inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis

Implementing Uniqueness and Ownership Transfer in the Universe Type System

Yoshimi Takano
ytakano@ethz.ch

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

September 2006 – March 2007

Advisors:

Arsenii Rudich
Prof. Dr. Peter Müller

Ownership is a powerful concept to structure the object store and to control aliasing and modifications of objects. Most extant ownership models do not support dynamic ownership transfers, which renders some object oriented-programming idioms inexpressible. Examples include the Abstract Factory design pattern and merging the representations of two linked lists. This report presents a solution to extend the Universe type system with ownership transfers and describes an implementation thereof.

Contents

1
Introduction

1.1 Setting the Scene

1.1.1 Ownership Type Systems

The concept of object ownership allows programmers to structure the object store hierarchically and to control aliasing and access between objects. Existing ownership models share fundamental concepts: Each object has at most one owner object. The set of all objects with the same owner is called a *context*. The *root context* is the set of objects with no owner. The ownership relation constitutes a tree order. The set of all object owned by an object (directly or indirectly) is called the object's *representation*.

1.1.2 Ownership Transfer

As part of a recent Master's Thesis [19], three ownership type systems (the Universe type system [17, 11], Ownership Types [4] and Ownership Domains [1]) have been evaluated for compatibility with object-oriented design patterns. One of the conclusions arrived at was that many design patterns require changing the owner of certain objects after their creation. This feature is missing in all of the reviewed ownership type systems. For instance, in the Abstract Factory pattern, it is desirable for the client to be the owner of the created product but not of the factory, since the factory, usually serving multiple clients, should be globally accessible from any ownership context. Other design patterns that may benefit from ownership transfers include the Composite pattern, the Decorator pattern and the Visitor pattern.

Another common application of ownership transfer is the combination of the representations of different objects. A prominent example, often attributed as a challenge for ownership type systems, is the concatenation of the encapsulated nodes of two linked lists.

In summary, it can be concluded that ownership transfer greatly enhances the expressiveness of ownership type systems.

1.1.3 Project Goals

The aim of this project is the implementation of a smooth ownership transfer model for the Universe type system. The solution should be intuitive and avoid additional annotation overheads, while still being expressive enough to handle the usual examples, such as the Abstract Factory design pattern and the merging of list representations.

1.1.4 Overview

After continuing this chapter with the necessary background on the Universe type system, Chapter 2 introduces the chosen ownership transfer model in an informal way. In Chapter 3, the type system is formalized for a simplistic toy language, making use of a static data flow analysis, which is then described in detail in Chapter 4. Chapter 5 is dedicated to the implementation of the proposed type system as part of the MultiJava compiler and the demonstration of some actual code examples. Finally, we present our conclusions in Chapter 6.

1.2 The Universe Type System

The *Universe type system* [17, 11] is an ownership type system that enforces the *owner-as-modifier* discipline: An object o may be referenced by any other object, but reference chains that do not pass through o's owner must not be used to modify o. This allows owner objects to control state changes of owned objects and thus maintain invariants. The owner-as-modifier discipline imposes weaker restrictions than the alternative *owner-as-dominator* discipline, which requires that *all* reference chains from an object in the root context to an object o in a different context go through o's owner. The owner-as-modifier discipline enables the Universe type system to handle common implementations where objects are shared between objects, such as collections with iterators.

Ownership Modifiers. A type in the Universe type system consist of an *ownership modifier* and a class name. The ownership modifier expresses object ownership relative to the current receiver object `this`. A program may contain the ownership modifiers `peer`, `rep` and `any`[1].

- `peer` expresses that an object has the same owner as `this`,
- `rep` expresses that an object is owned by `this` and
- `any` expresses that an object may have any owner.

The owner-as-modifier discipline is enforced by disallowing modifications of objects through `any` references. I.e., an expression of an `any` type may be used as receiver of field reads and calls to side-effect free (*pure*) methods, but not of field updates or calls to non-pure methods.To check this property, side-effect free methods are required to be annotated with the keyword `pure`. Due to the usage restrictions, `any` references can also be perceived as *read-only* references.

Type Combinator. The ownership modifier combinator outlined in Table 1.1 is used to determine the resulting ownership modifier of transitive accesses such as field accesses, method parameters/results and array element accesses. The special ownership modifier `this` acts as ownership modifier of the `this` reference. The `this` modifier cannot be accessed directly by a user, but it is used to simplify the formalization.

[1] In earlier descriptions and in currently implemented tools, `any` is called `readonly`

$\triangleright_U$	any	peer	rep
this	any	peer	rep
any	any	any	any
peer	any	peer	any
rep	any	rep	any

Table 1.1: Universe modifier combinator. The left-hand side argument is given in the left column, the right-hand side argument is given in the top row.

Subtype Relation. Since the peer and rep modifier are more specific than the any modifier, they are both defined to be a subtype of any. However, they are unrelated to each other. A type is a subtype of another type iff both the class components and the ownership modifier components are respective subtypes.

Object Creation. The owner of an object is to be specified at creation time by either a peer (the object will have the same owner as this) or a rep modifier (the object will be owned by this).

1.2.1 List Example

As an example, a linked list with an associated iterator in the Universe type system is typically typed according to the skeleton in Listing 1.1. Figure 1.1 depicts the corresponding ownership contexts.

```
class LinkedList {
    rep Node first, last;
}

class Node {
    any Object element;
    peer Node next, prev;
}

class Iterator {
    peer LinkedList list;
    readonly Node current;
}
```

Listing 1.1: Linked list with iterator in the Universe type system.

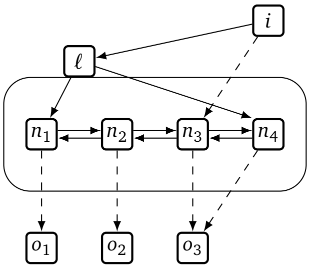


Figure 1.1: Linked list with iterator in the Universe type system. The nodes n_i are owned by the list object ℓ. The iterator object i is in the same context as the list object and maintains a read-only reference (dashed) into the list representation (solid rectangle).

2 Approach

This chapter begins with brief reviews of some extant concepts related to the notion of unique variables. Subsequently, an informal description of the approach taken to integrate ownership transfers into the Universe type system is presented. The techniques used are based on the idea of (external) uniqueness and the concept of alias burying and make use of a static, intra-procedural program analysis.

2.1 Uniqueness and its Variants

The following subsections present a short review of existing concepts related to alias management with unique variables and sets them in context with the list representation merging problem.

2.1.1 Unique Variables

The concept of *unique* variables (also known as *linear* variables) has been proposed by many researchers [22, 13, 3, 16] as an approach to managing aliasing in object-oriented programs. In its purest form, the uniqueness invariant requires that a unique variable is either `null` or else its value is the sole reference to an object, which is then referred to as an *unshared* object. The term "unique reference" is used alongside "unique variable" to emphasize the pointer value of a unique variable.

Note that in the Universe type system, `any` references do not need to be restricted, as they may not be used for modifications.

Clearly, an unshared object can be safely transfered into another ownership context, since we can be sure that there are no other references pointing to the object that might end up being ill-typed after the transfer. Consequently, an object to be transfered should be referenced by a unique variable.

In relation with the list merge example, however, plain unique variables would be of no big help, since the first node may not only be referenced by the list object, but also by its successor node (cf. Figure 1.1). This aliasing complicates the maintenance of the uniqueness invariant.

2.1.2 Working with Unique Variables

Once having described the uniqueness invariant, we naturally would like to maintain it when working with unique references.

Destructive Reads. One possible way to achieve this is to employ *destructive reads*: A unique variable is atomically nullified (set to `null`) whenever it is read. Consider Figure 2.1(a) for an example, where we suppose that `u` is a unique variable. Right after the value of `u` is obtained to perform the method call, its value becomes `null`.

It is immediate that this strategy is sufficient to prevent a unique variable from ever being aliased, since creating an alias necessarily involves reading the unique variable. Though being an easy concept both to understand and implement, destructive reads are unintuitive for a programmer and, even worse, the semantics of the programming language is changed. Moreover, destructive reads may cause the source code to get more complicated when it comes to restore nullified variables.

Alias Burying. Another approach for maintaining uniqueness is the notion of *alias burying* [5], which can do without destructive reads. The underlying idea is that aliases of a unique variable do not harm if they are not used anymore to access the referenced object at the moment the unique variable is read again.

Have a look at the code fragment in Figure 2.1(b) for an example, where u is supposed to be a unique variable. On line 2, an alias of the unique variable u is created, which in fact violates the uniqueness invariant. On line 4, the unique variable u is read again. The crucial observation is that if we are sure that the alias o is not used anymore to access the object again, then this has the same effect as though o did not exist at all. In consequence, if o is dead before line 4 in the example, then it might as well be "buried" to enforce that it will not be read again. This explains the term "alias burying", the intuition being that dead aliases can safely be buried.

Another way one can imagine the concept of alias burying is related to access rights: instead of having destructive reads, each alias of a unique variable is stripped off the right to access the unshared object. In other words, the alias is still there, but it cannot be used it anymore to access the unshared object (the variable is marked as *unusable*). The uniqueness invariant may be violated – but only in a safe way, in the sense that the invariant holds up whenever it is interesting, namely when a unique variable is read.

In summary, we need to make sure that all existing aliases of a unique variable are not used anymore to access the unshared object, or, equivalently, no buried variable (or no variable marked as unusable) must be read. To statically ensure this property, some extra annotations in method signatures and a data flow analysis are used. Among others, each method needs an annotation that indicates which variables it may read.

The benefit of the alias burying approach is the fact that there is no need for destructive reads. On the other hand, as alias burying relies on program analysis, the strength of this approach is sensitive to the precision and the efficiency of the underlying data flow analysis. Moreover, apart from being an overhead per se, the effect annotations on methods may introduce an abstraction problem in case of improper information hiding.

2.1.3 External Uniqueness

External uniqueness [7] loosens the conventional uniqueness constraint, requiring that there be only one reference to an *aggregate* of objects from the *outside* of the aggregate. Within the aggregate, arbitrary aliasing is allowed. In conjunction with ownership type

```
...
u.m();
// u is null here
...
```

(a) Desctructive reads.

```
...
o = u; // uniqueness invariant violated
...
u.m(); // ok if o is dead here (o is 'buried')
...
```

(b) Alias burying.

Figure 2.1: Destructive reads vs. alias burying.

systems, an ownership context can be regarded as defining an externally unique object aggregate. I.e., the following uniqueness invariant is formulated: An externally unique reference is the only (read-write) reference crossing an ownership context boundary from the outside to the inside. Note that this contrasts `rep` references in the Universe type system, where the owner might have more than one `rep` reference into its representation.

Applying the idea of external uniqueness to the list merging example, the list object would simply maintain an externally unique reference into its representation.

In summary, external uniqueness allows us to deal with object aggregates as opposed to only single objects. However, the solution proposed in the external uniqueness publication relies on ownership type systems that enforce the owner-as-dominator discipline. Moreover, it still resorts to destructive reads and additional annotations so as to maintain the (external) uniqueness invariant.

2.2 Universes Blending in

This section describes how the concepts of externally unique clusters of objects and alias burying can be applied in conjunction with the Universe type system to achieve a flexible

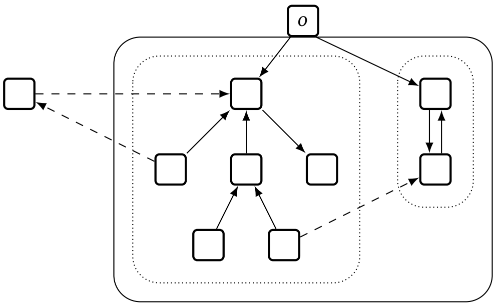


Figure 2.2: The representation of object o (solid rectangle) partitioned into two clusters (dotted rectangles). Dashed arrows represent read-only references, solid arrows denote read-write references.

ownership transfer model without the need for destructive reads and without the downsides of the original alias burying proposal. The presented approach is based on work by Arsenii Rudich and Peter Müller.

2.2.1 Controlled Clusters

As with external uniqueness, the basic idea is still to regard the representation of an object as an externally unique aggregate. However, to support the quite natural case where only a part of the object representation is meant to be transfered (think of, e.g., a factory that maintains encapsulated internal state objects), we partition an ownership context into so-called *clusters*.

Clusters act as the units of an ownership transfer: they are transfered always as a whole. There is one special cluster, the *this-cluster*, that may *not* be transfered. Thus, the this-cluster corresponds to the usual representation of the owner object, which renders the model is backward compatible to the original Universe type system in case there are no transferable clusters.

Clusters basically correspond to externally unique aggregates: there may be arbitrary aliasing within a cluster, but external references into a cluster should be controlled. Ultimately, we need to make sure that there are no remaining external references pointing into a cluster that may end up ill-typed when a cluster is transfered. Thus, in contrast to external uniqueness, multiple read-write references into a cluster are allowed, as long as it is guaranteed that at the moment of a transfer, there is indeed only a single reference pointing into the transfered cluster. In any case, `any` references are not restricted.

Cluster Declaration. There is a new ownership modifier, `uniq`, that may only be used in field declarations. A field declared `uniq` can be seen as a special kind of `rep` reference that additionally defines a transferable cluster. Moreover, the field points into the cluster it defines. In contrast, a field declared `rep` is interpreted as pointing into the this-cluster. Furthermore, the refined modifier syntax `rep[f]` is used to declare a field that points into the cluster defined by the field `f`. `f` must be declared `uniq` in the *enclosing* class (and not in a superclass). Simply put, this is motivated by the desire for modular checking, as a class needs to be aware of all fields that point into a certain cluster (this will become clearer in later sections).

Method Parameters and Return Values. The refined modifier syntax `rep[f]` can also be used in method signatures to denote references into a cluster defined by the field `f`, which, again, must be declared `uniq` in the enclosing class. As for fields, a plain `rep` modifier in a method signature is interpreted as a reference into the this-cluster. In anticipation, there will be an additional modifier, `free`, that can be used in method signatures. This is explained below.

Subtyping and Assignment Compatibility. Subtyping and assignment compatibility[1] among `rep` references that point into clusters is defined as follows: a reference pointing into a cluster C_f is a subtype of and assignable to a reference pointing into a cluster C_g iff $C_f = C_g$.

Cluster Inference for Local Variables. Local variables that point into the representation of `this` are to be declared solely with a plain `rep` modifier (as opposed to a `rep[f]` modifier). The actual cluster the variable may point into is statically inferred (by the data flow analysis described later). Consequently, a local `rep` variable can be seen as pointing into a "*wildcard*" cluster. Note that this contrasts a plain `rep` modifier in field declarations and method signatures, where it is interpreted as denoting a reference into the this-cluster. The cluster inference for local variables frees the user from the burden

[1] Refer to Section 3.6 for a more detailed explanation of the differences between the subtype relation and the assignable-to relation.

of unnecessary cluster details in conjunction with local variables. What is more, cluster inference for local variables leads to a higher expressiveness, as demonstrated by the fragment

```
if (...) x = f; else x = g;
```

where `x` is a local variable and `f` and `g` are fields pointing into distinct clusters.

2.2.2 Ownership Transfer

Capture and Release. An ownership transfer of a cluster happens in two steps: first, the cluster is *released* by the current owner, making it an *unowned*, a or *free* cluster. A free cluster may then be *captured* into an ownership context to complete the ownership transfer.

`free` Modifier. There is another new Universe modifier, `free`, which may solely be used in declarations of formal method parameters and return types. The `free` modifier denotes a reference to a released (or free) cluster, with the additional property that it is the only read-write reference to the free cluster whatsoever. Thus, a `free` reference is "really unique", as opposed to a reference into a transferable cluster, where there could be other references (from the stack of from the heap) pointing into it. This could be confusing since transferable clusters are declared with the keyword `uniq`.

The Universe modifier combinator for the `free` modifier is defined as follows:

$$x \rhd_U \texttt{free} := \texttt{free}$$
$$\texttt{free} \rhd_U x := \texttt{any},$$

where x stands for any Universe modifier and `free` $\rhd_U$ `free` is defined to yield `free`. Combining x with a `free` modifier can be seen as passing on an already released cluster. Hence, the resulting modifier is again `free`.

Implicit Transfers. To keep the notational overhead small, the capture and release operations will take place *implicitly.* It has yet to be decided as future work at which point exactly implicit capturings, releasing and ownership transfers happen. To give an example, an implicit ownership transfer (including both a releasing and a capturing) certainly happens in the following two cases:

- An assignment `x = y;` where `y` is of a `rep` type and `x` is of a `peer` type releases the cluster pointed into by `y` and captures it into the `peer` context of `this`.

- An assignment `x.f = y;` where `x` is of a `rep[g]` type, `f` is a `peer` field and `y` is of a `rep[g]` type (where `g` $\neq$ `h`) releases the cluster pointed into by `y` and captures it into the cluster pointed into by `x`, thereby merging the clusters.

2.2.3 Cluster Alias Controlling

As hinted at above, to safely transfer a cluster, we must be able to control external references that point into it, in particular at the moment the cluster is released.

Unusable Variables. In analogy to the concept of alias burying, we maintain a set of *unusable* variables that must not be read. The following set of rules describes the manipulation of the set of unusable variables:

- Whenever a (possible) release operation is inferred, all variables that point into the released cluster are marked as unusable. This makes sure that no aliases are used to access the released cluster again, corresponding to the state of a really externally unique cluster at the moment of the release operation.

- Before a non-pure `peer` method call, all local variable pointing into a non-free cluster are marked as unusable. The reason for this is to ensure that there be no references from the caller's stack frame into any cluster that might be released during the called method.

- Reading a `free` variable marks it as unusable.

- Assigning to a variable marks it as not unusable (anymore).

Note that `rep` method calls are of no concern, since, by virtue of the owner-as-modifier discipline, there cannot be any "upgoing" read-write references.

Section 3.9 formalizes the above rules, using a conservative approximation as to when a release operation takes place (i.e., there may be more variables marked as unusable than would actually be necessary).

Static Data Flow Analysis. As with alias burying, a static, intra-procedural data flow analysis is employed to check the alias constraints. In contrast to alias burying, there will be no need for extra effect annotations on methods. Simply put, the reason for this lies in the additional assurances guaranteed by owner-as-modifier property, e.g., as mentioned above, "upgoing" reads during a `rep` method call can only be made through `any` references. The data flow analysis performs the following checks:

- No unusable variables must be read.
- No fields must be unusable before a non-pure `peer` method call and upon method termination.

The second check ensures a valid field state in the sense that every field is usable in the pre and post state of a non-pure `peer` method invocation.

2.3 Example Ownership Transfer

Figure 2.3 outlines the single steps of an ownership transfer, resulting from an assignment of the `rep` variable `x` to a the field `g`. Note that without line 9, field `f` would be unusable upon the termination of method `m`, which would be illegal.

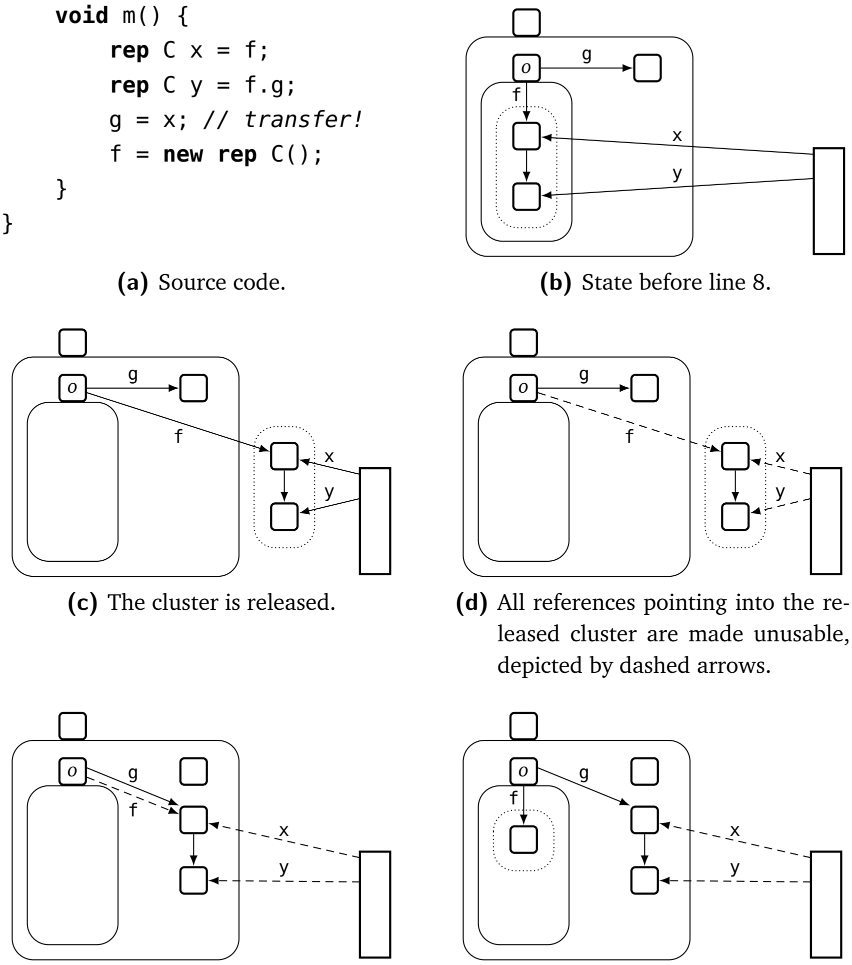
2.4 Summary

The notion of external uniqueness and the ideas from alias burying are integrated into the Universe type system. An object representation is partitioned into clusters, which act as the units of (implicit) ownership transfers. External references into clusters are controlled using a static, intra-procedural data flow analysis. There is no need for destructive reads or additional ownership transfer annotations (i.e., explicit release and capture statements). Moreover, on the strength of the owner-as-modifier discipline, no method effects need to be declared either.

The following itemization summarizes the possible usages of Universe modifiers that denote references pointing into the representation of `this` (as opposed to `peer` and `any` references):

- `uniq C f`: Defines a new transferable cluster associated with the field `f`. May only be used in field declarations.

```
 1  class C {
 2      uniq C f;
 3      peer C g;
 4
 5      void m() {
 6          rep C x = f;
 7          rep C y = f.g;
 8          g = x; // transfer!
 9          f = new rep C();
10      }
11  }
```



(a) Source code.

(b) State before line 8.

(c) The cluster is released.

(d) All references pointing into the released cluster are made unusable, depicted by dashed arrows.

(e) State after line 8. The cluster is captured into the peer context of *o*, thereby merged. This completes the ownership transfer.

(f) State after line 9, with field `f` re-assigned and not unusable anymore.

Figure 2.3: Example ownership transfer. Ownership contexts are depicted by solid rectangles, transferable clusters by dotted rectangles. The rectangle to the right symbolizes the stack. *o* is the receiver object.

- `rep[f]`: Denotes a reference pointing into the cluster defined by the field `f`. Field `f` must be declared `uniq` in the enclosing class. May only be used in field declarations and method signatures.
- `free`: Denotes a released (free, unowned) cluster. May only be used in method signatures.
- `rep`: The interpretation depends on context of usage. For fields and in method signatures, the modifier denotes a reference pointing into the this-cluster. For local variables, it denotes a reference pointing into a “wildcard” cluster, where the actual cluster will be inferred by the data flow analysis.

3 Formalization

This chapter formalizes the presented type system for a simplistic toy language, while introducing the static data flow analysis on a high level. The analysis in turn is fully covered in the subsequent chapter.

3.1 Toy Language Syntax

To give a formalization of the type system, we employ a simple, minimalistic toy language. We will describe the following syntactic categories:

P	$\in$ **TProg**	programs
CDecl	$\in$ **TCDecl**	class declarations
MDecl	$\in$ **TMDecl**	method declarations
W	$\in$ **TPure**	purity modifiers
S	$\in$ **TStmt**	statements
$T_f, T_p, T_r, T_l, T_n, T_c$	$\in$ **TType**	field types, formal parameter types, return value types, local variable types, object creation types, types in cast statements
$m_f, m_p, m_r, m_l, m_n, m_c$	$\in$ **TMod**	Universe modifiers for field types, formal parameter types, return value types, local variable types, object creation types and types in cast statements

Further, we make use of the following meta variables:

C, D	$\in$ **TClass**	class names (including `Object`)
m	$\in$ **TMethod**	method names
f	$\in$ **TField**	field names
x, y, z	$\in$ **TLoc**	local variable names (including `this` and `result`) and formal parameter names

We use $\overline{X}$ as shortcut for $X_1 \ \cdots \ X_k$, where X stands for *CDecl* or *MDecl*. Similarly, $\overline{X \ \ Y}$ is to be read as $X_1 \ \ Y_1$`;` $\ldots$`;` $X_k \ \ Y_k$`;` (note the semicolons), where $X \in$ **TType** and $Y \in$ **TLoc** or $Y \in$ **TField**. The syntax is given in Table 3.1

Let us further remark and comment some aspects:

- Except for the sequential composition, we only consider elementary statements. I.e., there are neither loops, nor branches, nor any other control flow constructs.

$$
\begin{array}{lcl}
P & ::= & \overline{\mathit{CDecl}} \\[1em]
\mathit{CDecl} & ::= & \texttt{class}\ C\ \texttt{extends}\ D\ \texttt{\{}\ \overline{T_f\ f}\ \overline{\mathit{MDecl}}\ \texttt{\}} \\[1em]
\mathit{MDecl} & ::= & W\ T_r\ m\texttt{(}T_p\ x\texttt{)}\ \texttt{\{}\ \overline{T_l\ y}\ S\ \texttt{\}} \\[1em]
W & ::= & \texttt{pure} \\
 & | & \texttt{nonpure} \\[1em]
S & ::= & x\ \texttt{=}\ y \\
 & | & x\ \texttt{=}\ \texttt{null} \\
 & | & x\ \texttt{=}\ \texttt{new}\ T_n \\
 & | & x\ \texttt{=}\ y.f \\
 & | & y.f\ \texttt{=}\ x \\
 & | & x\ \texttt{=}\ y.m\texttt{(}z\texttt{)} \\
 & | & x\ \texttt{=}\ \texttt{(}T_n\texttt{)}\ y \\
 & | & S_1\texttt{;}\ S_2 \\[1em]
T_i & ::= & m_i\ C \qquad \text{, for } i \in \{f, p, r, l, n, c\} \\[1em]
m_f & ::= & \texttt{any} \mid \texttt{peer} \mid \texttt{rep[}f\texttt{]} \mid \texttt{uniq} \\
m_p & ::= & \texttt{any} \mid \texttt{peer} \mid \texttt{rep[}f\texttt{]} \mid \texttt{free} \\
m_r & ::= & m_p \\
m_l & ::= & \texttt{any} \mid \texttt{peer} \mid \texttt{rep} \\
m_n & ::= & \texttt{peer} \mid \texttt{rep} \\
m_c & ::= & \texttt{any} \mid \texttt{peer} \mid \texttt{rep[}f\texttt{]} \mid \texttt{rep}
\end{array}
$$

Table 3.1: Syntax of the toy language.

- There are no nested expressions. Any program can be transformed to this form via introducing additional local variables which represent temporary values.
- We suppose that any method has exactly one parameter. It is not hard to generalize this case for an arbitrary number of parameters.
- We impose the rule that all fields be declared at the beginning of a class and, likewise, all local variables be declared at the beginning of a method. A program can be transformed to this form by collecting and rearranging the declarations.
- For each method call, the special variable `this` $\in$ **TLoc** denotes a reference to the current receiver object, as in Java. Note that there are no static methods in the toy language.
- Instead of using a `return` statement to return the result of a method invocation, we use a distinguished variable `result` $\in$ **TLoc** to which the return value is assigned. This also means that a method cannot terminate prematurely.
- We assume a special class `Object` $\in$ **TClass** marking the root of the class hierarchy.
- We suppose that all classes, fields and local variables have globally unique names. In particular, the sets **TClass**, **TMethod**, **TField** and **TLoc** are pairwise disjoint.
- If methods have equal names, then they have equal signatures. In other words, there is no method overloading.
- For simplicity, method overriding is ignored.
- A `new` statement is just concerned with the creation of a new object; no initializer method (constructor) is called.
- The syntax draws a distinction between Universe modifiers and types used in different places, namely:
  - field declarations (m_f and T_f),
  - formal method parameters return values (m_p and T_p, m_r and T_r, respectively),
  - local variable declarations (m_l and T_l),
  - `new` statements (m_n and T_n), and
  - cast statements (m_c and T_c).

This way, the allowed usages of Universe modifiers are already enforced by the syntax definition. As a presumably more concise alternative, the possible occurrences of Universe modifiers are summarized in Table 3.2. Note that the `uniq` and `free` modifiers can exclusively be used in field declarations and for method parameters and return values, respectively.

	`any`	`peer`	`rep`	`rep[f]`	`uniq`	`free`
Field declarations	✓	✓	✓	✓	✓	×
Formal parameters and return values	✓	✓	✓	✓	×	✓
Local variable declarations	✓	✓	✓	×	×	×
`new` statements	×	✓	✓	×	×	×
Cast statements	✓	✓	✓	✓	×	×

Table 3.2: Universe modifiers in the toy language.

3.2 Clusters

Let the set **TClust** represent the domain for possible cluster names to identify clusters. We assume **TClust** to be disjoint from each of **TClass**, **TMethod**, **TField** and **TLoc** so as to guarantee globally unique identifiers.Further, let the special cluster name $Cl_{\text{this}} \in \mathbf{TClust}$ denote the this-cluster.

As mentioned above, each field that is declared `uniq` defines one distinguished cluster. We assume an injective function

$$\text{definedCl}\colon \mathbf{TField} \rightarrow \mathbf{TClust}$$

which returns the name of cluster that is defined by a given field name. The result is undefined if the field is not declared `uniq`.

The function $\text{definedCls}\colon \mathbf{TClass} \rightarrow 2^{\mathbf{TClust}}$ then yields the clusters defined by a class C or a superclass (where the symbol ␣ denotes a “don’t care” placeholder):

$$\frac{}{\text{definedCls}(\texttt{Object}) = \{Cl_{\text{this}}\}}$$

$$\frac{\texttt{class } C \texttt{ extends } D \texttt{ \{ ...; uniq ␣ } f\texttt{; ... ␣ \}}}{\text{definedCls}(C) \supseteq \{\text{definedCl}(f)\} \cup \text{definedCls}(D)}$$

Note that the this-cluster Cl_{this} is contained in definedCls(C) for every class C.

3.3 Universe Modifier Translation

In the toy language, there are the following Universe modifiers:

$$\mathbf{TMod} = \{\texttt{this}, \texttt{any}, \texttt{peer}, \texttt{rep}, \texttt{uniq}, \texttt{free}\} \cup \left\{\texttt{rep[}f\texttt{]} : f \in \mathbf{TField}\right\},$$

where `this` corresponds to the Universe modifier of the `this` reference. I.e., the `this` reference has type $T = \texttt{this}\ C$, where C is the class declaring the method. A user cannot access the `this` modifier directly, but it will be used to simplify the formalization.

To represent the target clusters that are associated with each occurrence of a `rep`, `rep[`f`]`, `uniq` and `free` modifier in a program in the toy language, we perform a translation of each Universe modifier in **TMod** to a Universe modifier in the following set:

$$\mathbf{CMod} := \{\mathit{this}, \mathit{any}, \mathit{peer}, \mathit{rep}\langle ?\rangle\} \cup \{\mathit{rep}\langle \mathit{Cl}\rangle : \mathit{Cl} \in \mathbf{TClust}\}.$$

This can be viewed as a translation of the program from a *surface syntax* to a *core syntax* (**CMod** is supposed to stand for "core modifiers"). The core modifiers are distinguished from the surface syntax modifiers in that each *rep* modifier explicitly denotes its target cluster. This is either a *named* cluster $\mathit{Cl} \in \mathbf{TClust}$, denoted by $\mathit{rep}\langle \mathit{Cl}\rangle$, or the *wildcard cluster*, denoted by $\mathit{rep}\langle ?\rangle$. The translation is defined as follows:

$$\begin{array}{lcll}
\texttt{this} & \mapsto & \mathit{this} & \\
\texttt{any} & \mapsto & \mathit{any} & \\
\texttt{peer} & \mapsto & \mathit{peer} & \\
\texttt{rep} & \mapsto & \begin{cases} \mathit{rep}\langle \mathit{Cl}_{\text{this}}\rangle & \text{, for field declarations, formal} \\ & \quad\text{parameters and return values} \\ \mathit{rep}\langle ?\rangle & \text{, otherwise} \end{cases} & \\
\texttt{rep[}f\texttt{]} & \mapsto & \mathit{rep}\langle \text{definedCl}(f)\rangle & \text{, where } f \text{ must be declared } \texttt{uniq} \\
 & & & \quad\text{in the enclosing class} \\
\texttt{uniq} & \mapsto & \mathit{rep}\langle \text{definedCl}(f)\rangle & \text{, where } f \text{ is the declared field} \\
\texttt{free} & \mapsto & \mathit{rep}\langle ?\rangle &
\end{array}$$

As explained above, we require that the field f in a `rep[`f`]` modifier must be declared `uniq` in the enclosing class. The wildcard cluster captures the idea that the actual cluster of a local variable is unknown and will be inferred. Observe that the `free` modifier is also mapped to be pointing into the wildcard cluster ?. This can be interpreted by the fact that having a reference to a free cluster essentially corresponds to owning the cluster. Hence, it is of no concern what particular cluster is referenced by a free variable. Furthermore, we define $rep\langle ?\rangle \neq rep\langle Cl\rangle$ for any $Cl \in \mathbf{TClust}$ (which would correspond to $? \notin \mathbf{TClust}$).

Note that each field f has one of the type modifiers *any*, *peer* or $rep\langle Cl\rangle$, where $Cl \in \mathbf{TClust}$ (i.e., a field can never have the type modifier $rep\langle ?\rangle$). In case of a $rep\langle Cl\rangle$ modifier, the named cluster Cl is also referred to as the cluster associated with the field f.

Once the Universe modifiers have been translated, we will continue to work only with core Universe modifiers from this point on.

Example 3.1 Assuming C_g to be the cluster defined by field `g` (i.e., definedCl(`g`) $= C_g$), the following program on the left is translated to the program on the right:

```
class C extends Object {
    rep C f;
    uniq C g;
    rep[g] h;

    free C m(rep C p,
             rep[g] C q,
             free C r) {
        rep C l;
        l = new rep C;
        result = l;
    }
}
```

```
class C extends Object {
    $rep\langle Cl_{\text{this}}\rangle$ C f;
    $rep\langle C_g\rangle$ C g;
    $rep\langle C_g\rangle$ h;

    $rep\langle ?\rangle$ C m($rep\langle Cl_{\text{this}}\rangle$ C p,
                   $rep\langle C_g\rangle$ C q,
                   $rep\langle ?\rangle$ C r) {
        $rep\langle ?\rangle$ C l;
        l = new $rep\langle ?\rangle$ C;
        result = l;
    }
}
```

□

For a modifier $m \in \mathbf{CMod}$, we write $m = rep$ as shortcut for $m = rep\langle ?\rangle \vee m = rep\langle Cl\rangle$, where $Cl \in \mathbf{TClust}$. In other words, all we care about is that m is a *rep* modifier, regardless of whether m's cluster is unknown or a named cluster. Conversely though, as noted above, when we write $m = rep\langle Cl\rangle$, we always mean that $Cl \in \mathbf{TClust}$ and $m \neq rep\langle ?\rangle$.

3.4 Lookup Functions

This section introduces a couple of lookup functions to access pieces of static information. The functions are defined by inference rules, and we use the symbol ␣ to denote a "don't care" placeholder for a syntactic category.

- The function fields: **TClass** → $2^{\textbf{TField}}$ yields the identifiers of all fields that are declared in or inherited by a class C.

$$\frac{}{\text{fields}(\texttt{Object}) = \emptyset} \qquad \frac{\texttt{class } C \texttt{ extends } D\ \texttt{\{}\ \overline{T_f\ f\ \text{␣}}\ \texttt{\}}}{\text{fields}(C) = \{f_1, \ldots, f_k\} \cup \text{fields}(D)}$$

- The function fType: **TClass** × **TField** → **TType** yields the type of a field f as declared in class C. The result is undefined if f is not declared in C. Since identifiers are assumed to be globally unique, there is only one declaration for each field identifier.

$$\frac{\texttt{class } C \texttt{ extends }\ \text{␣}\ \texttt{\{ ...; } T_f\ f\texttt{; ... }\ \text{␣}\ \texttt{\}}}{\text{fType}(C, f) = T_f}$$

- The function mType: **TClass** × **TMethod** → **TPure** × **TType** × **TType** yields the signature of a method m as declared in class C. The result is undefined if m is not declared in C. As we do not allow overloading of methods, the method identifier is sufficient to uniquely identify a method.

$$\frac{\texttt{class } C \texttt{ extends }\ \text{␣}\ \texttt{\{}\ \text{␣} \cdots W\ T_r\ m\texttt{(}T_p\ \text{␣}\texttt{)}\ \texttt{\{}\ \text{␣}\ \text{␣}\ \texttt{\}} \cdots \texttt{\}}}{\text{mType}(C, m) = \big(W, T_p, T_r\big)}$$

- The function mLoc: **TClass** × **TMethod** → $2^{\textbf{TLoc}}$ yields the names of the local variables and formal parameters of a method m as declared in class C. The result is undefined if m is not declared in C.

$$\frac{\texttt{class } C \texttt{ extends }\ \text{␣}\ \texttt{\{}\ \text{␣} \cdots \text{␣}\ \text{␣}\ m\texttt{(}\text{␣}\ x\texttt{)}\ \texttt{\{}\ \overline{T_l\ y}\ \text{␣}\ \texttt{\}} \cdots \texttt{\}}}{\text{mLoc}(C, m) = \big\{y_1, \ldots, y_k\big\} \cup \{x\}}$$

- The function Δ: **TClass** × **TMethod** → **TLoc** → **TType** yields the static declaration environment for a method m as declared in class C. The result is undefined if m is not declared in C. We use the "maps-to" notation to describe the resulting function.

$$\frac{\texttt{class } C \texttt{ extends }\ \text{␣}\ \texttt{\{}\ \text{␣} \cdots \text{␣}\ T_r\ m\texttt{(}T_p\ x\texttt{)}\ \texttt{\{}\ \overline{T_l\ y}\ \text{␣}\ \texttt{\}} \cdots \texttt{\}}}{\Delta(C, m) = \{x \mapsto T_p, y_1 \mapsto T_{l,1}, \ldots, y_k \mapsto T_{l,k}, \texttt{this} \mapsto \texttt{this}\ C, \texttt{result} \mapsto T_r\}}$$

3.4.1 Type Projection Functions

To access the modifier and class name components of a type $T = m\ C$ we use the projection functions $\mathrm{mod}\colon \mathbf{TType} \to \mathbf{CMod}$ and $\mathrm{class}\colon \mathbf{TType} \to \mathbf{TClass}$, respectively:

$$\begin{aligned}\mathrm{mod}(m\ C) &\coloneqq m\\ \mathrm{class}(m\ C) &\coloneqq C\end{aligned}$$

Furthermore, let us introduce the following shortcut given a static declaration environment $\Gamma\colon \mathbf{TLoc} \to \mathbf{TType}$ and $x \in \mathbf{TLoc}$:

$$\Gamma_{\mathrm{m}}(x) \coloneqq \mathrm{mod}(\Gamma(x))\,.$$

3.5 Type Combinator

Table 3.3 displays the type combinator $\rhd_U\colon \mathbf{CMod} \times \mathbf{CMod} \to \mathbf{CMod}$

$\rhd_U$	*any*	*peer*	$\mathit{rep}\langle \mathit{Cl}\rangle$	$\mathit{rep}\langle ?\rangle$
this	*any*	*peer*	$\mathit{rep}\langle \mathit{Cl}\rangle$	$\mathit{rep}\langle ?\rangle$
any	*any*	*any*	*any*	$\mathit{rep}\langle ?\rangle$
peer	*any*	*peer*	*any*	$\mathit{rep}\langle ?\rangle$
$\mathit{rep}\langle \mathit{Cl}\rangle$	*any*	$\mathit{rep}\langle \mathit{Cl}\rangle$	*any*	$\mathit{rep}\langle ?\rangle$
$\mathit{rep}\langle ?\rangle$	*any*	$\mathit{rep}\langle ?\rangle$	*any*	$\mathit{rep}\langle ?\rangle$

Table 3.3: Core Universe modifier combinator.

In a more programmatic view, Table 3.3 can be summarized as follows, where x and y are to be seen as placeholders for Universe modifiers (the top-down order matters):

$$\begin{aligned}\mathit{this} \rhd_U x &= x\\ \mathit{peer} \rhd_U \mathit{peer} &= \mathit{peer}\\ \mathit{rep}\langle \mathit{Cl}\rangle \rhd_U \mathit{peer} &= \mathit{rep}\langle \mathit{Cl}\rangle\\ \mathit{rep}\langle ?\rangle \rhd_U \mathit{peer} &= \mathit{rep}\langle ?\rangle\\ x \rhd_U \mathit{rep}\langle ?\rangle &= \mathit{rep}\langle ?\rangle \qquad (3.1)\\ x \rhd_U y &= \mathit{any}\end{aligned}$$

The interesting line is (3.1): an arbitrary modifier combined with $rep\langle ?\rangle$ returns $rep\langle ?\rangle$ again, which corresponds to the `free` modifier combination described in Subsection 2.2.2. All other combinations agree with the standard Universe type system

We further introduce an overloaded version of the combinator operator for types (as opposed to Universe type modifiers), i.e., $\rhd_U : \textbf{TMod} \times \textbf{TMod} \to \textbf{TMod}$. For types $T_1 = m_1\ C_1$ and $T_2 = m_2\ C_2$, where $m_1, m_2 \in \textbf{CMod}$ and $C_1, C_2 \in \textbf{TClass}$, it is defined by

$$T_1 \rhd_U T_2 = (m_1\ C_1) \rhd_U (m_2\ C_2) := (m_1 \rhd_U m_2)\ \ C_2,$$

which means that the class part of the resulting type is the class part of the right-hand type (as in Java) and the Universe modifier of the resulting type is the combined modifier.

3.6 Assignable-To Relation

The subtype relation describes a is-a relation, in the sense that if type A is a subtype of type B, then the set of all variables of type A is contained in the set of all variables of type B. In contrast, the assignable-to relation regulates the expression types on the left and right side of an assignment operation. In conventional type systems, the subtype relation corresponds to the assignable-to relation. With ownership transfer, however, it does not, the reason being the fact that ownership transfers causes types to change.

Figure 3.1(a) displays the subtype relation among ownership modifiers. The hierarchy follows immediately by taking a being-less-specific-than point of view. The subtype relation will not be used any further in this formalization.

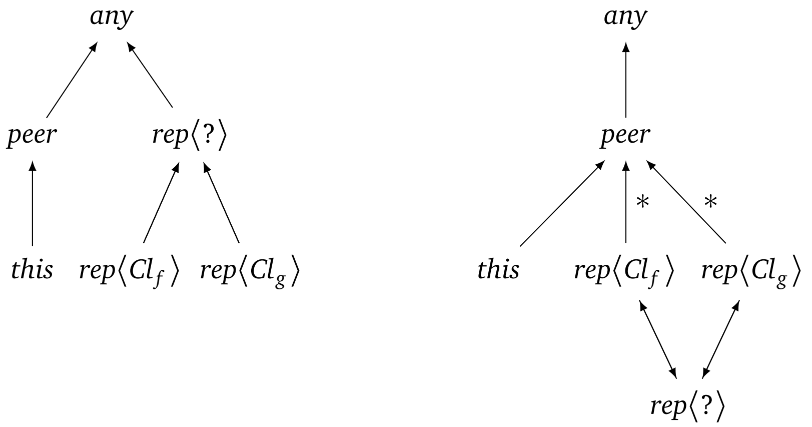
The assignable-to relation on types is defined by

$$\frac{m_1 <:_A m_2 \qquad C_1 <:_A C_2}{m_1\ C_1 <:_A m_2\ C_2}$$

where the the assignable-to relation on classes is given by

$$\frac{}{C <:_A C} \qquad \frac{C <:_A D \qquad D <:_A E}{C <:_A E} \qquad \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ \llcorner\!\lrcorner\ \llcorner\!\lrcorner\ \}}{C <:_A D}$$

(as in Java) and the assignable-to relation on type modifiers is outlined in Figure 3.1(b). This needs some explaining: For $Cl \in \textbf{TClust}$, $rep\langle Cl\rangle$ is defined to be a assignable-to *peer* to enable assignments from $rep\langle Cl\rangle$ to *peer* to allow for (implicit) ownership transfers. Note that, however, this is only valid if $Cl \neq Cl_{\text{this}}$, since the this-cluster must not be transfered. This would correspond to an additional condition $Cl \neq Cl_{\text{this}}$ on the edges marked $*$. In the formalization, however, a *rep*$\langle Cl \rangle$ is defined to be *always* assignable-to *peer* and the check for the condition $Cl \neq Cl_{\text{this}}$ is delegated to the static data flow analysis. This amounts to the view that the assignable-to relation consists of a static part as depicted in the figure, and of a dynamic part made up of information provided by the data flow analysis, for each point in the program. Moreover, a wildcard cluster type can be assigned to and from any non-wildcard cluster type, but two distinct non-wildcard cluster types are not assignable to each other ($Cl_f \neq Cl_g$ is supposed in the figure).



(a) Subtype relation.

(b) Assignable-to relation.

Figure 3.1: Subtype and assignable-to relation among core Universe modifiers.

3.7 Ternary Logic

To interface with the static data flow analysis, we will make use of a ternary logic with value set

$$\mathbb{T} := \{0, \tfrac{1}{2}, 1\}\,.$$

Whenever we query the data flow analysis whether a certain property holds at a certain point in the program, the analysis will deliver an answer $a \in \mathbb{T}$ with the following interpretation:

$a = 0$ ⇒ "The property does definitely *not* hold,
in all possible executions of the program."

$a = 1$ ⇒ "The property definitely holds,
in all possible executions of the program."

$a = ½$ ⇒ "The property holds in some executions,
but does not hold in other executions of the program."

Accordingly, the values 0 and 1 represent precise answers ("no" and "yes"), while ½ stands for an unknown state ("I don't know"). This way, we combine a "must" analysis with a "may" analysis. Note that a trivial implementation of the data flow analysis would be to answer every query with ½. However, this approach would of course have the least possible precision.

Using a ternary logic has two major benefits over the standard binary logic:

- The type checker can give more accurate error messages to the user (e.g., "Variable x is unusable" vs. "Variable x may be unusable").
- As will be explained in the subsequent chapter, there will in fact propose several data flow analyses – each one of a different precision (and of a different time complexity). The distinction between precise answers and "I don't know" answers enables us to use the analyses in the following way: We can start with the least precise analysis and switch to an analysis of higher precision in case the current analysis reports an imprecise answer. We do not need to increase the precision level if a precise answer is returned. Refer to Subsection 4.10.1 for more detail on such a hierarchical analysis solving method.

On the other hand, the apparent downside of the ternary logic approach is an increased complexity when dealing with analysis values.

Logical Operators. Table 3.4 displays the truth tables of logical operators that will be used in the following. As in conventional binary logic, the "and" operator $\wedge_T$ has 0 as controlling value (i.e., $x \wedge_T 0 = 0$) and 1 as non-controlling value (i.e., $x \wedge_T 1 = x$). The "or" operator $\vee_T$ is the dual of $\wedge_T$. The "join" operator $\sqcup_T$ reflects the intuition that we can only obtain a precise answer if we join two precise answers of the same kind.

$\wedge_T$	0	½	1
0	0	0	0
½	0	½	½
1	0	½	1

(a) Trivalent “and”.

$\vee_T$	0	½	1
0	0	½	1
½	½	½	1
1	1	1	1

(b) Trivalent “or”.

$\sqcup_T$	0	½	1
0	0	½	½
½	½	½	½
1	½	½	1

(c) Trivalent “join”.

Table 3.4: Ternary logic operators.

3.8 Static Data Flow Analysis – Introduction

In the most abstract view, the static data flow analysis tracks which variables point into which cluster and which variables are unusable. The analysis is *intra-procedural*. I.e., a single analysis run is executed for a fixed method m in class C and a static declaration environment $\Gamma = \Delta(C, m)$. In the following, free occurrences of m and C are meant to stand for this fixed method m in class C.

3.8.1 Analysis Values and Queries

An *analysis value* represents the property we are interested in. Together, the analysis values form the *property space* $\mathcal{L}$ or the *analysis universe*. Viewed abstractly, for each program point, the analysis computes an analysis value $L \in \mathcal{L}$ which holds before (at the entry to) the program point. The analysis value the offers two query functions for clients to obtain the analysis information. First, the predicate

$$\mathit{isUnusable}\colon \mathcal{L} \times \mathrm{mLoc}(C, m) \cup \mathrm{fields}(C) \to \mathbb{T}$$

returns whether a given field or local variable x is unusable. Referring to Section 3.7, we can interpret the function values as follows:

$\mathit{isUnusable}(L, x) = 0 \quad \Rightarrow$ “No, x is definitely *not* unusable,
in all possible executions of the program.”

$\mathit{isUnusable}(L, x) = 1 \quad \Rightarrow$ “Yes, x is definitely unusable,
in all possible executions of the program.”

$\mathit{isUnusable}(L, x) = ½ \quad \Rightarrow$ “I don’t know.”

Furthermore, the function

$$\mathit{pointsInto}\colon \mathcal{L} \times \mathrm{mLoc}(C,m) \cup \mathrm{fields}(C) \times \mathrm{definedCls}(C) \to \mathbb{T}$$

checks whether a given field or local variable x points into a given cluster Cl. The result values are interpreted accordingly.

There is a special analysis value $\iota \in \mathcal{L}$ standing for the value that holds at the beginning of the analysis. It can be described as follows:

- Each field points into its associated cluster.
- Each local variable points into a new cluster of its own.

3.8.2 Analysis Transition Functions

An analysis *transition function* operates on the analysis property space $\mathcal{L}$ and representing the effect a certain entity has on an analysis value $L \in \mathcal{L}$. Note that the conventional name for such a function would be a *transfer function*. However, so as to avoid possible confusion with the notion of ownership transfer, we will stick to the term "transition function".

For an analysis value $L \in \mathcal{L}$, variables $x, y \in \mathrm{mLoc}(C,m) \cup \mathrm{fields}(C)$ and a cluster $\mathit{Cl} \in \mathrm{definedCls}(C)$, the following compilation gives an informal description of the transition functions that will be used:

- $\mathit{Consume}(L,x)$ – x and all locals and fields that point into the same cluster as x are marked as unusable.
- $\mathit{Consume}(L,\mathit{Cl})$ – All locals and fields that point into cluster Cl are marked as unusable.
- $\mathit{Merge}\big(L,x,y\big)$ – x and all locals and fields that point into the same cluster as x are marked as pointing into the same cluster as y.
- $\mathit{Merge}(L,x,\mathit{Cl})$ – x and all locals and fields that point into the same cluster as x are marked as pointing into cluster Cl.
- $\mathit{Move}\big(L,x,y\big)$ – x is marked as pointing into the same cluster as y.
- $\mathit{Move}(L,x,\mathit{Cl})$ – x is marked as pointing into cluster Cl.

- *New*(L,x) – x is marked as not unusable and all other variables and fields are marked as not pointing into the same cluster as x.

- *ConsumeLocals*(L) – All locals pointing into cluster Cl are marked as unusable, for all clusters $Cl \in \text{definedCls}(C) \setminus \{Cl_{\text{this}}\}$.

The *Consume* and *ConsumeLocals* operations are used to make variables unusable. The *Merge* operation represents merging two clusters in case a reference between them is created. The *Move* operation marks a single variable as pointing to a certain cluster. Finally, the *New* operation makes a variable point into a new cluster of its own.

The transition functions will be defined formally in Chapter 4.

Implicit Consume Precondition. In order to keep the type rules (described below) simple, we assume an implicit precondition

$$pointsInto(L,x,Cl_{\text{this}}) = 0$$

for a *Consume*(L,x) transition function. This ensures that the this-cluster Cl_{this} is never transfered.

3.9 Analysis Value Transition Rules

The following set of rules describes the effect that each statement of the toy language has in terms of the analysis values. I.e., each statement of the toy language is translated into a (or possibly a sequence of) analysis transition functions. This can equivalently be seen as a translation of a statement of the toy language into a (or possibly a sequence of) statements in an abstract analysis language, where each such statement corresponds to a transition function. In fact, this is exactly what is done in the implementation (cf. Chapter 5, or more specifically Subsection 5.3.5).

To begin with, let us introduce a function handleArg: $\mathcal{L} \times \mathbf{CMod} \times \mathbf{CMod} \times \mathbf{CMod} \rightarrow \mathcal{L}$ that describes the handling of a method call argument z of type modifier m_z in a method call with a formal parameter of type modifier m_p on a receiver object y of type modifier m_y. It is defined as follows:

$$\text{handleArg}\big(L, y, z, m_y, m_z, m_p\big) :=$$
$$\begin{cases} \mathit{Merge}\big(L, y, z\big) & \text{, if } m_z = \mathit{rep} \wedge m_y = \mathit{rep}\langle ?\rangle \wedge m_p = \mathit{peer}; \\ \mathit{Consume}(L, z) & \text{, if } m_z = \mathit{rep} \wedge m_y \neq \mathit{rep}\langle ?\rangle \\ & \quad \wedge\, m_y \rhd_U m_p \in \{\mathit{peer}, \mathit{rep}\langle ?\rangle\}\,; \\ \mathit{Merge}(L, \mathit{Cl}, z) & \text{, if } m_z = \mathit{rep} \wedge m_y \rhd_U m_p = \mathit{rep}\langle \mathit{Cl}\rangle; \\ L & \text{, otherwise.} \end{cases}$$

The first case corresponds to a possible capturing of the wildcard cluster. The second case corresponds to a releasing of the cluster pointed into by z. Finally, the third case corresponds to a possible capturing of the cluster *Cl*. If z is not of a *rep* type, the analysis value does not change. The handleArg function is extracted in order to be able to reuse it in both the rules [L-Pre-Invk] and [T-Pre-Invk] (described below). In the rule [T-Pre-Invk], we need to get a reference to the analysis value after a method call argument is handled.

The rules are expressed using judgments of the form $\Gamma; L \vdash_{\mathcal{L}} S : L'$, which expresses that the statement S causes the analysis value changes from L to L' in a given static declaration environment Γ: **TLoc** → **TType**.

$$[\text{L-Assign}]\ \frac{L' = \begin{cases} \mathit{Consume}\big(L, y\big) & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{peer} \wedge \Gamma_{\mathrm{m}}\big(y\big) = \mathit{rep} \\ \mathit{Move}\big(L, x, y\big) & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{rep} \wedge \Gamma_{\mathrm{m}}\big(y\big) = \mathit{rep} \\ L & \text{, otherwise} \end{cases}}{\Gamma; L \vdash_{\mathcal{L}} x\ \texttt{=}\ y : L'}$$

$$[\text{L-Null}]\ \frac{L' = \begin{cases} \mathit{New}(L, x) & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{rep} \\ L & \text{, otherwise} \end{cases}}{\Gamma; L \vdash_{\mathcal{L}} x\ \texttt{=}\ \texttt{null} : L'}$$

$$[\text{L-New}]\ \frac{L' = \begin{cases} \mathit{New}(L, x) & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{rep} \\ L & \text{, otherwise} \end{cases}}{\Gamma; L \vdash_{\mathcal{L}} x\ \texttt{=}\ \texttt{new}\ T : L'}$$

$$\text{[L-Field-Read]}\ \dfrac{\begin{array}{c} m_f = \mathrm{mod}\big(\mathrm{fType}\big(\mathrm{class}\big(\Gamma\big(y\big)\big), f\big)\big) \\ L' = \begin{cases} \mathit{Consume}(L, \mathit{Cl}) & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{peer} \wedge \Gamma_{\mathrm{m}}\big(y\big) = \mathit{this} \wedge m_f = \mathit{rep}\big\langle \mathit{Cl}\big\rangle \\ \mathit{Consume}\big(L, y\big) & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{peer} \wedge \Gamma_{\mathrm{m}}\big(y\big) = \mathit{rep} \wedge m_f = \mathit{peer} \\ \mathit{Move}(L, x, \mathit{Cl}) & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{rep} \wedge \Gamma_{\mathrm{m}}\big(y\big) = \mathit{this} \wedge m_f = \mathit{rep}\big\langle \mathit{Cl}\big\rangle \\ \mathit{Move}\big(L, x, y\big) & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{rep} \wedge \Gamma_{\mathrm{m}}\big(y\big) = \mathit{rep} \wedge m_f = \mathit{peer} \\ L & \text{, otherwise} \end{cases} \end{array}}{\Gamma; L \vdash_{\mathcal{L}} x\ =\ y.f\ :\ L'}$$

$$\text{[L-Field-Write]}\ \dfrac{\begin{array}{c} m_f = \mathrm{mod}\big(\mathrm{fType}\big(\mathrm{class}\big(\Gamma\big(y\big)\big), f\big)\big) \\ L' = \begin{cases} \mathit{Consume}(L, x) & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{rep} \\ & \quad \wedge\ \Gamma_{\mathrm{m}}\big(y\big) \rhd_U m_f = \mathit{peer} \\ \mathit{Move}\big(\mathit{Merge}(L, \mathit{Cl}, x), f, \mathit{Cl}\big) & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{rep} \\ & \quad \wedge\ \Gamma_{\mathrm{m}}\big(y\big) = \mathit{this} \wedge m_f = \mathit{rep}\big\langle \mathit{Cl}\big\rangle \\ \mathit{Merge}\big(L, x, y\big) & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{rep} \\ & \quad \wedge\ \Gamma_{\mathrm{m}}\big(y\big) = \mathit{rep} \wedge m_f = \mathit{peer} \\ L & \text{, otherwise} \end{cases} \end{array}}{\Gamma; L \vdash_{\mathcal{L}} y.f\ =\ x\ :\ L'}$$

$$\text{[L-Pre-Invk]}\ \dfrac{\begin{array}{c} C = \mathrm{class}(\Gamma(\texttt{this})) \\ \big(W, T_p, T_r\big) = \mathrm{mType}(C, m) \\ L' = \mathrm{handleArg}\big(L, y, z, \Gamma_{\mathrm{m}}\big(y\big), \Gamma_{\mathrm{m}}(z), \mathrm{mod}\big(T_p\big)\big) \\ L'' = \begin{cases} \mathit{ConsumeLocals}\big(L'\big) & \text{, if } \Gamma_{\mathrm{m}}(z) \in \{\mathit{peer}, \mathit{this}\} \wedge W = \texttt{nonpure} \\ L' & \text{, otherwise} \end{cases} \end{array}}{\Gamma; L \vdash_{\mathcal{L}} y.m(z)\ :\ L''}$$

$$
[\text{L-Invk}]\ \dfrac{\begin{array}{c} C = \text{class}(\Gamma(\texttt{this})) \\ \left(W, T_p, T_r\right) = \text{mType}(C, m) \\ \Gamma; L \vdash_{\mathcal{L}} \mathtt{y.m(z)} : L' \\ L'' = \begin{cases} \mathit{Move}\left(L', x, \mathit{Cl}\right) & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{rep} \wedge \Gamma_{\mathrm{m}}\left(y\right) \rhd_U \text{mod}\left(T_r\right) = \mathit{rep}\langle \mathit{Cl} \rangle \\ \mathit{New}\left(L', x\right) & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{rep} \wedge \Gamma_{\mathrm{m}}\left(y\right) \rhd_U \text{mod}\left(T_r\right) = \mathit{rep}\langle ? \rangle \\ L' & \text{, otherwise} \end{cases} \end{array}}{\Gamma; L \vdash_{\mathcal{L}} \mathtt{x\ =\ y.m(z)} : L''}
$$

$$
[\text{L-Cast}]\ \dfrac{\begin{array}{c} L' = \begin{cases} \mathit{Consume}\left(L, y\right) & \text{, if mod}(T) = \mathit{peer} \wedge \Gamma_{\mathrm{m}}\left(y\right) = \mathit{rep} \\ \mathit{Merge}\left(L, \mathit{Cl}, y\right) & \text{, if mod}(T) = \mathit{rep}\langle \mathit{Cl} \rangle \wedge \Gamma_{\mathrm{m}}\left(y\right) = \mathit{rep} \\ L & \text{, otherwise} \end{cases} \\ L'' = \begin{cases} \mathit{Move}\left(L', x, \mathit{Cl}\right) & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{rep} \wedge \text{mod}(T) = \mathit{rep}\langle \mathit{Cl} \rangle \\ \mathit{Move}\left(L', x, y\right) & \text{, if } \Gamma_{\mathrm{m}}(x) = \text{mod}(T) = \mathit{rep}\langle ? \rangle \wedge \Gamma_{\mathrm{m}}\left(y\right) = \mathit{rep} \\ L' & \text{, otherwise} \end{cases} \end{array}}{\Gamma; L \vdash_{\mathcal{L}} \mathtt{x\ =\ (}T\mathtt{)\ y} : L''}
$$

$$
[\text{L-Seq}]\ \dfrac{\Gamma; L \vdash_{\mathcal{L}} S_1 : L' \qquad \Gamma; L' \vdash_{\mathcal{L}} S_2 : L''}{\Gamma; L \vdash_{\mathcal{L}} S_1\mathtt{;}\ \ S_2 : L''}
$$

Let us discuss some aspects.

- The *Consume* operation in the [L-Assign] rule corresponds to an implicit ownership transfer.
- The [L-Null] and the [L-New] rules have the same effect.
- Note the *ConsumeLocals* operation in case of a non-pure *peer* (or *this*) call in the rule [L-Pre-Invk] to mark all local variables that point into a non-free cluster as unusable.

3.10 Type Rules

This section describes the rules to type check the toy language. Let us begin with the rules for statements. A judgment has the form $\Gamma; L \vdash S$ and expresses that the statement S is well-typed in a static declaration environment $\Gamma\colon$ **TLoc** → **TType**, where L is the analysis value before type checking the statement S.

$$[\text{T-Assign}]\ \frac{\Gamma(x) :>_A \Gamma(y) \qquad \mathit{isUnusable}(L, y) = 0}{\Gamma; L \vdash x \texttt{ = } y}$$

$$[\text{T-Null}]\ \frac{}{\Gamma; L \vdash x \texttt{ = null}}$$

$$[\text{T-New}]\ \frac{\Gamma(x) :>_A T}{\Gamma; L \vdash x \texttt{ = new } T}$$

$$[\text{T-Field-Read}]\ \frac{\begin{array}{c} T_f = \text{fType}(\text{class}(\Gamma(y)), f) \\ \Gamma(x) :>_A \left(\Gamma(y) \rhd_U T_f\right) \\ \mathit{isUnusable}(L, y) = 0 \\ \Gamma_{\text{m}}(y) = \mathit{this} \Rightarrow \mathit{isUnusable}(L, f) = 0 \end{array}}{\Gamma; L \vdash x \texttt{ = } y.f}$$

$$[\text{T-Field-Write}]\ \frac{\begin{array}{c} T_f = \text{fType}(\text{class}(\Gamma(y)), f) \\ \left(\Gamma(y) \rhd_U T_f\right) :>_A \Gamma(x) \\ \Gamma_{\text{m}}(y) \neq \mathit{any} \\ \text{mod}\left(T_f\right) = \mathit{rep}\langle \mathit{Cl}\rangle \Rightarrow \Gamma_{\text{m}}(y) = \mathit{this} \\ \mathit{isUnusable}(L, x) = 0 \\ \mathit{isUnusable}(L, y) = 0 \end{array}}{\Gamma; L \vdash y.f \texttt{ = } x}$$

$$[\text{T-Pre-Invk}]\ \frac{\begin{array}{c} C = \text{class}(\Gamma(\texttt{this})) \\ \left(W, T_p, T_r\right) = \text{mType}(C, m) \\ \left(\Gamma(y) \rhd_U T_p\right) :>_A \Gamma(z) \\ W = \texttt{nonpure} \Rightarrow \Gamma_{\text{m}}(y) \neq \mathit{any} \\ \mathit{isUnusable}(L, z) = 0 \\ L' = \text{handleArg}\left(L, y, z, \Gamma_{\text{m}}(y), \Gamma_{\text{m}}(z), \text{mod}\left(T_p\right)\right) \\ \mathit{isUnusable}\left(L', y\right) = 0 \\ \Gamma_{\text{m}}(y) \in \{\mathit{peer}, \mathit{this}\} \wedge W = \texttt{nonpure} \Rightarrow \forall f \in \text{fields}(C):\ \mathit{isUnusable}\left(L', f\right) = 0 \end{array}}{\Gamma; L \vdash y.m(z)}$$

$$[\text{T-Invk}]\ \frac{\begin{array}{c} C = \text{class}(\Gamma(\texttt{this})) \\ \left(w, T_p, T_r\right) = \text{mType}(C, m) \\ \Gamma(x) :>_A \left(\Gamma(y) \rhd_U T_r\right) \\ \Gamma; L \vdash y.m(z) \end{array}}{\Gamma; L \vdash x\ \texttt{=}\ y.m(z)}$$

$$[\text{T-Cast}]\ \frac{\begin{array}{c} \Gamma(x) :>_A T \\ m_y = \Gamma_{\text{m}}(y) \\ m_T = \text{mod}(T) \\ m_T :>_A m_y \vee \left(m_T \in \{\mathit{peer}, \mathit{rep}\langle \mathit{Cl}\rangle\} \wedge m_y = \mathit{any}\right) \\ \mathit{isUnusable}(L, y) = 0 \end{array}}{\Gamma; L \vdash x\ \texttt{=}\ \texttt{(}T\texttt{)}\ y}$$

$$[\text{T-Seq}]\ \frac{\Gamma; L \vdash S_1 \qquad \Gamma; L \vdash_{\mathcal{L}} S_2 : L' \qquad \Gamma; L' \vdash S_2}{\Gamma; L \vdash S_1\texttt{;}\ S_2}$$

Let us again point out some details.

- The type rules perform the usual Java checks (e.g., if the right-hand side in an assignment is assignable to the left-hand side) but also check with the help of the analysis values that no unusable variable is read.

- The [T-Pre-Invk] rule enforces the check that no field be unusable before a non-pure *peer* (or *this*) call. Note that the check is carried out *after* the argument of the method call is handled. The same holds for the check that the receiver object y is not unusable. The reason for the latter are such cases as $z\texttt{.}m\texttt{(}z\texttt{)}$, which should yield an error if z is consumed.

- The [T-Cast] rule forbids casts from *peer* to *rep*.

The following type rule describes how a method declaration is type checked. A judgment of the form $C \vdash_{\mathcal{M}} W \;\; T_r \;\; m\texttt{(}T_p \;\; x\texttt{)} \;\; \texttt{\{} \;\; \overline{T_l \;\; y} \;\; S \;\; \texttt{\}}$ expresses that the method declaration in class C is well-formed. Note in particular the check that no field is unusable after the method, starting out with the initial analysis value $\imath$, has terminated yielding analysis value L.

$$\text{[T-MDecl]}\;\frac{\begin{array}{c} W = \texttt{pure} \Rightarrow \mathrm{mod}\left(T_r\right), \mathrm{mod}\left(T_p\right) \in \{\mathit{any}, \mathit{rep}\langle ? \rangle\} \\ \Delta(C,m); \imath \vdash S \\ \Delta(C,m); \imath \vdash_{\mathcal{L}} S : L \\ \forall f \in \mathrm{fields}(C) : \; \mathit{isUnusable}\left(L, f\right) = 0 \\ \mathit{isUnusable}(L, \texttt{result}) = 0 \end{array}}{C \vdash_{\mathcal{M}} W \;\; T_r \;\; m\texttt{(}T_p \;\; x\texttt{)} \;\; \texttt{\{} \;\; \overline{T_l \;\; y} \;\; S \;\; \texttt{\}}}$$

Finally, we present the type rules for class declarations and programs. They should be self-explanatory.

$$\text{[T-CDecl]}\;\frac{C \vdash_{\mathcal{M}} M_1 \qquad \cdots \qquad C \vdash_{\mathcal{M}} M_k}{\vdash_{\mathcal{C}} \texttt{class} \;\; C \;\; \texttt{extends} \;\; D \;\; \texttt{\{} \;\; \overline{T_f \;\; f} \;\; \overline{M} \;\; \texttt{\}}}$$

$$\text{[T-Program]}\;\frac{\vdash_{\mathcal{C}} \mathit{CDecl}_1 \qquad \cdots \qquad \vdash_{\mathcal{C}} \mathit{CDecl}_k}{\vdash_{\mathcal{P}} \overline{\mathit{CDecl}}}$$

4

Data Flow Analysis

This chapter describes in detail the techniques used in conjunction with the static data flow analysis. Much of the contents is based on material exhibited in the book "Principles of Program Analysis" [18], with some adaptations applied so as to better fit our purposes.

After introducing an abstract analysis language and describing the construction of a control flow graph, we will present multiple analyses, defined both concretely and abstractly using the notion of a Monotone Framework. Each analysis variant leads to a different precision level and time and space complexity. We continue by presenting various worklist algorithms for solving data flow equations and conclude with a presentation of some benchmarks to evaluate the practical performance of the analyses.

4.1 Syntax of the Analysis Language

The static data flow analysis does neither operate directly on the Java source, nor on the toy language presented in Chapter 3. Instead, its target language is an abstract language, hereafter referred to as the "analysis language", especially designed for the analysis. This separation facilitates the design of solver algorithms as well as testing and debugging. Since all operations not relevant to the analysis are removed during the abstraction, this has the further advantage of a reduced amount of code to deal with, thus improving the speed and the scalability of the analysis. However, in order to seamlessly support as many common Java constructs as possible, the analysis language is rather rich, featuring, e.g., `switch` statements and `try`/`catch`/`finally` clauses.

There is only one syntactic category in the analysis language, namely **AStmt** describing the analysis statements:

$$S \in \mathbf{AStmt} \quad \text{analysis statements.}$$

Further, we assume some countable set **AVar** of variable names as well as some countable set **ALab** of labels for labeled statements are given to use the following meta variables:

$$\begin{array}{lll} x, y & \in \mathbf{AVar} & \text{analysis variable names} \\ \ell & \in \mathbf{ALab} & \text{analysis statement labels.} \end{array}$$

In the following we shall describe the abstract syntax for a statement the analysis language. The syntax is abstract in the sense that it specifies the abstract syntax trees of the language; the presented grammar, however, does contain ambiguities. We shall use indenting in textual representations of abstract syntax trees to disambiguate it.

First of all, there are elementary statements that perform a certain operation on the analysis values:

$$\begin{array}{rl} S ::= & \texttt{skip} \\ \mid & \texttt{merge(}x\texttt{, }y\texttt{)} \\ \mid & \texttt{new(}x\texttt{)} \\ \mid & \texttt{move(}x\texttt{, }y\texttt{)} \\ \mid & \texttt{consume(}x\texttt{)} \\ \mid & \texttt{consumeLocals} \\ \mid & \ldots \end{array}$$

These are the statements that correspond to the transition functions used in the rules that describe the effect a statement in the toy language given in Section 3.9. The exact semantics of the operations will be formally defined further below.

Secondly, there are elementary statements which do not actually represent an operation on analysis values but some change in the control flow:

$$\begin{array}{lr}
S ::= \ldots & \\
\quad \mid \texttt{break} \mid \texttt{break}\ \ell & \text{(unlabeled and labeled break)} \\
\quad \mid \texttt{continue} \mid \texttt{continue}\ \ell & \text{(unlabeled and labeled continue)} \\
\quad \mid \texttt{exit} & \text{(method termination)} \\
\quad \mid \ldots &
\end{array}$$

The unlabeled and labeled `break` and `continue` statements have the same meaning and usage restrictions as in Java. The `exit` statement is used to mark an immediate method termination, just like the `return` statement in Java.

Finally, there are composite statements with other statements as children, defining various control flow constructs:

$$\begin{array}{lr}
S ::= \ldots & \\
\quad \mid S_1\texttt{;}\ S_2 & \\
\quad \mid \texttt{if}\ S_1\ \texttt{then}\ S_2\ \texttt{else}\ S_3 & \\
\quad \mid \texttt{while}\ S_1\ \texttt{do}\ S_2 & \text{(while loop)} \\
\quad \mid \texttt{do}\ S_1\ \texttt{while}\ S_2 & \text{(do loop)} \\
\quad \mid \ell\texttt{:}\ S & \text{(labeled statement)} \\
\quad \mid \texttt{switch}\ S_1\ \texttt{case}\ S_2 & \\
\qquad\qquad\quad \ldots & \\
\qquad\qquad\quad \texttt{case}\ S_k & \\
\quad \mid \texttt{try}\ S_1\ \texttt{catch}\ S_2 & \\
\qquad\qquad \ldots & \\
\qquad\qquad \texttt{catch}\ S_k & \\
\qquad\qquad \texttt{finally}\ S_{k+1} &
\end{array}$$

where $k \geq 2$. These statements directly correspond to their counterparts in Java. It should be pointed out, however, that the conditions in `if`, `while`, `do` and `switch` statements are ordinary analysis statements and there is no notion of an expression. Hence, a `while` loop in the analysis language might look like this:

```
while merge(x, y); consume(x) do
    new(x);
    move(z, x)
```

The reason for this design is the desire to support Java expressions whose evaluations have side effects that are relevant for the analysis, such as `(x.f = y) == (z = x)`. The ultimate boolean value of such a condition expression, however, is of no relevance to the analysis.

As another remark, the `while` and `do` loops are treated separately since the common translation of a `do` loop

do S_1 while S_2

into a `while` loop

S_1; while S_2 do S_1

is inaccurate if S_1 contains `break` or `continue` statements. (Note that, in contrast, a Java `for` loop *can* be equivalently expressed by a `while` loop.)

4.2 Flow Graphs

The data flow analysis operates on a graph representation of a given analysis statement (or analysis program), the *flow graph*. This section describes in a formal way how such a flow graph is constructed.

The *nodes* or *elementary blocks* in a flow graph correspond to elementary analysis statements. We shall thus use the terms "elementary statement", "node" and "elementary block" as synonyms. Edges in the flow graph represent possible control flow transitions from one node to another. Most of these edges are straight-forward and agree with what a programmer would expect. However, care has to be taken when modeling the transition edges associated with "unusual" statements such as `break`, `switch` and `try`/`catch`/`finally` statements.

Indexed Elementary Statements. For convenience in both the formalization and implementation, we assume that each elementary statement (node or elementary block, respectively) is labeled by an index $i \in \mathbf{AIndex}$ which uniquely identifies the elementary statement. The index set **AIndex** does not need to be further defined, but can be thought of as being equal to $\mathbb{N}$. In what follows, we will use the terms "elementary statement i" and "elementary statement with index i" interchangeably. Furthermore, the notation $[S]^i$ is used to denote a particular elementary statement S with index i, for instance,

$$[\texttt{skip}]^i,\ [\texttt{merge(}x\texttt{, }y\texttt{)}]^i \text{ or } [\texttt{break}]^i.$$

To define the construction of the flow graph out of an analysis statement, we will make use of a number of functions operating on analysis statements. They are all inductively defined over the structure of a statement.

Initial Index. The first one of those functions is

$$\mathit{init}\colon \mathbf{AStmt} \to \mathbf{AIndex}$$

which returns the *initial index* (the index of the initial node) of a statement. The definition should contain no surprises:

$$\begin{aligned}
\mathit{init}\big([S]^i\big) &:= i \text{ , for any elem. statement } S\\
\mathit{init}(S_1\texttt{;}\ S_2) &:= \mathit{init}(S_1)\\
\mathit{init}(\texttt{if}\ S_1\ \texttt{then}\ S_2\ \texttt{else}\ S_3) &:= \mathit{init}(S_1)\\
\mathit{init}(\texttt{while}\ S_1\ \texttt{do}\ S_2) &:= \mathit{init}(S_1)\\
\mathit{init}(\texttt{do}\ S_1\ \texttt{while}\ S_2) &:= \mathit{init}(S_1)\\
\mathit{init}(\ell\texttt{:}\ S) &:= \mathit{init}(S)\\
\mathit{init}(\texttt{switch}\ S_1\ \texttt{case}\ S_2\ \cdots &\\
\texttt{case}\ S_k) &:= \mathit{init}(S_1)\\
\mathit{init}(\texttt{try}\ S_1\ \texttt{catch}\ S_2\ \cdots &\\
\texttt{catch}\ S_k\ \texttt{finally}\ S_{k+1}) &:= \mathit{init}(S_1)
\end{aligned}$$

Final Indices. We also need a function

$$\mathit{final}\colon \mathbf{AStmt} \to 2^{\mathbf{AIndex}}$$

to return the set of *final indices* (the indices of the final nodes) of a statement. While a statement has only a single entry, it may have multiple exits (as, e.g., in the conditional):

$$\begin{aligned}
\mathit{final}\Big(\texttt{[skip]}^{i}\Big) &:= \{i\}\\
\mathit{final}\Big(\texttt{[merge(}x\texttt{, }y\texttt{)]}^{i}\Big) &:= \{i\}\\
\mathit{final}\Big(\texttt{[new(}x\texttt{)]}^{i}\Big) &:= \{i\}\\
\mathit{final}\Big(\texttt{[move(}x\texttt{, }y\texttt{)]}^{i}\Big) &:= \{i\}\\
\mathit{final}\Big(\texttt{[consume(}x\texttt{)]}^{i}\Big) &:= \{i\}\\
\mathit{final}\Big(\texttt{[consumeLocals]}^{i}\Big) &:= \{i\}\\
\mathit{final}\Big(\texttt{[break]}^{i}\Big) &:= \emptyset\\
\mathit{final}\Big(\texttt{[break }\ell\texttt{]}^{i}\Big) &:= \emptyset\\
\mathit{final}\Big(\texttt{[continue]}^{i}\Big) &:= \emptyset\\
\mathit{final}\Big(\texttt{[continue }\ell\texttt{]}^{i}\Big) &:= \emptyset\\
\mathit{final}\Big(\texttt{[exit]}^{i}\Big) &:= \emptyset\\
\mathit{final}\big(S_1\texttt{; }S_2\big) &:= \mathit{final}\big(S_2\big)\\
\mathit{final}\big(\texttt{if }S_1\texttt{ then }S_2\texttt{ else }S_3\big) &:= \mathit{final}\big(S_2\big) \cup \mathit{final}\big(S_3\big)\\
\mathit{final}\big(\texttt{while }S_1\texttt{ do }S_2\big) &:= \mathit{final}\big(S_1\big)\\
\mathit{final}\big(\texttt{do }S_1\texttt{ while }S_2\big) &:= \mathit{final}\big(S_2\big)\\
\mathit{final}(\ell\texttt{: }S) &:= \mathit{final}(S)\\
\mathit{final}\big(\texttt{switch }S_1\texttt{ case }S_2\ \cdots\\
\texttt{case }S_k\big) &:= \mathit{final}\big(S_k\big)\\
\mathit{final}\big(\texttt{try }S_1\texttt{ catch }S_2\ \cdots\\
\texttt{catch }S_k\texttt{ finally }S_{k+1}\big) &:= \mathit{final}\big(S_{k+1}\big)
\end{aligned} \tag{4.1}$$

A few things to note here: Firstly, the final indices are used to describe the connecting control flow edges between the child statements of composite statements. Consequently, `break`, `continue` and `exit` statements do not count as final nodes in this formalization and thus have no final indices, as their successor statements are unreachable. The control flow edges given rise to by `break` and `continue` statements are accounted for separately. Secondly, the final indices of the loop statements are the final indices of the loop condition statements, since the loops terminate immediately after the condition has evaluated to false.

Nodes. Let us now turn to a function that extracts the elementary statements out of a given analysis statement. These elementary statements will incorporate the nodes of our flow graph. The function is called “nodes” (rather than “elemStmt” or similar) to emphasize the graph domain. Since we represent the nodes by their indices, the function returns a set of indices:

$$\mathit{nodes}\colon \mathbf{AStmt} \to 2^{\mathbf{AIndex}}.$$

An elementary statement amounts to a single node having the same index. The node set of a composite statement is simply the union of the nodes of the child statements:

$$\begin{aligned}
\mathit{nodes}\big([S]^i\big) &:= \{i\} \text{ , for any elem. statement } S \\
\mathit{nodes}(S_1\texttt{;}\ S_2) &:= \mathit{nodes}(S_1) \cup \mathit{nodes}(S_2) \\
\mathit{nodes}(\texttt{if}\ S_1\ \texttt{then}\ S_2\ \texttt{else}\ S_3) &:= \mathit{nodes}(S_1) \cup \mathit{nodes}(S_2) \cup \mathit{nodes}(S_3) \\
\mathit{nodes}(\texttt{while}\ S_1\ \texttt{do}\ S_2) &:= \mathit{nodes}(S_1) \cup \mathit{nodes}(S_2) \\
\mathit{nodes}(\texttt{do}\ S_1\ \texttt{while}\ S_2) &:= \mathit{nodes}(S_1) \cup \mathit{nodes}(S_2) \\
\mathit{nodes}(\ell\texttt{:}\ S) &:= \mathit{nodes}(S) \\
\mathit{nodes}(\texttt{switch}\ S_1\ \texttt{case}\ S_2\ \cdots \\
\texttt{case}\ S_k) &:= \mathit{nodes}(S_1) \cup \cdots \cup \mathit{nodes}(S_k) \\
\mathit{nodes}(\texttt{try}\ S_1\ \texttt{catch}\ S_2\ \cdots \\
\texttt{catch}\ S_k\ \texttt{finally}\ S_{k+1}) &:= \mathit{nodes}(S_1) \cup \cdots \cup \mathit{nodes}(S_{k+1})
\end{aligned}$$

Edges. We are now ready to define the (directed) edges between nodes (or node indices, to be more precise) in the flow graph corresponding to a analysis statement:

$$\mathit{edges}\colon \mathbf{AStmt} \to 2^{\mathbf{AIndex}\times\mathbf{AIndex}}.$$

The following abbreviations come in handy to succinctly describe a set of edges: For sets $I, J \subseteq \mathbf{AIndex}$ of indices and an index $j \in \mathbf{AIndex}$,

$$\begin{aligned}
(I, j) &:= \begin{cases} \{(i, j) : i \in I\} & \text{, if } I \neq \emptyset; \\ \emptyset & \text{, otherwise;} \end{cases} \\
(I, J) &:= \bigcup_{j \in J} (I, j)\,.
\end{aligned}$$

We first give the definition of the *edges* function and discuss it below:

$$
\begin{aligned}
\mathit{edges}\Big(\texttt{[break]}^i\Big) &:= \{\mathit{breakEdge}(i)\}\\
\mathit{edges}\Big(\texttt{[break }\ell\texttt{]}^i\Big) &:= \{\mathit{labeledBreakEdge}(\ell, i)\}\\
\mathit{edges}\Big(\texttt{[continue]}^i\Big) &:= \{\mathit{continueEdge}(i)\}\\
\mathit{edges}\Big(\texttt{[continue }\ell\texttt{]}^i\Big) &:= \{\mathit{labeledContinueEdge}(\ell, i)\}\\
\mathit{edges}\Big(\texttt{[}S\texttt{]}^i\Big) &:= \emptyset \text{ , for any other elem. statement } S\\
\mathit{edges}\big(S_1\texttt{; }S_2\big) &:= \mathit{edges}\big(S_1\big) \cup \mathit{edges}\big(S_2\big)\\
&\quad\ \cup \big(\mathit{final}\big(S_1\big), \mathit{init}\big(S_2\big)\big)\\
\mathit{edges}\big(\texttt{if }S_1\texttt{ then }S_2\texttt{ else }S_3\big) &:= \mathit{edges}\big(S_1\big) \cup \mathit{edges}\big(S_2\big) \cup \mathit{edges}\big(S_3\big)\\
&\quad\ \cup \big(\mathit{final}\big(S_1\big), \mathit{init}\big(S_2\big)\big)\\
&\quad\ \cup \big(\mathit{final}\big(S_1\big), \mathit{init}\big(S_3\big)\big)\\
\mathit{edges}\big(\texttt{while }S_1\texttt{ do }S_2\big) &:= \mathit{edges}\big(S_1\big) \cup \mathit{edges}\big(S_2\big)\\
&\quad\ \cup \big(\mathit{final}\big(S_1\big), \mathit{init}\big(S_2\big)\big)\\
&\quad\ \cup \big(\mathit{final}\big(S_2\big), \mathit{init}\big(S_1\big)\big)\\
\mathit{edges}\big(\texttt{do }S_1\texttt{ while }S_2\big) &:= \mathit{edges}\big(S_1\big) \cup \mathit{edges}\big(S_2\big)\\
&\quad\ \cup \big(\mathit{final}\big(S_1\big), \mathit{init}\big(S_2\big)\big)\\
&\quad\ \cup \big(\mathit{final}\big(S_2\big), \mathit{init}\big(S_1\big)\big)\\
\mathit{edges}(\ell\texttt{: }S) &:= \mathit{edges}(S)\\
\mathit{edges}\big(\texttt{switch }S_1\texttt{ case }S_2\ \cdots &\\
\texttt{case }S_k\big) &:= \mathit{edges}\big(S_1\big) \cup \cdots \cup \mathit{edges}\big(S_k\big)\\
&\quad\ \cup \textstyle\bigcup_{i=2}^{k} \big(\mathit{final}\big(S_1\big), \mathit{init}\big(S_i\big)\big)\\
&\quad\ \cup \textstyle\bigcup_{i=2}^{k-1} \big(\mathit{final}\big(S_i\big), \mathit{init}\big(S_{i+1}\big)\big)\\
\mathit{edges}\big(\texttt{try }S_1\texttt{ catch }S_2\ \cdots &\\
\texttt{catch }S_k\texttt{ finally }S_{k+1}\big) &:= \mathit{edges}\big(S_1\big) \cup \cdots \cup \mathit{edges}\big(S_{k+1}\big)\\
&\quad\ \cup \textstyle\bigcup_{i=2}^{k} \big(\mathit{nodes}\big(S_1\big) \setminus \mathit{final}\big(S_1\big), \mathit{init}\big(S_i\big)\big)\\
&\quad\ \cup \big(\mathit{nodes}\big(S_1\big), \mathit{init}\big(S_{k+1}\big)\big)\\
&\quad\ \cup \big(\mathit{final}\big(S_{k+1}\big), \mathit{nodes}\big(S_1\big) \setminus \big\{\mathit{init}\big(S_1\big)\big\}\big)\\
&\quad\ \cup \textstyle\bigcup_{i=2}^{k} \big(\mathit{final}\big(S_i\big), \mathit{init}\big(S_{k+1}\big)\big)
\end{aligned}
\tag{4.2}
$$

`break` Statements

To compute the edge corresponding to a unlabeled or labeled `break` statement, the *break target statement* needs to be resolved first. The only possible breakable statements that can act as break targets are

- a labeled statement,
- `while` and `do` loops,
- a `switch` statement.

Once the break target statement has been determined, the break destination node then is the node to which the control flow transfers immediately after the break target statement terminates (if there is such a node).

The function

$$\mathit{breakEdge}\colon \mathbf{AIndex} \to \mathbf{AIndex} \times \mathbf{AIndex}$$

takes the index i of an unlabeled `break` statement and returns the edge from the `break` statement to the break destination node. Informally, starting at node i, it traverses the parent hierarchy in the abstract syntax tree until a breakable statement is encountered. This innermost breakable statement will be the break target statement.

Similarly to the case of unlabeled breaks, the function

$$\mathit{labeledBreakEdge}\colon \mathbf{AIndex} \times \mathbf{ALab} \to \mathbf{AIndex} \times \mathbf{AIndex}$$

takes the index i and the label ℓ of a labeled `break` statement and returns the edge from the `break` statement to the break destination node. Informally, starting at i, it traverses the parent hierarchy in the abstract syntax tree until a labeled statement with label ℓ is encountered. This labeled statement will be the break target.

The formal definitions of the functions *breakEdge* and *labeledBreakEdge* are omitted here.

In the implementation, so as to (i) make sure that every breakable statement has in fact a successor node acting as break destination node and (ii) simplify resolving this break destination node, any breakable statement S is actually regarded as a sequence statement S`; skip`, where an additionally inserted `skip` statement acts as break destination node of S.

Example 4.1 For the statement

```
while [move(x, y)]^0 do
    if [skip]^1 then
        [break]^2
    else
        [new(x)]^3;
[skip]^4;
[consume(y)]^5
```

we have *breakEdge*$(2) = (2,4)$, whereas the statement

```
while [skip]^0 do
    while [move(x, y)]^1 do
        if [skip]^2 then
            [break]^3
        else
            [new(x)]^4;
    [skip]^5;
[skip]^6;
```

yields *breakEdge*$(3) = (3,5)$. The planted `skip` statements are italicized. □

`continue` Statements

`continue` statements are handled similarly to `break` statements. First, the *continue target statement* needs to be resolved. As in Java, only loop statements (`while` or `do` loops) can act as continue targets. Once the continue target statement has been determined, the continue destination node then is the initial node of the loop condition statement, which corresponds to starting a new iteration.

The function

$$\mathit{continueEdge}\colon \textbf{AIndex} \to \textbf{AIndex} \times \textbf{AIndex}$$

returns the edge from an unlabeled `continue` statement with index i to the continue destination node. Informally, starting at node i, it traverses the parent hierarchy in the abstract syntax tree until a loop statement is encountered. This innermost loop statement will be the continue target statement.

Likewise, the function

$$\mathit{labeledContinueEdge}\colon \mathbf{AIndex} \times \mathbf{ALab} \to \mathbf{AIndex} \times \mathbf{AIndex}$$

returns the edge from an labeled `continue` statement with index i and label ℓ to the continue destination node. Informally, starting at node i, it traverses the parent hierarchy in the abstract syntax tree until a labeled statement with label ℓ is encountered. The immediate child of the labeled statement, which must be a loop statement, will be the continue target statement.

The formal definitions of the functions *continueEdge* and *labeledContinueEdge* are omitted here.

In contrast to `break` statments, there always exists a continue destination node in a loop and, what is more, can be resolved without having to add any extra helper statements.

`switch` Statement

As for the `switch` statement, the end of the statement representing the switch expression is linked to the beginnings of all case group statements. Furthermore, there is an edge from the end of each case group to the beginning of the next case group, which models “fall-throughs”. Note that if a case group ends in a `break` statement the “fall-through” edge is prevented, since a `break` statement has *no* final indices, as defined in (4.1).

`try/catch/finally` Statement

The edges induced by a `try/catch/finally` statement are slightly more involved than the ones for the other statements.

The event of a statement throwing an exception which is caught by one of the `catch` clauses is modeled by flow graph edge that leads from the node that was executed immediately *before* the current statement to the beginning of the `catch` clause in question. Note that we cannot simply add an edge from the current statement to the beginning of the `catch` clause, because this edge would mean a successful termination of the statement. Since there is no predecessor node for the initial node of the `try` clause, we add an artificial `skip` statement right at the beginning of the `try` clause. I.e., we treat the `try` clause as `skip;` S, where S is the original `try` clause. Since in the abstract analysis language we have no knowledge whatsoever about which statements may throw an exception, we conservatively assume that every statement may do so. Moreover, we neither are aware of what kind of exceptions are caught in which `catch` clauses. Consequently, we add a corresponding edge to the beginning of every `catch` clause. Thus, the first category of edges go from every statement in the `try` clause – except for the inserted `skip` statement – to the beginning of each `catch clause`, corresponding to the case an exception is thrown by a node in the `try` body and caught by a `catch` clause.

Moreover, there needs to be an edge to the beginning of the `finally` clause in case a statement terminates abruptly without being caught by a `catch` clause. It is important to note that the `finally` clause may not solely be "invoked" by an uncaught exception, but also by a `break` statement (which always terminates abruptly) enclosed in the `try` clause. In order to perform a possible transfer of control (e.g., the jump out of the loop for a `break` statement) *after* the execution of the `finally` clause, we additionally need a back edge leading from the end of the `finally` clause to the node which caused control to transfer to the `finally` clause. Consider Example 4.2 to illustrate the need for such a kind of edges. To approximate these contingencies conservatively, we add an edge from every node in the `try` clause to the beginning of the `finally clause` and an edge from the end of the `finally` clause back to every node in the `try` clause except the inserted `skip` statement.

Finally, we need to link the end of every `catch` clause to the beginning of the `finally` clause.

In summary, a `try/catch/finally` statement produces

- an edge from every non-final node in the `try` clause to the beginning of each `catch` clause;
- an edge from every node in the `try` clause to the beginning of the `finally` clause;
- a back edge from the end of the `finally` clause to every non-initial node in the `try` clause; and
- an edge from the end of every `catch` clause to the beginning of the `finally` clause;

The edges are described in this order in the definition of the *edges* function, (4.2).

Undoubtedly, this amounts to a very crude exception handling model and remains to be refined as future work, e.g., by taking into account what kind of exceptions are caught or may be thrown.

Example 4.2 The statement

```
while [skip]$^0$ do
    try
        [skip]$^1$;                        (this is the planted node)
        if [consume(x)]$^2$ then
            [break]$^3$
        else
            [move(x, y)]$^4$;
        [merge(x, y)]$^5$
    catch
        [new(x)]$^6$
    catch
        [new(y)]$^7$; [merge(x, y)]$^8$
    finally
        [new(x)]$^9$; [consume(x)]$^{10}$;
[move(x, y)]$^{11}$
```

leads to the following edges:

$\{(0,1)\}$	edge from the while condition to the body
$\cup\{(10,0)\}$	loop edge from the while body to the cond.
$\cup\{(3,11)\}$	break edge
$\cup\{(1,2),(2,3),(2,4),(4,5)\}$	edges within the try clause
$\cup\{(7,8)\}$	edges within the second catch clause
$\cup\{(9,10)\}$	edges within the finally clause
$\cup\{(1,6),(1,7),(2,6),(2,7),$ $(3,6),(3,7),(4,6),(4,7)\}$	edges from each non-final node in the try clause to the begin. of each catch clause
$\cup\{(1,9),(2,9),(3,9),(4,9),$ $(5,9)\}$	edges from each node in the try clause to the beginning of the finally clause
$\cup\{(6,9),(8,9)\}$	edges from the end of each catch clause to the beginning of the finally clause
$\cup\{(10,2),(10,3),(10,4),(10,5)\}$	back edges from the end of the finally clause to every non-initial node of the try clause □

Once the nodes and edges are ready, the flow graph of a program S is now defined by the tuple $(\mathit{nodes}(S), \mathit{edges}(S))$.

Example 4.3 Let S be the following program:

```
while [skip]$^{0}$ do
    [new(z)]$^{1}$;
    [merge(x, z)]$^{2}$;
    [move(y, z)]$^{3}$;
[skip]$^{4}$;
[consume(z)]$^{5}$;
if [skip]$^{6}$ then
    while [skip]$^{7}$ do
        [new(y)]$^{8}$;
        [merge(x, w)]$^{9}$;
    [skip]$^{10}$
else
    [move(z, x)]$^{11}$;
    [new(y)]$^{12}$
[move(x, w)]$^{13}$;
[skip]$^{14}$
```

We have $\mathit{init}(S) = 0$, $\mathit{final}(S) = \{14\}$ and a representation of the corresponding flow graph $(\mathit{nodes}(S), \mathit{edges}(S))$ is shown in Figure 4.1. □

4.3 Analysis Variables of Interest

Recall from Chapters 2 and 3 that the analysis is intra-procedural and thus, for each run of the analysis, we have a fixed method $m \in \mathbf{TMethod}$ in a class $C \in \mathbf{TClass}$ as well as a static declaration environment $\Gamma = \Delta(C, m)$ of the toy language.

The following sets define the *variables of interest* for the analysis:

$$\begin{aligned}
\mathbf{ALoc} &:= \{x \in \mathrm{mLoc}(C, m) : \ \Gamma_{\mathrm{m}}(x) = \mathit{rep}\} \\
\mathbf{AField} &:= \{f \in \mathrm{fields}(C) : \ \mathrm{fType}(C, f) = \mathit{rep}\langle \mathit{Cl} \rangle\} \\
\mathbf{AClust} &:= \mathrm{definedCls}(C)\,.
\end{aligned}$$

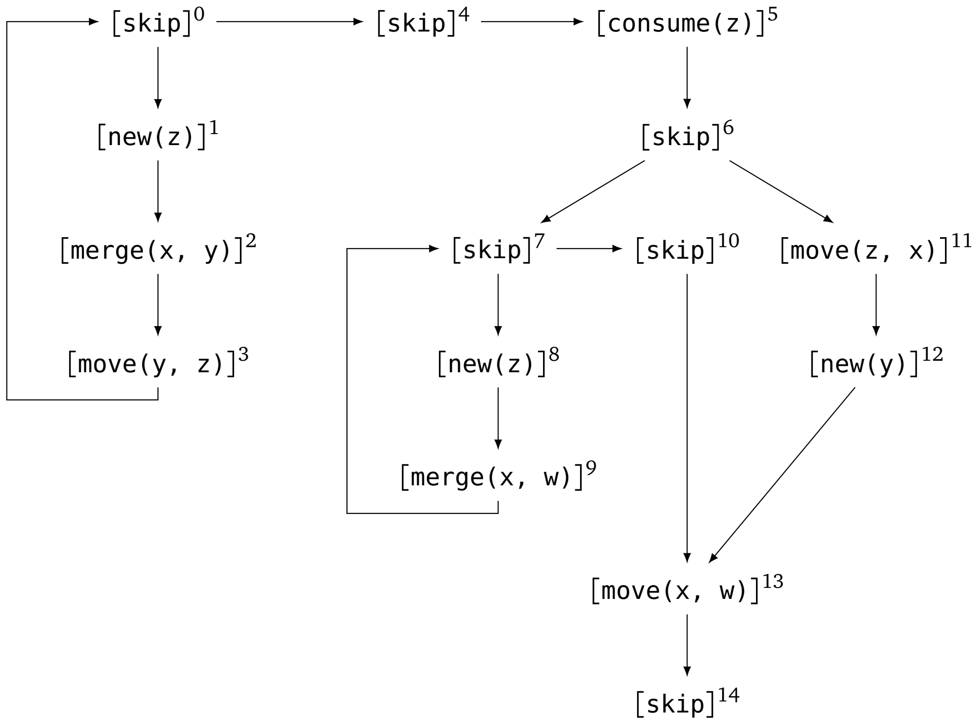


Figure 4.1: Example flow graph $(\mathit{nodes}(S), \mathit{edges}(S))$.

The set **ALoc** includes all local variables and parameters of a *rep* type. Likewise, **AField** is the set of all fields of type $rep\langle Cl\rangle$ visible in class C. Finally, the set **AClust** contains all clusters defined by `uniq` fields of class C or of a superclass. Note that **ALoc**, **AField** and **AClust** are pairwise disjoint (due to the assumption that all identifiers are globally unique) and finite. The analysis only needs to consider the variables of interest since all other variables may never get unusable.

Let us further define a function *cluster*: **AField** → **AClust** that returns the cluster associated with a given field:

$$\frac{\mathrm{fType}(C,f) = rep\langle Cl\rangle}{cluster(f) = Cl}$$

Letting S denote the program (the "top-level" analysis statement) to analyze, the input to the data flow analysis can be viewed as 4-tuple $(S, \mathbf{ALoc}, \mathbf{AField}, \mathbf{AClust})$.

4.4 Partition Sets and Analysis Definition

We may embark on formalizing and defining the analysis.

4.4.1 Set Partitions

Before we continue, let us first intersperse a general section about set partitions so as to fix some terminology and notation as well as to introduce a few concepts that are needed hereafter.

Partitions and Blocks. As usual, a *partition* of a set A is a set of pairwise disjoint and non-empty *blocks*[1] $A_i \subseteq A$ that together cover A, i.e.,

$$\begin{aligned}&\forall i: A_i \neq \emptyset,\\ &\forall i,j: i \neq j \Rightarrow A_i \cap A_j = \emptyset \text{ and}\\ &\textstyle\bigcup_i A_i = A.\end{aligned} \tag{4.3}$$

[1] Some authors also use the terms *partitioning* and *partition* for what we call a partition and a block, respectively. To avoid confusion, we shall use only the latter terminology throughout this document.

We use Part(A) to denote the set of all partitions of a set A. E.g.,

$$\begin{aligned}\mathrm{Part}\big(\{x,y,z\}\big) = \{\, &\{\{x\},\{y\},\{z\}\},\\ &\{\{x\},\{y,z\}\},\\ &\{\{y\},\{x,z\}\},\\ &\{\{z\},\{x,y\}\},\\ &\{\{x,y,z\}\}\,\}.\end{aligned}$$

We shall also use the following, more readable notation for partitions, using "|" to separate the blocks of a partition:

$$\begin{aligned}\mathrm{Part}\big(\{x,y,z\}\big) = \{\, &x \mid y \mid z,\\ &x \mid yz,\\ &y \mid xz,\\ &z \mid xy,\\ &xzy\,\}.\end{aligned}$$

For a set A, a partition $P \in \mathrm{Part}(A)$ and an element $x \in A$, we denote the block of P that contains x with $P^{[x]} \subseteq A$. It follows from the partition properties (4.3) that such a block exists and is unique. Note that for any two variables $x, y \in A$, $P^{[x]}$ and $P^{[y]}$ are either equal or disjoint.

The number of partitions for an n-element set is counted by the Bell number B_n, which has an exponential asymptotic limit [20].

In what follows, we shall use Greek uppercase letters to denote *sets* of partitions of a given set to clearly distinguish them from single partitions, which will be designated with usual uppercase letters, as in, e.g.,

$$\Phi = \{\underbrace{x \mid yz}_{=P}, \underbrace{xy \mid z}_{=Q}\} \in 2^{\mathrm{Part}(\{x,y,z\})}.$$

Exchanging Blocks. Let us further introduce two helper functions which allow as to modify partitions by adding and removing blocks: For a set A, a partition $P \in \mathrm{Part}(A)$ and a set $V \subseteq A$,

$$P + V := \begin{cases} P & \text{, if } V = \emptyset;\\ P \cup \{V\} & \text{, otherwise;}\end{cases}$$

$$P - V := \begin{cases} P & \text{, if } V = \emptyset; \\ P \setminus \{V\} & \text{, otherwise.} \end{cases}$$

It is immediate that these operations – viewed by themselves – may violate the partition properties (4.3).

New. For a set A, a partition $P \in \mathrm{Part}(A)$ and an element $x \in A$, the function

$$\mathit{new}\colon \mathrm{Part}(A) \times A \to \mathrm{Part}(A)$$

returns a partition in which x constitutes a singleton block:

$$\mathit{new}(P, x) := P - P^{[x]} + P^{[x]} \setminus \{x\} + \{x\}\,. \tag{4.4}$$

It is not hard to see that $\mathit{new}(P, x)$ indeed satisfies the partition properties (4.3).

Merge. For a set A, a partition $P \in \mathrm{Part}(A)$ and elements $x, y \in A$, the function

$$\mathit{merge}\colon \mathrm{Part}(A) \times A \times A \to \mathrm{Part}(A)$$

returns a partition in which the former blocks of x and y are unified:

$$\mathit{merge}\big(P, x, y\big) := P - P^{[x]} - P^{[y]} + P^{[x]} \cup P^{[y]}. \tag{4.5}$$

Note that $\mathit{merge}\big(P, x, y\big) = P$ if $P^{[x]} = P^{[y]}$. As for the *new* operation, we note that $\mathit{merge}\big(P, x, y\big)$ preserves the partition properties (4.3).

Move. Finally, for a set A, a partition $P \in \mathrm{Part}(A)$ and elements $x, y \in A$, the function

$$\mathit{move}\colon \mathrm{Part}(A) \times A \times A \to \mathrm{Part}(A)$$

returns a partition in which x is removed from its former block added to the block of y:

$$\mathit{move}\big(P, x, y\big) := \begin{cases} P & \text{, if } P^{[x]} = P^{[y]}; \\ P - P^{[x]} - P^{[y]} + P^{[x]} \setminus \{x\} + P^{[y]} \cup \{x\} & \text{, otherwise.} \end{cases} \tag{4.6}$$

The following simple lemma shows that the *move* operation can in fact be expressed by a *new* and a *merge* operation. This allows us to transfer properties which hold for the *new* and *merge* operations immediately to the *move* operation. E.g., since both *new* and *merge* preserve the partition properties (4.3), so does *move*.

Lemma 4.4 *For a set A, a partition $P \in \mathrm{Part}(A)$ and elements $x, y \in A$,*

$$move\big(P, x, y\big) = merge\big(new(P, x), x, y\big).$$

Proof. Let $P' := new(P, x)$. According to (4.4), this results in $P'^{[x]} = \{x\}$.

If $P^{[x]} \neq P^{[y]}$, we have $P'^{[y]} = P^{[y]}$ and thus

$$\begin{aligned} merge\big(new(P, x), x, y\big) &= merge\big(P', x, y\big) = \\ &= P' - P'^{[x]} - P'^{[y]} + P'^{[x]} \cup P'^{[y]} = \\ &= P - P^{[x]} + P^{[x]} \setminus \{x\} + \cancel{\{x\}} - \cancel{\{x\}} - P^{[y]} + \Big(\{x\} \cup P^{[y]}\Big) = \\ &= move\big(P, x, y\big) \end{aligned}$$

Otherwise, if $P^{[x]} = P^{[y]}$, it follows that $P'^{[y]} = P^{[x]} \setminus \{x\}$ and thus

$$\begin{aligned} merge\big(new(P, x), x, y\big) &= merge\big(P', x, y\big) = \\ &= P' - P'^{[x]} - P'^{[y]} + P'^{[x]} \cup P'^{[y]} = \\ &= P - \cancel{P^{[x]}} + \cancel{\Big(P^{[x]} \setminus \{x\}\Big)} + \cancel{\{x\}} - \cancel{\{x\}} - \cancel{\Big(P^{[x]} \setminus \{x\}\Big)} \\ &\qquad + \underbrace{\Big(\{x\} \cup \Big(P^{[x]} \setminus \{x\}\Big)\Big)}_{=\cancel{P^{[x]}}} = \\ &= P = move\big(P, x, y\big) \end{aligned}$$

■

Example 4.5 For $A := \{x, y, z\}$, we have

$$\begin{aligned} new\big(xy \mid z,\ x\big) &= x \mid y \mid z \\ new\big(x \mid yz,\ x\big) &= x \mid yz \\ merge\big(xy \mid z,\ x, y\big) &= xy \mid z \\ merge\big(x \mid yz,\ x, y\big) &= xyz \\ move\big(xy \mid z,\ x, y\big) &= x \mid yz \\ move\big(x \mid yz,\ x, y\big) &= xyz. \end{aligned}$$

□

4.4.2 Analysis Values

As described in Section 3.8.1, the analysis values represent the property we are interested in and together form the property space or the analysis universe.

In order to motivate the choice of the property space for the analysis, we have to anticipate the general concept of the analysis. Broadly speaking, the analysis computes two analysis values for each node in the flow graph: one that holds at the entry to the node and one that holds at the exit from the node. The entry and exit value of a specific node are related by the operation the node stands for (the *transition function*). Furthermore, the entry value of a specific node is obtained by joining the exit values of the node's predecessor nodes into a new analysis value.

In our case, so as to implement the transition functions described in Subsection 3.8.2, the analysis values should enable us to track

- which analysis variables point into a specific cluster,
- which analysis variables point into the same cluster as a specific analysis variable and
- which analysis variables are unusable.

To this end, we will partition the set of variables such that two variables point into the same cluster iff they are in the same block of the partition. To identify the clusters we include the cluster variables as markers in the variable set. Similarly, we use another special marker variable, `unusable`, to distinguish the unusable variables. In summary, we let

$$\mathbf{PVar} := \mathbf{ALoc} \cup \mathbf{AField} \cup \underbrace{\mathbf{AClust} \cup \{\texttt{unusable}\}}_{\text{marker variables}}$$

and use *partitions* of **PVar** as analysis values to begin with.

Example 4.6 For $\mathbf{ALoc} := \{v, w, x, y, z\}$, $\mathbf{AField} := \{f\}$ and $\mathbf{AClust} := \{C_f, Cl_{\text{this}}\}$, the partition

$$\left\{\left\{v, f, C_f\right\}, \{w, Cl_{\text{this}}\}, \{x, y\}, \{z, \texttt{unusable}\}\right\}$$

contains the information that, e.g.,

- v and f point into the same cluster C_f,
- neither v nor f are unusable,

- w points into the this-cluster Cl_{this},
- w is not unusable,
- x and y point into the same cluster different from C_f and Cl_{this},
- neither x nor y are unusable,
- z is unusable. □

Remains the question as to how to join the predecessor nodes' exit analysis values to compute the entry value for nodes where multiple paths of execution come together (i.e., nodes with multiple incoming edges). In order to achieve a high level of precision, one approach is not to work with single partitions as analysis values, but with *sets* of partitions. This allows us to unify the partitions at the exit of the predecessor nodes using the usual set union operator. Consequently, each path of execution reaching a node corresponds to a partition in the set of partitions associated with the node in question. Of course, this is no bijection, since the same partition may be obtained through multiple paths. However, every path is represented by a partition in the partition set.

Alternatively, we could think of a way to combine the predecessor partitions into one *single* partition. This compact representation would certainly be more efficient than the approach with sets of partitions, which may grow exponentially in the size of the program (consider Benchmark 1 in Section 4.11 for such an example). However, the inevitable loss in precision turned out to be too large for our needs. Consequently, this approach was abandoned. We will nevertheless present some more suitable alternatives to sets of partitions as analysis values in Sections 4.8 and 4.9.

In conclusion, we use sets of partitions of **PVar** as analysis values and our analysis property space is

$$2^{\mathrm{Part}(\mathbf{PVar})}.$$

4.4.3 Queries

To implement the *isUnusable* and *pointsInto* query functions introduced in Section 3.8 for partition sets, we will make use of the following basic predicate that checks whether two given variables $x, y \in \mathbf{PVar}$ are in the same block of partitions contained in a given partition set Φ:

$$\mathit{areInSameBlock}\colon 2^{\mathrm{Part}(A)} \times A \times A \to \mathbb{T}$$

(for an arbitrary set A). Its definition is

$$\mathit{areInSameBlock}(\Phi, x, y) := \begin{cases} 0 & \text{, if } \forall P \in \Phi \colon x \notin P^{[y]}; \\ 1 & \text{, if } \forall P \in \Phi \colon x \in P^{[y]}; \\ \tfrac{1}{2} & \text{, otherwise.} \end{cases}$$

In other words, the function returns 1 if x and y are in the same block in *all* partitions in the partition set and 0 if x and y are in different blocks in *all* partition sets. In any other case, i.e., if x and y are in the same block in some partition and in different blocks in another partition, the function returns ½.

Example 4.7 Let $A := \{w, x, y, z\}$ and $\Phi := \{wxy \mid z,\ wx \mid yz,\ wxy \mid z\}$. Then, we have

$$\begin{aligned} \mathit{areInSameBlock}(\Phi, w, x) &= 1 \\ \mathit{areInSameBlock}(\Phi, w, y) &= \tfrac{1}{2} \\ \mathit{areInSameBlock}(\Phi, w, z) &= 0. \end{aligned}$$

□

Since the analysis value obtained through each path of execution leading to the node at hand is represented by a partition in the partition set Φ, the function values have the desired interpretation described in Section 3.8, i.e.,

$$\begin{aligned} \mathit{areInSameBlock}(\Phi, x, y) = 0 \quad &\Rightarrow \quad \text{"No, } x \text{ and } y \text{ are definitely in different blocks,} \\ &\qquad\quad \text{in all possible executions of the program."} \\ \mathit{areInSameBlock}(\Phi, x, y) = 1 \quad &\Rightarrow \quad \text{"Yes, } x \text{ and } y \text{ are definitely in the same block,} \\ &\qquad\quad \text{in all possible executions of the program."} \\ \mathit{areInSameBlock}(\Phi, x, y) = \tfrac{1}{2} \quad &\Rightarrow \quad \text{"I don't know."} \end{aligned}$$

With the help of the *areInSameBlock* primitive we can now easily formulate the above mentioned query functions, which will merely be more readable wrapper functions:

$$\begin{aligned} \mathit{isUnusable}(\Phi, x) &:= \mathit{areInSameBlock}(\Phi, x, \texttt{unusable}) \\ \mathit{pointsInto}(\Phi, x, Cl) &:= \mathit{areInSameBlock}(\Phi, x, Cl), \end{aligned}$$

for $\Phi \in 2^{\mathrm{Part}(\mathbf{PVar})}$, $x \in \mathbf{ALoc} \cup \mathbf{AField}$ and $Cl \in \mathbf{AClust}$. As a concluding remark, we note that the domain of the second argument of the function *isUnusable* as given in Section 3.8 is in actual fact $\mathrm{mLoc}(C, m) \cup \mathrm{fields}(C)$ (for the method m in class C being analyzed), and not just the subset $\mathbf{ALoc} \cup \mathbf{AField}$. This is because the function is used in the type rules, where it needs to be defined for all locals and fields of the method, not only for the locals and fields which are of interest to the analysis. In order not to complicate the function definition, we just let the function implicitly return 0 for a second argument $x \in (\mathrm{mLoc}(C, m) \cup \mathrm{fields}(C)) \setminus (\mathbf{ALoc} \cup \mathbf{AField})$. A similar discussion goes for the *pointsInto* predicate.

4.4.4 Partition Invariants

During the analysis, we maintain the following invariant for a partition set Φ:

$$\forall Cl \in \mathbf{AClust}\colon\ \mathit{areInSameBlock}(\Phi, Cl, \texttt{unusable}) = 0,$$

meaning that a cluster variable is never unusable. This is motivated by statements which lead to a *Merge* operation of a cluster, such the cast statement

```
x = (rep[f] C) y;
```

where `f` is a field declared `uniq` and `y` is a variable of a *rep* type. According to the analysis value transition rules given in Section 3.9, the above cast statement amounts to a $\mathit{Merge}\left(\Phi, C_f, y\right)$ operation, where C_f is the cluster variable associated with the field `f`. In such a case, the cluster variable C_f should not be unusable.

Moreover, we disallow two different cluster variables marking the same variable. In other words,

$$\begin{aligned}&\forall x \in \mathbf{PVar}, C_1, C_2 \in \mathbf{AClust}\colon\\ &\qquad\big(\mathit{pointsInto}\big(\Phi, x, C_1\big) \neq 0 \wedge \mathit{pointsInto}\big(\Phi, x, C_2\big) \neq 0\\ &\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow C_1 = C_2\big)\end{aligned}$$

should hold as additional invariant. This corresponds to the limitation that we cannot have a reference from inside a cluster to a different cluster. This appears not to be a severe restriction for practical applications.

As a side-effect, both invariants also result in a simplification in the analysis implementation.

4.4.5 Transition Functions

The *transition function*

$$Trans_i : 2^{\text{Part}(\mathbf{PVar})} \longrightarrow 2^{\text{Part}(\mathbf{PVar})}$$

associated with a node i describes the effect (the semantics) of node i in terms of analysis values. We would now like to formulate the transition functions in conjunction with partition sets as analysis values to achieve the effects that are informally described in Subsection 3.8.2.

Merge and New as Building Blocks. The *Merge* and *New* transition functions are defined as application of the corresponding partition operations to each partition P in the input partition set Φ:

$$Merge(\Phi, x, y) := \{merge(P, x, y) : P \in \Phi\} \tag{4.7}$$

$$New(\Phi, x) := \{new(P, x) : P \in \Phi\}, \tag{4.8}$$

for $x, y \in \mathbf{PVar}$. We shall also use the following shortcut to apply the *New* operation repeatedly for a set $X = \{x_1, x_2, \dots, x_k\} \subseteq \mathbf{PVar}$ of variables:

$$New(\Phi, X) := New(\dots New(New(\Phi, x_1), x_2) \dots, x_k). \tag{4.9}$$

By virtue of our partition set up, the definition of the *Merge* operation amounts to the desired effect of taking into account all “aliases” of the arguments that point into the same respective cluster. Moreover, we verify that the *New* operation corresponds to the high-level description given in Subsection 3.8.2, too.

Note that the arguments of both functions are general analysis variables in **PVar**. In particular, they may also stand for cluster variables. This is because our formalization, using marker variables, allows us to make no difference between, e.g., $Merge(\Phi, x, y)$ and $Merge(\Phi, x, Cl)$ for $x, y \in \mathbf{ALoc}$ and $Cl \in \mathbf{AClust}$.

Move and Consume. All further transition functions can now be expressed using *Merge* and *New* as building blocks (possibly indirectly, as for the *ConsumeLocals* operation described below):

$$
\begin{aligned}
\mathit{Move}(\Phi, x, y) &:= \{\mathit{move}(P, x, y) : P \in \Phi\} = \\
&= \{\mathit{merge}(\mathit{new}(P, x), x, y) : P \in \Phi\} = \\
&= \mathit{Merge}(\mathit{New}(\Phi, x), x, y) && (4.10) \\
\mathit{Consume}(\Phi, x) &:= \mathit{New}(\mathit{Merge}(\Phi, x, \texttt{unusable}), && (4.11) \\
&\qquad \{\mathit{Cl} \in \mathbf{AClust}\colon \mathit{pointsInto}(\Phi, x, \mathit{Cl}) \neq 0\}). && (4.12)
\end{aligned}
$$

The derivation steps in (4.10) are immediate consequences of Lemma 4.4. Consuming a variable x is carried out by merging x with the marker variable `unusable`. This guaranties that all variables that point into the same cluster are as marked unusable as well. In addition, all cluster variables possibly marking a block of x in any partition $P \in \Phi$ are renewed. This is to ensure that the cluster variables that might possibly have become unusable as result of the merging are restored, as required by the first partition invariant defined in Subsection 4.4.4.

ConsumeLocals. To specify the *ConsumeLocals* operation, we need two more helper functions operating on partition sets. First, similarly to (4.9), we introduce an abbreviation to iteratively apply a *Consume* operation for a set $X = \{x_1, x_2, \ldots, x_k\} \subseteq \mathbf{PVar}$ of variables:

$$\mathit{Consume}(\Phi, X) := \mathit{Consume}(\ldots \mathit{Consume}(\mathit{Consume}(\Phi, x_1), x_2) \ldots, x_k).$$

Moreover, let us enumerate the field analysis variables by $\mathbf{AField} = \{f_1, \ldots, f_k\}$ (remember that **AField** is finite). The function $\mathit{RestoreFields}\colon 2^{\mathrm{Part}(\mathbf{PVar})} \to 2^{\mathrm{Part}(\mathbf{PVar})}$ is then defined by

$$
\begin{aligned}
&\mathit{RestoreFields}(P) := \\
&\quad \mathit{Move}(\ldots \mathit{Move}(\mathit{Move}(\Phi, f_1, \mathit{cluster}(f_1)), f_2, \mathit{cluster}(f_2)) \ldots, f_k, \mathit{cluster}(f_k)).
\end{aligned}
$$

This looks more complicated than it is – it simply moves each field variable f_i to its associated cluster $\mathit{cluster}(f_i)$. At last, we can give the definition of the *ConsumeLocals* operation:

$$\mathit{ConsumeLocals}(\Phi) := \mathit{RestoreFields}(\mathit{Consume}(\Phi, \mathbf{AClust} \setminus \{\mathit{Cl}_{\mathrm{this}}\})). \qquad (4.13)$$

This definition has the desired effect of consuming all variables that point into a named cluster (other than the this-cluster). The *RestoreFields* operation is used for reasons of simplicity and to avoid introducing a new basic function besides *Merge* and *New*. Note that the partition invariants are maintained, since the cluster variables are restored as part of the *Consume* operations.

Transition Function Definition. In conclusion, the transition function $\mathit{Trans}_i(\Phi)$ for a node i is trivially defined according to the node's type:

$$\begin{aligned}
[\texttt{merge(}x\texttt{, }y\texttt{)}]^i \quad &\Rightarrow \quad \mathit{Trans}_i(\Phi) := \mathit{Merge}\big(\Phi, x, y\big) \\
[\texttt{new(}x\texttt{)}]^i \quad &\Rightarrow \quad \mathit{Trans}_i(\Phi) := \mathit{New}(\Phi, x) \\
[\texttt{move(}x\texttt{, }y\texttt{)}]^i \quad &\Rightarrow \quad \mathit{Trans}_i(\Phi) := \mathit{Move}\big(\Phi, x, y\big) \\
[\texttt{consume(}x\texttt{)}]^i \quad &\Rightarrow \quad \mathit{Trans}_i(\Phi) := \mathit{Consume}(\Phi, x) \\
[\texttt{consumeLocals}]^i \quad &\Rightarrow \quad \mathit{Trans}_i(\Phi) := \mathit{ConsumeLocals}(\Phi)\,.
\end{aligned}$$

For the remaining types of nodes, the transition function is defined to be the identity on partition sets:

$$\begin{aligned}
&[\texttt{skip}]^i, [\texttt{break}]^i, [\texttt{break }\ell]^i, [\texttt{continue}]^i, [\texttt{continue }\ell]^i, [\texttt{exit}]^i \\
&\qquad \Rightarrow \quad \mathit{Trans}_i(\Phi) := \Phi.
\end{aligned}$$

4.4.6 Analysis Definition

Having fixed the analysis property space and the transition functions, we will now formally define the analysis in terms of data flow equations to be solved, along the lines already suggested in Subsection 4.4.2.

For an analysis program S, let us denote the analysis values computed by the analysis by the following functions:

$$\Phi_{\text{entry}}, \Phi_{\text{exit}} \colon \mathit{nodes}(S) \to 2^{\text{Part}(\mathbf{PVar})}.$$

That is, Φ_{entry} maps a node to the analysis value at its entry, and Φ_{exit} maps a node to the an the analysis value at its exit, respectively.

$$\Phi_{\text{entry}}(i) = \bigcup_{\substack{(i',i)\in \\ \mathit{edges}(S)}} \Phi_{\text{exit}}\big(i'\big) \cup \iota^i_{\mathit{init}(S)} \tag{4.14}$$

$$\Phi_{\text{exit}}(i) = \mathit{Trans}_i\Big(\Phi_{\text{entry}}(i)\Big) \tag{4.15}$$

where

$$\iota_j^i := \begin{cases} \mathit{RestoreFields}(\{\{\{x\} : x \in \mathbf{PVar}\}\}) & \text{, if } i = j; \\ \emptyset & \text{, otherwise.} \end{cases} \tag{4.16a}$$

Equation (4.14) means that the entry value of a node is computed by joining the exit values of all predecessor nodes and joining the initial value for the initial node of the program. Equation (4.15) describes the effect of a single node, linking the entry value to the exit value with the node's transition function.

The initial value represents the information available at the start of the program. In our case, (4.16a) defines it to be a partition set containing a single partition in which every element is in its own block and, by virtue of the *RestoreFields* operation, every field is in the same block as its associated cluster.

Example 4.8 Let **ALoc** $:= \{x, y, z\}$, **AField** $:= \{f, g, h\}$ and **AClust** $:= \{C_f, Cl_{\text{this}}\}$. Moreover, let $\mathit{cluster}(f) = \mathit{cluster}(g) = C_f$ and $\mathit{cluster}(g) = Cl_{\text{this}}$. Then, the initial analysis value is

$$\{x \mid y \mid z \mid f\,g\,C_f \mid h\,Cl_{\text{this}}\}\,. \qquad \square$$

We note that we are interested in the *least solution* (the *least fixed point*) to the data flow equations, the intuition being that we do not want to have unnecessarily big partition sets. The analysis is a *forward analysis* in the sense that it determines the values of successor nodes by joining the values of predecessor nodes. Furthermore, the analysis is *flow-sensitive*, since it calculates one analysis value for each node in the flow graph, as opposed to ignoring control flow information and computing a solution for the whole method/program. Finally, as already mentioned before, the analysis is intra-procedural.

4.4.7 Preconditions and Early Type Checking

So as to maintain the partition invariants described in Subsection 4.4.4, we have to check some special *preconditions* before the transition functions of certain node types may be applied during the analysis. If the required preconditions are not satisfied, the analysis immediately terminates with an appropriate error message.

As an optimization (and in order to make sure that propagated errors do not lead to unsound results), we also check already *during* the analysis that no unusable variables are read. This can be seen as an early type checking and has the advantage that some type errors may be recognized early during the analysis and the analysis does not need to to be run for the entire program in case such an error is encountered. It should be noted, however, that a special kind of analysis solving method is needed in conjunction with such an anticipated type checking (cf. Subsection 4.7.1 for more detail on this).

The compilation coming next describes the checks that are performed as preconditions before the transition functions of certain node types is executed.

Merge. For a $[\texttt{merge(}x\texttt{, }y\texttt{)}]^i$ node, the following two preconditions are checked before applying the transition function:

$$\begin{aligned}&\forall C_1, C_2 \in \mathbf{AClust}: \\ &\qquad \big(\mathit{pointsInto}\big(\Phi, x, C_1\big) \neq 0 \land \mathit{pointsInto}\big(\Phi, y, C_2\big) \neq 0 \\ &\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow C_1 = C_2\big)\end{aligned}$$

and

$$\mathit{isUnusable}(\Phi, x) = 0 \land \mathit{isUnusable}\big(\Phi, y\big) = 0,$$

where $\Phi := \Phi_{\text{entry}}(i)$. The first precondition assures that no different clusters are merged, hence enforcing the second partition invariant. Further, the second precondition checks that neither of the involved variables is unusable. As mentioned above, this is to check that no unusable variable is read as part of the *Merge* operation and corresponds to an early type checking. Note that a *Merge* operation can indeed only be generated by a statement in the toy language that requires reading both arguments of the *Merge* operation.

Move. A $[\texttt{move(}x\texttt{, }y\texttt{)}]^i$ has the following precondition:

$$\mathit{isUnusable}\big(\Phi, y\big) = 0,$$

where $\Phi := \Phi_{\text{entry}}(i)$. Similarly to the *Merge* operation, this anticipatorily checks that the second argument is not unusable when read. Note that a *Move* operation corresponds to reading the *second* argument and assigning it to the first argument in the source code (turn back to the analysis value transition rules in Section 3.9 the to verify this).

Consume. A `[consume(`x`)]`^{i} node takes the following two preconditions:

$$\begin{gathered} \mathit{pointsInto}(\Phi, x, \mathit{Cl}_{\text{this}}) = 0 \text{ and} \\ \mathit{isUnusable}(\Phi, x) = 0, \end{gathered}$$

where $\Phi := \Phi_{\text{entry}}(i)$. The first precondition prevents consuming the this-cluster $\mathit{Cl}_{\text{this}}$, as described in Subsection 3.8.2 (there referred to as "implicit precondition"). The second precondition is another instance of an early type checking strategy, ensuring that the variable to be consumed is not unusable.

4.5 Abstract Analysis Definition

In view of general analysis solver algorithms and to facilitate the introduction of alternative analysis values, we would now like to generalize the analysis and define abstract requirements the property space and the transition functions have to fulfill.

4.5.1 Abstract Analysis Values

The following definition describes an abstract analysis property space.

Definition 4.9 A *bounded join-semilattice* is an algebraic structure $(\mathcal{L}, \sqcup, \bot)$ satisfying the following conditions:

- $\mathcal{L}$ is a non-empty set.
- $\sqcup\colon \mathcal{L} \times \mathcal{L} \to \mathcal{L}$ is a binary operator (the *join operator*) which is
  - idempotent: $\forall L \in \mathcal{L}\colon L \sqcup L = L$,
  - commutative: $\forall L_1, L_2 \in \mathcal{L}\colon L_1 \sqcup L_2 = L_2 \sqcup L_1$,
  - associative: $\forall L_1, L_2, L_3 \in \mathcal{L}\colon \big(L_1 \sqcup L_2\big) \sqcup L_3 = L_1 \sqcup \big(L_2 \sqcup L_3\big)$.
- $\bot \in \mathcal{L}$ is a distinguished element (the *least element*) such that

$$\forall L \in \mathcal{L}\colon L \sqcup \bot = L. \qquad \square$$

In summary, a join-semilattice fixes the property space $\mathcal{L}$ and defines the least lattice value $\perp$, which will be used in the solver algorithm described later. Moreover, the semilattice's join operator $\sqcup$ defines how analysis values are combined in control flow joins. Intuitively, the commutativity and associativity of the operation mean that it does not matter in which order we combine information from different paths.

For a join-semilattice $(\mathcal{L}, \sqcup, \perp)$ we can establish a partial ordering $\sqsubseteq$ on $\mathcal{L}$ by setting, for $L_1, L_2 \in \mathcal{L}$,

$$L_1 \sqsubseteq L_2 \quad :\Longleftrightarrow \quad L_2 \sqcup L_2 = L_2.$$

It is easy to verify that this defines indeed a partial ordering on $\mathcal{L}$. When we discuss a join-semilattice in the following, we will take the liberty of referring to this induced partial ordering without explicitly introducing it.

In addition to be a bounded join-semilattice, the abstract analysis values need to satisfy the so-called Ascending Chain condition.

Definition 4.10 A partially ordered set $(\mathcal{L}, \sqsubseteq)$ satisfies the *Ascending Chain condition* iff each ascending chain $L_1 \sqsubseteq L_2 \sqsubseteq L_3 \cdots$ eventually stabilizes. I.e., $\exists\, k \colon L_k = L_{k+1}$. □

This simply means that there are no infinite ascending chains of elements of the join-semilattice.

4.5.2 Abstract Transition Functions

The abstract transition function

$$\mathit{Trans}_i \colon \mathcal{L} \to \mathcal{L}$$

associated with a node i describes the effect (the semantics) of node i in terms of analysis values. It is natural to demand that each transition function F_i is monotone, i.e.,

$$\forall L_1, L_2 \in \mathcal{L}\colon \quad L_1 \sqsubseteq L_2 \quad \Rightarrow \quad \mathit{Trans}_i(L_1) \sqsubseteq \mathit{Trans}_i(L_2)\,.$$

Intuitively, this says that an increase in our knowledge about the input must give rise to an increase in out knowledge about the output (or at least that we know the same as before).

4.5.3 Abstract Analysis Definition

We can now introduce the well-established notion of a *Monotone Framework* [18, 14].

Definition 4.11 Given a program S to analyze, a *Monotone Framework* consists of

- a join-semilattice $(\mathcal{L}, \sqcup, \bot)$ satisfying the Ascending Chain condition[2];
- a monotone transition function Trans_i for each $i \in \mathit{nodes}(S)$; □

So as to define the analysis as *instance* of a Monotone Framework, we additionally require a distinguished value $\iota \in \mathcal{L}$, the *initial value*, defining the value the initial node of the program S starts out with. Note that the initial value is not necessarily equal to the least value.

Having all ingredients together, let us denote the analysis values computed by the analysis by the functions

$$L_{\text{entry}}, L_{\text{exit}} \colon \mathit{nodes}(S) \to \mathcal{L}$$

which mapping a node to the analysis value at its entry and exit, respectively. The Monotone Framework instance then gives rise to a set of equations whose form should be familiar from Subsection 4.4.6:

$$L_{\text{entry}}(i) = \bigsqcup_{\substack{(i',i) \in \\ \mathit{edges}(S)}} L_{\text{exit}}\big(i'\big) \sqcup \iota^{i}_{\mathit{init}(S)} \tag{4.17}$$

$$L_{\text{exit}}(i) = \mathit{Trans}_i\Big(L_{\text{entry}}(i)\Big) \tag{4.18}$$

where

$$\iota^{i}_{j} := \begin{cases} \iota & \text{, if } i = j; \\ \bot & \text{, otherwise.} \end{cases}$$

[2] As an aside, requiring that $\mathcal{L}$ be a complete lattice satisfying the Ascending Chain condition is equivalent to demanding that $\mathcal{L}$ be a bounded join-semilattice satisfying the Ascending Chain condition [18]. We use the latter definition as it fits better with the adoption of alias matrices as analysis values, as will be seen later.

4.6 Partition Sets Revisited

Let us now return to partition sets and quickly verify that the analysis formalization indeed gives rise to a Monotone Framework to make sure the above described solver algorithms terminate with the correct result.

Lemma 4.12 *The partition set analysis formalization constitutes a Monotone Framework.*

Proof. We have $\mathcal{P} := 2^{\mathrm{Part}(\mathbf{PVar})}$ as property space with the set union operator $\cup$ as join operator and the empty set $\emptyset$ in the role of the least value. $\mathcal{P}$ being a power set, $(\mathcal{P}, \cup, \emptyset)$ is clearly a bounded join-semilattice. Moreover, the satisfaction of the Ascending Chain condition follows because Part(**PVar**) is finite.

What is left to confirm is the monotonicity of the transition functions. According to Subsection 4.4.5, all transition functions can be reduced to a sequence of *Merge* and *New* operations. Hence, it suffices to demonstrate that the latter two functions are monotone. For this purpose, let $\Phi, \Psi \in \mathcal{P}$ and $\Phi \subseteq \Psi$. Enumerating the elements in both partition sets, we arrive at the following picture (note that both Φ and Ψ are finite):

$$\Phi = \{P_1, \ldots, P_k\} \subseteq \{P_1, \ldots, P_k, Q_1, \ldots, Q_l\} = \Psi.$$

Since, by (4.7) and (4.8), the *Merge* and *New* operations are defined as an element-wise application of *merge* or *new*, respectively, the monotonicity follows immediately:

$$F(\Phi) = \{f(P_1), \ldots, f(P_k)\} \subseteq \{f(P_1), \ldots, f(P_k), f(Q_1), \ldots, f(Q_l)\} = F(\Psi).$$

where the functions F and f stand for *Merge* and *merge* or *New* and *new*, respectively. ■

4.7 Analysis Solver Algorithms

To solve the data flow equations (4.17) and (4.18) we focus on iterative algorithms [18, 14]. The underlying principle is simple: Starting with an appropriate initialization, the analysis values are iteratively updated for each node by (i) joining the analysis values from predecessor nodes (Equation (4.17)) and (ii) applying transition functions until all values are stable (Equation (4.18)).

So as to organize this iteration in a less chaotic way, we employ a *worklist* which contains edges (as opposed to nodes). The presence of an edge (i, i') in the worklist indicates that the analysis value has changed at the entry to (and therefore also at the exit of) node i and so must be recomputed at the entry to the successor node i'. Keeping edges instead of nodes in the worklist has the advantage that the entry analysis value of a node can be selectively updated by joining the exit values from single predecessor nodes, as opposed to having to join the exit values from *all* predecessor nodes for each update.

To enable different worklist organizations, let us introduce an abstraction of a worklist providing the following iterator-like interface:

- **operation** initialize(S) – initialize the worklist using a given program S
- **operation** isEmpty() – return true iff there are no more edges in the worklist, meaning that the iteration is done
- **operation** next() – return the next edge to be treated
- **operation** update(e) – update the worklist to record that edge e was affected by the last iteration

Algorithm 4.1, adapted from [18], outlines a basic worklist solver. Making use of the monotonicity property and the Ascending Chain condition guaranteed by the Monotone Framework, it computes the desired least solution (the least fixed point) to the data flow equations. For a complete proof of this fact, the reader is kindly referred to [18].

Time Complexity. Regardless of the concrete worklist implementation, the worklist algorithm has a worst-case time complexity of $O(|edges(S)| \cdot h \cdot a)$, where S is the program to analyze, h the length of the longest ascending chain in the join-semilattice $\mathcal{L}$ and a the cost needed to perform a basic operation (an application of a transition function or an application of a join operation) during the iteration. Consider [18] for more detail on this. Note that for programs written using standard constructs (including `goto` statements) the number of edges in the flow graph is $O(\text{number of nodes in the flow graph})$ [14].

4.7.1 Standard Worklist

A straight-forward implementaiton of a worklist is given in Algorithm 4.2. It maintains a usual linked list and simply delegates all calls to the list. For the initialization, all flow graph edges are added to the list. It does not matter whether the elements are added in a LIFO or FIFO manner.

Algorithm 4.1: Worklist Solver

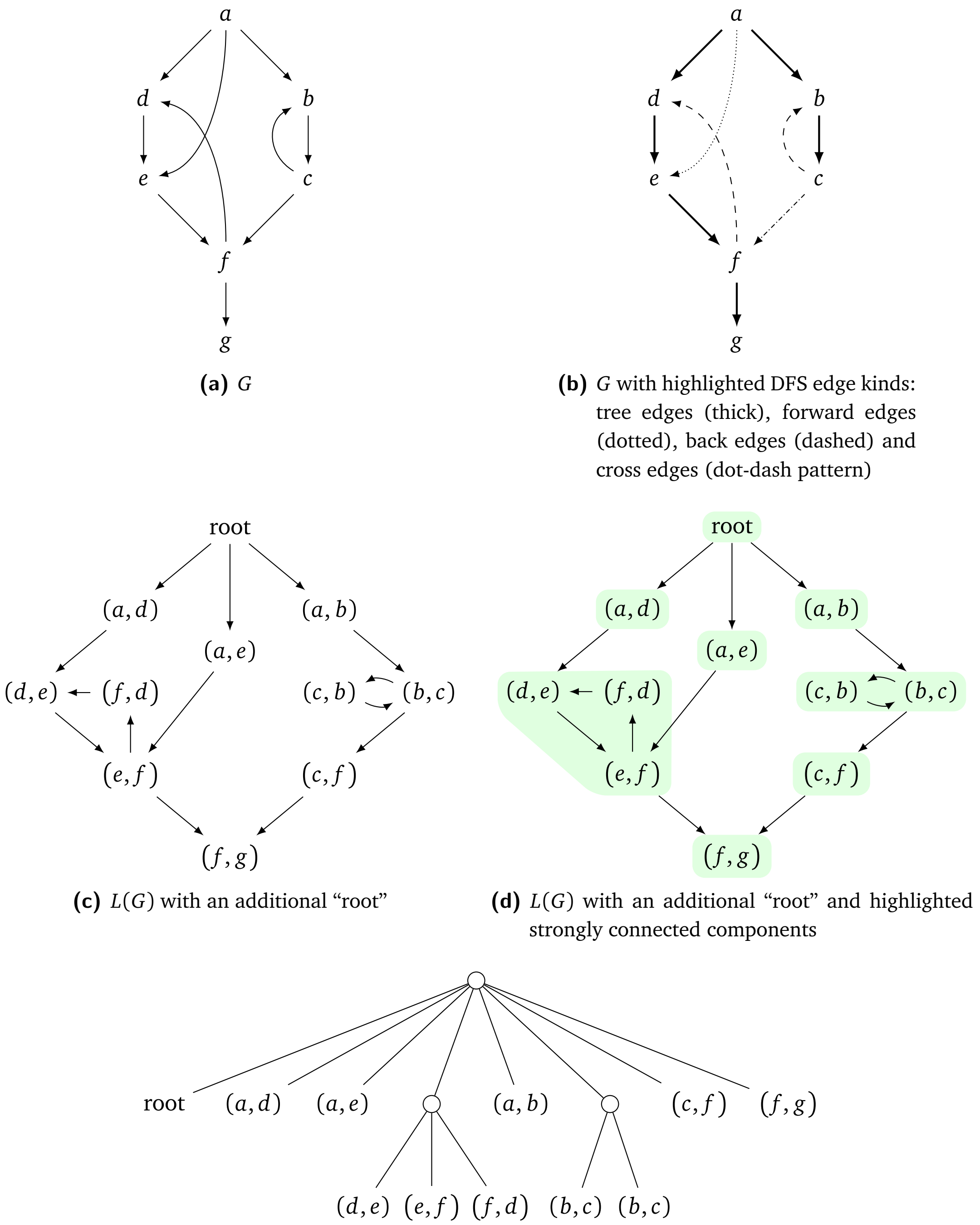
```
Input : a program $S$ and analysis equations defined by an instance of a Monotone
        Framework
Output: the least analysis solution in form of $L_{\text{entry}}(i)$ for each $i \in nodes(S)$

for each $i \in nodes(S) \setminus \{init(S)\}$
    $L_{\text{entry}}(i) := \bot$            // initialize all nodes except the initial one to the least value
$L_{\text{entry}}(init(S)) := \iota$                    // initialize the initial node to the initial value
$W$.initialize($S$)                      // initialize the worklist and begin the iteration

while $\neg W$.isEmpty()
    $(i, i') := W$.next()
    $new := Trans_i\big(L_{\text{entry}}(i)\big)$                               // $new = L_{exit}(i)$
    if $new \not\sqsubseteq L_{\text{entry}}(i')$                 // $new \sqsupset L_{entry}(i') \Leftrightarrow L_{entry}(i')$ will change
        $L_{\text{entry}}(i') := L_{\text{entry}}(i') \sqcup new$                    // update the analysis value
        for each $(i', i'') \in edges(S)$
            $W$.update$(i', i'')$              // update the worklist with each outgoing edge
```

However, this simple approach bares a fatal flaw in association with the preconditions and early type checks that no unusable variable is read that are performed before applying certain transition functions *during* the analysis (cf. Subsection 4.4.5) and the fact that the edges are in no particular order in the worklist: Consider the flow graph G from Figure 4.2(a). Suppose that node f is a `[consume(x)]`^{i} node and we retrieve edge (f, g) from the worklist. This means that, as part of the preconditions for a `consume` node, a check is performed whether the variable x is unusable. If $isUnusable\big(L_{\text{entry}}(f), x\big) = 1$, then the solver emits a precise error message and the analysis terminates. However, it may be that one of the incoming edges (c, f) and (e, f) has not yet been retrieved from the worklist, but would have caused the value at the entry to node f to become imprecise, i.e., $isUnusable\big(L_{\text{entry}}(f), x\big) = ½$. This means that the solver would report a spurious precise error.

To solve this problem we have to organize the edges in the worklist in some kind of order that makes sure that, coming back to our example, the edges (c, f) and (e, f) are retrieved *before* the edge (f, g).



(a) G

(b) G with highlighted DFS edge kinds: tree edges (thick), forward edges (dotted), back edges (dashed) and cross edges (dot-dash pattern)

(c) $L(G)$ with an additional "root"

(d) $L(G)$ with an additional "root" and highlighted strongly connected components

(e) Loop tree of $L(G)$ with an additional "root"

Figure 4.2: Example graph, line graph and loop tree.

Algorithm 4.2: Standard Worklist

```
**State variables**: *list*

**operation** initialize($S$)
    *list* := emtpy list
    **for each** $e \in edges(S)$
        add $e$ to *list*

**operation** isEmpty()
    **return true** iff *list* is empty

**operation** next()
    $e$ := remove next edge from *list*
    **return** $e$

**operation** update($e$)
    add $e$ to *list*
```

4.7.2 Reverse Postorder Worklist

Before we try to tackle the problem described in the previous subsection, let us first briefly review two common concepts.

DFS Graph Traversals. A depth-first search (DFS) traversal visits all nodes of a graph by recursively visiting all not yet visited successor nodes. Thereby, it constructs a spanning tree (or forest, if the graph is not connected) consisting of those edges that led to a successor that had not yet been visited. We can categorize the edges in the original graph as follows:

- *Tree edges* are present in the DFS spanning tree (or forest). As explained above, this signifies that the destination node of the graph was first visited via this edge.

- *Forward edges* go from a node to a proper descendant in the tree, but are no tree edges.

- *Back edges* go from descendants to ancestors in the tree.

- *Cross edges* go between nodes that are unrelated by the ancestor and descendant relations.

Consider Figure 4.2(b) for an example DFS run using a "leftmost successor first" strategy to visit the successor nodes.

While traversing the tree, the encountered nodes can be numbered (or printed) either in *preorder* or in *postorder*. A preorder numbering means that a node is numbered (or printed) as soon as it is visited, before the successor nodes are recursed into. In contrast, a postorder numbering numbers (or prints) a node only after all of its successor nodes have been recursed into. Finally, a reverse postorder numbering is, expectedly, the reverse of a postorder numbering.

One important property of a reverse postorder numbering is the fact that it topologically sorts the nodes by all edges except back edges. In particular, it also takes cross edges into account. The following itemization displays possible node orderings of the graph G given in Figure 4.2(a), again using a "leftmost successor first" strategy to visit the successor nodes. This makes plain that the preorder and the breadth-first search (BFS) discipline do not necessarily sort the nodes according to cross edges, as the end nodes c and f of the cross edge (c, f) are in the wrong order:

- DFS postorder : g, f, e, d, c, b, a
- DFS reverse postorder : a, b, c, d, e, f, g
- DFS preorder : a, d, e, f, g, b, c
- BFS : a, d, e, b, f, c, g

Line Graphs. Given a (simple) directed graph $G = (V(G), E(G))$, the *line graph* $L(G)$ of G is the graph whose vertices are the edges of G. There is an edge between two nodes in $L(G)$ iff the corresponding edges in G share an endpoint:

$$\begin{aligned} V(L(G)) &:= E(G) \\ E(L(G)) &:= \{((u,v),(u',v')) \in E(G) \times E(G) \colon v = u'\} . \end{aligned}$$

Line graphs are often used to translate properties about edges in the original graph into properties about vertices in the line graph. When dealing with a flow graph G that corresponds to a program S in the analysis language, G has a single initial node *init*(S) from which there is a path to every other node in the graph (assuming there are no unreachable statements). However, the line graph $L(G)$ of G does not have this property if $\mathit{init}(S)$ has multiple successors. To overcome this, we can simply add an artificial "root" node to $L(G)$ and connect it to the nodes in $L(G)$ that correspond to the outgoing edges of $\mathit{init}(S)$ in G. Figure 4.2(c) shows an example of a line graph with such an added "root" node.

Coming back to the problem of a better organization of the worklist, we observe that all we need to do is to make sure that for every node, all incoming edges – excluding back edges – are retrieved prior to the outgoing edges. But this is exactly the property of a reverse postorder numbering. Hence, we can compute the line graph of the original flow graph, keep the edges sorted in reverse postorder and retrieve them in this order from the worklist. Note that an edge in a line graph captures the same idea as an edge in the original flow graph, namely that updating the source node influences the destination node.

We can achieve the same effect if we number the nodes in reverse postorder and impose a total order on edges by comparing them alphabetically by the reverse postorder numbers of the source and destination node of the edge, in this order. This way, we can avoid using a line graph.

Example 4.13 For the example graph G of Figure 4.2(a), we have already seen a valid reverse postorder of the nodes: a, b, c, d, e, f, g. Ordering the edges of G alphabetically by the reverse postorder numbers of the source and destination node then yields:

$$(a,b),\ (a,d),\ (a,e),\ (b,c),\ (c,b),\ (c,f),\ (d,e),\ (e,f),\ (f,g)\,. \qquad \square$$

In consequence, the implementation of a reverse postorder worklist looks just like the standard worklist of Algorithm 4.2, except that the edges in the aggregated list are retrieved in reverse postorder.

4.7.3 Strongly Connected Components Worklist

An even more advanced organization of the worklist is based on the structure of the *strongly connected components* of a graph. Recall that a strongly connected component (SCC or strong component in short) of a directed graph G is a maximal subgraph $S \subseteq G$ such that every node in S is reachable from every other node in S. Being the equivalence classes of an equivalence relation, the SCCs partition the nodes in the graph. Further, viewing the SCCs as supernodes produces a DAG and hence, the SCCs are topologically sortable.

The basic idea is now to organize the worklist in such a way that we iterate over loops until they are stabilized, stabilize inner loops before their outer loops and visit the nodes in reverse postorder within each loop [18, 14, 15]. To accomplish this, we employ the same techniques as described in [15], with the difference that we want the worklist to return edges instead of nodes. This can be achieved by simply applying the procedures to the line graph (enriched with an artificial "root" node) of the flow graph instead of using the flow graph directly.

The entry node to a loop is referred to as the *loop head*. A loop head in the line graph of a flow graph corresponds to an edge whose source node is a loop head in the flow graph, i.e., an edge leading into the loop in the flow graph. Such a kind of edge will be referred to as a *loop head edge*.

Example 4.14 Consider once more Figure 4.2. The nodes d and b are loop heads in the graph G. The corresponding line graph $L(G)$ has the nodes (d, e) and (b, c) as loop heads. Note that the latter edges indeed correspond to an edge leading into the loop in G. □

When the worklist is about to return such a loop head edge, we may either (i) return the edge – corresponding to another iteration over the loop body – or, (ii) return the next edge *after* the loop, which corresponds to jumping over the nodes of the loop body. To capture such an update sequence, including the decision points at edges leading into a loop, the so-called *loop tree* is calculated out of the SCCs of the line graph in a preprocessing step. The loop tree is a tree with ordered children. Inner nodes represent loops (except for the root node of the whole tree), while leaf nodes represent nodes of the original graph, in our case, the line graph of the flow graph.

The construction of a loop tree given an arbitrary graph G is shown in the recursive Algorithm 4.4, adapted from [15]. The algorithm terminates since removing all incoming edges from a loop head of an SCC lets this component fall apart into at least two subcomponents. In the implementation, the strongly connected components are computed using Tarjan's algorithm [21], which conveniently delivers them already in (reverse) topological order. Figure 4.2(d) highlights the SCCs of the line graph $L(G)$ (with an additional "root" node) of the example graph G, and Figure 4.2(e) displays the corresponding loop tree. Note that the first (leftmost) child of a sub loop tree is always a leaf and – except for the additional "root" node – corresponds to a loop head in the line graph, or to a loop head edge leading into a loop in the original graph.

The constructed loop tree then defines the following update sequence: the leaf nodes are retrieved from left to right in tree order, jumping back to the first (leftmost) child from the last (rightmost) node of a subtree. At loop heads (decision points), a check is performed whether the loop is stable, indicating a fixed point for the loop. If so, the ordinary evaluation sequence from left to right is broken and the iteration continues with the first node after the loop.

Algorithm 4.3, adapted from [15], sketches the corresponding worklist iterator. The loop tree of the line graph is created in the initialization method. During the iteration, the helper function checkFirstChildStable(e) takes an edge e which must be the first child of a subtree, meaning that it is a loop head edge leading into a loop. The function then checks if the loop head (the source node of e) has changed since the last time it was encountered. If so, then the loop is not stable and the worklist returns the edge e, meaning that one more loop iteration is performed. Otherwise, the loop is stable and the next node of e's parent is calculated, which corresponds to jumping over the loop body.

In summary, the described update scheme has the benefit of a reduced number of computations required to stabilize each loop. In the practical examples that have been analyzed (including the benchmarks from Section 4.11), however, no overwhelming difference could be observed between the efficiency of an SCC worklist and a simpler reverse postorder worklist from the previous section; the performances were similar. Nonetheless, the analysis solver in the current implementation uses an SCC worklist.

4.8 Alias Matrices

Instead of tweaking the analysis solver algorithm for a better performance, we shall now embark on some strategies to replace the bulky partition sets as analysis values with more compact representations which, unlike partition sets, may not grow exponentially in size. As it turns out, however, this results in a possible loss in precision.

Given a set of partitions Φ, all analysis queries can be reduced to the basic predicate *areInSameBlock* (cf. Subsection 4.4.3). This leads to the idea that we may just store the values of the latter relation (for each pair of variables) instead of maintaining a full set of partitions. Thus, we will only have to deal with a quadratic matrix and so, the required space is quadratically bounded. Since the *areInSameBlock* relation tracks if variables point into the same cluster, or, more broadly speaking, are aliases, we call such a matrix an *alias matrix A*.

Algorithm 4.3: Loop Tree Worklist

```
State variables: $loopTree$, $current$

operation initialize($S$)
    $G := (nodes(S), edges(S))$
    $LG := L(G)$
    add a node $root$ to $LG$                                   // dummy root
    for each $e = (init(S), i) \in edges(S)$
        add an edge $(root, e)$ to $LG$                  // connect the dummy root
    $loopTree$ := createLoopTree($LG$)
    $current$ := first child of $loopTree$              // this is the dummy root

operation isEmpty()
    return true iff next($current$) = first child of $loopTree$

operation next()
    $current$ := calculateNext($current$)
    return $current$

operation calculateNext($e$)
    if $e$ is last child of $e$.parent
        return checkFirstChildStable(first child of $e$.parent)
    if next sibling of $e$ is a leaf
        return next sibling of $e$
    // e's next sibling is an inner node
    return checkFirstChildStable(leftmost leaf of $e$'s next sibling)

operation checkFirstChildStable($e$)
    if the source node of $e$ has changed since last time
        return $e$                                           // remain in the loop
    else
        return calculateNext($e$.parent)                   // jump out of the loop

operation update($e$)
    // nothing to do
```

Algorithm 4.4: createLoopTree

```
Input  : a graph $G$
Output: the loop tree of $G$

$r$ := new root node
$SCC$ := strongly connected components of $G$
for each $S \in SCC$ in topological order
    if $S$ contains a single node $n$
        add $n$ as the next leaf to the root $r$
    else
        $h$ := a loop head of $S$
        remove all incoming edges of $h$
        add createLoopTree($S$) as next child to the root $r$        // recursion
return $r$
```

More formally, we introduce a function

$$\mathrm{proj}\colon 2^{\mathrm{Part}(\mathbf{PVar})} \to \mathbf{PVar} \times \mathbf{PVar} \to \mathbb{T}$$

that projects a set of partitions $\Phi \in 2^{\mathrm{Part}(\mathbf{PVar})}$ to an alias matrix $A\colon \mathbf{PVar} \times \mathbf{PVar} \to \mathbb{T}$ such that

$$A(i,j) = \mathrm{proj}(\Phi)(i,j) := \mathit{areInSameBlock}(\Phi, i, j)\,. \tag{4.19}$$

Note that, for technical reasons, we regard A as a function rather than an actual matrix as element of $\mathbb{T}^{n \times n}$. It is immediate that A is symmetric, i.e., $A(i,j) = A(j,i)$ and that its diagonal consists of ones, i.e., $A(i,i) = 1$.

Queries. Using an alias matrix $A = \mathrm{proj}(\Phi)$, the query predicates on analysis values can be expressed just as described in Subsection 4.4.3, substituting $A(i,j)$ for every occurrence of $\mathit{areInSameBlock}(\Phi, i, j)$.

Transition Functions. Since all transition functions with partition sets can be reduced to (possibly a sequence of) *New* and *Merge* operations (cf. Subsection 4.4.5), it suffices to define (overloaded versions) the two latter operations for alias matrices. To begin with, the *New* operation is defined by

$$A'(i,j) = \mathit{New}(A,x)(i,j) := \begin{cases} 1 & \text{, if } i = j = x; \\ 0 & \text{, if } i = x \textbf{ xor } j = x; \\ A(i,j) & \text{, otherwise.} \end{cases} \tag{4.20}$$

This captures the intuition that x is in its own block in every partition of the resulting partition set.

The *Merge* operation is given by:

$$\begin{aligned} A'(i,j) = \mathit{Merge}(A,x,y)(i,j) := {} & A(i,j) && (4.21a) \\ & \vee_T \left(A(i,x) \wedge_T A(j,y)\right) && (4.21b) \\ & \vee_T \left(A(j,x) \wedge_T A(i,y)\right). && (4.21c) \end{aligned}$$

Alternatively, by writing out the ternary logic constructs in (4.21b) and (4.21c), it can be expressed as follows:

$$A'(i,j) := \begin{cases} \max(A(i,j)\,, 0) & \text{, if } (A(i,x) = 0 \vee A(j,y) = 0) \\ & \quad \wedge \left(A(i,y) = 0 \vee A(j,x) = 0\right); \qquad (4.22a) \\ \max(A(i,j)\,, 1) & \text{, if } (A(i,x) = 1 \wedge A(j,y) = 1) \\ & \quad \vee \left(A(i,y) = 1 \wedge A(j,x) = 1\right); \qquad (4.22b) \\ \max(A(i,j)\,, ½) & \text{, otherwise.} \qquad (4.22c) \end{cases}$$

As in binary logic, the maximum operator is equivalent to the "or" connective, where the elements of $\mathbb{T} = \{0, ½, 1\}$ are interpreted as real numbers. We used the maximum operator so as to emphasize that a value cannot be decreased by a *Merge* operation. This corresponds to the fact that a *Merge* operation only unifies blocks of partitions and never separates two variables that are in the same block. Case 4.22b signifies that i and j will certainly end up in the same block if i is in the same block as x and j is in the same block as y (or in the symmetric case). Case 4.22a is the dual to case 4.22b. Otherwise, we cannot give a precise answer.

Note that the preconditions and early type checks (cf. Subsection 4.4.7) can also be easily translated for alias matrices.

4.8.1 Alias Matrices as Monotone Framework

Let $\mathcal{A}$ be the set of all alias matrices A: **PVar** × **PVar** → $\mathbb{T}$ that are symmetric and have a diagonal consisting of ones. $\mathcal{A}$ will act as analysis property space. Moreover, we define a join operator $\sqcup_M : \mathcal{A} \to \mathcal{A}$ as component-wise application of the ternary logic join operator $\sqcup_T$ specified in Table 3.4(c):

$$(A \sqcup_M B)(i,j) := A(i,j) \sqcup_T B(i,j)$$

for $i, j \in \mathbf{PVar}$. As described at the end of Section 3.7, this expresses the intuition that the resulting value is unknown unless both of the joined values are precise.

Finally, we add an "artificial" least value $\bot_M$ and define it to have the desired property that

$$A \sqcup_M \bot_M = A$$

for all $A \in \mathcal{A}$ [14]. This is legitimate due to the fact that the least value $\bot_M$ can only be the result of a join operation if both operands are equal to the least value.

Now we can prove that this amounts in fact to a Monotone Framework, which enables us to use the analysis solver algorithms outlined in Section 4.7.

Lemma 4.15 *The alias matrix analysis formalization constitutes a Monotone Framework.*

Proof. The $\sqcup_M$ operator clearly is idempotent, commutative and associative, as required by Definition 4.9. Since, in addition, there is a least element $\bot_M$, $(\mathcal{A}, \sqcup_M, \bot_M)$ is a bounded join-semilattice. Moreover, the satisfaction of the Ascending Chain condition follows because **PVar** × **PVar** is finite. As mentioned in Subsection 4.5.1, the bounded join-semilattice gives rise to a partial order $\sqsubseteq_M$ such that, for $A, B \in \mathcal{A}$,

$$A \sqsubseteq_M B \Leftrightarrow A \sqcup_M B = B.$$

What is left to confirm is the monotonicity of the transition functions (with respect to $\sqsubseteq_M$). According to Subsection 4.4.5, all transition functions can be reduced to a sequence of *Merge* and *New* operations. Hence, it suffices to demonstrate that the latter two functions are monotone. For this purpose, let $A, B \in \mathcal{A}$ and $A \sqsubseteq_M B$ or, equivalently, $A \sqcup_M B = B$.

For the *New* operation and a variable $x \in \mathbf{PVar}$, let us write $A' := \mathit{New}(A, x)$ and $B' := \mathit{New}(B, x)$. From the definition of the *New* operation, (4.20), it follows that the operation affects only the x-column and the x-row in the input matrix: all entries in the x-column and the x row are set to 0 except for the entry (x,x), which is set to 1. All other entries are unchanged by the *New* operation. Since it holds that $A'(i,j) \sqcup_T B'(i,j) = B'(i,j)$ for entries in the x-column and x-row as well as for all remaining entries (by the assumption $A \sqcup_M B = B$), we have $A' \sqsubseteq_M B'$, as desired.

The corresponding proof for the *Merge* operation will require some case splits. For variables $x, y \in \mathbf{PVar}$, let us write $A' := \mathit{Merge}(A,x,y)$ and $B' := \mathit{Merge}(B,x,y)$. Suppose for a contradiction that $A' \sqcup_M B' \neq B'$, i.e., there is an entry (i,j) such that $A'(i,j) \sqcup_T B'(i,j) \neq B'(i,j)$. By looking at the definition of the $\sqcup_T$ operator, Table 3.4(c), it follows that this case can only occur if either

$$B'(i,j) = 1 \wedge A'(i,j) \neq 1 \text{ or, likewise,} \tag{4.23}$$

$$B'(i,j) = 0 \wedge A'(i,j) \neq 0. \tag{4.24}$$

Via the definition of the *Merge* operation, (4.22b), case $B'(i,j) = 1$ implies that either

$$\begin{gathered} B(i,j) = 1 \text{ or} \\ \big(B(i,x) = 1 \wedge B(j,y) = 1\big) \vee \big(B(i,y) = 1 \wedge B(j,x) = 1\big). \end{gathered}$$

Using the assumption $A \sqcup_M B = B$, it follows that A must have a 1 entry wherever B has a 1 entry. I.e., it holds that either

$$\begin{gathered} A(i,j) = 1 \text{ or} \\ \big(A(i,x) = 1 \wedge A(j,y) = 1\big) \vee \big(A(i,y) = 1 \wedge A(j,x) = 1\big). \end{gathered}$$

But this means, again by definition of the *Merge* operation, that $A'(i,j) = 1$, which contradicts (4.23).

Case $B'(i,j) = 0$ can be handled in a dual way. ■

4.8.2 Alias Matrices as Conservative Approximation of Partition Sets

In this subsection we will demonstrate that using alias matrices instead of partition sets is a conservative approximation. In other words, if we start off with projecting a partition set to an alias matrix and perform the same sequence of transition functions and join operations with the original partition set and the projected alias matrix, then we may incur a loss in precision, but, on the other hand, no spurious precise answers can be produced by the alias matrices.

The following lemma is even stronger: it is to say that under a *New* operation, an alias matrix is as precise as a partition set.

Lemma 4.16 *For a partition set $\Phi \in 2^{\mathrm{Part}(\textbf{\textit{PVar}})}$ and a variable $x \in$ **PVar**,*

$$\mathrm{proj}(New(\Phi, x)) = New\big(\mathrm{proj}(\Phi), x\big).$$

In other words, the following diagram is commutative:

$$\begin{tikzcd}
\Phi \arrow[r, "\mathrm{proj}(.)"] \arrow[d, "New(.{,}x)"'] & A \arrow[d, "New(.{,}x)"] \\
\Phi' \arrow[r, "\mathrm{proj}(.)"'] & A'
\end{tikzcd}$$

Proof. Having $\Phi' := New(\Phi, x)$, it follows from (4.4) that for a partition $P' \in \Phi'$ and a variable $i \in \mathbf{PVar}$

$$P'^{[i]} = \begin{cases} \{x\} & \text{, if } i = x; \\ P^{[i]} \setminus \{x\} & \text{, otherwise,} \end{cases}$$

meaning that in P', i is in its own block and, consequently, no other variable is in the same block as i. Now let $A := \mathrm{proj}(\Phi)$ and $A' := New(A, x)$. For variables $i, j \in \mathbf{PVar}$, the above observation leads to

$$areInSameBlock\big(\Phi', i, j\big) = \begin{cases} 1 & \text{, if } i = j = x; \\ 0 & \text{, if } i = x \textbf{ xor } j = x; \\ areInSameBlock\big(\Phi, i, j\big) & \text{, otherwise.} \end{cases} \tag{4.25}$$

Comparing (4.25) with (4.20) immediately yields $\mathrm{proj}\big(\Phi'\big)\big(i, j\big) = A'\big(i, j\big)$. ∎

For the *Merge* operation, the analogous correspondance

$$\mathrm{proj}\big(Merge\big(\Phi, x, y\big)\big) \overset{?}{=} Merge\big(\mathrm{proj}\big(\Phi, x, y\big)\big)$$

turns out to be invalid and thus, the following diagram is *not* commutative:

$$\begin{tikzcd}
\Phi \arrow[r, "\mathrm{proj}(.)"] \arrow[d, "Merge(.{,}x{,}y)"'] & A \arrow[d, "Merge(.{,}x{,}y)"] \\
\Phi \arrow[r, "\mathrm{proj}(.)"'] & A'
\end{tikzcd}$$

Consider the diagrams in Figure 4.3 for a counter example, where the partition sets in the top-left and top-right corner have the same projection to an alias matrix, but a different projection after a *Merge* transition function has been applied. Since the corresponding transition function in the alias matrix domain naturally needs to be deterministic, there is no way we could define such an operation without a loss in precision (indicated by '?' in the figure).

$$\begin{tikzcd}
\{xb \mid y \mid a,\ x \mid b \mid ya\} \arrow[r, "\text{proj}(.)"] \arrow[d, "Merge(.,x,y)"] & \begin{array}{c|cccc} & x & y & a & b \\ \hline x & 1 & 0 & 0 & \tfrac{1}{2} \\ y & & 1 & \tfrac{1}{2} & 0 \\ a & & & 1 & 0 \\ b & & & & 1 \end{array} \arrow[d, "?"] \arrow[dr, "?"] & & \{x \mid y \mid a \mid b,\ xb \mid ya\} \arrow[ll, "\text{proj}(.)"'] \arrow[d, "Merge(.,x,y)"] \\
\{xyb \mid a,\ xya \mid b\} \arrow[r, "\text{proj}(.)"] & \begin{array}{c|cccc} & x & y & a & b \\ \hline x & 1 & 1 & \tfrac{1}{2} & \tfrac{1}{2} \\ y & & 1 & \tfrac{1}{2} & \tfrac{1}{2} \\ a & & & 1 & 0 \\ b & & & & 1 \end{array} & \begin{array}{c|cccc} & x & y & a & b \\ \hline x & 1 & 1 & \tfrac{1}{2} & \tfrac{1}{2} \\ y & & 1 & \tfrac{1}{2} & \tfrac{1}{2} \\ a & & & 1 & \tfrac{1}{2} \\ b & & & & 1 \end{array} & \{xy \mid a \mid b,\ xyab\} \arrow[l, "\text{proj}(.)"']
\end{tikzcd}$$

Figure 4.3: Precision loss with alias matrices: the projections of both partition sets result in the same alias matrix, but the projections of the respective partition sets after applying the *Merge* operation are different[3].

Nevertheless, the following lemma states that under a *Merge* operation, an alias matrix is a conservative approximation of a partition set, in the sense that a precise value in the alias matrix implies the same precise value in the partition set.

Lemma 4.17 *For a partition set $\Phi \in 2^{\text{Part}(\textbf{\textit{PVar}})}$ and variables $x, y \in$ **PVar**, let $\Phi' := Merge(\Phi, x, y)$, $A := \text{proj}(\Phi)$ and $A' := Merge(A, x, y)$. Then, for variables $i, j \in$ **PVar**,*

(i) $A'(i,j) = 1 \Rightarrow areInSameBlock(\Phi', i, j) = 1$

(ii) $A'(i,j) = 0 \Rightarrow areInSameBlock(\Phi', i, j) = 0$

[3] It may be argued that this example is a special case since the partition set in the top-right corner is minimizable in the sense of Section 4.9. However, it has been verified that examples without this property can be constructed as well.

Proof.

(i) Similarly to demonstrating the monotonicity of a *Merge* operation in the proof of Lemma 4.15, it follows from the definition of the *Merge* operation that $A'(i,j) = 1$ requires either

$$A(i,j) = 1 \text{ or}$$
$$\big(A(i,x) = 1 \wedge A(j,y) = 1\big) \vee \big(A(i,y) = 1 \wedge A(j,x) = 1\big)\,.$$

Case $A(i,j) = 1$ implies $\mathit{areInSameBlock}(\Phi, i, j) = 1$. Since the *Merge* operation does not separate blocks, we have $\mathit{areInSameBlock}(\Phi', i, j) = 1$.

Case $A(i,x) = 1 \wedge A(j,y) = 1$ implies

$$\mathit{areInSameBlock}(\Phi, i, x) = 1 \wedge \mathit{areInSameBlock}(\Phi, j, y) = 1.$$

In other words, for every partition $P \in \Phi$, we have $P^{[i]} = P^{[x]}$ and $P^{[j]} = P^{[y]}$. But then, for every partition, i and j will end up in the same block after the blocks of x and y have been unified by the *Merge* operation. This results in $\mathit{areInSameBlock}(\Phi', i, j) = 1$.

Likewise, case $A(i,y) = 1 \wedge A(j,x) = 1$ implies

$$\mathit{areInSameBlock}(\Phi, i, y) = 1 \wedge \mathit{areInSameBlock}(\Phi, j, x) = 1.$$

Using the same argument as in the previous case immediately leads to the desired $\mathit{areInSameBlock}(\Phi', i, j) = 1$.

(ii) Once more, the proof of the second part of the lemma is completely dual to the proof of the first part and therefore omitted here. ■

Combining the two of the previous lemmas and the conservative nature of the $\sqcup_M$ operation, we conclude with the main theorem of this subsection.

Theorem 4.18 *Alias matrices are a conservative approximation of partition sets as analysis values: there may be a loss in precision (an alias matrix may signal an unknown answer, while the corresponding partition set reports a precise answer), but precise answers given by alias matrices are always sound.*

Proof. The $\sqcup_T$ operation clearly cannot introduce spurious precise answers, since the joined value is only precise if both operands are precise. Consequently, the same holds for the $\sqcup_M$ operation.

Furthermore, since every sequence of transition functions can be expressed as a sequence of *New* and *Merge* operations and since the analysis terminates after applying a finite sequence of transition functions and join operations, the above observation and Lemmas 4.16 and 4.17 can be inductively applied to obtain the statement of the lemma.■

For reference, the Listing 4.1 displays a program[4] that corresponds to the settings in the example incurring a precision loss given in Figure 4.3: An analysis solver using alias matrices reports the that on line 21, the variable `b` may be unusable, whereas a solver using partition sets accepts the program.

4.9 Minimized Partition Sets

Another way to possibly overcome the potentially exponential size of a partition set uses the notion of *finer* and *coarser* partitions.

Finer and Coarser Partitions. For a set A and partitions $P, Q \in \mathrm{Part}(A)$, P is *finer* than or equal to Q, $P \leq_P Q$, iff all blocks of P are contained in a block of Q:

$$P \leq_P Q \quad :\Longleftrightarrow \quad \forall p \in P : \exists q \in Q : p \subseteq q.$$

Conversely, Q is said to be *coarser* than or equal to P.

Example 4.19 For $A := \{w, x, y, z\}$, the following partitions in $\mathrm{Part}(A)$ are examples related by $\leq_P$:

$$\begin{gathered} wx \mid y \mid z \leq_P wx \mid yz \\ wxy \mid z \leq_P wxyz \\ wz \mid x \mid y \leq_P wxz \mid y. \end{gathered}$$

[4] The source file can be found at `org/multijava/mjc/testcase/universes/uniqueness/Analysis_Precision.java` in the MultiJava source tree.

```
class C extends Object {

    peer C f;

    void m() {
        rep C x = null;
        rep C y = null;
        rep C a = null;
        rep C b = null;
        peer C z;

        if (x == null) {
            x.f = b; // merge(x, b)
        } else {
            y.f = a; // merge(y, a)
        }

        x.f = y; // merge(x, y)

        z = a; // consume(a);
        z = b; // consume(b);
    }

}
```

Listing 4.1: Precision loss with alias matrices.

In contrast, the following partitions are not related by $\leq_P$:

$$\begin{gathered} wx \mid y \mid z,\ w \mid x \mid yz \\ wxy \mid z,\ wx \mid yz \\ wz \mid x \mid y,\ wx \mid y \mid z. \end{gathered} \qquad \square$$

As an aside, we remark that $\big(\mathrm{Part}(A), \leq_P\big)$ in fact constitutes a complete lattice, for any set A.

Minimize Operation. The idea is now to reduce (minimize) the size of a given partition set by throwing away all partitions which are finer than another partition contained in the set. Thus, for a set A, we now define a minimize operation on partition sets

$$\mathrm{minimize}\colon 2^{\mathrm{Part}(A)} \to 2^{\mathrm{Part}(A)}$$

which removes all partitions that are finer than another partition in the set:

$$\begin{aligned} \mathrm{minimize}(\Phi) := \text{“remove all partitions } P \in \Phi \text{ if} \\ \text{there is a partition } Q \in \Phi \text{ such that } P <_P Q\text{”} \end{aligned}$$

Example 4.20 For $A := \{w, x, y, z\}$, we have

$$\mathrm{minimize}\big(\{wx \mid y \mid z,\ wx \mid yz\}\big) = \{wx \mid yz\}\,. \tag{4.26}$$

$\square$

Note that the trivial upper bound for the cost of minimizing a partition set Φ is $O\Big(|\Phi|^2\Big)$, since every pair of partitions has to be examined.

Adapted Query Functions. It clearly holds that

$$\mathit{areInSameBlock}\big(\Phi, x, y\big) = 0 \quad \Rightarrow \quad \mathit{areInSameBlock}\big(\mathrm{minimize}(\Phi), x, y\big) = 0.$$

However, the analogous statement for a positive answer,

$$\mathit{areInSameBlock}\big(\Phi, x, y\big) = 1 \quad \overset{?}{\Rightarrow} \quad \mathit{areInSameBlock}\big(\mathrm{minimize}(\Phi), x, y\big) = 1,$$

is invalid, as seen in (4.26), where

$$\mathit{areInSameBlock}(\{wx \mid yz\}, y, z) = 1$$

would be a spurious precise answser. Thus, we define a new conservative version of the *areInSameBlock* predicate:

$$\mathit{areInSameBlock}'(\Phi, x, y) := \begin{cases} 0 & \text{, if } \mathit{areInSameBlock}(\Phi, x, y) = 0; \\ 1/2 & \text{, otherwise.} \end{cases}$$

Since the value 1 is never returned, this function actually has only a two valued logic range[5]. The query predicates for minimized partition sets are then expressed just as described in Subsection 4.4.3, substituting $\mathit{areInSameBlock}'(\Phi, x, y)$ for every occurrence of $\mathit{areInSameBlock}(\Phi, x, y)$.

Compared with the (unminimized) partition set formalization, the only operation that needs to be modified is the join operation, which now additionally performs the minimization of the resulting partition set. The transition functions (including the preconditions and the early type checks) and the analysis property space do not change. Thus, it follows that minimized partition sets still define a Monotone Framework.

Minimized partition sets may require exponentially less time and space than partition sets without loosing precision (as in Benchmark 1, cf. Section 4.11). However, there are cases of partition sets that are not minimizable, even exponentially big ones (see Example 4.21 below). Thus, minimized partition sets have an even larger worst-case time complexity than partition sets due to the additional cost of the minimize operation. Furthermore, since only a two valued logic is used in analysis queries, the precision may suffer substantially.

Example 4.21 For $n \in \mathbb{N}$, consider a variable set $A := \{x_1, \ldots, x_{2n}\}$ of size $2n$. We define a partition set $\Phi \in 2^{\mathrm{Part}(A)}$ to be

$$\Phi := \left\{ \{V, A \setminus V\} : V \in \binom{A}{n} \right\}.$$

In other words, a partition $P \in \Phi$ consists of the blocks V and $A \setminus V$, where V is an n-element subset of A. Since each partition $P \in \Phi$ has different blocks, it follows that Φ

[5] This reflects the development process of the project, where the ternary logic versions of the query functions were added later.

cannot be minimized. For reasons of symmetry, each V will also occur exactly once as $A \setminus V$. Thus, we have

$$|\Phi| = \frac{1}{2}\left|\binom{A}{n}\right| = \frac{1}{2}\binom{2n}{n} = \frac{(2n)!}{2\,(n!)^2} \sim \frac{4^n}{2\sqrt{\pi n}},$$

using Stirling's formula. This proves that there are exponentially large partition sets that cannot be minimized. □

4.10 Three Different Kinds of Analyses – Summary

This section summarizes the pro and cons of the three different analysis values we have seen: partition sets, alias matrices and minimized partition sets. Moreover, we present a way of combining them to form a meta analysis solver.

Partition sets have the highest precision, but may require exponential time and space. The time and space requirements for alias matrices are bounded quadratically, but they may suffer from a loss in precision as compared with partition sets. Finally, minimized partition sets even have a larger worst-case time complexity than partition sets, but they may be exponentially faster than partition sets without loosing precision. Furthermore, as they use only a two valued logic, the precision may suffer substantially.

It should be noted that both alias matrices and minimized partition sets are conservative approximations to partition sets, but there is no such relation between alias matrices and minimized partition sets, as there are examples where alias matrices are more precise than minimized partition sets and vice versa.

Integrating the quadratic bound for the cost of a basic operation (an application of a transition function or an application of a join operation) into the worst-case cost equation for a general worklist solver (cf. Section 4.7), the overall worst-case bound for a run of an alias matrix solver for an analysis program S is $O\Big(|nodes(S)|^3 \cdot h\Big)$, where we used $O(|edges(S)|) = O(|nodes(S)|)$ and h stands for the length of the longest ascending chain in the alias matrix join-semilattice. The value of h may be be accurately analyzed as future work.

A partition set solver and a minimized partition set solver both have an exponential worst-case time complexity.

It is yet to be determined whether minimized partition sets provide a useful alternative to partition sets and alias matrices for practical examples.

4.10.1 Hierarchical Solver Approach

As hinted at above, we can construct a meta analysis solver that contains a set of solvers, each with an associated priority. A smaller priority means that the solver is run first. Then, as long as there is an imprecise answer to a query (or as long as the analysis is imprecisely terminated due to an unsatisfied precondition), the solver of the next higher priority is run, if there is any. This corresponds to Algorithm 4.5.

Algorithm 4.5: Hierarchical Solver (Sketch)

```
solve the analysis with solver of $currentPriority$
$answer$ := getAnswer()
while $answer$ is imprecise $\wedge$ $currentPriority < highestPriority$
    increase $currentPriority$
    solve the analysis with solver of $currentPriority$
    $answer$ := getAnswer()
```

In the current implementation, the following solvers and priority assignments are used:

Solver	Priority
Alias matrix solver	1
Minimized partition set solver	2
Partition set solver	3

In other words, the alias matrix solver is run first, which makes sense, as it is the only solver that has a non-exponential worst-case bound. Should there be an imprecise answer, the minimized partition set solver follows to be executed. Finally, if there is still an imprecise answer, the partition set worklist solver is invoked. Note that even though there is no “is more precise than” relation between the alias matrix solver and the minimized partition set solver, there will eventually be a solver of higher priority, namely the partition set solver, which is at least as precise as any of the alias matrix solver and the minimized partition set solver.

4.11 Benchmarks

This section presents some benchmarks for contrived worst-case examples where the data flow analysis is expected to perform badly. The data structures used in the implementations are described in Section 5.4.

All benchmarks are measured on a 2.8 GHz Pentium-4 CPU with 1024 MB of physical RAM and 512 MB of VM heap space. The given times are for the solution only, meaning that, e.g., the time to build up the test flow graph is not taken into account. All time measurements are averaged over 3 runs. “oom” denotes an out of memory error, “PS” stands for partition sets, “Min. PS” for minimized partition sets, “AM” for alias matrices.

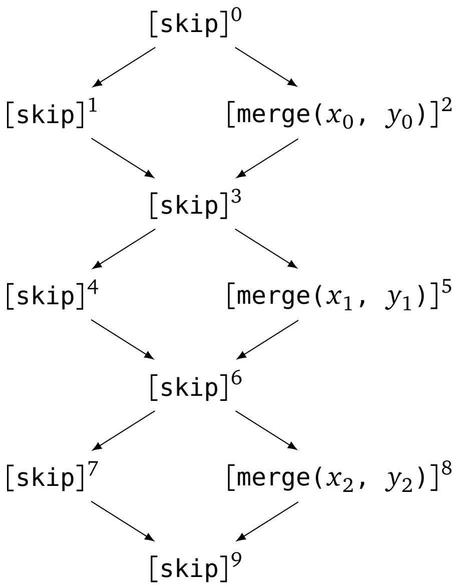
It should be pointed out that all solvers acquire the same level of precision for all presented benchmarks. I.e., the alias matrix solver and the minimized partition solver do not incur a precision loss in comparison with the partition set worklist solver.

Benchmark 1. The purpose of this benchmark is to test a worst-case example where the partition sets grow exponentially. The test program uses of a sequence of k branches, such that there is one unique partition for every possible execution path taken. Since there are exponentially many such paths, we obtain the desired exponential (in k) size of the partition sets. The flow graph consists of

- $2k$ local variables $(\{x_0, \ldots, x_{k-1}, y_0, \ldots, y_{k-1}\})$,
- $3k + 1$ flow graph nodes and
- $4k$ flow graph edges.

Due to their special form, the resulting partition sets can be minimized to size one. I.e., when using a minimized partition set solver there will be only one partition in the partition sets, instead of exponentially many.

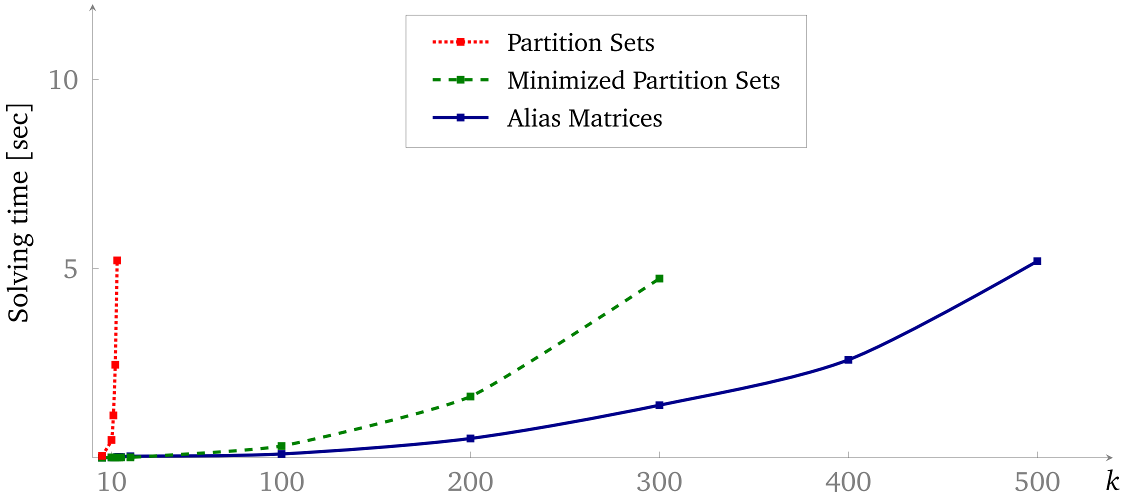
Figure 4.4 displays the result of running Benchmark 1. As expected, the minimized partition set solver performs substantially better than the partition set solver. The alias matrix solver’s curve also corresponds to the expected growth, which is at least cubic. Note that for $k = 600$, which amounts to 1200 variables and around 1800 flow graph nodes, even the alias matrix solver runs out of memory.


[skip]^0
[skip]^1
[merge(x_0, y_0)]^2
[skip]^3
[skip]^4
[merge(x_1, y_1)]^5
[skip]^6
[skip]^7
[merge(x_2, y_2)]^8
[skip]^9

(a) Flow graph for $k = 3$.

k	PS	Min. PS	AM
5	0.05	0.00	0.00
10	0.47	0.01	0.01
11	1.12	0.01	0.01
12	2.46	0.01	0.01
13	5.22	0.01	0.02
14	oom	0.01	0.02
15		0.01	0.02
20		0.01	0.04
100		0.31	0.10
200		1.62	0.51
300		4.74	1.39
400		oom	2.59
500			5.20
600			oom

(b) Solving times in seconds against k.



(c) Solving times in seconds against k.

Figure 4.4: Benchmark 1.

Benchmark 2. Benchmark 2 consists of a loop of a sequence of $k - 1$ move operations that needs k iterations to reach a fixed point. In total, we have

- $2k$ variables (as in Benchmark 1),
- $2k + 1$ flow graph nodes and
- $2k + 1$ flow graph edges.

In contrast to Benchmark 1, the partition sets cannot be minimized in this case. Consequently, the minimized partition set worklist solver is expected to perform worse than the partition set worklist solver, since it the quadratic minimize operation is performed in vain. The results presented in Figure 4.5 confirm this. The alias matrix solver again is clearly more efficient than both partition set solvers, though reaching its scalability limits for high values of k.

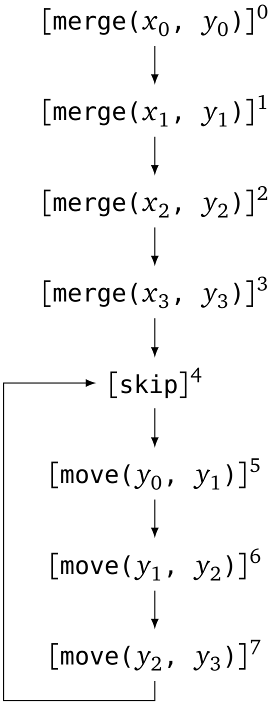
Benchmark 2a. This is a variant of Benchmark 2 where we have ℓ nested loops of the same type as in Benchmark 2. Each sub loop operates on a dedicated set of $2k$ variables. Totally, this amounts to

- $\ell \cdot 2k$ local variables,
- $\ell \cdot (2k + 1)$ flow graph nodes and
- $\ell \cdot (2k + 2) - 1$ flow graph edges.

As presented in Figure 4.6, the performance of the partition set worklist solver is very poor, resulting in an early out of memory error. Since the minimized partition set performs even worse (as the partition sets still cannot be minimized), its measurements are not shown in the figure. In similarity to the previous benchmarks, the alias matrix worklist solver is acceptable.

Benchmark 2b. Another variant of Benchmark 2 with ℓ nested loops of the same type as in Benchmark 2, plus conditional labeled `break` statements from each loop to the top level. This gives rise to

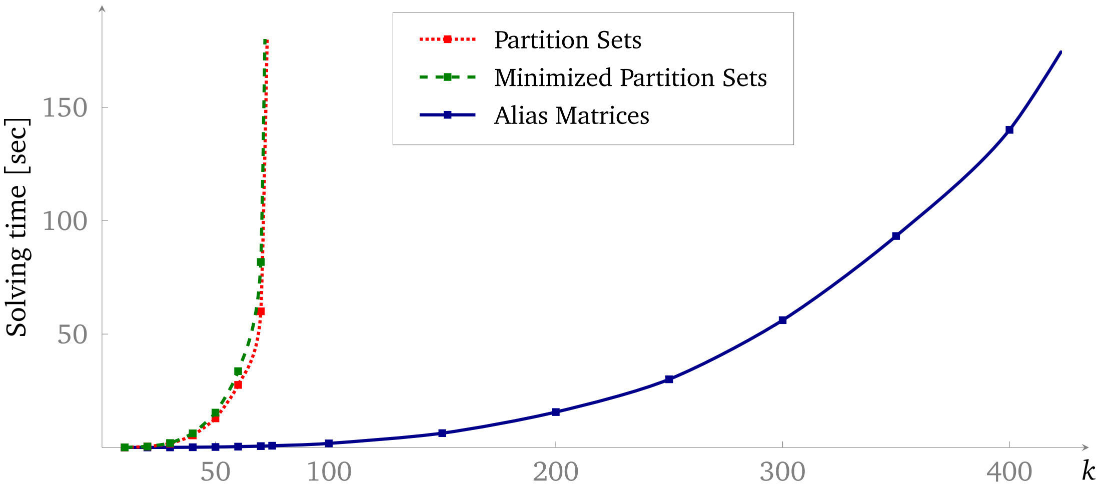
- $\ell \cdot 2k$ local variables,
- $\ell \cdot (2k + 4) + 1$ flow graph nodes and
- $\ell \cdot (2k + 6)$ flow graph edges.


[merge(x_0, y_0)]^0
[merge(x_1, y_1)]^1
[merge(x_2, y_2)]^2
[merge(x_3, y_3)]^3
[skip]^4
[move(y_0, y_1)]^5
[move(y_1, y_2)]^6
[move(y_2, y_3)]^7

(a) Flow graph for $k = 4$.

k	PS	Min. PS	Alias Matrix
10	0.06	0.06	0.04
20	0.40	0.43	0.02
30	1.71	1.95	0.05
40	5.33	6.23	0.11
50	12.85	15.36	0.21
60	27.59	33.64	0.37
70	60.04	81.77	0.60
75	299.42	412.15	0.77
100			1.80
150			6.31
200			15.61
250			30.07
300			56.13
350			93.22
400			140.04
450			220.13

(b) Solving times in seconds against k.



(c) Solving times in seconds against k.

Figure 4.5: Benchmark 2.

k	PS	AM
5	4.20	0.05
10	11.34	0.14
15	oom	0.43
20		1.03
25		2.09
30		3.58
35		5.82
40		8.75
45		12.88

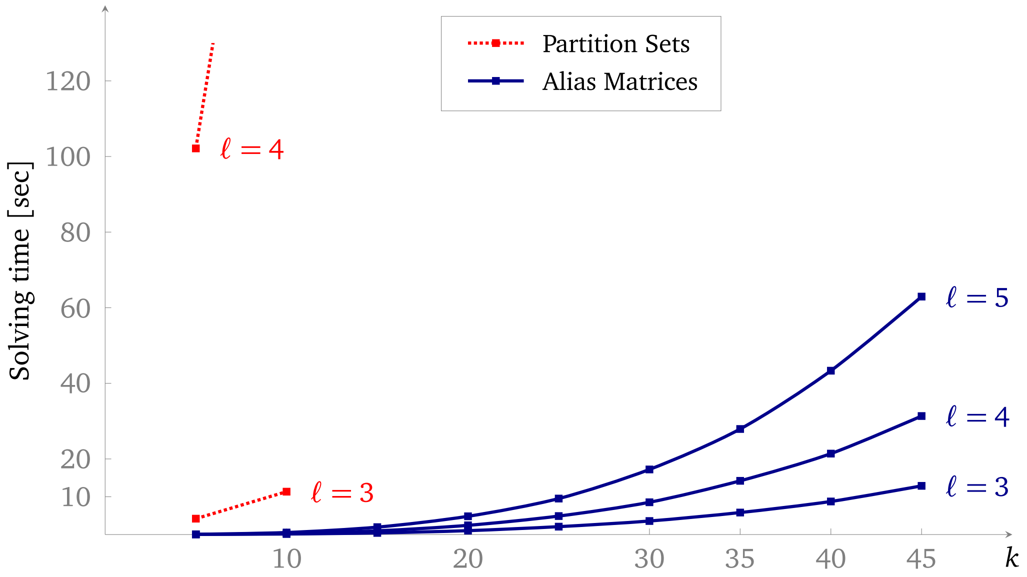
(a) Solving times in seconds against k for $\ell = 3$.

k	PS	AM
5	102.14	0.04
10	249.54	0.27
15	oom	0.98
20		2.44
25		4.89
30		8.54
35		14.21
40		21.41
45		31.35

(b) Solving times in seconds against k for $\ell = 4$.

k	PS	AM
5	oom	0.07
10		0.54
15		1.96
20		4.83
25		9.52
30		17.22
35		27.92
40		43.34
45		62.94

(c) Solving times in seconds against k for $\ell = 5$.



(d) Solving times in seconds against k for $\ell = 3, 4, 5$.

Figure 4.6: Benchmark 2a.

The results, displayed in Figure 4.7, resemble the ones of Benchmark 2a, though the absolute timings are slightly lower.

Benchmark 3 Finally, the flow graph of Benchmark 3 consists of a a sequence of $\binom{k}{2}$ branches and

- k local variables $(\{x_0, \ldots, x_{k-1}\})$,
- $3\binom{k}{2} + 1$ flow graph nodes and
- $4\binom{k}{2}$ flow graph edges.

Unlike Benchmark 1, the partition sets do not grow to an exponential, but only to a quadratic size. Yet, as in Benchmark 1, they can be minimized. The results of the benchmark runs are depicted in Figure 4.6. For low values of k, all three solvers perform efficiently. However, the partition set solver runs out of memory very early, for $k = 15$, while the minimized partition set solver and the alias matrix solver prove a better scaling.

Conclusion. In conclusion, we recognize that the alias matrix solver clearly outperforms both partition set solvers, as expected. There are examples where the minimizing the partition sets leads to a remarkable efficiency improvement over unminimized partition sets, but we also saw cases where futile minimizing causes the performance to decay. It is to be analyzed as future work whether the precision loss pertaining to alias matrices is of great hindrance for *practical* examples. Further, let us remark that worst-case behavior is often not indicative of typical performance.

4.12 Existing Points-To Analyses

Let us conclude this chapter by briefly investigating into the question as to why there exists no points-to analysis (also called pointer analysis) that we could use in an off-the-shelf manner, given the long history of pointer analysis research.

First of all, it has to be remarked that there has been little work done in the area of pointer analysis for object-oriented languages. What is more, most of the existing

k	PS	AM
5	2.72	0.06
10	5.63	0.13
15	oom	0.36
20		0.80
25		1.57
30		2.65
35		4.28
40		6.33
45		9.20

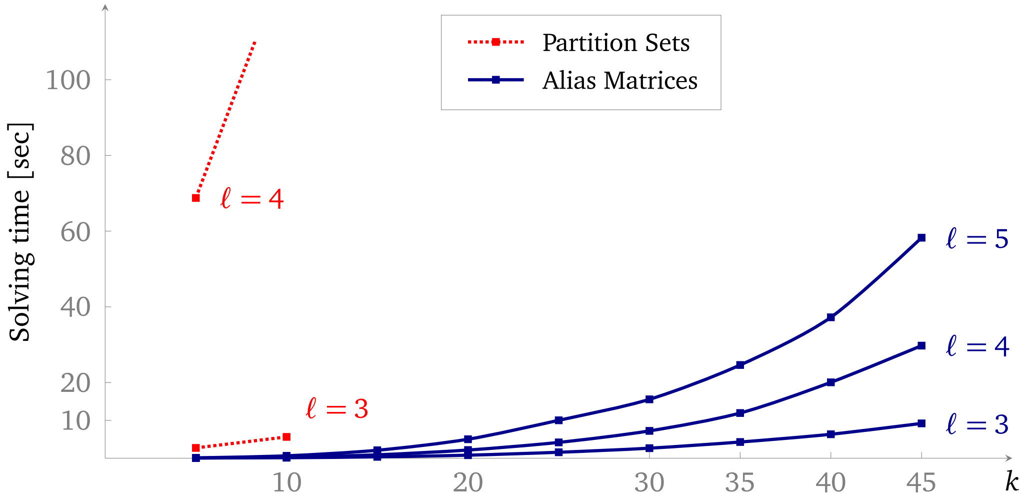
(a) Solving times in seconds against k for $\ell = 3$.

k	PS	AM
5	68.77	0.04
10	132.21	0.28
15	oom	0.90
20		2.16
25		4.18
30		7.22
35		11.94
40		20.05
45		29.75

(b) Solving times in seconds against k for $\ell = 4$.

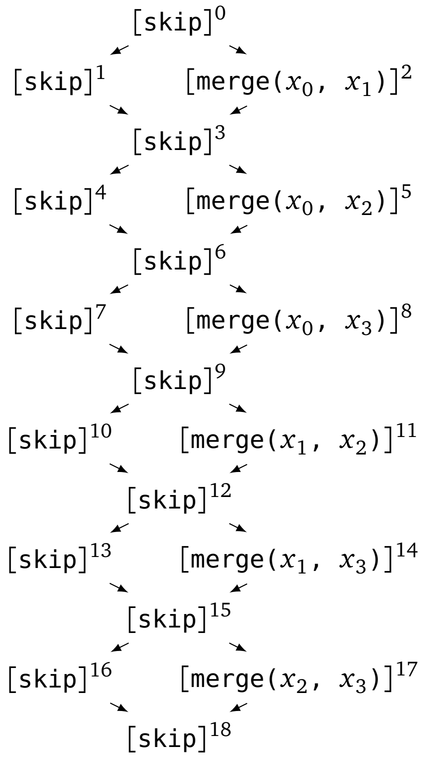
k	PS	AM
5	oom	0.093
10		0.635
15		2.101
20		5.013
25		10.018
30		15.557
35		24.629
40		37.232
45		58.266

(c) Solving times in seconds against k for $\ell = 5$.



(d) Solving times in seconds agains k for $\ell = 3, 4, 5$.

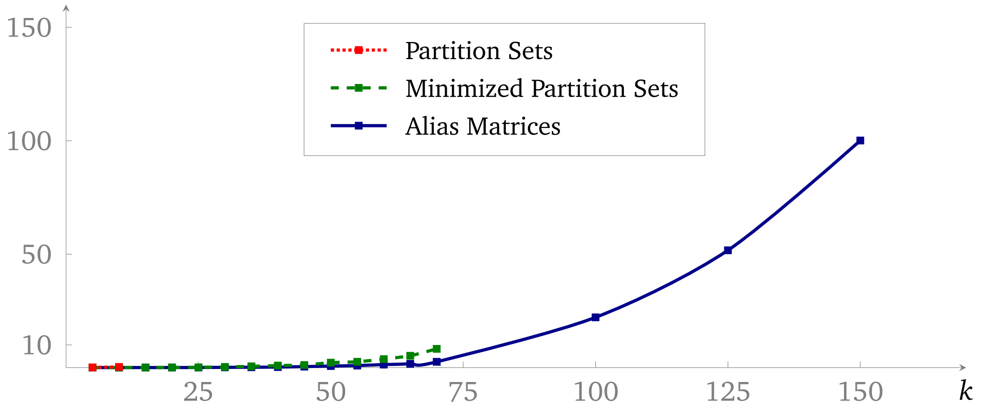
Figure 4.7: Benchmark 2b.



(a) Flow graph for $k = 4$.

k	PS	PS (Min.)	AM
5	0.08	0.03	0.01
10	0.27	0.04	0.02
15	oom	0.04	0.02
20		0.06	0.03
25		0.12	0.04
30		0.28	0.11
35		0.55	0.15
40		0.89	0.24
45		1.14	0.43
50		2.17	0.69
55		2.56	0.90
60		3.74	1.33
65		5.21	1.65
70		8.26	2.55
100		oom	22.16
125			51.72
150			100.12
200			oom

(b) Solving times in seconds against k.



(c) Solving times in seconds against k.

Figure 4.8: Benchmark 3.

analysis are flow-insensitive, whereas our application seems to rely on the precision of a flow-sensitive analysis.

Generally, the precision and efficiency requirements depend greatly on the client application – there is no magic bullet analysis. Using partition sets as analysis values delivers a high precision, but the scalability seems to be poor. On the other hand, alias matrices scale better, but may suffer from a loss in precision. Using alias matrices as analysis values goes into the direction of existing analyses that use points-to pairs (x points to y) or alias pairs (x and y point to the same object) as analysis values. However, the *Merge* and *New* operations, which are inherently different, still make our setting special.

Furthermore, it seems to be difficult to categorize existing pointer analysis into groups that offer adequate precision and efficiency characteristics for the certain classes of client problems.

Hind presents a survey of pointer analysis research issues and open problems with the meaningful title "Pointer Analysis: Haven't we solved this problem yet?" [12]. Therein, Manuel Fähndrich is cited saying

> "I think there are two distinct uses of pointer analysis, 1) optimizations, and 2) error detection/program understanding. These two uses have vastly different requirements on pointer analysis. For optimizations, there seems to be some upper bound on how much precision is useful because taking advantage of more precision usually translates into specializing more code, which needs to be bounded. In my opinion, the spectrum of analyses mostly covers the needs for optimizations. For error detection and program understanding, the picture is different. For these applications, there seems to be a lower bound on precision, below which, pointer information is pretty useless. Gearing pointer analysis towards error detection requires more work on precision and scaling issues."

Clearly, our application falls in the second domain, which may be another reason why there does not seem to be a suitable analysis for our needs.

5 Implementation and Examples

The proposed type system has been implemented as part of the MultiJava [8, 9] compiler and the JML [6] tools. This chapter illuminates various aspects of the implementation and concludes with the demonstration of some actual code examples.

5.1 MultiJava and JML

In previous works, the Universe type system has been integrated into the MultiJava compiler and the JML tools, featuring static type checking as well as runtime and byte code generation support.

Since JML builds on top of the MultiJava compiler and its utility classes, the major part of the uniqueness and ownership transfer extensions pertains to the MultiJava code, as JML automatically benefits from them. Since, in fact, the modifications to JML merely consisted of altering grammar definition files and delegation code to MultiJava, this chapter is mainly concerned with the MultiJava implementation.

5.2 New Command Line Options

The existing `--universesx` command line switch of the MultiJava/JML compiler accepts a list of comma separated options that control which part of the Universe type system implementation are to be enabled. Two new options have been added:

- `uniq`: Enable the uniqueness and ownership extension, using an alias matrix solver (cf. Section 4.8) to solve the data flow analysis.
- `uniq-prec`: Enable the uniqueness and ownership extension, using a hierarchical analysis solver as described in Subsection 4.10.1. `-prec` is meant to stand for "precision" or "precise".

Note that the uniqueness options are interpreted as "grading up" each given Universe options to make them cope with uniqueness and ownership transfer. They do not have an effect by themselves. I.e., `--universesx parse,check` parses the Universe keywords and type checks the code without the uniqueness and ownership transfer extension, while `--universesx parse,check,uniq` parses the Universe keywords taking the new uniqueness keywords into account and performs type checks with the uniqueness and ownership transfer extensions enabled.

5.3 Type Checking

The MultiJava compiler uses a number of separate passes to process an abstract syntax tree (AST). A prominent pass is the type check pass, which performs the Java type checking on the AST. To support the uniqueness extensions, two new compiler passes have been added to surround the Java type checking pass, though the first one only walks a portion of the AST. Figure 5.1 shows a high-level outline of the uniqueness type checking process as implemented in MultiJava. The single steps are discussed in the following.

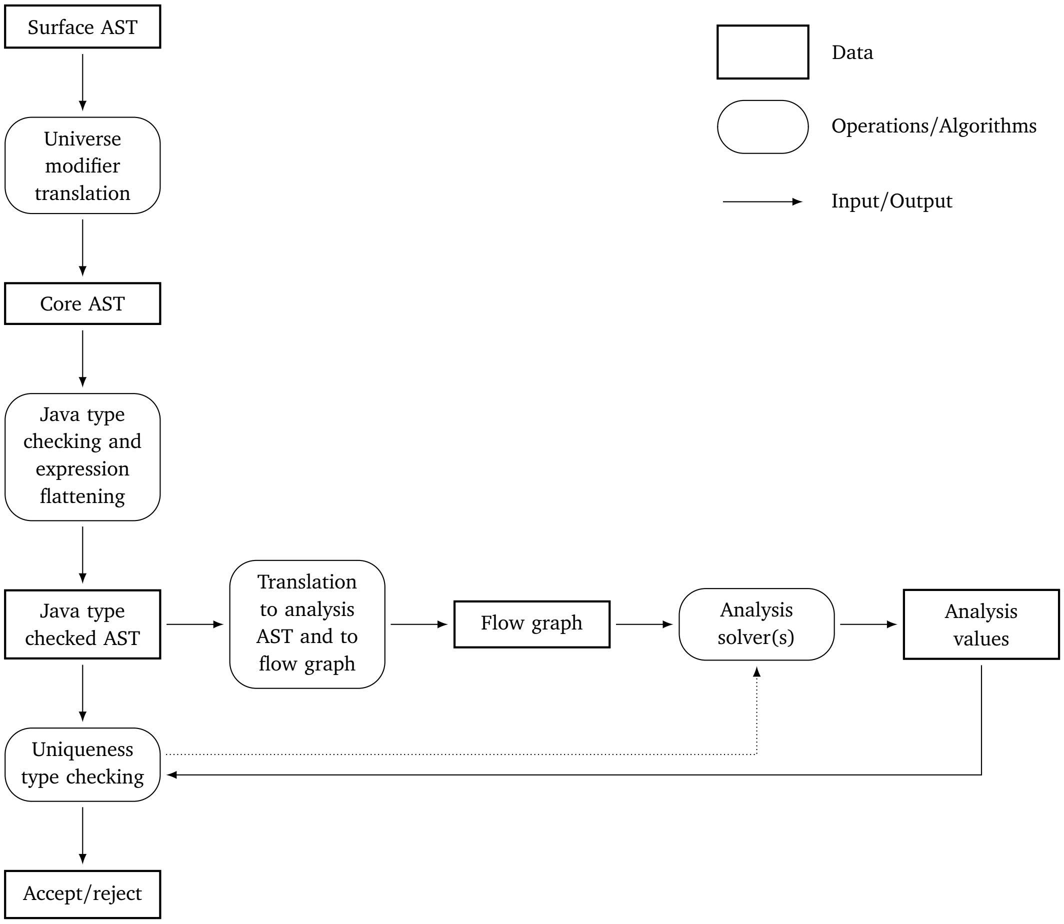


Figure 5.1: Schematic of the uniqueness type checking process implemented in Multi-Java. The dotted edge represents multiple analysis solver runs in conjunction with a hierarchical solver.

5.3.1 Parsing

A first change to the code affects the ANTLR grammar definition files, where the definition of possible Universe modifiers has been extended to include `uniq` and `free` (which are also declared as new keywords), as well as to support an optional identifier enclosed in brackets after a `rep` modifier. There are no additional checks related to the new Universe modifiers during parsing, so they are accepted wherever a Universe modifier is. The checks as to whether a certain use of a new Universe modifier is legal are performed after the parsing pass, as explained below.

5.3.2 Clusters

A cluster is represented by the newly added class `CUniverseRepCluster`. There is one `CUniverseRepCluster` instance per declared cluster (i.e., per field declared `uniq`), plus two instances that represent the this-cluster Cl_{this} and the wildcard cluster ?, respectively.

The `CUniverseRep` class represents a `rep` modifier and used to be a class with a singleton instance in the original implementation. Now, each `CUniverseRep` object maintains a reference to a `CUniverseRepCluster` object to represent a $rep\langle Cl\rangle$ modifier. Note that two `CUniverseRep` instances are defined to be equal iff their associated clusters are equal. This guarantees compatibility with the internal type cache mechanism of MultiJava (refer to the class `CTopLevel` for more detail).

5.3.3 Universe Modifier Translation

A new compiler pass of the name `InitUniverseUniquenessTask` has been added to be executed prior to the (Java) type checking pass. A general task name has been chosen to allow for possible future extensions without having to rename the task. Currently though, the only work performed by the task is resolving the clusters for *rep* modifiers for fields and and in method signatures, corresponding to the translation of the surface Universe modifiers to core modifiers as explained in Section 3.3. At the source level, this amounts to setting a reference from any `CUniverseRep` object to a `CUnverseRepCluster` object.

Note that processing needs to be done before type checking, since there might be a reference to a field or a call to a method that has not yet been type checked (e.g., if declared textually posterior to the reference or the call).

In case of illegal modifiers (e.g., a `free` modifier not in a method signature) or unresolvable clusters (e.g., a `rep[f]` modifier where field `f` is not declared `uniq` in the enclosing class), a corresponding error message is emitted.

The functionality is implemented in `initUniverseUniqueness( ... )` methods added to the relevant Java AST classes.

5.3.4 Java Type Checking and Expression Flattening

Some work related to uniqueness type checking has been integrated into the Java type check pass. To begin with, the clusters of the remaining *rep* modifiers (e.g., in local variable declarations) are resolved and checked for legal usage, in the same way as for fields and method signatures described in the previous subsection.

Expression Flattening. What is more, nested Java expressions are transformed into the form of statements in the toy language[1] (*flattened*) by introducing temporary variables. It is needless to say that the actual Java expressions are *not* modified by the flattening operation (this would be pointless), but the flattening is stored in association with the original expression. The subsequent uniqueness type checking pass (see below) then in fact works on the flattened expressions, and not on the original Java expressions (unless they are already in flattened form).

The expression flattening needs to be done since evaluating a (sub)expression may have side-effects on the analysis value. E.g., in the expression

```
m(x) == m(x)
```

the variable `x` might get unusable as result of the first, left-hand side method call (e.g., if the formal parameter is of type modifier `free`) and hence the read of `x` in the second, right hand side method call is illegal. Other examples that need to be flattened include

```
if ((x = y) == z) ...
```

or, in general, rather bizarre expressions such as

```
x.m((x = y).f.m().g) == ((y.i++ == 0) ? (y = z) : (y = x)).
```

[1] To be precise, a flattened statement may also contain an `if` statement due to the flattening of a conditional expression (`b ? e1 : e2`).

The base class of all classes representing Java expressions, `JExpression`, is extended to include a method `flatten( ... )` to perform the flattening. Let us further point out that no techniques are employed so as to minimize the number of additional temporary variables needed.

As a last remark, the expression flattening is integrated in the Java type checking pass because the flattening statements also need to be type checked in order to resolve bindings (i.e., the mapping from string names to nodes in the AST). To this end, the context objects used in the Java type checking pass can be reused.

5.3.5 Checking Uniqueness

The remaining uniqueness type checks that use data flow analysis information (e.g., whether is a certain variable is unusable) are performed in another newly created compiler pass, namely `CheckUniverseUniquenessTask`, which is to be run after the Java type checking pass. The functionality is implemented in `checkUniverseUniqueness( ... )` methods added to the relevant Java AST classes.

Translation to an Analysis AST

For each method declaration, the Java statement of the body is translated into a corresponding statement in the analysis language. This is accomplished by the method `getAnalysisStmt( ... )` that has been added to the relevant Java AST classes. Note that this translation (as well as running the analysis) is *not* performed in a separate pass, but within the `CheckUniverseUniquenessTask` pass before recursing into the method body.

Toy Language Statements. The Java statements that correspond to the statements of the toy language are translated into an elementary statement (or possibly a sequence of elementary statements) of the analysis language according to the analysis value transition rules given in Section 3.9. To give a concrete example, the translation of an assignment statement $x = y$ can be described as follows, where S stands for the resulting analysis statement:

$$\frac{S = \begin{cases} \texttt{consume(}y\texttt{)} & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{peer} \wedge \Gamma_{\mathrm{m}}(y) = \mathit{rep} \\ \texttt{move(}x\texttt{, }y\texttt{)} & \text{, if } \Gamma_{\mathrm{m}}(x) = \mathit{rep} \wedge \Gamma_{\mathrm{m}}(y) = \mathit{rep} \\ \texttt{skip} & \text{, otherwise} \end{cases}}{\Gamma \vdash x \texttt{ = } y : \mathtt{S}}$$

in perfect correspondance to the [L-Assing] rule. Note the latter rules also ensure that each variable occurring as argument of a transition function or of a statement in the analysis language, respectively, is indeed contained in **ALoc**∪**AClust**∪**AField**, as required by the analysis interface (cf. Section 4.3).

Composite Java Statements. Composite Java statements, which are not formalized in the toy language, are recursively mapped in a straight-forward way to the corresponding statements in the analysis language, if such a statement exists. For instance, a Java `if` statement is mapped to an analysis `if` statement, a Java `while` statement is mapped to an analysis `while` statment, a Java `switch` statement is mapped to an analysis `switch` statement and so on. As noted Section 4.1, a Java `for` loop is translated into an equivalent `while` loop, in contrast to a Java `do` loop, which is translated to an equivalent analysis `do` loop. Moreover, note that the Java expressions in conditions of `if` statements and the like, which have been flattened in the previous pass, are mapped to analysis *statements* (there is no such thing as an analysis expression), as mentioned in Section 4.1, too.

Break and Continue Statements. Labeled and unlabeled Java `break` and `continue` statements and labeled Java statements are translated to their respective counterparts in the analysis language.

Constructor Calls. A Java constructor call `new C(z)` is treated like `o = new C; o.m(z); o`, where `o` is the newly created object and the method `m` corresponds to the constructor. The `void` method call `o.m(z);` is handled according to the rules [L-Pre-Invk] and [T-Pre-Invk].

Return Statement. The translation of a Java `return` z statement to an analysis statement involves two steps. First the argument z is handled in the same way as for a method call `this.m(`z`)`, where the formal parameter of the imaginary method `m` has the same type as the return type of the current method, with the exception that there is no *ConsumeLocals* statement, as there is no actual method call. In addition to the handling of the argument, an `exit` statement is added to ensure the premature method termination is reflected in the flow graph.

To add a formal touch to the above explanation, the analysis transition rule for a `return` z statement can be described as follows, where the setup corresponds the rules described in Section 3.9:

$$[\text{L-Return}]\ \frac{\begin{array}{c} \left(W, T_p, T_r\right) = \text{“signature of the enclosing method”} \\ L' = \text{handleArg}\left(L, \texttt{this}, z, \mathit{this}, \Gamma_{\mathrm{m}}(z), \text{mod}\left(T_r\right)\right) \\ L'' = \text{“add an } \texttt{exit} \text{ statement to } L'\text{”} \end{array}}{\Gamma; L \vdash_{\mathcal{L}} \texttt{return}\ z : L''}$$

Note that we need reference to the enclosing method to obtain its signature. The intention to avoid such an unnecessary complication of the rules is in fact the reason why the `return` statement was not included in the toy language.

Throw Statement. Finally, a Java `throw` statement is handled similarly to a `return` statement, though the support for exceptions is still to be redesigned as future work (e.g., a desirable way would be to have the exceptions transferred into the context of the current handling method during propagation [10].

Performing the Type Checks

Once the abstraction to a statement in the analysis language is carried out, the corresponding flow graph is created in accordance to Section 4.2. Then, depending on the command line option, either an alias matrix analysis solver or a hierarchical analysis solver (cf. Subsection 4.10.1) is executed to solve the analysis.

If the analysis could successfully be solved (i.e., without being aborted due to an unsatisfied precondition of a transition function), the body of the method is recursed into to perform the type checks as described in Section 3.10, using the analysis values computed by the solver. Note that if a hierarchical solver is used, the analysis may be rerun with a solver of higher priority in case of an imprecise answer to a query. The dotted edge in Figure 5.1 represents these feedback loops.

Return Statement. For the sake of completeness, the following rule pseudo-formally describes the handling of a `return` z statement, in analogy to the translation rule [L-Return] given above:

$$\text{[T-Return]}\ \frac{\begin{array}{c}\left(W, T_p, T_r\right) = \text{“signature of the enclosing method”} \\ T_r :>_A \Gamma(z) \\ \mathit{isUnusable}(L, z) = 0 \\ L' = \text{handleArg}\big(L, \texttt{this}, z, \mathit{this}, \Gamma_{\mathrm{m}}(z), \text{mod}(T_r)\big) \\ W = \texttt{nonpure} \Rightarrow \forall f \in \text{fields}(C) :\ \mathit{isUnusable}\big(L', f\big) = 0\end{array}}{\Gamma; L \vdash \texttt{return}\ z}$$

Note that, in similarity to [L-Return], the rule can be seen as instance of the rule to type check a method call `this.m(`z`)`, where the formal parameter of the imaginary method `m` has the same type as the return type of the current method. In particular, this entails the required check as to whether the fields are unusable upon method termination (if the method is non-pure). In summary, this means that, unlike in the toy language formalization, the check whether the fields are unusable upon method termination is performed when type checking a `return` z statement. This handling is necessary to support premature method terminations in conjunction with `return` z statements.

5.3.6 Purity

For the time being, the method purity checker has been modified to

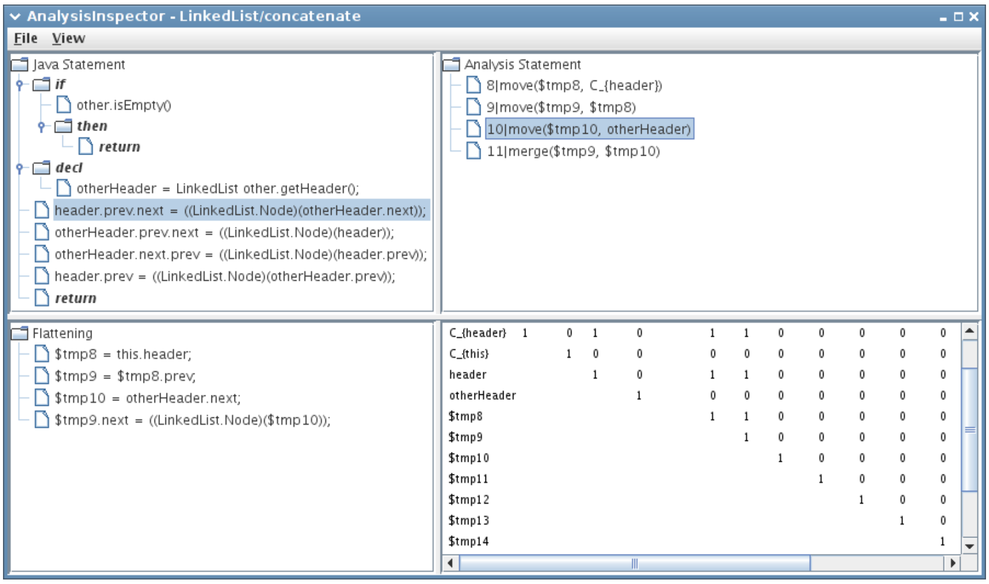
- allow `free` formal parameters in pure methods,
- allow non-pure method calls on something other than `this` with a `free` formal parameter and
- allow constructor calls with a `free` formal parameter.

These changes are motivated by the fact that, simply put, `free` parameters should be accepted (or can be returned) by any method. The implementation of a method purity checker in conjunction with ownership transfer is yet to be further investigated as future work.

5.3.7 Miscellaneous

For a more convenient debugging and to facilitate seeing behind the curtain, the distribution includes a graphical “analysis inspector” tool. The “analysis inspector” presents information such as the translated statement in the analysis language, the performed flattenings and the actually computed analysis values (i.e., the analysis solution) for each program point of the method being analyzed (cf. Figure 5.2 for a screenshot).

Currently, the analysis inspector tool can only be accessed when manually stepping through the code in debugging mode, since a MultiJava run always results in a call to `System.exit( ... )`, which causes the whole application to be terminated.



```
Java Statement
  if
    other.isEmpty()
    then
      return
  decl
    otherHeader = LinkedList other.getHeader();
  header.prev.next = ((LinkedList.Node)(otherHeader.next));
  otherHeader.prev.next = ((LinkedList.Node)(header));
  otherHeader.next.prev = ((LinkedList.Node)(header.prev));
  header.prev = ((LinkedList.Node)(otherHeader.prev));
  return
```

```
Analysis Statement
  8|move($tmp8, C_{header})
  9|move($tmp9, $tmp8)
  10|move($tmp10, otherHeader)
  11|merge($tmp9, $tmp10)
```

```
Flattening
  $tmp8 = this.header;
  $tmp9 = $tmp8.prev;
  $tmp10 = otherHeader.next;
  $tmp9.next = ((LinkedList.Node)($tmp10));
```

```
C_{header}  1  0  1  0  1  1  0  0  0  0  0
C_{this}       1  0  0  0  0  0  0  0  0  0
header            1  0  1  1  0  0  0  0  0
otherHeader          1  0  0  0  0  0  0  0
$tmp8                   1  1  0  0  0  0  0
$tmp9                      1  0  0  0  0  0
$tmp10                        1  0  0  0  0
$tmp11                           1  0  0  0
$tmp12                              1  0  0
$tmp13                                 1  0
$tmp14                                    1
```

Figure 5.2: Analysis inspector debugging tool.

5.4 Static Data Flow Analysis

In this section, some aspects regarding the implementation of the data flow analysis and, in particular, the representations of the various analysis values are highlighted

5.4.1 Partitions

In order to represent sets of partitions in regard of a partition set analysis solver, we first have to come up with an adequate data structure for a single partition that supports efficient implementations of the operations *merge*, *new* and *move* (cf. Section 4.4.1). Moreover, we need to be able to efficiently decide whether two elements are in the same block of the partition.

Union-Find Data Structures. A tried and trusted data structure to represent a single set partition is the classic union-find data structure. As implied by its name, a union-find structure offers two basic operations:

- Union: Unify two given blocks into a single block.
- Find: Return an identifier of the block a given element is contained in.

A union-find structure represents the elements of a block in a rooted tree, where the root acts as identifier for the block. The union operation combines – in $O(1)$ worst-case time – two trees by attaching the root of the tree of lower height to the root of the tree of larger height. This heuristic guarantees a logarithmic height of the resulting tree. The find operation follows the parent pointers all the way up to the root and returns the root as the identifier of the set. Applying one of various path compression techniques to shorten the path from the element to the root yields an amortized cost per find operation of $O(\alpha(n)) \approx O(1)$ for a set of n elements and α representing the inverse Ackermann function.

The union operation of a union-find data structure allows us to implement the *merge* operation efficiently. Further, we can efficiently check whether two given elements are in the same block with the help of the find operation. However, the *new* operation, which takes a given element and makes it a singleton block, is not supported by a classic union-find structure.

Union-Find-Delete Data Structures. To overcome this, we resort to an extension of a union-find structure that offers a constant time delete operation [2]:

- Delete: Remove a given element from the block it is contained in.

The proposed data structure still maintains the elements of a block in a rooted tree. In a standard union-find structure, each tree node corresponds to an element. In contrast, a tree node in the new data structure is either *occupied* (i.e., containing an element) or *vacant* (i.e., not containing an element). A delete operation marks the node that contains the element as vacant. Note that the node cannot simply be removed from the tree, since it may be the root of the tree, which acts as identifier of the block. To ensure that the tree does not contain too many vacant nodes (which might cause the space needed to store the tree and the time needed to process a find operation to become too large), the data structure uses a simple collection of local operations to tidy up the tree after a delete operation.

Using a union-find-delete structure, the *new* operation can be synthesized by deleting the element from the containing block and adding it back as singleton block. Both of these operations run in constant time. Since the *move* operation can be expressed by an *new* and an *merge* operation, a union-find-delete structure fulfills all of our needs.

5.4.2 Partition Sets

To maintain *sets* of partitions we can use a dictionary (e.g., a binary tree or a sorted list) or a hash set of single partitions (represented by union-find-delete structures). With a sorted list, we can perform the set operations in time linear in the sizes of the operand sets (à la merge sort). However, we naturally have to define a (total) order on union-find-delete structures, respectively. For hash sets, we need to come up with an adequate hash function.

Let us define a function that calculates the so-called *block finger print* of a given partition. It is defined in the following way: We impose a total ordering on the elements and for each unvisited element in order, we traverse all nodes of its tree in the union-find-delete structure, marking the elements with an increasing number, starting with 0. This corresponds to numbering the connected components of the forest in the union-find-delete structure by a graph traversal. Hence, calculating the block finger print of a partition can be done time linear in the number of elements.

Example 5.1 For an alphabetical element order, the following table shows the block finger prints for various partitions in Part($\{a, b, c, d, e\}$):

Partition	Block Finger print
$\{ac \mid be \mid d\}$	$[0,1,0,2,1]$
$\{abcd \mid e\}$	$[0,0,0,0,1]$
$\{ac \mid b \mid d \mid e\}$	$[0,1,0,2,3]$

□

The block finger print can be used to compare partitions by interpreting the values as digits of a number. A hash function can be generated out of the block finger print by using the hash function for `int[]` arrays provided by the Java utility class `Arrays` (since Java 1.5). As an additional optimization, the block finger print of a partition is cached in a field and only updated when necessary.

Empirical test runs (including the benchmarks shown in Section 4.11) revealed that a hash set implementation has a significantly better performance than an implementation with a sorted list. As a result, the current implementation solely employs hash sets of union-find-delete data structures.

As a concluding remark, in addition to the possible exponential size of a partition set to begin with, the linear time required to compute the hash function of a partition seems to be the most prominent factor that negatively impacts the performance of a partition set backed by a hash set.

5.4.3 Alias Matrices

An alias matrix is implemented as a array of *trit sets*. A trit set represents a sequence of values in $\mathbb{T}$ and is the analogue of a bit set, which represents a sequence of values in $\{0,1\}$. In turn, a trit set is implemented by two bit sets, mapping $\mathbb{T}$ to $\{0,1\}^2$. The bit sets use 64 bit chunks to represent the bit values. For a fixed mapping of $\mathbb{T}$ to $\{0,1\}^2$, the trivalent “and”, “or” and “join” operations can be equivalently expressed by bivalent “and” and “or” operations. What is more, the *Merge* operation on alias matrices can be efficiently implemented by only *one* bivalent operation (as opposed to the equations given in (4.21a)-(4.22c)). Please refer directly to the source code for more detail.

5.4.4 Analysis Solver

The implementations of the basic worklist solver and the various worklist models correspond to a large extent to their descriptions given in Chapter 4. As mentioned in Subsection 4.7.3, the current implementation uses an SCC worklist, applying Tarjan's algorithm to compute the SCCs as part of the loop tree creation.

As an optimization in the worklist solver iteration, the test whether the resulting analysis value will change and the join operation $\sqcup$ (cf. lines 9 and 8 of Algorithm 4.1) are combined into one single operation, namely a join operation that returns true iff the value changed as a result of the operation. This avoids some extra equality tests on analysis values.

5.4.5 Package Organization

For reference, Table 5.1 gives a summary of the package organization of the classes implementing the static data flow analysis.

5.5 Testing

To emphasize robust code, there is a junit test case in the `org.multijava.universes.uniqueness.analysis.testing` package for every newly added class that is to perform some non-trivial operations . Moreover, the directory `org/multijava/mjc/testcase/universe/uniqueness` contains a number of Java source test files that are integrated into the MultiJava testing framework where each source file is compiled and the compiler output is checked against an expected string.

5.6 Code Examples

Appendix A lists some examples making use of ownership transfers that can be successfully expressed in the presented type system.

5.6.1 Abstract Factory Pattern

The Abstract factory design pattern (A.1) can easily be type set in the model. Note the `readonly` reference from the client to the factory (and so the purity modifier of the product creation method) as well as the implicit release and capture operations.

Package Name	Description
`analysis`	General interfaces and abstractions, including exception classes
`analysis.graph`	Flow graph related classes, including a loop tree representation and an SSC computer (implementing Tarjan's algorithm)
`analysis.solvers`	Abstract worklist solvers and worklist implementations (standard worklist, reverse postorder worklist, SSC worklist)
`analysis.solvers.aliasmatrix`	Alias matrix solver, including trit set and bit set representations
`analysis.solvers.partitionset`	Partition set solver and minimized partition set solver, including a union-find-delete implementation
`analysis.solvers.singlepartition`	Solver using a single partition set (abandoned approach)
`analysis.statements`	AST nodes for analysis language statements
`analysis.util`	Utility classes
`analysis.testing`	Unit test cases (mirrors the above directory structure) and benchmarks

Table 5.1: Package organization of the classes implementing the static data flow analysis. The package name is to be prefixed with `org.multijava.universes.uniqueness`.

5.6.2 List Representation Merging

The list representation merging is demonstrated in A.2. The list implementation is modeled after the one in the Java library, using a dummy node `header` and being cyclicly arranged. The capture operation in the concatenation method corresponds to the Factory pattern example.

Note that we need a special method `release( ... )` to release the `header` node in the `getHeader()` method. The reason for this lies in the fact that the type system enforces a one-to-one correspondence between *static clusters* (a set of fields) and *dynamic clusters* (a set of objects), which, in this example, is enforced by the additional *Merge* operation in the rule [T-Field-Write]. In other words, the `header` field is always associated with the cluster *cluster*(`header`). Consequently, if the commented assignment statement

```
rep Node result = header;
```

was used instead of the `release` method, the `header` field would still be marked as unusable, since the variable `result`, which points into the `header` cluster, is marked as unusable due to the `return` statement. Moreover, note that the release method has to be pure, for, otherwise, the unusability of the `header` field before the `peer` call to the `release` method would trigger an error.

The `release( ... )` method in the Decorator pattern example (A.4) has the same raison-d'être.

While this minor limitation appears to be a little grain of salt, it will be lifted in the future.

5.6.3 Construction and Traversal of a rep Tree

Thanks to ownership transfer mechanism, a `rep` tree (A.3) can be conveniently built up using the constructor or the setter methods for the children, which may already have been created. Without support for ownership transfer solely the parent node could create its children.

Note that the linked list in the traverse method is accepted as argument, modified and returned again. After a recursive call, we need to re-assign the returned list in order to capture it back. The use of *borrowed* parameters [5] to only temporarily grant write access rights to an argument may render this additional assignment unnecessary. The Composite design pattern (with a deep ownership structure) could be expressed according to the rep tree example.

5.6.4 Decorator Design Pattern

The Decorator design pattern (A.4) combines the capture techniques from the rep tree example (to add a responsibility to the decoree) and the release mechanisms from the linked list example (to remove a responsibility from the decoree).

5.6.5 Visitor Design Pattern

The Visitor design pattern (with a deep ownership structure of the elements to be visited and a modifying visitor) fails to be expressed in the type system, no matter if the traversal order is determined by the visitor (a) or by the visited element (b), cf. the code fragments below. In case (a), the resulting stack references can be perceived as mutual read-write references between the visitor and the visited elements that essentially cause all elements to be in the same ownership context. In case (b), the failure can be attributed to the peer nature of the this reference that is passed as argument to the visitor callback method.

```
class ConcreteVisitor1 implements Visitor {
    void visitConcreteElement1(free ConcreteElement1 e) {
        // modify e
        e.getChild().accept(this); // (a) traversal triggered by the
                                   // visitor
    }
    ...
}

class ConcreteElement1 implements Element {
    void accept(free Visitor v) {
        v.visitConcreteElement1(this); // callback
        getChild().accept(v); // (b) traversal triggered by the element
    }
    ...
}
```

As with the traversal methods in the `rep` tree example, support for borrowed parameters might help solve the problem.

6

Conclusion and Future Work

6.1 Conclusion

A flexible ownership transfer model for the Universe type system has been implemented that is able to express the usual examples, including the Abstract Factory design pattern, the Composite design pattern and the Decorator design pattern, as well as the merging of linked list representations. In particular, the solution involves no destructive reads and no additional specification overhead. There are, however, some minor limitations in the expressibility (e.g., in case of the list merge example or the Visitor design pattern). The implementation handles the usual Java constructs.

6.2 Future Work

The following compilation addresses possible extensions to the project to be carried out as future work.

Arrays. Due to time constrains regarding this project, the support for arrays is yet to be implemented. One way to achieve this is to treat array element accesses like field accesses.

Performance Evaluation. Is the alias matrix data flow analysis solver precise enough for practical examples? How about the performance? Case Studies?

Runtime Support. Design and implementation of a runtime model including, in particular, an ownership transfer function. Moreover, it has to be investigated when implicit cluster releasings and capturings ought to happen.

Complete Type Inference for Local Variables. Design and implementation of a complete type inference for local variables such that local variables do not need to be annotated with a Universe modifier anymore.

Static vs. Dynamic Clusters. Investigate into the issue of a strict correspondence between static and dynamic clusters, aiming at getting rid of the additional `release` methods, e.g., in the list merging example.

Borrowed Parameters. Investigate if a support for borrowed parameters would help express such cases as the Visitor design pattern.

Exception Handling Model. How should exceptions be handled? Should they be transferred into the context of the handler during propagation? What modifiers can be used to declare exceptions?

At the moment of writing, the design and implementation of a runtime model as well as the complete type inference for local variables are already being realized as part of other master's theses.

Acknowledgments. The author would like to thank Arsenii Rudich and Peter Müller as well as the whole SCT group for their contributions, support and a very enjoyable collaboration.

Bibliography

[1] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. pages 1–25.

[2] S. Alstrup, I. L. Gørtz, T. Rauhe, M. Thorup, and U. Zwick. Union-find with constant time deletions. In *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP 2005). Lecture Notes in Computer Science*, volume 3580, pages 78–89. Springer-Verlag, jul 2005.

[3] Henry G. Baker. üse-oncev̈ariables and linear objects: storage management, reflection and multi-threading. *SIGPLAN Not.*, 30(1):45–52, 1995.

[4] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, Louisiana, January 2003.

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

[6] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[7] D. Clarke and T. Wrigstad. External uniqueness is unique enough, 2003.

[8] Curtis Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, November 2001. Available from archives.cs.iastate.edu.

[9] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3), May 2006.

[10] W. Dietl and P. Müller. Exceptions in ownership type systems. In E. Poll, editor, *Formal Techniques for Java-like Programs*, pages 49–54, 2004.

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

[12] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, 2001.

[13] John Hogg. Islands: aliasing protection in object-oriented languages. 26(11):271–285, November 1991.

[14] Susan Horwitz, Alan Demers, and Tim Teitebaum. An efficient general iterative algorithm for dataflow analysis. *Acta Inf.*, 24(6):679–694, 1987.

[15] Jonas Lundberg and Welf Löwe. A scalable flow-sensitive points-to analysis. In *Compiler Construction - Advances and Applications, Festschrift on the occasion of the retirement of Prof. Dr. Dr. h.c. Gerhard Goos*. Springer Verlag, 2007. accepted.

[16] Naftaly H. Minsky. Towards alias-free pointers. In *ECCOP '96: Proceedings of the 10th European Conference on Object-Oriented Programming*, pages 189–209, London, UK, 1996. Springer-Verlag.

[17] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[18] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[19] Stefan Nägeli. Ownership in design patterns. Master's thesis, ETH Zurich, 2006.

[20] N. J. A. Sloane. Sequence a000110—bell numbers. Available from `http://www.research.att.com/~njas/sequences/`, 2007. On-Line Encyclopedia of Integer Sequences.

[21] R. E. Tarjan. Depth first search and linear graph algorithms. 1(2):146–160, June 1972.

[22] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

A
Code Examples

A.1 Abstract Factory Design Pattern

The corresponding source file can be found at `org/multijava/mjc/testcase/universes/uniqueness/AbstractFactory.java` in the MultiJava source tree.

```java
/**
 * Testing uniqueness and ownership transfer in the Universe type
 * system: Abstract Factory design pattern example.
 * <p>
 * Expected result: no errors
 *
 * @author ytakano
 */
abstract class AbstractFactory {

    public static readonly AbstractFactory getFactory() {
        if (true) {
            return new peer Factory1();
        } else {
            return new peer Factory2();
        }
    }

    public abstract pure free Product createProduct();

}

class Factory1 extends AbstractFactory {

    public pure free Product createProduct() {
        return new rep Product1(); // release
    }

}

class Factory2 extends AbstractFactory {

    public pure free Product createProduct() {
        return new rep Product2(); // release
    }

}

interface Product {

    public int getId();

}

class Product1 implements Product {

    public pure Product1() {}

    public int getId() { return 1; }

}

class Product2 implements Product {

    public pure Product2() {}

    public int getId() { return 2; }

}

class AbstractFactoryClient {

    private rep Product product;

    public AbstractFactoryClient() {
        readonly AbstractFactory factory = AbstractFactory.getFactory();
        product = factory.createProduct(); // capture
    }

    public void printProductId() {
        System.out.println(product.getId());
    }

    public static void main(String[] args) {
        AbstractFactoryClient client = new AbstractFactoryClient();
        client.printProductId(); // 1
    }

}
```

A.2 Merging List Representations

The corresponding source file can be found at `org/multijava/mjc/testcase/universes/uniqueness/LinkedList.java` in the MultiJava source tree.

```
/**
 * Testing uniqueness and ownership transfer in the Universe type
 * system: linked list implementation.
 * <p>
 * Expected result: no errors
 *
 * @author ytakano
 */

public class LinkedList {

    protected uniq Node header;

    public LinkedList() {
        header = new rep Node(null, null, null);
        header.next = header.prev = header;
    }

    public void addFirst(readonly Object o) {
        addBefore(o, header.next);
    }

    public void addLast(readonly Object o) {
        addBefore(o, header);
    }

    protected void addBefore(readonly Object o, rep[header] Node node) {
        rep Node newNode = new rep Node(o, node, node.prev);
        newNode.prev.next = newNode;
        newNode.next.prev = newNode;
    }

    public boolean isEmpty() {
        return header.next == header;
    }

    protected pure free Node release(free Node x) { return x; }

    protected free Node getHeader() {
        // rep Node result = header;
        rep Node result = release(header); // release

        header = new rep Node(null, null, null);
        header.next = header.prev = header;

        return result;
    }

    public void concatenate(peer LinkedList other) {
        if (other.isEmpty()) {
            return;
        }

        rep Node otherHeader = other.getHeader();

        header.prev.next = otherHeader.next;
        otherHeader.prev.next = header;

        otherHeader.next.prev = header.prev;
        header.prev = otherHeader.prev;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("[␣");

        readonly Node current = header.next;
        while (current != header) {
            // the cast to peer is needed since there is no method
            // StringBuffer#append(readonly Object)
            buffer.append((peer Object) current.element).append("␣");
            current = current.next;
        }

        buffer.append("]");
        return buffer.toString();
    }

    private static class Node {

        readonly Object element;

        peer Node next;

        peer Node prev;

        Node(readonly Object element, peer Node next, peer Node prev) {
            this.element = element;
            this.next = next;
            this.prev = prev;
        }

    }

    public static void main(String[] args) {
        LinkedList l1 = new LinkedList();
        l1.addLast(new Integer(1));
        l1.addLast(new Integer(2));
        l1.addLast(new Integer(3));
        System.out.println(l1); // [ 1 2 3 ]

        LinkedList l2 = new LinkedList();
        l2.addLast(new Integer(4));
        l2.addLast(new Integer(5));
        System.out.println(l2); // [ 4 5 ]

        l1.concatenate(l2);
        System.out.println(l1); // [ 1 2 3 4 5 ]
        System.out.println(l2); // [ ]
    }

}
```

A.3 rep Tree Construction and Traversal

The corresponding source file can be found at `org/multijava/mjc/testcase/universes/uniqueness/Tree.java` in the MultiJava source tree.

```java
/**
 * Testing uniqueness and ownership transfer in the Universe type
 * system: rep tree construction and traversal.
 * <p>
 * Expected result: no errors
 *
 * @author ytakano
 */

class Tree {

    private readonly Object element;

    private rep Tree left;

    private rep Tree right;

    public Tree(readonly Object element) {
        this(element, null, null);
    }

    public Tree(readonly Object element, free Tree left, free Tree right) {
        this.element = element;
        this.left = left; // capture
        this.right = right; // capture
    }

    public void setLeft(free Tree left) {
        this.left = left; // capture
    }

    public void setRight(free Tree right) {
        this.right = right; // capture
    }

    public readonly LinkedList getElements() {
        rep LinkedList l = new rep LinkedList();
        return collectElements(l);
    }

    // we use a LinkedList to support adding readonly Objects
    free LinkedList collectElements(free LinkedList l) {
        if (left != null) {
            l = left.collectElements(l);
        }

        l.addLast(element); // inorder traversal

        if (right != null) {
            l = right.collectElements(l);
        }

        return l;
    }

}

class TreeClient {

    TreeClient() {
        Tree t1 = new Tree(new Integer(2));
        t1.setLeft(new rep Tree(new Integer(1)));
        t1.setRight(new rep Tree(new Integer(3)));

        readonly LinkedList l1 = t1.getElements();
        System.out.println((peer LinkedList) l1); // [ 1 2 3 ]

        Tree t2 = new Tree(new Integer(4),
                           new rep Tree(new Integer(2),
                                        new rep Tree(new Integer(1)),
                                        new rep Tree(new Integer(3))
                           ),
                           new rep Tree(new Integer(6),
                                        new rep Tree(new Integer(5)),
                                        null
                           )
        );

        readonly LinkedList l2 = t2.getElements();
        System.out.println((peer LinkedList) l2); // [ 1 2 3 4 5 6 ]
    }

    public static void main(String[] args) {
        new TreeClient();
    }

}
```

A.4 Decorator Design Pattern

The corresponding source file can be found at `org/multijava/mjc/testcase/universes/uniqueness/Decorator.java` in the MultiJava source tree.

```java
/**
 * Testing uniqueness and ownership transfer in the Universe type
 * system: Decorator design pattern example.
 * <p>
 * Expected result: no errors
 *
 * @author ytakano
 */

class Component {

    private int x;

    public void setX(int x) { this.x = x; }

    public pure int getX() {
        return x;
    }

}

class Decorator {

    private uniq Component component;

    public Decorator(free Component component) {
        this.component = component; // capture
    }

    private pure free Component release(free Component x) { return x; }

    public free Component getComponent() {
        rep Component result = release(component); // release
        component = null;
        return result;
    }

    public void clearX() {
        component.setX(0);
    }

    public pure int getX() {
        return component.getX();
    }

}

class DecoratorClient {

    DecoratorClient() {
        rep Component c = new rep Component();
        c.setX(18);
        System.out.println(c.getX()); // 18

        // add responsibility
        Decorator d = new Decorator(c);
        d.clearX();
        System.out.println(d.getX()); // 0

        // remove responsibility
        c = d.getComponent();
        c.setX(99);
        System.out.println(c.getX()); // 99
    }

    public static void main(String[] args) {
        new DecoratorClient();
    }
```

```

}
```