$$\textbf{OutputCaps}(\alpha,\, x) = \begin{cases} (\emptyset, \emptyset) & \alpha = \texttt{unique} \\ (\emptyset, \emptyset) & \alpha = \texttt{shared} \\ (\{\ x.f \mid f \in \overline{f}\ \}, \emptyset) & \alpha = \texttt{borrowed}(\overline{f}) \end{cases}$$

Figure 5.16: Auxiliary Functions for Well-formed Methods and Classes

capabilities κ_{out}, for a well-formed method declaration, the method body e needs to be type checked under E and κ_{in}, and the result type should be a subtype of the declared return type. The final capabilities should be κ_{out}.

Similarly, the rule CONSTRUCTOR type checks a constructor declaration. Notice that a constructor declaration does not contain type for receiver or returned value. Also, the constructor body has to conform to a particular format: a list of field assignments, followed by returning `this`. Finally, it is required that all the not-null fields should appear in the field assignments, to ensure they are initialized.

The rule CLASS defines the well-formedness of a class declaration. It is straightforward: for a class declaration *defn* to be well-formed, all the field and method declarations in *defn* are required to be well-formed.

Lastly, the rule PROG defines what is a well-formed program in the system. All the class declarations in the program need to be well-formed.

5.4 Examples

In this section we go through several examples that demonstrate how the various pieces of the type system works together.

5.4.1 Example 1: Illegal Consumption

First, let us consider an example that is adopted from Retert's thesis [Ret09]:

```
class UniqueDemo {
  unique notnull Object o1;
  unique notnull Object o2;

  ...

  void bad() borrowed(o1,o2) {
    this.o1 = this.o2;
    null
  }
}
```

In this example, class `UniqueDemo` contains two unique non-null fields `o1` and `o2`. For the method `bad`, if we allow it to be checked, then the uniqueness of object initially pointed by `o2` will be violated, since after the method call it would be pointed by both `o1` and `o2`. Here, this method will not be type checked, since in default the method receiver is `borrowed`($\overline{f}$), and the field capability for `this.o2` is required to be in the output capabilities after checking the method, but in this case it does not.

5.4.2 Example 2: Restoring Consumed Field Capability

This example is taken from the paper on comprehending annotations using fractional permissions, by Boyland, Retert and Zhao [BRZ09]. It is slightly modified here:

```
class UNode {
  unique nullable UNode next;
  shared nullable Object datum;

  ...

  void append(borrowed(next) notnull Unode n,
              unique notnull UNode m) borrowed() {
    let x = n.next in
      n.next = (m.next = x; m);
    null
  }
}
```

The method body of `append` can be checked by the LetRestore rule. Here, since method parameter `n` is borrowed, the method `append` has the field capability $\texttt{n.next}^\bullet$ both as input and output. In the method body, the capability is first consumed as effect of passing to `m.next`, and then immediately restored by consuming the unique argument `m`.

5.4.3 Example 3: Searching in a Linked List

This is another example for linked list. The function `member` looks for `data` in the linked list starting from node `head`. It returns the `data` if any of the node contains it. Otherwise, if the `data` is not found in any of the list node, it returns `null`.

```
1   class Node {
2     shared nullable Data val;
3     unique nullable Node next;
4   }
5
6   shared nullable Object
7   member(borrowed(val, next) nullable Node head,
8          shared nullable Object data) borrowed() {
9     if (head == 0) then null
10    else if (head.val == data) head.val
11    else member(head.next, data);
12  }
```

For the `member`, its method type is:

$$((\texttt{borrowed}(\texttt{val}, \texttt{next})\ \texttt{nullable Node}), (\texttt{shared nullable Object})) \longrightarrow \texttt{shared nullable Object}$$

The interesting part in this example is the recursive call on the `member` function, at line 11. Since the result target for `head.next` is {`head.next`}, and field capability `head.next`$^\bullet$ is available at the entry of the call, the SubBorrowed rule is applied and `head.next`$^\bullet$ is removed from the input. It is then restored after the call. For the `head.val` at line 10, it can also be type checked, because the capability `head.val`$^\bullet$ is available at that point. The field capability will not be pinned since the field `val` is shared.

Chapter 6

Soundness of the Conservative Type System

In this chapter, we describe how the soundness of the conservative type system defined in the last chapter can be proved. Similar to the non-null type system described in Chap. 4, the soundness is proved by converting each component in the conservative type system to the corresponding parts under fractional permission system. The goal is that, since the fractional permission system is already proven sound, if we can show that the conversion to fractional permissions is valid, then the conservative type system itself is sound as well. In particular, we will need to show that if an expression can be type checked under the conservative type system, and if all the environments, input and output can be converted to the corresponding pieces in the fractional permission system, then the same expression can be type checked under the latter.

6.1 Conversion to Fractional Permissions

In this section, we describe the conversion from the conservative type system to fractional permission system, in a bottom-up order.

6.1.1 Conversion for Field Types

The first step is to transform each class definition to a class predicate [BS11]. A class predicate describes invariants about the class. It is (roughly) a set of invariants about each field inside the class. Invariants are expressed through fractional permissions. For instance, whether this field is not null or possibly null, or whether this field is unique or shared. Therefore, on the lower level we need to convert each field declaration to the corresponding permission about the field.

Recall that in Sec. 5.1, a field has annotated type $\alpha\ \varepsilon\ c$, while α can be either `unique` or `shared`, and ε can be `notnull`, which indicates the field can never be null, or `nullable`, which indicates the field may or may not be null.

Depending on the nullness of the field, we need to construct two different forms of permissions. In the following context we shall assume the field is f of an object o. For the case that f is possibly null (`nullable`), the result permission is of the format:

$$\exists r \cdot ((o.f \to r) + (r = 0\ ?\ \emptyset : \pi_f))$$

where π_f is the rest of permissions for f. Depending on the annotation of the field, π_f may be different. We shall show how this is done later in the section.

For the case where f is not null (`notnull`), the corresponding permission is of the format:

$$\exists r \cdot ((r \neq 0) + (o.f \to r) + (r = 0\ ?\ \emptyset : \pi_f))$$

In this case, besides the conditional permission, there is also the fact that r is not null. This can be used to "unlock" the conditional permission, and obtain the permission in the "else" branch, i.e., π_f. One may wonder why we do not choose to use the more concise format:

$$\exists r \cdot ((r \neq 0) + (o.f \rightarrow r) + \pi_f)$$

This is because in the SUB rule (Fig. 5.7) we allow a not-null field to be treated as a possibly-null field, and correspondingly, we need to transform the permission for a field target where the field is not-null:

$$\exists r \cdot (((o.f \rightarrow r) + (r = 0 \,?\, \emptyset : \pi_f)) \multimap \pi)$$

to a permission for field target where the field is possibly-null:

$$\exists r \cdot (((r \neq 0) + (o.f \rightarrow r) + (r = 0 \,?\, \emptyset : \pi_f)) \multimap \pi)$$

Here, the field permission is encumbered in some other permission π. Since permission transformation inside a encumbered permission is very restrictive, this cannot be done if the field permission is not a conditional permission.

To construct the permission π_f, we need to consider annotation α. Suppose α is `unique`, the result permission is:

$$\pi_f \equiv r.\text{All} \rightarrow 0 + p(r)$$

That is, along with the class predicate $p(r)$ for the field's class, the whole permission for accessing r is included in π_f.

For `shared` annotation, the result permission is:

$$(r.\text{All} \rightarrow 0) \prec 0.\text{All}$$

In this case, π_f merely contains the fact that the whole permission for r is nested inside the "all" field of the null (or *world*) object (0). Thus, in order to obtain the whole permission,

one has to first obtain the permission for the special permission $0.\text{All} \to 0$, and then carve out the permission $r.\text{All} \to 0$ from it.

Conversion for field types is defined by the function **ty2perm** in Fig. 6.1 of Sec. 6.1.3. With this function, a field type $\alpha\ \tau$ for f of object r is converted to

$$\exists r \cdot (o.f \to r + \mathbf{ty2perm}(G, \alpha, \tau, r))$$

In Sec. 6.1.3, we shall describe this function in detail.

6.1.2 Conversion for Classes

In this section, we describe how to transform a class declaration to a class predicate in terms of fractional permissions.

As described in the last section, to construct the permission for a field f, the class predicate of f needs to be available. However, to get a class predicate we need all of its field permissions (which may include the permission for f). Hence, they are mutually-dependent.

A naive way would be for each class c, to construct class predicate for each of its field on the fly, that is, when processing a class c, we first construct predicates for each of its fields, and then construct the predicate for c, and fill the entry in the predicate map. However, in practice, as we have discussed in Sec. 4, the resulting permission would not be equivalent to its unfolded version, and thus the proof would not be sound. The other approach, therefore, is the same as we did for the non-null type system: we first construct class predicates all at once, and store them in a special construct *predicate map*. The difference is that, with the first approach we may construct more than one predicates for a class c, while the second approach guarantees that for each class c, exactly one predicate is constructed.

A predicate map is defined as follow:

$$G ::= \cdot \mid G, c : p(x)$$

where p is predicate for class c. When processing a class c, if there is no associated predicate for c, we first associate the class with a newly created variable, and then use it to construct c's actual predicate. After the predicate is constructed, we use it to replace the variable for c, and then the algorithm moves on to the next unseen class. At the end of this process, for each class in the program, we will get a corresponding entry in the predicate map G, which maps the class identifier to its predicate.

For constructing the predicate for a class c, a similar algorithm is used to obtain all field permissions for c. The fields are processed one by one. Whenever we see a field whose associated class has no predicate, a new variable is used to create the actual predicate for the field's class. This process is similar to a depth-first-traversal on all the fields of c.

6.1.3 Conversion for Input and Output

The next step of the transformation is to convert the input and output of a type rule:

$$P; E; \kappa_0 \vdash e : (\tau, \beta) \dashv \kappa_1$$

to the corresponding permissions. For the input, it consists of the permissions from the environment E and the input capabilities κ_0. For the output, besides the permissions from E and output capabilities κ_1, it also includes the permissions from the type (τ, β) for the result value. The goal is that, after the transformation is done, we can type check the same expression under the fractional permission system, using the corresponding input and output permissions. In the following sections, we describe how the transformation is done for input and output.

Conversion for Input

For fractional permissions, the input is a program type (a mapping from procedures to their types) and a set of input permissions. In Sec. 6.1.4 we will define how methods in the program are converted to a program type under fractional permissions.

In Fig. 6.1, the function **ty2perm** defines how an annotated type $\alpha\ \tau$ is converted to permissions. Given the predicate map G constructed in the last section, an annotation α, a type τ, and a variable r, **ty2perm** constructs permissions for the type and the variable r. Note, to convert for a borrowed annotation, the rule does not need to know the set of fields. In a slight abuse of notation, instead of using $Borrowed\overline{f}$, we will use `borrowed` for this rule. In several occasions in the following we will need to use this function without having an existing borrowed annotation, and therefore with this change we do not need to obtain an arbitrary set of fields from nothing.

The output of this function varies depending on the nullness and the type of the annotation. Also, the conversion for the nullness is orthogonal to the conversion for the annotation α.

$$\textbf{ty2perm}(G,\ \alpha,\ \varepsilon\ c,\ r) =$$

$$\begin{cases} r \neq 0 + r = 0\ ?\ \emptyset : p(r) & \varepsilon = \texttt{notnull}, \alpha = \texttt{borrowed} \\ r = 0\ ?\ \emptyset : p(r) & \varepsilon = \texttt{nullable}, \alpha = \texttt{borrowed} \\ r \neq 0 + r = 0\ ?\ \emptyset : r.\text{All} \rightarrow 0 + p(r) & \varepsilon = \texttt{notnull}, \alpha = \texttt{unique} \\ r = 0\ ?\ \emptyset : r.\text{All} \rightarrow 0 + p(r) & \varepsilon = \texttt{nullable}, \alpha = \texttt{unique} \\ r \neq 0 + r = 0\ ?\ \emptyset : r.\text{All} \prec 0.\text{All} + p(r) & \varepsilon = \texttt{notnull}, \alpha = \texttt{shared} \\ r = 0\ ?\ \emptyset : r.\text{All} \prec 0.\text{All} + p(r) & \varepsilon = \texttt{nullable}, \alpha = \texttt{shared} \end{cases}$$

where $G(c) = p$

Figure 6.1: Converting Annotated Type to Permissions

Fig. 6.2 defines how an environment E is converted to a set of formulae. A formula,

unlike a permission, can be freely duplicated and discarded. The formulae generated here represent the facts we know about each method argument. The base case is defined EMPTY, where an empty environment is simply converted to an empty permission. For shared or borrowed types (NONEMPTY-SHARED and NONEMPTY-BORROWED), they are converted to the corresponding permissions using **ty2perm**. For NONEMPTY-BORROWED, it uses the annotation `borrowed` without the set of fields, as explained above. In NONEMPTY-UNIQUE, instead of `unique`, the `borrowed` is used for calling **ty2perm**. This is because the permissions for a unique variable come from the associated capabilities, not the environment.

$$\textsc{Empty}\quad \frac{}{G \vdash \cdot \Rightarrow \emptyset} \qquad \textsc{NonEmpty-Shared}\quad \frac{G \vdash E \Rightarrow \pi_0 \qquad \pi = \textbf{ty2perm}(G, \texttt{shared}, \tau, x)}{G \vdash E, x : \texttt{shared}\ \tau \Rightarrow \pi_0 + \pi}$$

$$\textsc{NonEmpty-Borrowed}\quad \frac{G \vdash E \Rightarrow \pi_0 \qquad \pi = \textbf{ty2perm}(G, \texttt{borrowed}, \tau, x)}{G \vdash E, x : \texttt{borrowed}(\overline{f})\ \tau \Rightarrow \pi_0 + \pi}$$

$$\textsc{NonEmpty-Unique}\quad \frac{G \vdash E \Rightarrow \pi_0 \qquad \pi = \textbf{ty2perm}(G, \texttt{borrowed}, \tau, x)}{G \vdash E, x : \texttt{unique}\ \tau \Rightarrow \pi_0 + \pi}$$

Figure 6.2: Converting Environment to Permissions

To convert capabilities to permissions, there are two parts: field capabilities ($x.f^\bullet$ or **s-tgt**$^\bullet$) and object capabilities (x).

For field capabilities, the related rules are defined in Fig. 6.3. The rule EMPTY handles the base case, where an empty capability set is simply converted to empty permission. Rule NONEMPTYCAP-UNIQUE defines how a field capability $x.f^\bullet$ is converted to permissions. It first looks up the type for the field f, and then uses **ty2perm** to generate the field permissions. Note that in this rule variable x must have `unique` annotation; if x is shared,

the capability on the target would be mapped on to the shared target, and therefore would be included in the capability **s-tgt**$^\bullet$ instead. Also, conditional permission is used in the result, regardless of the nullness of variable x. In case the x is not null, the fact $x \neq 0$ can be obtained from the permissions converted from the environment E.

In rule NonEmptyCap-Shared, the shared target is converted to the unique permission on the *All* field of the null object (0) encoding the "world" object. Since the shared target can only appear at most once in a capability set (guaranteed by the Method or Constructor rule in Fig. 5.3.4), at most one copy of this permission will appear in the output.

$$\textsc{EmptyCap}\quad \frac{}{P; E; G \vdash \cdot \Rightarrow \emptyset}$$

$$\textsc{NonEmptyCap-Unique}\quad \frac{P; E; G \vdash \kappa_f \Rightarrow \pi_0 \qquad E(x) = \alpha\ \varepsilon\ c \qquad \alpha = \texttt{unique} \vee \alpha = \texttt{borrowed}(\overline{f}) \qquad \mathbf{ftype}(P, c, f) = \alpha_f\ \tau \qquad \pi = \mathbf{ty2perm}(G, \alpha_f, \tau, r)}{P; E; G \vdash \kappa_f, x.f^\bullet \Rightarrow \pi_0 + (x = 0\ ?\ \emptyset : \exists r \cdot (x.f \rightarrow r + \pi))}$$

$$\textsc{NonEmptyCap-Shared}\quad \frac{P; E; G \vdash \kappa_f \Rightarrow \pi_0}{P; E; G \vdash \kappa_f, \texttt{shared}^\bullet \Rightarrow \pi_0 + 0.\mathrm{All} \rightarrow 0}$$

Figure 6.3: Converting Field Capabilities to Permissions

Fig. 6.4 defines how object capabilities are converted to permissions. In general, for each variable x in the set, we first collects permissions for every field of the object referred by x, and then encumbers these permissions from the unique permission for x ($x.\mathrm{All} \rightarrow 0$).

With conversion for both field and object capabilities done, the rule Caps2Perm defined in Fig. 6.5 defines how capabilities κ are converted to permissions. It is a simple combination of permissions from field capabilities and object capabilities.

EMPTY

$$\frac{}{P; E; G \vdash \cdot \Rightarrow \emptyset}$$

NONEMPTY

$$\frac{\begin{array}{cc} E(x) = \alpha\ \varepsilon\ c & P; E; G \vdash \kappa_o \Rightarrow \pi_0 \\ \forall f_i \in \mathbf{fields}(c) : \mathbf{ftype}(P, c, f_i) = \alpha_i\ \tau_i & \pi_{f_i} = \exists r_i \cdot (x.f_i \to r_i + \mathbf{ty2perm}(G, \alpha_i, \tau_i, r_i)) \end{array}}{P; E; G \vdash \kappa_o, x \Rightarrow \pi_0 + (x = 0\ ?\ \emptyset : ((\pi_{f_1} + \cdots + \pi_{f_n}) \multimap x.\mathrm{All} \to 0))}$$

Figure 6.4: Converting Object Capabilities to Permissions

CAPSTOPERM

$$\frac{P; E; G \vdash \kappa_f \Rightarrow \pi_f \qquad P; E; G \vdash \kappa_o \Rightarrow \pi_o}{P; E; G \vdash (\kappa_f, \kappa_o) \Rightarrow \pi_f + \pi_o}$$

Figure 6.5: Converting Capabilities to Permissions

In Chap. 5 we mentioned that to consume an object r as a whole, we need to:

- Remove all field capabilities for all fields of r
- Remove r from the object capabilities

In fractional permissions, with permissions from all the field capabilities of r, and the permissions from the object capability r, we can reconstruct the unique permission for r:

$$r = 0\ ?\ \emptyset : \exists r_1 \cdot (r.f_1 \to r_1 + \pi_{f_1}) + \cdots +$$

$$r = 0\ ?\ \emptyset : \exists r_n \cdot (r.f_n \to r_n + \pi_{f_n}) +$$

$$r = 0\ ?\ \emptyset : ((\exists r_1 \cdot (r.f_1 \to r_1 + \pi_{f_1}) + \cdots +$$

$$\exists r_n \cdot (r.f_n \to r_n + \pi_{f_n})) \multimap r.\mathrm{All} \to 0)$$

$$\models \text{(through linear } \textit{modus-ponens} \text{ rule)}$$

$$r = 0\ ?\ \emptyset : r.\mathrm{All} \to 0$$

With additional information from the environment (e.g., nullness, class predicate), we can obtain the unique permission for r and be able to consume it (e.g., passing to a unique field).

With the above definitions, the input permissions for checking an expression is the permissions from the environment E, combined with the permissions from the input capabilities.

Conversion for Output

For converting outputs from the conservative type system to the fractional permission system, in addition to the conversions for environment and capabilities, we also need to convert the output reference type (a pair of type τ and source B) and the output capabilities to the corresponding permissions.

Fig. 6.6 defines the conversion from a pair of type and source to permissions. The rule has the following format:

$$P; E; G; (o, r, v) \vdash (\tau, \beta) \Rightarrow \pi$$

This says that, under program P, environment E, and predicate map G, reference type (τ, β) can be converted to permission π for reference variables o, r, and permission variable v. Here, variable o represents the final value for the corresponding expression after evaluating it, and v represents some *unknown* permission that we do not track. This is used to encode a fresh target. Variable r represents some additional value that o could be equal to.

The rules for this conversion are defined in Fig. 6.6:

First, in case the source is shared (**shared**), it is simply converted to shared permissions using **ty2perm**. This is defined by the rule SHARED. The permission only contains the formula specifying that the unique permissions for the result value is nested in the unique permission for the null (0) object. To access these permissions, one needs to have permission $0.\text{All} \rightarrow 0$, which can be obtained from the **s-tgt$^\bullet$** capability.

SHARED

$$\frac{}{P; E; G; (o, r, v) \vdash (\tau, \textbf{shared}) \Rightarrow \textbf{ty2perm}(G, \texttt{shared}, \tau, o)}$$

NONLINEAR

$$\frac{\textbf{non-obj-tgts}(\psi) = \emptyset \qquad o \vdash \textbf{obj-tgts}(\psi) \Rightarrow \gamma}{P; E; G; (o, r, v) \vdash (\tau, \psi) \Rightarrow \gamma + \textbf{ty2perm}(G, \texttt{borrowed}, \tau, o)}$$

UNIQUE

$$\frac{\begin{array}{c}\textbf{non-obj-tgts}(\psi) \neq \emptyset \qquad o \vdash \textbf{obj-tgts}(\psi) \Rightarrow \gamma \\ \pi_1 = \textbf{ty2perm}(G, \texttt{unique}, \tau, r) \\ P; E; G \vdash \textbf{non-obj-tgts}(\psi)^\bullet \Rightarrow \pi_2 \\ \pi_3 = v \text{ if } \textbf{f-tgt} \in \psi \text{ else } \emptyset\end{array}}{P; E; G; (o, r, v) \vdash (\tau, \psi) \Rightarrow o = r \,?\, (\pi_1 + (\pi_1 \multimap (\pi_2 + \pi_3))) : \gamma}$$

Figure 6.6: Converting Reference Type to Permissions

EMPTY

$$\frac{}{o \vdash \cdot \Rightarrow \bot}$$

NONEMPTY

$$\frac{o \vdash \psi \Rightarrow \gamma}{o \vdash \psi, x \Rightarrow \gamma \vee o = x}$$

Figure 6.7: Converting Object Targets to Permissions

When the result source is unique, the case is more interesting. There are two sub cases: when there are only object targets in the result targets, or when there are field targets in the targets. Note that a target set ψ will never be empty, as enforced by the type rules.

For the first case (NONLINEAR) where all targets are object targets, there is no actual permission associated with the result value. Instead, the object targets are converted into a list of object equal relations between the result value (o) and each of the variables represented by the object targets. This conversion is described by rules defined in Fig. 6.7. They are quite straightforward.

Note the rule also produces permission for result value o using TY2PERM. Annotation `borrowed` is used since there is no actual permission for o.

With this, expression such as:

```
x = if (...) a else b
```

would not require any permission from either variable `a` or `b` until some field for `x` is actually used.

In the second case (UNIQUE), the result target set ψ not only may contain object targets, but also field targets, shared target, or fresh target. This means that there are pinned capabilities for the targets. In terms of fractional permissions, those capabilities are represented by permissions that are encumbered by the permissions for the result value. In addition to all the variables from the object targets in ψ, the result value o could also be equal to another variable r, for which the whole unique permission is available. This permission is encumbered in permissions for the pinned capabilities, represented by π_2 and π_3 in the rule. Therefore, unlike in NONLINEAR, **ty2perm** in rule UNIQUE takes a `unique` annotation as input, and therefore the whole permission (represented by the permission π_1 in the rule) is available when the result value o equals to the variable r.

For instance:

```
if (...) a else b.f
```

In this example, depending on the result from the condition part (omitted with ...), result value o for the "if" expression could be an alias to the result of `a`, or `b.f`. In the former case, we only have the knowledge that o is equal to result value of `a`, but have no actual permission on o, while in the latter case, not only we know that o is equal to the result value of `b.f`, but also that we have unique permission on the value. The unique permission comes from the permissions for the result value of `b.f`.

In the Sec. 5.3.1 of Chap. 5, we mentioned that expression such as:

```
if (..) then a else new C()
```

cannot be checked under the current type system, as restricted by the rule UNIQUE2UNIQUE defined in Fig. 5.8. One may wonder why we cannot obtain a common target set $\{\texttt{a}, \textbf{f-tgt}\}$ for this expression. In terms of fractional permissions, the left hand side of the above expression generates:

$$\exists o \cdot (o = r_a + \pi_{r_a})$$

Note π_{r_a} is generated through **ty2perm** where the input annotation is `borrowed`. Therefore, it does not contain any actual permission for r_a, merely formulae.

To derive a common target set $\{\texttt{a}, \textbf{f-tgt}\}$, the final permissions need to be:

$$\exists o, r, v \cdot (o = r \ ? \ \pi_r + (\pi_r \multimap v) : (o = r_a))$$

Note the π_r contains the full permission, including π_r..All for variable r. In order to prove that the rule is valid, we need to show that first set of permissions can be transformed into the second set of transformations. This is not possible because there is no way to derive π_r from π_{r_a}, in case r_a is not null.

In UNIQUE rule, a fresh target **f-tgt** is represented with the permission variable v. This represents some unknown permission that we do not track.

Finally, the rule EXPR in Fig. 6.8 describes how outputs from typing an expression e in the conservative type system are converted to fractional permissions. It is a combination of the 1), the permissions from the environment, 2) the permissions from the output capabilities, and 3), the permissions from the result type and source.

Also in Fig. 6.8, the rule COND defines how outputs from typing a boolean expression are converted to permissions. It is simpler than UNIQUE, in that there is no result type and source. The result is simply the permissions from the environment combined with permissions from the output capabilities.

$$\textsc{Expr}\ \frac{G \vdash E \Rightarrow \pi_1 \qquad P; E; G \vdash \kappa \Rightarrow \pi_2 \qquad P; E; G; (o, r, v) \vdash (\tau, \beta) \Rightarrow \pi_3}{P; E; G \vdash \langle (\tau, \beta), \kappa \rangle \Rightarrow \exists o, r, v \cdot (\pi_1 + \pi_2 + \pi_3)}$$

$$\textsc{Cond}\ \frac{G \vdash E \Rightarrow \pi_1 \qquad P; E; G \vdash \kappa \Rightarrow \pi_2}{P; E; G \vdash \kappa \Rightarrow (\pi_1 + \pi_2)}$$

Figure 6.8: Converting To Output Permissions

6.1.4 Conversion for Method Type

Each method type in the conservative type system corresponds to a procedure type under fractional permission system. In order to obtain a program type ω (defined in Chap. 3) from a program P, we need to iterate over all the methods and perform conversion for each method type. This is similar to what we did in Chap. 4.

Recall that a method type is of the following format:

$$(\alpha_1\ \tau_1, \ldots, \alpha_n\ \tau_n)\ \longrightarrow\ \alpha\ \tau$$

and a procedure type in fractional permissions is of the following format:

$$\forall_{\overline{r}} \forall_{\Delta}\ \Pi \rightarrow \exists_{r_0} \exists_{\Delta'}\ \Pi'$$

With the help of the definitions in the last few sections, we can define how a method type is converted to a procedure type in fractional permissions. This is shown in Fig. 6.9. First, each annotated parameter type $\alpha_i\ \tau_i$ as well as the result type $\alpha\ \tau$ is converted to permission π_i on a fresh variable x_i, using **ty2perm**. Then, functions **InputCaps** and **OutputCaps** (defined in Fig. 5.16) are used to collect the input and output capabilities, which are then converted to the corresponding input and output permissions for type checking the method body.

We also need to convert environment E to permissions and include them in both input and output. These contain information about each method argument, such as its nullness and class predicate.

METHTYPE

$$\frac{\begin{array}{c} G \vdash E \Rightarrow \pi_E \\ \pi_i = \mathbf{ty2perm}(G, \alpha_i, \tau_i, x_i) \qquad \pi_r = \mathbf{ty2perm}(G, \alpha, \tau, x_r) \\ \kappa_{in} = \bigcup \mathbf{InputCaps}(\alpha_i, \tau_i, x_i) \cup (\{\textbf{s-tgt}^{\bullet}\}, \emptyset) \\ \kappa_{out} = \bigcup \mathbf{OutputCaps}(\alpha_i, \tau_i, x_i) \cup (\{\textbf{s-tgt}^{\bullet}\}, \emptyset) \\ P; E; G \vdash \kappa_{in} \Rightarrow \pi_{in} \qquad P; E; G \vdash \kappa_{out} \Rightarrow \pi_{out} \end{array}}{\begin{array}{c} P; E; G \vdash (\alpha_1\ \tau_1, \ldots, \alpha_n\ \tau_n)\ \longrightarrow\ \alpha\ \tau \Rightarrow \\ \forall x_1, \ldots, x_n \cdot (\pi_E + \pi_1 + \ldots + \pi_n + \pi_{in}) \rightarrow \exists x_r \cdot (\pi_E + \pi_r + \pi_{out}) \end{array}}$$

Figure 6.9: Converting Method Type to Procedure Type

6.2 Soundness of the Transformation

The proof process for the conservative type system is similar to the proof we demonstrated for the nonnull type system in Chap. 4. To prove the final soundness theorem, the following lemmas need to be proved first:

lemma1 For any expression e in the kernel language, if e can be type checked under the conservative type system under consistent program P, environment E, with input capabilities κ_{in} and output capabilities κ_{out}, and has result type τ and source β. Then, with ω converted from P, input permissions π_{in} converted from E and κ_{in}, and output permissions π_{out} converted from E, τ, β and κ_{out}, e can also be type checked under fractional permission system.

For this lemma, it can be proved by case analysis on each of the expressions in the kernel language. Most of the cases are straightforward, except for LetRestore and LetSwap. For the former, in order to restore the uniqueness of field f, we need to preserve the field permission $(o.f \rightarrow o')$, and use it for the assignment.

Unlike Read, which produces (roughly) the following permission:

$$\exists o, r \cdot (o = r + r = 0 \,?\, \emptyset : (r.\text{All} \rightarrow 0 + p(r)) +$$

$$r = 0 \,?\, \emptyset : (r.\text{All} \rightarrow 0 + p(r)) \multimap$$

$$\exists r_f \cdot (x'.f \rightarrow r_f + r_f = 0 \,?\, \emptyset : (r_f.\text{All} \rightarrow 0 + p(r_f)))$$

When converting to fractional permissions, the result permissions from checking the expression $x'.f$ are (roughly):

$$\exists r \cdot (x'.f \rightarrow r + (r = 0 \,?\, \emptyset : (r.\text{All} \rightarrow 0 + p(r)))$$

Here, the field permission $x'.f \rightarrow r$ in the above is passed along when checking the let body e, and then is used to check the expression $x'.f = e$. The conditional permission in the above is converted to the unique permission needed for the let-bound variable x. After the "let" expression is checked, any remaining permission for x is discarded.

The approach for handling LETSWAP rule is similar. The conditional permission is also passed along when checking the expression e, but remains in the final permissions for the **f-tgt**.

lemma2 For any method declaration in the kernel language, if it is well-typed under consistent environment P in the conservative type system, then with the converted program type ω, the converted procedure type for the method can also be type checked under the fractional permission system.

To prove this lemma, we first obtain a procedure type using the rule METHTYPE. Then, we show that the input permissions for the procedure type can be transformed to the input permissions converted from input environment and capabilities. With the help of **lemma1**, the method body can be type checked under fractional permission system. Finally, we show that the output permissions for checking the method body can be transformed to the output permissions of the procedure type.

With the above, the soundness theorem for the conservative type system is as follow:

soundness For every program g in the kernel language, if g is well-typed under consistent environment P, then with the converted program type ω, g can also be type checked under the fractional permission system.

To prove this theorem, we can use the lemma above and show that for all method declarations in the program, they can be converted to procedure type and is well-typed under

fractional permission system.

Chapter 7

Discussion

In this chapter, we discuss some of the difficulties we encountered during designing the conservative type system and proving its correctness. We also describe some of the limitations of the current system, and possible future work that can be done.

7.1 Proofs In Twelf

The conservative type system and the soundness theorem described in the last two chapters are proved sound using mechanized proofs realized in the Twelf programming language [PS99, PS02]. In this section we discuss some of the advantages and disadvantages by proving in this approach.

The advantages of using a proof system such as Twelf are:

- *Much more confidence on the correctness of the proof.* Mechanized proof offers much greater reliability on its correctness. Compared to a hand-written proof, which is checked by human, a mechanized proof is checked by computer programs. Computers are much less error-prone than manual checkings.

- *Easier to detect errors.* Since a proof system checks all the possible cases in a proof, this gives the proof writer better chances to discover errors in the proof. In my personal experience, during the writing process for this thesis, I often got stuck when proving some theorems that I initially thought were trivial, only to find out later that there were some details that I had overlooked. As result, changes were required for the theorem or related judgements. In this aspect, having the assistance of a proof system is very helpful.

- *Easier to maintain.* It is often easier to update and maintain mechanized proofs. For instance, during the proof process for this thesis, I frequently found that some definitions needed to be changed, while there were already many theorems related to them. In this case, a proof system will detect which parts of the proof need to be updated and report all the places for me. I consider this as a big advantage compared to maintaining a hand-written proof.

Not all aspects of using a proof system like Twelf are pleasant. In my opinion, there are also some disadvantages, listed in the following:

- *Much more tedious to write.* Everything comes with a price. What comes with the rigorous nature of mechanized proof is the fact that it is *much harder* to write. The mechanized proof for the fractional permission type system took Boyland about 500 hours[BS11], and for me, much more for the conservative type system. The tediousness, in my opinion, mainly comes from the fact that the proof system requires precise definitions for all the judgements and theorems about the programming language. One cannot ignore any of the cases, even though some cases may seem trivial and often are ignored in a hand-written proof. In general, one needs to spend much more

time writting a mechanized proof than a hand-written proof.

- *Hard to debug.* Twelf does not have very good support for debugging. This is further aggravated by the fact that it tries to unify variables when there are no explicit annotations to differentiate them. This often often results to coverage errors, which means there are cases that are not covered by the proof. In this situation, Twelf will print out all the cases that are missing. For theorems with a few cases this is not a big issue, but for theorems that have thousands or even more cases, which is very common for this thesis, this can be a huge headache, since Twelf will try to first collect all the missing cases and then output them to the console all at once. The "collecting" process is quite time consuming, and when the result is not ready, the console will appear to be frozen and often have to be interrupted manually to avoid waiting. The debugging information for the missing cases is also not so useful. As a result, I often just explicitly add annotations for all the variables in the proof. This makes the proof more tedious to write, and harder to read.

- *Lack good library and module system support.* Twelf does not come with good library support by default, and lacks module system. Boyland has written some library support [Boy] for basic data types such as natural numbers, sets, maps, etc., and a primitive module system using the C++ preprocessor. However, this does not solve all the issues. First, when using a particular data type defined in the library, one needs to search through all the properties about the data type and look for the one that fits the need. Since there are a large number of properties (theorems) for each data type, this searching process is time consuming. In many cases, the library does not contain the exact property one needs, and therefore new theorems need to be defined

and proved. Second, the library only has limited coverage. For instance, in Twelf the **extend-fld** function in Fig. 5.5 is implemented by a many-to-one mapping from a set of natural numbers into another set of natural numbers. This is not present in the library. Similarly, in Chap. 6, often a set of field or object capabilities or needs to be mapped to corresponding permissions, and many properties about this conversion need also be proved in Twelf.

- *Transformation in fractional permissions* One important part of the proof is to convert fractional permissions from one format into another, and this is done through permission transformations. When there are many permissions, it can become rather complex, because one needs to keep track of the positions of all the permissions, and apply different rules to split/combine them or move them around. Permission transformations consist of a large portion of the Twelf proof. They are also hard to maintain. A small change in the input permissions often involves changes in many places in the sequence of transformations.

7.2 Future Work

This section discusses some of the possible directions for future work. In the following, Sec. 7.2.1 discusses several possible enhancements for the conservative type system; Sec. 7.2.2 discusses the implementation for the conservative type system as an annotation checker; Sec. 7.2.3 discusses how a "liberal" annotation checker, alongside the conservative annotation checker, can help to further improve the annotation checking process.

7.2.1 Improvements on the Type System

In this section, we first discuss some of the limitations for the current type system. In particular, a few examples that appear to be obviously correct are unable to type check under the system. We then discuss some possible future work that can be done to enhance the type system and lift these restrictions.

Restore Consumed Capabilities

Although the rule LetRestore offers a way to restore consumed capabilities, it is limited in a "let" expression, and often a source program needs to be rewritten in order to be type checked. Another more general, albeit more complex approach is to introduce *compromised field capability* into the type system:

$$\kappa_f \in \textbf{FieldCapabilities} \quad ::= \quad \cdot \mid \kappa_f, x.f^{\bullet} \mid \kappa_f, \textbf{s-tgt}^{\bullet} \mid \kappa_f, x.f^{\circ}$$

A compromised field capability is of the form: $x.f^{\circ}$. It is used to track those capabilities on fields that have been consumed, but can be restored later via assignments. Also, a normal field capability $x.f^{\bullet}$ is stronger than a compromised field capability $x.f^{\circ}$: all places that a $x.f^{\circ}$ is used, one can substitute it with a $x.f^{\bullet}$.

With the compromised field capability, the relevant rules that need to be changed are shown in Fig. 7.1.

In the rule SubUniqueNew, different from the original rule SubUnique, the pinned field capabilities for targets ψ are added back to the capabilities as compromised capabilities. Note that the capability **s-tgt**$^{\bullet}$ cannot be restored because the current type system tracks these using the **s-tgt**, instead of the actual unique fields that they came from. To allow this would require a bigger change on the system.

SubUnique

$$\frac{\varepsilon_1\ c_1 <: \tau_2 \qquad \alpha \neq \texttt{borrowed}(\overline{f}) \qquad \kappa' = (\mathbf{extend\text{-}flds}(\psi, \mathbf{fields}(c_1))^\bullet, \mathbf{obj\text{-}tgts}(\psi)) \qquad \kappa' \subseteq \kappa \qquad \psi' = \mathbf{non\text{-}obj\text{-}tgts}(\psi) \setminus \{\mathbf{s\text{-}tgt}\}}{\kappa \vdash (\varepsilon_1\ c_1, \psi) <: \alpha\ \tau_2 \dashv \kappa \setminus \kappa' \cup (\psi'^\circ, \emptyset)}$$

WriteNew

$$\frac{P; E; \kappa_0 \vdash e_1 : (\texttt{notnull}\ c, \beta_1) \dashv \kappa_1 \qquad \mathbf{ftype}(P, c, f) = \alpha\ \tau \qquad \mathbf{write\text{-}field}(\beta_1, \kappa_1, f) = \psi, \kappa_2 \qquad P; E; \kappa_2 \vdash e_2 : (\tau_2, \beta_2) \dashv \kappa_3 \qquad \kappa_3 \vdash (\tau_2, \beta_2) <: \alpha\ \tau \dashv \kappa_4 \qquad \mathbf{consider\text{-}ftype}(\psi, \kappa_4, \alpha) = \beta_3, \kappa_5}{P; E; \kappa_0 \vdash e_1.f = e_2 : (\tau, \beta_3) \dashv \kappa_5}$$

Figure 7.1: New Rules for Write (Partial)

$$\mathbf{write\text{-}field}(\beta,\ (\kappa_f, \kappa_o),\ f) = \begin{cases} (\{\mathbf{s\text{-}tgt}\}, (\kappa_f \setminus \{\mathbf{s\text{-}tgt}^\bullet\}, \kappa_o)) & \beta \equiv \mathbf{shared} \\ (\psi_2 \cup \mathbf{non\text{-}obj\text{-}tgts}(\psi), (\kappa_f \setminus (\psi_3^\bullet \cup \psi_4^\circ), \kappa_o)) & \beta \equiv \psi \\ & \psi_2 \equiv \mathbf{extend\text{-}fld}(\psi, f) \wedge \\ & \psi_2 \equiv \psi_3 \cup \psi_4 \wedge \\ & \psi_3 \cap \psi_4 = \emptyset \wedge \\ & \psi_3^\bullet \subset \kappa_f \ \wedge \psi_4^\circ \subset \kappa_f \end{cases}$$

Figure 7.2: New Auxiliary Rules (Partial)

A consumed whole object cannot be restored as well. Such objects must be unique method parameters. The unique permission on the *all* field for those parameters have been lost, and therefore access to any field in the objects are no longer available.

In WriteNew, the only change is that **read-field** is now replaced with **write-field**, defined in Fig. 7.2, which also restores the compromised capabilities. Either normal or compromised field capabilities can be used to write a field. Both capabilities are removed from input capabilities and pinned. They can be restored once the expression e_1 in rule WriteNew is no longer needed.

A compromised capability $x.f$ is converted to the following (assuming x is not null) permission:

$$\exists r \cdot ((x \neq 0) + (x.f \rightarrow r) + (r = 0 \,?\, \emptyset : p(r)))$$

Notice that we don't have any permission on r, only class predicate.

Further changes need to be made with regards to how output permissions are converted. Consider the following example:

```
x.u = y;
b.f = z;
```

Assume the variable `y` has targets: $\{\texttt{a}, \texttt{b.f}, \texttt{c.g}\}$, the output permissions after checking `y` are of the following format:

$$\exists o, r \cdot (o = a \vee o = r +$$

$$r = 0 \,?\, \emptyset : r.\text{All} \rightarrow 0 + p(r) +$$

$$(r = 0 \,?\, \emptyset : r.\text{All} \rightarrow 0 + p(r)) \multimap$$

$$((\exists r \cdot (b.f \rightarrow r + r = 0 \,?\, \emptyset : r.\text{All} \rightarrow 0 + p(r))) +$$

$$(\exists r \cdot (c.g \rightarrow r + r = 0 \,?\, \emptyset : r.\text{All} \rightarrow 0 + p(r)))))$$

Here, the permissions corresponding to compromised field capabilities $\texttt{a.f}^\circ$ and $\texttt{b.g}^\circ$ are inside the encumbered permissions, and therefore would not be available for checking the expression `b.f = z`, which needs the permission:

$$\exists r \cdot ((b.f \rightarrow r) + (r = 0 \,?\, \emptyset : p(r)))$$

for accessing `b.f`. Therefore, the above permissions need to be transformed to:

$$\exists o, r \cdot (o = a \,?\, \emptyset :$$

$$(o = r \,?\, (r = 0 \,?\, \emptyset : r.\text{All} \rightarrow 0 + p(r)) +$$

$$r = r_a \,?\, ((b.f \rightarrow r_a + p(r_a)) + \exists r \cdot ((c.g \rightarrow r) + (r = 0 \,?\, \emptyset : p(r))))$$

$$: (r = r_b \,?\, ((c.g \rightarrow r_b + p(r_b)) + \exists r \cdot ((b.f \rightarrow r) + (r = 0 \,?\, \emptyset : p(r)))) : \top) : \top))$$

In general, the result permissions corresponding to a pinned field capability $x.f^\bullet$ cannot be encumbered permissions as we defined in Fig. 6.6 of Chap. 6. Instead, we need to convert the capability into a field permission (assuming r is possibly-null) fo the format:

$$x.f \rightarrow r + p(r)$$

Where r is the variable representing the result value. The permission also needs to be in a conditional permission similar to that in the example above. In this way, if the targets are consumed for the expression, the field permission can be transformed into existential permission in the format:

$$\exists r \cdot (x.f \rightarrow r + p(r))$$

corresponding to the compromised capability $x.f^\circ$, and the conditional permission can then be removed since all the branches will have the same permissions. Or, in case the capability is unpinned, the original permissions for $x.f^\bullet$ can be restored in a similar way.

Better LetUnqiue Rule

Another restriction of the current type system is the LetUnique rule. To summarize, there are three restrictions:

1. All the unique targets associated with the "let" binding expression are treated as consumed and taken away from the input capabilities. They are no longer available when checking the "let" body.

2. For the let-bound variable, none of its fields can be consumed inside the "let" body.

3. The let-bound variable cannot appear anywhere in the last expression of the "let" body.

For instance, the following example:

```
let x = new C() in a.f = x
```

cannot be type checked, because the unique value `x` is consumed in the "let" body.

It is possible to lift the second and third restriction by performing substitution on both result capabilities and targets, after the "let" body is checked. Lifting the restrictions can potentially be done in the following rewrite of LetUnique: The difference in the rule

LetUniqueNew

$$\frac{\begin{array}{c} P; E; \kappa_0 \vdash e_0 : (\varepsilon_0\ c_0, \psi) \dashv \kappa_1 \\ \kappa_2 = (\textbf{extend-flds}(\psi, \textbf{fields}(c_0))^\bullet, \textbf{obj-tgts}(\psi)) \qquad \kappa_2 \subseteq \kappa_1 \\ \kappa_3 = (\{\ x.f^\bullet \mid f \in \textbf{fields}(c_0)\ \}, \{x\}) \\ x \notin E \qquad P; E, x : \texttt{unique}\ \varepsilon_0\ c_0; \kappa_1 \setminus \kappa_2 \cup \kappa_3 \vdash e_1 : (\tau_1, \beta) \dashv (\kappa_{f_4}, \kappa_{o_4}) \\ \text{if } x \in \kappa_{o_4} \text{ then } \kappa_o = \textbf{obj-tgts}(\psi) \text{ else } \kappa_o = \emptyset \\ s = \{\ f \mid x.f^\bullet \in \kappa_{f_4}\ \} \qquad \text{if } s = \textbf{fields}(c_0)\ \psi_4 = \textbf{non-obj-tgts}(\psi) \text{ else } \psi_4 = \emptyset \\ \psi_5 = \{\ x'.f \mid x.f \in \kappa_{f_4}, x' \in \textbf{obj-tgts}(\psi)\ \} \qquad \beta' = \textbf{sub-source}(\psi, \beta) \end{array}}{P; E; \kappa_0 \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : (\tau_1, \beta') \dashv (\kappa_{f_5} \cup \psi_4^\bullet \cup \psi_5^\bullet, \kappa_{o_5} \cup \kappa_o) \setminus \kappa_3}$$

Figure 7.3: New Rule for let

LetUniqueNew is that after the "let" body is checked, it substitutes all field capabilities and targets that x occurs with the original ones they come from.

The rule first checks whether let-bound x is in the result object capabilities after checking

the body expression. If so, it means the object represented by x is not consumed, so that all the object capabilities that x comes from ($\mathbf{obj\text{-}tgts}(\psi)$) can then be restored.

For field capabilities, there are two cases to consider: the pinned field capabilities for the let binding expression e_0 in the rule, and the field capabilities obtained by extending object targets of e_0 (field capabilities in κ_2). For the former, they can only be restored if *none of the capability for x has been consumed.* In terms of permissions, these correspond to the encumbered permissions (π_2 in Fig. 6.6), and they can only be restored if the permission for the result value is intacted. For the latter, the original field capabilities are obtained by replacing x with each of the object targets from e_0.

The substitution for targets is done through the auxiliary function **sub-source** in Fig. 7.4. If the result source is shared, nothing needs to be done. Otherwise, for each target in the result target set, there are three cases: if the target is x, then it is simply replaced by the original targets for e_0; if it is $x.f$ for some field f, then the original targets are obtained by extending the object targets that x comes from with f, unioned with all non-object targets of x; otherwise, x does not occur in the target and therefore no substitution needs to be done. Here, the same target could be added multiple times, but since the result is a set, the operation is idempotent.

$$\mathbf{sub\text{-}source}(\psi,\, \beta) = \begin{cases} \bigcup_{\rho \in \psi_1} \mathbf{sub\text{-}target}(\rho, \psi) & \beta \equiv \psi_1 \\ \beta & \beta \equiv \mathbf{shared} \end{cases}$$

$$\mathbf{sub\text{-}target}(\rho,\, \psi) = \begin{cases} \psi & \rho \equiv x \\ \mathbf{extend\text{-}fld}(\psi, f) \cup \psi \setminus \mathbf{obj\text{-}tgts}(\psi) & \rho \equiv x.f \\ \{\rho\} & \text{otherwise} \end{cases}$$

Figure 7.4: Substitution for Targets

Note that the new rule does not address the first restriction, and still removes capabilities for all object targets in the result of e_0. Therefore, expression such as:

```
let x =
  if (..) a else b.f
in (a.g = null; x)
```

cannot be checked, because capability for `a.g` is not available when checking the let body. The type system does not track the relation between the variable x and those targets that it comes from. If we allow the preceding example, then in

```
let x =
  if (..) a else b.f
in (c.f = a.f; c.f = x.f; x)
```

`a.f` would be consumed twice illegally.

Fine-grained Capabilities

The current conservative type system does not differentiate between read and write effects. A possible future work is to define two types of capabilities: a read capability and a write capability. For reading a field, only the former is required.

With this addition, a method can be declared as the following:

```
class Demo {
  unique nonnull A f;

  ...

  foo(borrowed(read f) nonnull A a1, borrowed(
     read f) nonnull A a2) {
```

```
        ...
    }
}
```

To differentiate between a read and write capability, notation `borrowed(read f)` is used instead of `borrowed(f)`. This indicates that only read capability for field `f` is needed inside the method.

A read or write capability can be split into multiple read capabilities. Therefore, the following is allowed:

```
foo(this.f, this.f)
```

as long as we hold either read or write capability on `this.f`.

Obviously, a write capability cannot be split into multiple write capabilities.

With the introduction of read capability, the method type for `foo` can be converted to the following procedure type:

$$\forall z_1, z_2, r_{a_1}, r_{a_2} \cdot (z_1 \exists r \cdot (r_{a_1}.f \rightarrow r \ + r \neq 0 + p(r)) + z_2 \exists r \cdot (r_{a_2}.f \rightarrow r + r \neq 0 + p(r))) \rightarrow \exists \cdot (z_1 \exists r \cdot (r_{a_1}.f \rightarrow r + r \neq 0 + p(r)) + z_2 \exists r \cdot (r_{a_2}.f \rightarrow r + r \neq 0 + p(r)))$$

Note that for different read capabilities, we need to bind different fraction variables. Also, to convert a read capability to fractional permissions, we need to know how to "split" an existing fraction. In expressions such as method calls, a read or write capability can be splitted into multiple copies, and we need to calculate how many copies to generate. A simple solution is to collect all the usages of the capability and divide the fraction variable by that number.

"From" return type

Currently for a method return type only `shared` or `unique` are supported. It would be interesting to think about supporting the "from" return type [BRZ07], so that usages such as iterators can also be benefited from aliasing control using annotations.

7.2.2 A Conservative Annotation Checker

An important follow-up work is to implement the conservative type system as an annotation checker, perhaps on the Fluid framework [GHS03], and integrate it with the existing checker based on fractional permissions. The implementation should be built directly on the type system, and does not involve any operations on the fractional permission level. This potentially could make annotation checking much faster.

The implementation should be followed by evaluations with large real world programs, to test how the multi-tiered approach will improve the overall efficiency of annotation checking. Based on the feedback, perhaps more improvements could be done on the conservative type system and the annotation checker, to increase its input program coverage, reduce the space or runtime overhead, etc.

7.2.3 A Liberal Annotation Checker

The conservative type system and the annotation checker based on it only makes checking *correct* program faster. For those incorrect programs, they would still be passed to the heaveweight permission checker once they failed the conservative checker, which could make the issue even worse. As mentioned in Chap. 1, one possible solution is to design a *liberal* annotation checker that is also lightweight, which detects those "obviously wrong" cases. For

instance, it would be clearly wrong to store a borrowed method parameter into a unique field, or use a unique variable more than one time as a unique method parameter. For these, the liberal checker should be able to reject them immediately.

Chapter 8

Conclusion

Annotations are a useful tool for specifying high-level design intent for computer programs. Unlike a built-in type system, an annotation system can function as an pluggable type system, and can work separately from the main compiler and runtime system. This makes them very flexible, and a great way for enhancing an existing programming language without modifying the language itself.

Researchers have proposed many different annotations for tracking changes on mutable states, such as uniqueness, object ownership, immutability, method effects, nullness, etc. Even though they are very useful, when putting them together under the same context, their interactions with each other are hard to reason about. Moreover, these annotations are often expressed in a high-level concepts, and their precise semantic meanings are often difficult to pin down. The fractional permission system by Boyland and Retert [Boy03, BR05] offers a powerful tool for specifying semantics for the various annotations mentioned above. However, the trade-off for its expressiveness is its complexity. The current implementation [Ret09] only handles a portion of the features for the system, yet it is already very complex and has both

high space and runtime overhead. Therefore, it is not practical for real world usage.

This thesis proposes that, instead of using a monolithic type system to check every input program, we can adopt a multi-tiered approach. On the top we can use a more lightweight type system that sacrifices expressiveness for efficiency. The lightweight system can only handle part of the inputs, and for those it cannot handle, they shall be passed to the more powerful system in the next tier. The expectation for this approach is that most of the input shall be handled by the lightweight system very quickly, while only a small number of input need to be checked by the more heavyweight system. In general, the overall efficiency should be greatly improved.

Based on this idea, this thesis also presents a conservative type system built on top of fractional permissions. The type system uses high-level types, whose semantics can be translated to that of the fractional permission system. It is more lightweight, and therefore type checking is much faster.

The type system is also accompanied with a machine-checked proof for its soundness, which, unlike a natural language proof, provides greater assurance about its correctness. The proof is written in a novel approach: since the soundness for the fractional permission type system is also proved through mechanized proof, the soundness of the conservative type system is proved indirectly by piggy-packing onto that of the former. By doing this, there is no need to define the dynamic semantics of the target language, and also no need to prove progress and preservation directly.

Some possible future work for this thesis includes:

- Enhancements for the conservative type system by making it more expressive, while still preserving the efficiency. Some possible directions include:

  - Better support for restoring consumed capabilities.
  - Better handling for let expression when the let variable is unique.
  - Fine-grained capabilities; differentiate between read and write capabilities.
  - Supporting a "from" return type.

- Implement the type system as an annotation checker, perhaps on top of the Fluid IR. Since the implementation is only concerned with high-level types, and does not involve operations on the permissions, it should run much faster.

- Implement a liberal type checker that works

Bibliography

[AC04] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In Martin Odersky, editor, *ECOOP'04 — Object-Oriented Programming, 18th European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25, Berlin, Heidelberg, New York, 2004. Springer.

[BA07] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. *SIGPLAN Not.*, 42(10):301–320, October 2007.

[BCO06] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 115–137, Berlin, Heidelberg, 2006. Springer-Verlag.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

[BML97] Joseph A. Bank, Andrew C. Myers, and Barbara Liskov. Parameterized types for java. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 132–145, New York, NY, USA, 1997. ACM.

[BNR01] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalization of uniqueness and read-only. In Jørgen Lindskov Knudsen, editor, *ECOOP'01 — Object-Oriented Programming, 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Heidelberg, New York, 2001. Springer.

[Boy] John Boyland. Boyland's twelf library. `https://github.com/boyland/twelf-library`.

[Boy01a] John Boyland. Alias burying: Unique variables without destructive reads. *Software Practice and Experience*, 31(6):533–553, May 2001.

[Boy01b] John Boyland. The interdependence of effects and uniqueness. Paper from Workshop on Formal Techniques for Java Programs, 2001, June 2001.

[Boy03] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

[Boy09] John Boyland. An operational semantics including volatile for safe concurrency. *Journal of Object Technology*, 8(4):33–53, June 2009.

[Boy10a] John Boyland. Generating bijections between HOAS and the natural numbers. In *LFMTP: Logical Frameworks and Meta-languages: Theory and Practice*, July 2010.

[Boy10b] John Boyland. Semantics of fractional permissions with nesting. *ACM Transactions on Programming Languages and Systems*, 32(6):Article 22, August 2010.

[BR05] John Boyland and William Retert. Connecting effects and uniqueness with adoption. In *Conference Record of POPL 2005: the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 283–295, New York, 2005. ACM Press.

[Bra04] Gilad Bracha. Pluggable type systems. In *In OOPSLAâĂŹ04 Workshop on Revival of Dynamic Languages*, 2004.

[BRZ07] John Boyland, William Retert, and Yang Zhao. Iterators can be independent "from" their collections. In *IWACO '07: International Workshop on Aliasing Confinement and Ownership*, July 2007.

[BRZ09] John Boyland, William Retert, and Yang Zhao. Comprehending annotations on object-oriented programs using fractional permissions. In Matthew Parkinson, editor, *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, New York, 2009. ACM Press.

[BS11] John Boyland and Chao (Chris) Sun. Proving the correctness of fractional permissions for a javalike kernel language. In *Informal Proceedings of 18th International Workshop on Foundations of Object-Oriented Languages*, FOOL'11, November 2011.

[BSBR03] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. 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*, volume 38, pages 324–337, New York, May 2003. ACM Press.

[BV99] Boris Bokowski and Jan Vitek. Confined types. In *OOPSLA'99 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, number 10, pages 82–96, New York, October 1999. ACM Press.

[CD02] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect, November 2002.

[Chl10] Adam Chlipala. A verified compiler for an impure functional language. *SIGPLAN Not.*, 45(1):93–106, January 2010.

[Cla01] David Clarke. *Object Ownership and Containment.* PhD thesis, University of New South Wales, Sydney, Australia, 2001.

[CNW10] Nicholas Cameron, James Noble, and Tobias Wrigstad. Tribal ownership. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 618–633, New York, NY, USA, 2010. ACM.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, number 10, pages 48–64, New York, October 1998. ACM Press.

[CW03] David Clarke and Tobias Wrigstad. External uniqueness is unique enough. In Luca Cardelli, editor, *ECOOP'03 — Object-Oriented Programming, 17th European Conference*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200, Berlin, Heidelberg, New York, 2003. Springer.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of the Twenty-sixth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 262–275, New York, 1999. ACM Press.

[DF01] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, volume 36, pages 59–69, New York, May 2001. ACM Press.

[DM05] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, October 2005.

[DPJ08] Dino Distefano and Matthew J. Parkinson J. jstar: Towards practical verification for java. *SIGPLAN Not.*, 43(10):213–226, October 2008.

[Eva96] David Evans. Static detection of dynamic memory errors. *SIGPLAN Not.*, 31(5):44–53, May 1996.

[FD02] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, volume 37, pages 13–24, New York, May 2002. ACM Press.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 192–203, New York, NY, USA, 1999. ACM.

[FL03] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA'03 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, number 11, pages 302–312, New York, November 2003. ACM Press.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[GB99] Aaron Greenhouse and John Boyland. An object-oriented effects system. In Rachid Guerraoui, editor, *ECOOP'99 — Object-Oriented Programming, 13th European Conference*, volume 1628 of *Lecture Notes in Computer Science*, pages 205–229, Berlin, Heidelberg, New York, 1999. Springer.

[GHS03] Aaron Greenhouse, T. J. Halloran, and William L. Scherlis. Using Eclipse to demonstrate positive static assurance of Java program concurrency design intent. In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 99–103, October 2003.

[Gir87] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, January 1987.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[HLL+12] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, June 2012.

[HO10] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 354–378, Berlin, Heidelberg, 2010. Springer-Verlag.

[Hog91] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 271–285, New York, NY, USA, 1991. ACM.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. 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*, pages 14–26, New York, 2001. ACM Press.

[KT01] Günter Kniesel and Dirk Theisen. Jac—access right based encapsulation for java. *Softw. Pract. Exper.*, 31(6):555–576, May 2001.

[Lei98] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, number 10, pages 144–153, New York, October 1998. ACM Press.

[LLP+00] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. Jml: notations and tools supporting detailed design in java. In *IN OOPSLA 2000 COMPANION*, pages 105–106. ACM, 2000.

[LP06] Yi Lu and John Potter. Protecting representation with effect encapsulation. *SIGPLAN Not.*, 41:359–371, January 2006.

[LPHZ02] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, volume 37, pages 246–257, New York, May 2002. ACM Press.

[Luc87] John M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming.* PhD thesis, MIT, Cambridge, Massachussetts, USA, September 1987.

[Min96] Naftaly Minsky. Towards alias-free pointers. In Pierre Cointe, editor, *ECOOP'96 — Object-Oriented Programming, 10th European Conference*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209, Berlin, Heidelberg, New York, July 1996. Springer.

[MPH00] Peter Müller and Arnd Poetzsch-Heffter. A type system for controlling representation exposure in Java. In Sophia Drossopolou, Susan Eisenbach, Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *2nd ECOOP Workshop on Formal Techniques for Java Programs*, 2000.

[MR07] Peter Müller and Arsenii Rudich. Ownership transfer in universe types. *SIGPLAN Not.*, 42(10):461–478, October 2007.

[NBAB12] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. *SIGPLAN Not.*, 47(1):557–570, January 2012.

[OYR04] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. *SIGPLAN Not.*, 39(1):268–280, January 2004.

[PB05] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. *SIGPLAN Not.*, 40(1):247–258, January 2005.

[PB08] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 75–86, New York, NY, USA, 2008. ACM.

[PB13] Matthew Parkinson and Gavin Bierman. Aliasing in object-oriented programming. chapter Separation Logic for Object-oriented Programming, pages 366–406. Springer-Verlag, Berlin, Heidelberg, 2013.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction: Automated Deduction*, CADE-16, pages 202–206, London, UK, UK, 1999. Springer-Verlag.

[PS02] Frank Pfenning and Carsten Schürmann. Twelf user's guide, version 1.4. Available at `http://www.cs.cm.edu/~twelf`, 2002.

[Ret09] William S. Retert. *Implementing Permission Analysis.* PhD thesis, University of Wisconsin–Milwaukee, Department of EE & CS, 2009.

[Rey02] John Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74, Los Alamitos, California, July 22–25 2002. IEEE Computer Society.

[SWM00] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In Gert Smolka, editor, *ESOP'00 — Programming Languages and Systems, 9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, Berlin, Heidelberg, New York, 2000. Springer.

[Wad90] Philip Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*. Elsevier, North-Holland, 1990.

[WCM00] David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.

[WM01] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Types in Compilation: Third International Workshop, TIC 2000*, volume 2071 of *Lecture Notes in Computer Science*, pages 177–206, Berlin, Heidelberg, New York, September 2001. Springer.

Chapter 9

Appendix
About the Mechanized Proof

We have mechanized all the proofs for the conservative type system in about 65K lines of Twelf code including about 1800 theorems. The current release of the Twelf proof on the conservative type system is available at
`https://github.com/sunchao/reftype/archive/release.zip`. It includes all the dependencies such as Boyland's proof for the fractional permission type system, and requires Twelf 1.5R3.

The syntax for the kernel language is defined in `simple-concur/syntax.elf`, and the type system is defined in `typing.elf`. Some of the main Twelf metatheorems are listed in below. They are defined in the file `conversion.thm`.

1. `reftyping-ok`. Corresponds to the **lemma1** in the Chap. 6.
2. `methmapmatch-implies-progtypematch`. Corresponds to the **lemma2** in the Chap. 6.
3. `env2progtype-total`. Corresponds to the **soundness** theorem in the Chap.6.

CURRICULUM VITAE

Title of Dissertation

A Conservative Type System Based On Fractional Permissions

Full Name

Chao Sun

Place and Date of Birth

Zhuzhou, Hunan, China April 14, 1983

Colleges and Universities,	Years attended and degrees
Shaanxi University of Science & Technology	2001-2005, B.S.
Major: Computer Science	
Nanjing University	2005-2008, M.E.
Major: Computer Science	
University of Wisconsin-Milwaukee	2008-2016, PhD.