Error Reporting for Universe Types with Transfer

Benjamin Lutz

Semester Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

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

May 2008

Supervised by:
Arsenii Rudich
Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

The Universe Type System is a type system for use in object-oriented programming, whose purpose it is to model relationships between objects in more detail. It introduces the concept of object ownership, through which object modification is controlled. With the universe type system, it is possible to catch programming errors such as unintended concurrent data modification from different objects within a program at compile time. The Universe Type System has been implemented in the MultiJava compiler.

The Universe Type System has in previous work been extended to allow transferring of objects from one owner to another. While this feature is very useful, and indeed, required for many problems, it can make analyzing programming mistakes difficult. This work tries to assist with that analysis by improving the information that is returned by the MultiJava compiler if there is a universe type error. Is does this by providing a backtrace through a program, similar in appearance to a classic framestack backtrace, that explains the circumstances that lead to the type error.

The backtrace method relies mostly on information already required by the existing Universe Type System analysis, although some of the data structures have been extended. Care has been taken so that the memory scalability of the MultiJava compiler does not become notably worse, i.e. that it is still quadratically bound.

Contents

1. Introduction

This chapter first gives a brief overview over the Universe type system and its extension with ownership transfer, which serve as the basis for this work. They are explained in much more detail in [1].

1.1. The Universe Type System

The *Universe type system* is a type system which models ownership relation between objects, with the ultimate goal of restricting modification of objects. In the Universe type system, references are extended with a universe type, which is orthogonal to the standard object type. The following universe types are available to express the ownership relationship between an object x holding a reference to an object o:

rep expresses that x is the owner of o,

peer expresses that x has the same owner as o, and

any[1] expresses that o may have any owner; x may thus not modify o.

A **rep** reference is essentially the same as a standard reference as it is known from many programming languages. As the owner of o, x may call any method and access any field of o (of course, visibility restrictions still apply).

If x has a **peer** reference to o, x may also call any method and access any field of o, even though it does not own o. Having **peer** references is useful with data structures, in which the objects that make up the data structure can modify each other. This allows such a data structure to change it's internal structure, even though it is owned by another object that is unaware of that internal structure and therefore unable to change it.

Last, if x has an **any** reference to o, x may only call side-effect free (*pure*) methods of o, and it may read (but not write to) fields of o (again, within the existing visibility constraints.)

The example in listing 1.1 demonstrates the use of universe types by showing how they can be applied to a very common data structure, the linear linked list.

1.2. Universe Types with Transfer

In previous work [1, 2], the Universe type system was extended to allow ownership of an object to be transferred. This extended version of the Universe type system is called *Universe Types with Transfer*, or *UTT*. It adds several new concepts: uniqueness, clusters and the actual ownership transfer mechanism.

[1] in previous texts and some tools, `readonly` was used in place of **any**

```
1  class LinkedList {
2      rep Node first = null;
3
4      void append(any Object o) {
5          if (first == null) {
6              first = new Node();
7              first.element = o; // this owns first, so modification is allowed
8          } else {
9              Node node = first;
10             while (node.next != null) {
11                 node = node.next; // node.next is a peer of node/first, so this owns node.next too...
12             }
13             node.next = new Node(); // ... and may therefore modify it
14             node.next.element = o;
15         }
16     }
17 }
18
19 class Node {
20     any Object element = null;
21     peer Node next = null;
22 }
```

Listing 1.1: Linked List Example

1.2.1. Uniqueness

The idea of using a *uniqueness invariant* to control aliasing of an object has been previously proposed and discussed in several works [3–6]. A uniqueness invariant, in it's purest form, expresses that an object may only have a single reference pointing to it. A unique reference must therefore either be **null**, or it must be the sole reference to an object. In the context of the Universe type system this definition is relaxed slightly: only **rep** and **peer** references are considered when determining uniqueness, since an object may not be modified through **any** references.

Note that we only consider uniqueness as an attribute of references to objects; we do not statically declare the objects themselves as unique. Therefore, the uniqueness invariant applies to an object only for as long as it is referenced by a unique reference.

In order to maintain the uniqueness invariant, the semantics of reading a reference must be changed. Consider what happens when a unique reference is copied into another reference. One possible way to handle the situation is the implementation of *destructive reads*: at the moment an unique reference is read, it is atomically set to null. The new reference may or may not actually be unique; if it is not, the uniqueness invariant no longer applies to the referenced object.

While destructive reads prevent unique reference aliasing and are relatively easy to understand, the change in the semantics of the programming language is severe. Another solution is known as *alias burying*. It allows unique references to be aliased freely, however, as soon as the unique reference is dereferenced, the aliases may no longer be used; this is enforced by marking them as *unusable*. The slightly strange name comes from the following image: if an alias is for the last time used sometime before the moment when a unique reference is first used after the creation of that alias, it is at that moment dead, and may be buried safely (marked unusable). It is a programming error to read (for dereferencing or aliasing[2]) an unusable reference.

[2] while it is not strictly required to forbid aliasing an unusable reference, such an action would serve no purpose, as the

```
1 class ClusterExample {
2     uniq   Object x; // create a new cluster, C_x, and have x point into it
3     rep[x] Object y; // y points into C_x too
4 }
```

Listing 1.2: Defining Clusters

1.2.2. Clusters

To allow some design patterns, it is useful to split the representation of an object into distinct *clusters*. The factory pattern is a good example for this: when a factory has created a new object x, ownership of x is passed away from the factory, and the factory should no longer hold any **rep** references to x, its representation, or any **peer** objects of x. Such behavior is possible by splitting the representation of the factory into disjoint clusters, one of which contains x and the peers of x.

The notion of unique references is extended: it applies no longer just to a single object, but to the whole cluster which contains the object. Thus, if the factory passes ownership of x away, and the new owner holds a unique reference to x, the factory no longer holds a **rep** or **peer** reference to x or the peers of x. In this way, clusters become the unit of ownership transfer, and allow (and enforce for all objects in the same cluster) several objects to be transferred simultaneously.

Declaration

Clusters are declared in two ways. They may be defined explicitly using the **uniq** keyword with a field definition, as shown in listing 1.2. This will create a new cluster and place the field inside it. The **uniq** keyword may only be used for field definitions. Note that **uniq** references are not a new type of reference, they are also **rep** references. We shall label the cluster defined by the **uniq** field x C_x

An invariant condition applies to classes containing **uniq** references: no two **uniq** references may point into the same cluster.

The second way to create clusters is implicit: Whenever a new object is created and assigned to a local variable, a new, anonymous cluster is created for that object; the new cluster may then be merged into an existing cluster.

Referencing

Once a cluster C_x has been defined, other fields may be placed inside it. by declaring them as being of type **rep**[x]. The implication is that only the names of fields declared as **uniq** may be used as parameter of the **rep**[x] keyword. Only the names of **uniq** fields declared in the enclosing class (not a superclass) may be used for **rep**[x] declarations; this restriction exists to facilitate modular checking. The **rep**[x] keyword may be used for field declarations and in method signatures.

Every object contains a special cluster, the **this**-cluster. The **this**-cluster contains all fields declared with plain **rep**.

No universe type specifier needs to be given for local variable declarations; their universe type and the cluster they refer to is statically inferred.

created alias would be unusable too

It is allowed to have several **rep**-references to objects in the same cluster, as long as there is only a single **rep**-reference when the cluster is transferred to a different owner. References which have been marked unusable by this point are ignored.

Operations

Clusters support three types of operation:

MakeNew Clusters may be created inside of method bodies by creating a new object and storing it in a local variable; a new, anonymous cluster is created for it. In most cases, the cluster will then be merged with an existing cluster.

Merge Two clusters may be merged, unless they are **this**-clusters.

Move An object may be moved from one cluster to another. With this operation, the **this**-cluster may be modified.

These operations suffice to add and remove objects from clusters, and to transfer clusters. When two clusters with different owners are merged, one owner will give up ownership; this mechanism for ownership transfer is described in more detail below.

Ownership Transfer

Transferring of a cluster to a new owner happens in a two step release-capture process. It can happen implicitly with an assignment:

- `x = y`, where `x` is of type **peer**and `y` is of type **rep**`[g]`: The cluster pointed into by `y` is released and then captured by the owner of **this**. This means that the cluster pointed into by `y` is merged into the cluster in which **this** (and `x`) resides.
- `x.f = y`, where `x` is of type **rep**`[g]`, `f` is of type **peer** and `y` is of type **rep**`[h]` with $C_g \neq C_h$. Again, the cluster pointed to by `y` is released; the cluster is then captured into the the cluster of `x`.

Ownership transfer may also happen in a more explicit way: the old owner release a cluster by passing it as a **free** parameter to a method, or returning it as a **free** return value from one. The keyword **free** specifies a unique **rep**-reference into a released cluster, which may be captured. **free** may only be specified in method signatures.

The new owner captures a **free**-reference by assigning it to a field, which merges the cluster than is being transferred into another cluster.

The reference which was passed as **free** parameter, or returned as **free** return value, is marked unusable.

1.3. UTT Implementation and Project Goal

In previous work, UTT has been implemented in a modified version of the MultiJava compiler. The standard MultiJava compiler already supports universe types; in the modified version, support for ownership transfer based on clusters and alias burying, as outlined in the previous sections, has been added. These modifications include a static (compile-time), intraprocedural data flow analysis which tracks the universe types of local variables, including the cluster they point to.

Note: further references to "MultiJava" in this text shall always refer to the modified MultiJava version, which served as platform for this thesis.

The results of this thesis will also be implemented in MultiJava, specifically in the version inherited from Annetta Schaad [2]; she in term inherited the code base from Yoshimi Takano [1].

1.3.1. Motivation

Experimentation with the ownership transfer features of MultiJava has created the wish for better analysis tools when it comes to universe type errors. MultiJava by default gives rather terse descriptions of universe type errors. For example, the following program:

```
1  class Consumer {
2      void consume(free Object o) { }
3  }
4
5  class Example {
6      uniq Object a;
7      rep Consumer consumer;
8
9      void error() {
10         consumer.consume(a);
11
12         Object l = a;
13     }
14 }
```

Will produce the following error message by default:

```
File "Example.java", line 12 error: Field "a" is unusable. [Universes (Uniqueness)]
```

This project tries to provide more information about such errors by providing a backtrace through the method, outlining the ownership relations that led to to the errors in the hope that this information will be useful to the programmer in understanding and fixing the type errors.

In the above example, such a backtrace would look like this[3]:

```
Backtrace:
    File "Example.java", line 12, move(variable "l", field "a")
        [cluster "unusable" = field "a"]
    File "Example.java", line 10, merge(field "a", ANY/UNUSABLE)
        []
```

The backtrace says: `a` is unusable because it has been merged with the `unusable`-cluster.Of course, this is a rather trivial example, where the error is easily spotted without a backtrace. Listing 1.3 shows a more complex example, where it's more difficult to pinpoint the error. The corresponding backtrace is shown in listing 1.4, and is to be understood as follows:

- `a` is unusable, because
- `f` has been merged with the unusable cluster, and `a` and `f` were in the same cluster before that, because

[3]The backtrace has been redacted slightly; the lines related to expression flattening were removed. The full backtrace is shown in appendix B

- `e` and `d` have been merged, and `a` and `d` as well as `e` and `f` were in the same cluster before that, because
- `b` and `c` have been merged, and `a` and `b`, `c` and `d` as well as `e` and `f` were in the same cluster before that, because
  - there is a path in which `a` and `b` have been merged,
  - there is a path in which `c` and `d` have been merged,
  - there is a path in which `e` and `f` have been merged

  and these paths are joined together.

Fixing the error can happen in any line that the backtrace mentions. For example, the programmer might realize that in line 20, he meant to write `a.f = a`, which fixes the type error. Alternatively, he might change line 28 to `d.f = a`, which fixes the error too.

1.3.2. Project Goal

The goal of this thesis is to develop an algorithm for producing error backtraces, whose memory consumption is quadratically bound, and to implement it in the MultiJava compiler.

```
1  class Consumer {
2      void consume(free Object o) { }
3  }
4
5  class Foo {
6      peer Object f;
7  }
8
9  class Example {
10     uniq Foo a;
11     uniq Foo b;
12     uniq Foo c;
13     uniq Foo d;
14     uniq Foo e;
15     uniq Foo f;
16     rep Consumer consumer;
17
18     void error(boolean t, boolean u, boolean v) {
19         if (t) {
20             a.f = b;
21         } else if (u) {
22             c.f = d;
23         } else if (v) {
24             e.f = f;
25         }
26
27         b.f = c;
28         d.f = e;
29
30         consumer.consume(f);
31
32         Object l = a;
33
34         a = new Foo();
35         b = new Foo();
36         c = new Foo();
37         d = new Foo();
38         e = new Foo();
39         f = new Foo();
40     }
41 }
```

Listing 1.3: complex introductory example

```
Backtrace:
    File "Example.java", line 32, move(variable "l", field "a")
        [cluster "unusable" = field "a"]
    File "Example.java", line 30, merge(field "f", ANY/UNUSABLE)
        [field "f" = field "a"]
    File "Example.java", line 28, merge(field "e", field "d")
        [field "f" = field "e" ^ field "d" = field "a"]
    File "Example.java", line 27, merge(field "c", field "b")
        [field "f" = field "e" ^ field "d" = field "c" ^ field "b" = field "a"]
    JOIN:
    /
    |     [field "b" = field "a"]
    | File "Example.java", line 20, merge(field "b", field "a")
    |     []
    \
    /
    |     [field "d" = field "c"]
    | File "Example.java", line 22, merge(field "d", field "c")
    |     []
    \
    /
    |     [field "f" = field "e"]
    | File "Example.java", line 24, merge(field "f", field "e")
    |     []
    \
```

Listing 1.4: backtrace for the example in listing 1.3

2. Simple Graph Backtracking

This chapter shows how a universe type error can be backtraced through simple programs that do not contain branches or loops. We will use it to introduce several key concepts, which serve as the basis for later chapters.

2.1. Flow Analysis Overview

The universe type checking method implemented in the MultiJava compiler employs a code flow analysis to find type errors in programs. As it traces a method, it builds a flow graph representing the discrete steps in a method's execution. The analysis calculates the *cluster state* for each node in the graph. In the following work, we will always look at only a single method, and not at the interaction between individual methods. We will use the terms *method* and *program* as synonyms.

In his Master's thesis [1], Takano evaluates three different algorithms for flow analysis. Only one of them, the alias matrix-based one, was chosen for enhancement in this work, as the other two algorithms can be inefficient; their complexity and memory requirements may be exponential. The alias matrix-based flow analysis is relatively memory efficient: the memory requirements are in $O(mn^2)$, where m is the size of the method under analysis, and n is the number of references used in a method; the algorithm complexity is quadratically bound too. It is a goal of this work to not worsen the scalability of the memory requirements with the introduction of new features, and as often as possible to rely on data structures already created by the data flow analysis.

A cluster state is a data structure containing for each pair of reference variables that exist at the given point in the method, information about whether they point into the same cluster. To handle branching and looping, tri-state logic is used: the question of whether reference x and reference y are in the same cluster may have the answer `TRUE`, `FALSE` or `UNKNOWN`[1] at a given point in the method. We will use the following syntax in this text to describe such relations:

- $x =_C y$: x and y are in the same cluster, or $(x =_C y) =$ `TRUE`,
- $x \neq_C y$: x and y are not in the same cluster, or $(x =_C y) =$ `FALSE`,
- $x \stackrel{?}{=}_C y$: x and y may or may not be in the same cluster, or $(x =_C y) =$ `UNKNOWN`.

The situation where $x \stackrel{?}{=}_C y$ may appear after two branches rejoin: If at the end of one branch, $x =_C y$ is true, while at the end of the other branch, $x \neq_C y$ is true, these two relations will be merged into $x \stackrel{?}{=}_C y$ at the point where the branches join. This behavior is defined by the alias matrix-based flow analysis algorithm. Other algorithms are feasible that have no need for a $\stackrel{?}{=}_C$ relation; in [1] two more methods are presented, and it is shown that the alias matrix method is the most practical of them. In the rest of this work, we will always assume an alias matrix-based flow analysis.

In addition to the references defined by the program, the flow analysis adds markers to the cluster state to identify special clusters.:

[1] The value `UNKNOWN` is also called `DONT_KNOW` in other works

- C_{unusable}: the cluster containing those references that have been marked unusable,
- C_{any}: the cluster containing **`any`**-references,
- C_{this}: the **`this`**-cluster, and
- C_{peer}: the **`peer`**-cluster.

We will use two notation to visualize cluster states in this work. A matrix will be shown to accurately depict a cluster state. For example, the following diagram shows the cluster state s involving the four variables a, b, c and d, in which a and b are in the same cluster, and c and d may be in the same cluster.

s	a	b	c	d
a	$\top$	$\top$	$\bot$	$\bot$
b		$\top$	$\bot$	$\bot$
c			$\top$	?
d				$\top$

We leave the lower half of the matrix empty for clarity. Since $(a =_C b) = (b =_C a)$, the full matrix is symmetrical. The values on the diagonal are trivially true, since a reference always points into the same cluster as itself.

We will also express cluster states using the following notation:

$$s = \{a\, b \mid c\, d\}$$

All the relevant variables in a cluster state are listen in curly braces, and clusters are separated by bars. Iff two variables a and b are not separated by a bar, they are in the same cluster. This second notation does not allow us to express the relation $a \stackrel{?}{=}_C b$, however it does have the advantage of being more concise. We shall only use if the cluster state contains no $\stackrel{?}{=}_C$ relations.

2.2. Error Formula

We represent universe type errors in an *error formula*. It serves as an integral part of the error backtrace, both for calculating the backtrace and for showing its results to the user. The error formula is a logic formula consisting of a conjunction of terms. Each term in the error formula consists of a statement expressing that two variables are in the same cluster; this statement is made using the $=_C$ operator.

The error formula in the following example represents the error that the reference a has been marked unusable. C_{unusable} means the "unusable cluster" marker:

$$a =_C C_{\text{unusable}}$$

An error formula expressing that it is an error that the references a and b point into the same cluster, and that c and d point into the same cluster, looks like this:

$$(a =_C b) \wedge (c =_C d)$$

To check whether an error specified by an error formula F exists at some point i in the program, we evaluate F against the cluster state s, which will have previously been determined by the flow

analysis. In this way, the error formula becomes a function which maps cluster states to tri-state logic values. We will use the following syntax to express that F is evaluated against s:

$$F(s)_3$$

Evaluating F against s is defined as evaluating each term of F in s, and AND-ing the results. Evaluating a term t means simply looking up the statement contained in t in the cluster state s, which will result in a tri-state logic value. We will use the syntax $t(s)$ for this operation. The function AND which operates on tri-state logic values is defined as folllows:

AND(a,b)		b		
		TRUE	FALSE	UNKNOWN
a	TRUE	TRUE	FALSE	UNKNOWN
	FALSE	FALSE	FALSE	FALSE
	UNKNOWN	UNKNOWN	FALSE	UNKNOWN

If $F = t_1 \wedge t_2 \wedge \cdots \wedge t_{n-1} \wedge t_n$, then we define:

$$F(s)_3 := \text{AND}(t_1(s), \text{AND}(t_2(s), \ldots, \text{AND}(t_{n-1}(s), t_n(s))))$$

We interpret $F(s)_3$ as follows:

- $F(s)_3 =$ FALSE: The type error is not present.
- $F(s)_3 =$ UNKNOWN: There is at least one path through the program which leads to the type error.
- $F(s)_3 =$ TRUE: All paths through the program lead to the type error.

If an error formula evaluates to TRUE or UNKNOWN with a given cluster state, there exists a program error. Since the distinction is not important to us (after all, even if just one of many paths through a program contains an error, the whole program is incorrect). We shall therefore define a simplified evaluation $F(s)$ which returns standard binary logic values, and define it as follows:

$$F(s) := \begin{cases} false, & \text{if } F(s)_3 = \texttt{FALSE} \\ true, & \text{otherwise} \end{cases}$$

We shall impose some restrictions on the terms that may be added to an error formula:

- The variables mentioned in them shall be ordered according to some arbitrary (but consistent) ordering. In this text, lexical ordering shall be used, thus $(a =_C b)$ is allowed, but $(b =_C a)$ is not. The expressiveness of error formulas is not reduced by this restriction, as $(a =_C b) = (b =_C a)$ always.
- The variables mentioned in a term must be different. Thus, the term $(a =_C a)$ is not allowed. In other words, we don't allow error formulas to contain tautologies. If any algorithm produces an error formula which contains terms that are tautologies, these terms are removed from the error formula immediately.
- An error formula may not contain duplicate terms. The error formula $(a =_C b) \wedge (a =_C b)$ is therefore invalid.

The purpose of these restrictions is to allow more efficient handling of error formulas, and the following definition: the empty error formula denoted by $F_\emptyset$, i.e. the error formula which contains no terms, evaluates to *false* always:

$$\begin{aligned} & F_\emptyset(s)_3 := \texttt{FALSE} \quad \forall s \\ \Leftrightarrow \quad & F_\emptyset(s) := false \quad \forall s \end{aligned}$$

2.3. Statement Backtracking

To calculate the cluster states, the flow graph analysis maps a program to a flow graph containing different types of nodes. In the context of the flow graph analysis, the nodes are also called *analysis statements* or just *statements*. There are two sorts of statements: control flow related statements and elementary statements. The control flow statements consist of the *breakable*, *do*, *if*, *labeled*, *loop*, *sequence*, *switch*, *try-catch-finally* and *while* statements. They create the structure of the flow graph and serve as containers for elementary statements. The flow graph analysis uses them to build a list of edges linking elementary statements; for backtracking, we shall rely on those edges only and ignore the control flow statements altogether.

The elementary statements are used to calculate the cluster states. Each elementary statement has a *transfer* function which cluster states to cluster states. It is used to calculate the cluster state at the next node give the cluster state at the current node. For example, a *merge* statement might do the following:

$$\{a\, b \mid c\} \longrightarrow \text{merge}(b, c) \longrightarrow \{a\, b\, c\}$$

To allow backtracking of errors, we equip each elementary statement with a *backtrace* function. It can be understood as a reverse analog of the transfer function: in the same way that the cluster state at node n_i is mapped to the cluster state at node n_{i+1} by the transfer function, the error formula at the node n_{i+1} is mapped to the error formula at node n_i by the backtrace function. Given the above example, this might look like this:

$$(a =_C b) \longleftarrow \text{merge}(b, c) \longleftarrow (a =_C c)$$

The above example would mean the following: after the merge statement, the type error is that a points into the same cluster as c. a came to point into the same cluster as c when b was merged with c at the shown merge statement, because a pointed into the same cluster as b before. If we change the program so a no longer points into the same cluster as b before the merge statement, then a will not point into the same cluster as c afterwards[2] , and the error will be fixed.

In this section, we will discuss the elementary statements and show how their backtrace functions are defined. We will use the following symbols:

- F: The error formula after the current statement; this is a parameter to the backtrace functions,
- $t_1, \ldots, t_n$: The terms in F,
- s: The cluster state before the current statement; this is a parameter to the transfer functions,

2.3.1. Break

The *break* statement's transfer function is the identity function, i.e. it does not change the cluster state. Therefore, the backtrace function is also the identity function:

$$\text{backtrace}_{\text{break}}(F, s) := F$$

[2] of course, $a \neq_C c$ must hold too!

2.3.2. Consume

The *consume*(a) statement is essentially the same as the merge$(a, \mathrm{C}_{\text{unusable}})$ statement (see section 2.3.6.) The only difference is that after the merge, the consume statement moves any cluster markers that have been merged into the unusable-cluster into a new cluster of their own. That difference is not relevant for a backtrace, since a cluster marker C_m that is in a cluster of its own cannot appear in an error formula term: The only non-false term would be $\mathrm{C}_m =_C \mathrm{C}_m$, which is a tautology and is therefore not permitted in an error formula. The consume(a) statement is therefore treated as a merge$(a, \mathrm{C}_{\text{unusable}})$ statement:

$$\text{backtrace}_{\text{consume}(a)}(F, s) := \text{backtrace}_{\text{merge}(a, \mathrm{C}_{\text{unusable}})}(F)$$

2.3.3. Continue

The *continue* statement's transfer function is the identity function, i.e. it does not change the cluster state. Therefore, the backtrace function is also the identity function:

$$\text{backtrace}_{\text{continue}}(F, s) := F$$

2.3.4. Exit

The *exit* statement's transfer function is the identity function, i.e. it does not change the cluster state. Therefore, the backtrace function is also the identity function:

$$\text{backtrace}_{\text{exit}}(F, s) := F$$

2.3.5. Invariant Restore

The *invariant_restore*(x) statement checks whether x is of type **peer**. If it isn't, nothing happens. Otherwise, all local variables that point into the same cluster as a field are marked as pointing into the **any**-cluster instead, and the restore field statement's transfer function is executed (see section 2.3.9.) We define the backtrace function as follows:

$$\text{backtrace}_{\text{invariant\_restore}(x)}(F, s) := \begin{cases} F & \text{if } F(s) \\ (x =_C \textbf{peer}) \wedge \left(\bigwedge_i \begin{cases} t_i & \text{if } t_i(s) \neq \texttt{FALSE} \\ h(t_i, s) & \text{otherwise} \end{cases} \right) & \text{otherwise} \end{cases}$$

The helper function h accepts terms of the form $(l =_C ANY)$ and is defined as follows:

$$h((l =_C ANY), s) := (f =_C l) \mid (f =_C l)(s) \neq \texttt{FALSE}, f \text{ is a field}$$

All terms that are ever passed to the helper function x are of the form $(l =_C ANY)$: All terms in F are true after the invariant restore statements, and we replace only those terms that are false before. The transfer function of the invariant restore statement only moves local variables into the **any**-cluster. The only terms in the error formula which can be false in s are those which contain a local variable that has been moved. Since the term is true after the statement, it must be of the form $(l =_C ANY)$.

There will always be a field f for which $(f =_C l)(s)$ is not FALSE since l is only moved if it points into the same cluster as a field. If there are several such fields, any one of them may be used.

2.3.6. Merge

The $merge(a, b)$ statement merges the two clusters containing a and b. We define the backtrace function as follows:

$$\text{backtrace}_{\text{merge}(a,b)}(F, s) := \bigwedge_i \begin{cases} t_i & \text{if } t_i(s) \neq \texttt{FALSE} \\ (a =_C x) \wedge (b =_C y) & \text{if } ((a =_C x) \wedge (b =_C y))(s) \mid t_i = (x =_C y) \\ (a =_C y) \wedge (b =_C x) & \text{otherwise} \mid t_i = (x =_C y) \end{cases}$$

If $t_i(s) = (x =_C y)(s) = \texttt{FALSE}$, then either $((a =_C x) \wedge (b =_C y))(s)$ or $((a =_C y) \wedge (b =_C x))(s)$ (but not both) will always be `TRUE` or `UNKNOWN`: Since $(x =_C y)(s)$ is `FALSE`, x and y were in different clusters. However, $x =_C y$ is `TRUE` or `UNKNOWN` after the cluster merge effected by this merge statement, so x and y have to be in the two clusters containing a and b respectively, which are the two clusters affected by the merge.

Note that in the cases where one or both of x and y is the same as a and b, we get terms that are tautologies that are not added to the error formula. It is therefore possible that a merge statement removes terms from the formula without adding any new terms.

2.3.7. Move

The $move(a, b)$ statement moves a into the cluster containing b. We define the backtrace function as follows:

$$\text{backtrace}_{\text{move}(a,b)}(F, s) := \bigwedge_i \begin{cases} t_i & \text{if } t_i(s) \neq \texttt{FALSE} \\ (b =_C y) & \text{if } (a =_C x)(s) \neq \texttt{FALSE} \mid t_i = (x =_C y) \\ (b =_C x) & \text{otherwise} \mid t_i = (x =_C y) \end{cases}$$

If $t_i(s) = (x =_C y)(s) = \texttt{FALSE}$, then a will be the same as either x or y: Since $(x =_C y)(s) = \texttt{FALSE}$, x and y were in different clusters. However, $x =_C y$ is `TRUE` or `UNKNOWN` after a has moved into b's cluster. Since no variable other than a has changed clusters, a has to be the same as one of x and y.

a cannot be the same as both x and y: This would imply that $x = y$, which would mean that t is a tautology, and tautologies are never added to the error formula. Of course, it would also mean that t_i never evaluates to `FALSE`, so it would be skipped when looking for terms to replace in this statement's backtrace function.

2.3.8. New

The $new(a)$ statement creates a new cluster and moves a into it. The backtrace function is the identity function: since after the new(a) statement, a is in a cluster of its own, the error formula cannot contain any term (other than a tautology, which would not be added to the error formula) which contains a and is not `FALSE`. Therefore we define:

$$\text{backtrace}_{\text{new}}(F, s) := F$$

2.3.9. Restore Fields

The *restore fields* statement restores proper field types my merging fields that point into different clusters but are declared as pointing into the same cluster. This may result in several merges, or none at all.

To backtrace the restore fields statement, we backtrace each of its merges in reverse order (see section 2.3.6). Thus, if a restore fields statement's transfer function effects the merge operations $\text{merge}(a_1, b_1), \ldots, \text{merge}(a_n, b_n)$, we define the backtrace function as follows. Let $s_1, \ldots, s_n$ be the results of the transfer functions of the successive $\text{merge}(a_1, b_1), \ldots, \text{merge}(a_n, b_n)$ operations:

$$\text{backtrace}_{\text{restore\_fields}}(F, s) :=$$

$$\text{backtrace}_{\text{move}(a_1,b_1)}(\ldots \text{backtrace}_{\text{move}(a_{n-1},b_{n-1})}(\text{backtrace}_{\text{move}(a_n,b_n)}(F, s_n), s_{n-1}), \ldots, s_1)$$

2.3.10. Skip

The *skip* statement's transfer function is the identity function, i.e. it does not change the cluster state. Therefore, the backtrace function is also the identity function:

$$\text{backtrace}_{\text{skip}}(F, s) := F$$

2.4. Linear Backtracking Algorithm

We now present a simplified version of the backtracking algorithm. It backtraces an error in a linear flow graph, i.e. a flow graph without branches[3] or loops. It has the following parameters:

- n: The node in the flow graph where the error is found, and
- F: The error formula describing the type error.

Note that this algorithm does not concern itself with reporting or presenting the trace it creates. We discuss the presentation of a backtrace in section 4.2.

linear_backtrace(F, n) :

1. If $F = F_\emptyset$ (f is empty, i.e. contains no terms), return.
2. If n has no incoming edges, return.
3. For the edge $(n_{\text{from}}, n_{\text{to}})$ with $n_{\text{to}} = n$ do:
   i) Let s be the cluster state at n_{from}.
   ii) Let $F' := \text{backtrace}_{n_{\text{from}}}(F, s)$. The backtrace function that is one of the backtrace functions defined in section 2.3: We choose the one that corresponds to the type of n_{from}[4].
   iii) Call linear_backtrace(F', n_{from}). This is a recursive call.

The recursion is always of finite depth: eventually the first node in the flow graph will be reached. This node has no incoming edges, so the recursion stops at step 2 of the algorithm.

[3] Actually, the algorithm handles branches that do not rejoin.

[4] Edges only point from one elementary statement to another, so n_{from} will always be an elementary statement and thus will always have a backtrace function.

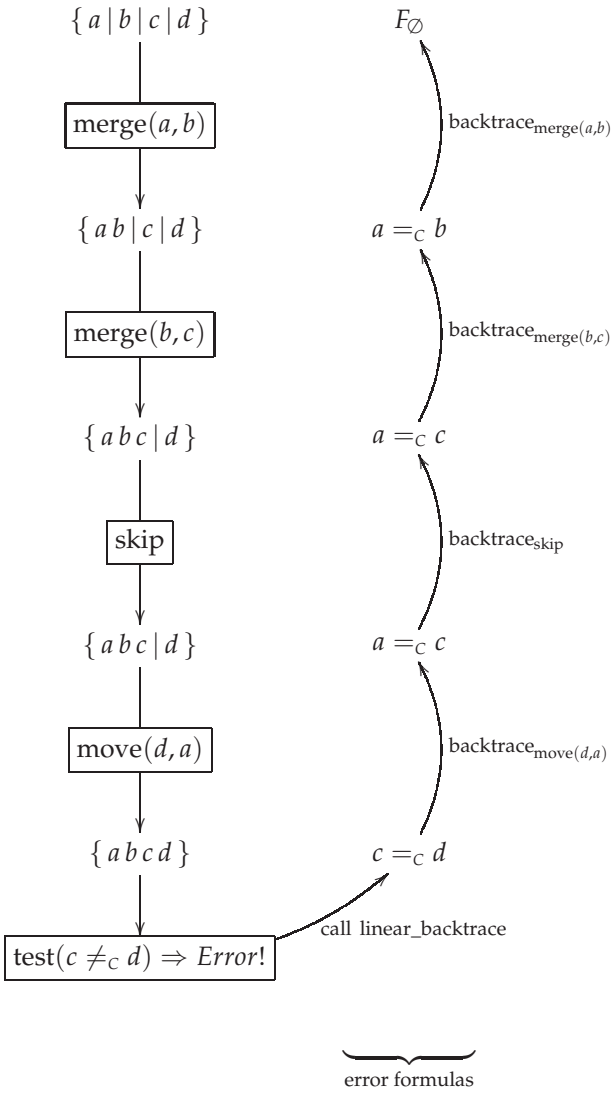


Figure 2.1: backtracing a program without branches

Figure 2.1 shows a small example that demonstrates the linear_backtrace backtracking algorithm. The left column shows the flow graph that was created by the flow graph analysis. At the end of the flow graph, we have a test that fails; a backtrace is initiated according to the linear_backtrace algorithm. Eventually, the backtrace leads to the empty formula $F_\emptyset$, and stops.

Note that a backtrace always backtrace to the top of a method. The backtrace will stop earlier if there's nothing that can be changed from that point on that would prevent the error. A trivial example is shown in figure 2.2.

2.5. Dealing with Branches

We will now present an extended version of linear_backtrace which is also capable of dealing with programs that contain branches:

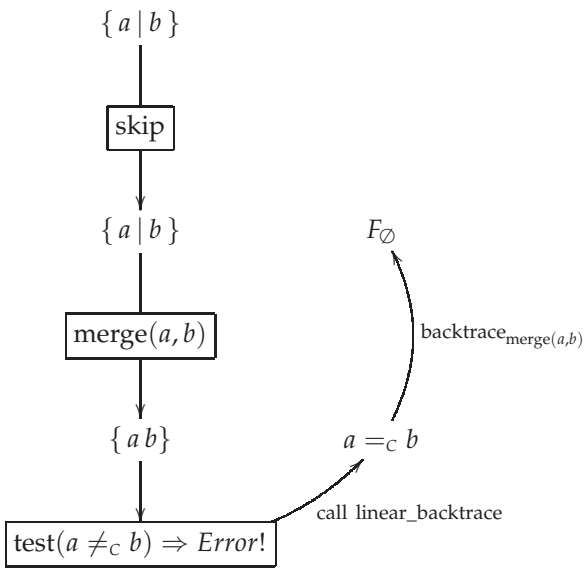


Figure 2.2: a backtrace that stops before it reaches the top of the method.

simple_branching_backtrace(F, n) :

1. If $F = F_\emptyset$ (f is empty, i.e. contains no terms), return.
2. If n has no incoming edges, return.
3. For every edge $(n_{\text{from}}, n_{\text{to}})$ with $n_{\text{to}} = n$ do:
   i) Let s be the cluster state at n_{from}.
   ii) Let $s' :=$ transfer$_{n_{\text{from}}}(s)$.
   iii) If $F(s') = false$, continue at step 3 with the next edge.
   iv) Let $F' :=$ backtrace$_{n_{\text{from}}}(F, s)$. The backtrace function that is one of the backtrace functions defined in section 2.3: We choose the one that corresponds to the type of n_{from}.
   v) Call simple_branching_backtrace(F', n_{from}). This is a recursive call.
   vi) Return.

The difference between simple_branching_backtrace and simple_backtrace is that the former can deal with programs that have rejoining branches. The node in the flow graph at the point where n branches join will have n incoming edges, and it is possible that the error was caused by some, but not all of the branches. Therefore, simple_branching_backtrace will try to find an edge which leads to a node where the error still exists, and once it does, continues the backtrace at that node.

Since we are interested in only one path through the program that shows the error, we add step 3.vi.

Figure 2.3 shows a program that contains two rejoining branches. When the backtrace gets to the branch join point, the left branch is evaluated but not taken because the error formula evaluates to $false$ in it.

simple_branching_backtrace does not handle all situations correctly. If a program contains two rejoining branches, but no single branch contains the error, the simple_branching_backtrace algorithm will stop the backtrace prematurely. An example for this behavior is shown in figure 2.4.

If we encounter such a situation, to accurately present an explanation for the error, the backtrace must evolve into a tree and trace both branches, splitting the terms of the error formula between the branches. We present the algorithm branching_backtrace, and and apply it to the previous example. Figure 2.5 shows how branching_backtrace succeeds where simple_branching_backtrace failed.

At this point, we need to make explicit the distinction between *entry* and *exit states*, or more specifically, the distinction between a node's entry cluster state, and the parent node's exit cluster state. Up until now, whenever we've referred to a node's cluster state, we were referring to the node's entry cluster state, i.e. the cluster state as it exists before the node's transfer function is applied to it. In the figures 2.1–2.5, this is the cluster state above each node. Now, let us define:

- A node's *entry state* is the cluster state that is used as parameter for the node's transfer function.
- A node's *exit state* is the cluster state that is the result of calling the node's transfer function with the entry state as parameter.

Inside a linear (non-branching) part of a program, a node's entry state equals to its parent node's exit state. If a node has more than one parent, i.e. it is a branch join point, this is no longer true: the node's entry state is the result of all its parents' exit states joined together.

In the following backtracing algorithm, we will use the expression *false parent*. A node's parent is a false parent, if the node's error formula evaluates to false against the parent's exit state.

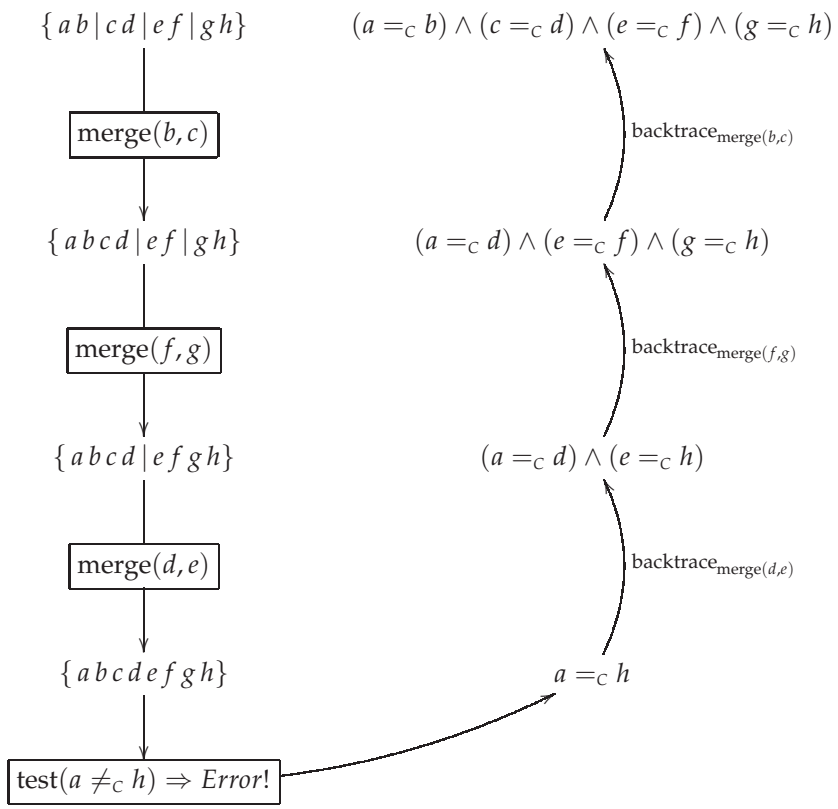
branching_backtrace(F, n) :

1. If $F = F_\emptyset$ (f is empty, i.e. contains no terms), return.
2. If n has no incoming edges, return.
3. Let P_f be the list of false parents, initialized to $\emptyset$.
4. For every edge $(n_\mathrm{from}, n_\mathrm{to})$ with $n_\mathrm{to} = n$ do:
   i) Let s be the entry state at n_from.
   ii) Let $s' :=$ transfer$_{n_\mathrm{from}}(s)$.
   iii) If $F(s') = false$, add n_from to P_f and continue at step 4 with the next edge.
   iv) Let $F' :=$ backtrace$_{n_\mathrm{from}}(F, s)$. The backtrace function that is one of the backtrace functions defined in section 2.3: We choose the one that corresponds to the type of n_from.
   v) Call branching_backtrace(F', n_from). This is a recursive call.
   vi) Return.
5. Let $A : p_\mathrm{f} \longrightarrow F_p$ be a mapping from false parents to error formulas, initialized with $A(p_\mathrm{f}) = F_\emptyset \,\forall\, p_\mathrm{f} \in P_\mathrm{f}$
6. For every term t in f do:
   i) Find the first $p_\mathrm{f} \in P_\mathrm{f}$, which has an exit state s' and $t(s') \neq$ `FALSE`. Add t to $A(p_\mathrm{f})$.
7. For every $p_\mathrm{f} \in P_\mathrm{f}$ do:
   i) If $A(p_\mathrm{f}) = F_\emptyset$, continue at step 7 with the next p_f.
   ii) Let s be the entry state at p_f.
   iii) Let $F' :=$ backtrace$_{p_\mathrm{f}}(A(p_\mathrm{f}), s)$.
   iv) Call branching_backtrace(F', p_f). This is a recursive call.

2.6. Worst-Case Scenarios

2.6.1. Arbitrarily Large Error Formulas

Since the merge statement's backtrace function can add new terms to the error formula, it is possible to construct programs which, when backtraced, lead to a large error formulas that contain every single variable in the program. Consider the following situation:



If we have N variables, then this program can be described in the following way:

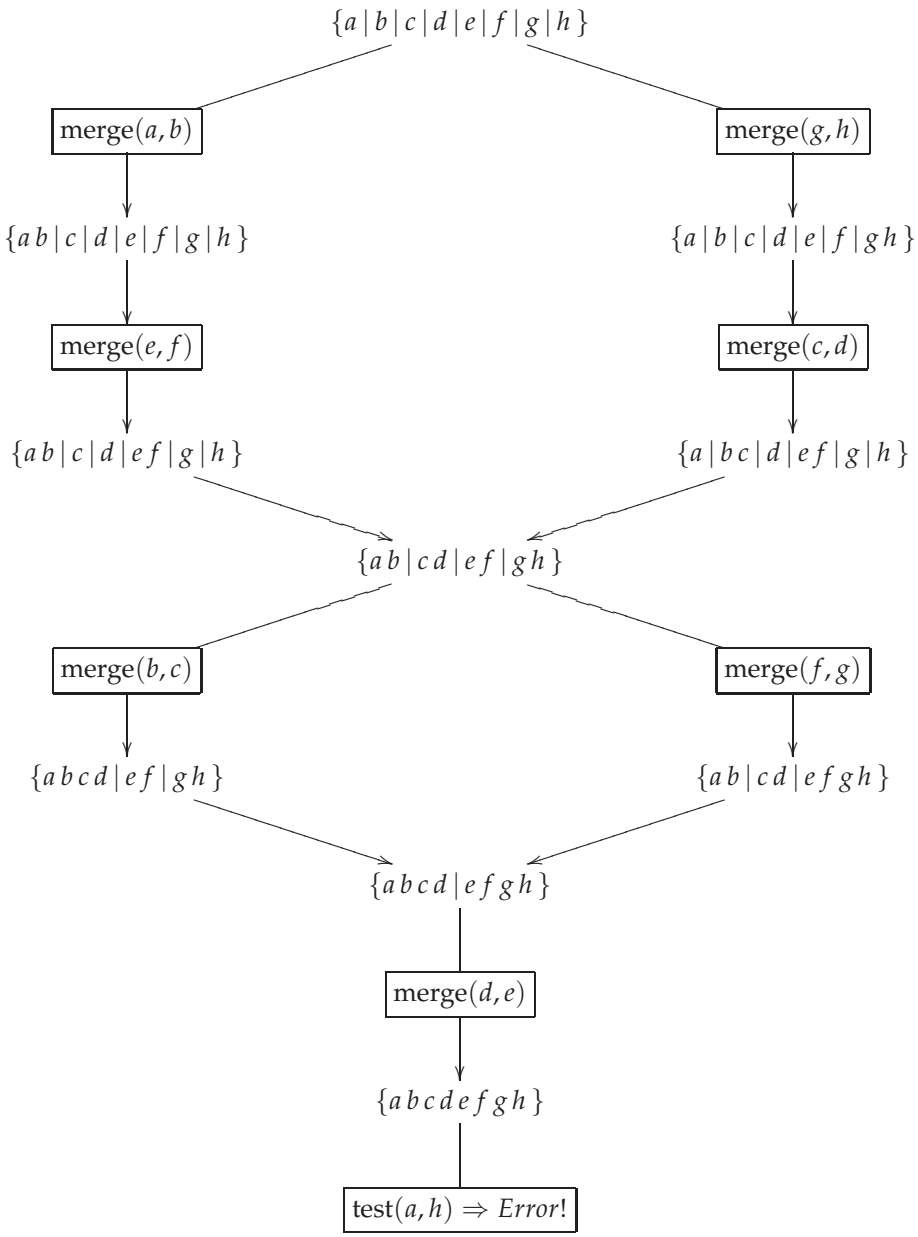
Error Formula Blow-Up(N)

1. Let there be $\frac{N}{2}$ pairs of variables, and let each pair be in a distinct cluster.
2. While there is more than 1 cluster:
   i) Select two of the smallest clusters, and merge them using variables that have not been mentioned in merge statements before.
3. Let a and b be the two variables that have not yet been mentioned in merge statements before. Test for $a \neq_C b$.

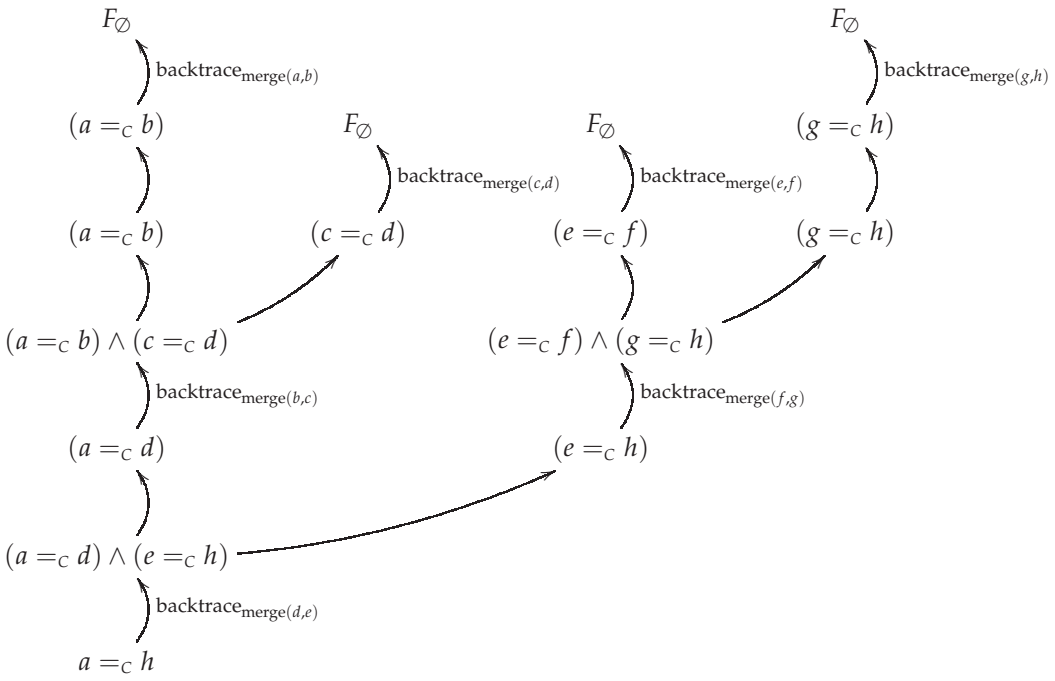
Such a program will contain $\log_2(N)$ merge statements and result in a backtrace containing $\frac{N}{2}$ terms. This is well within our goal of having quadratic bounds for the memory requirements of the backtracing algorithm.

2.6.2. Arbitrarily Many Backtraces

Since at branch join points, a backtrace can split into two backtraces, it is possible to construct programs which, when backtraced, lead to a large number of backtraces. Consider the following program with 8 variables:



We'll show the corresponding error backtrace separately:



To generalize: it is possible to construct a program that contains $2N$ variables and $2N - 1$ merges, and splits the backtrace into a tree with N leaves. This results in an exponential runtime, however, since we can develop the error backtrace tree depth-first, storage requirements are still quadratically bounded.

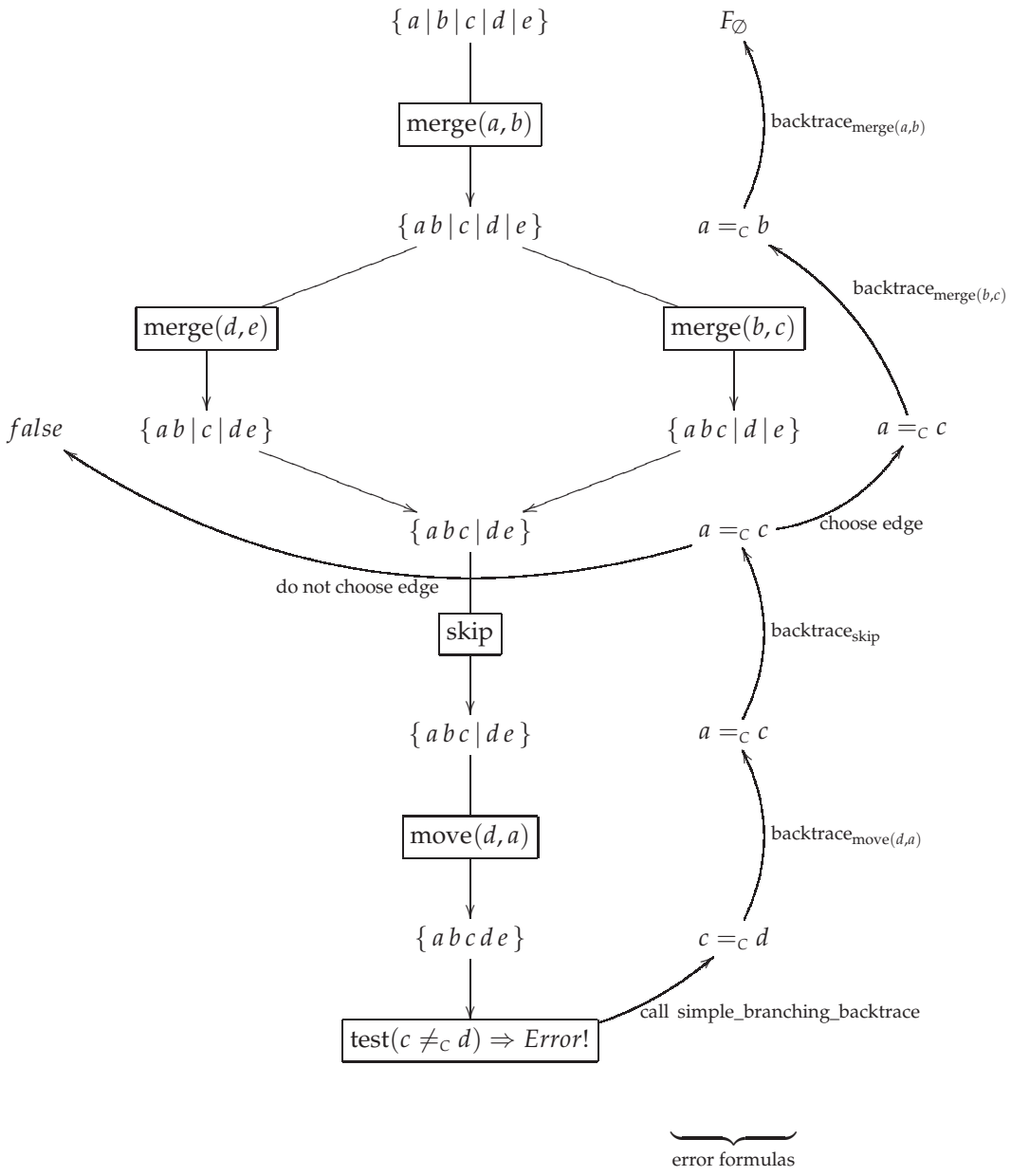


Figure 2.3: backtracing a program with branches

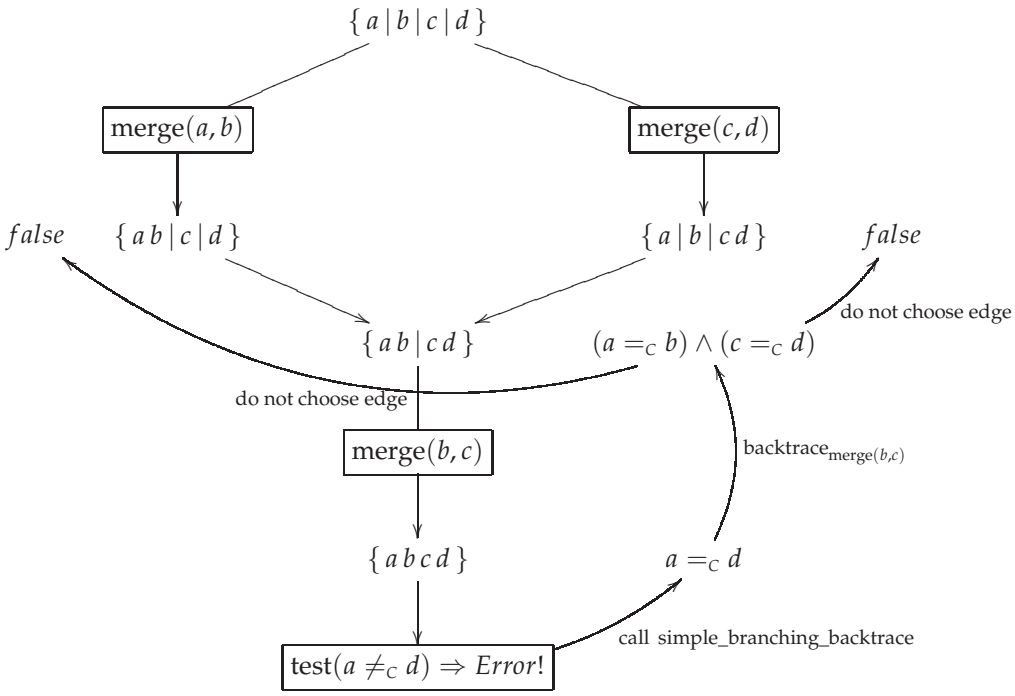


Figure 2.4: Backtracing a program with branches, where no single branch contains the error. The simple_branching_backtrace algorithm stops early.

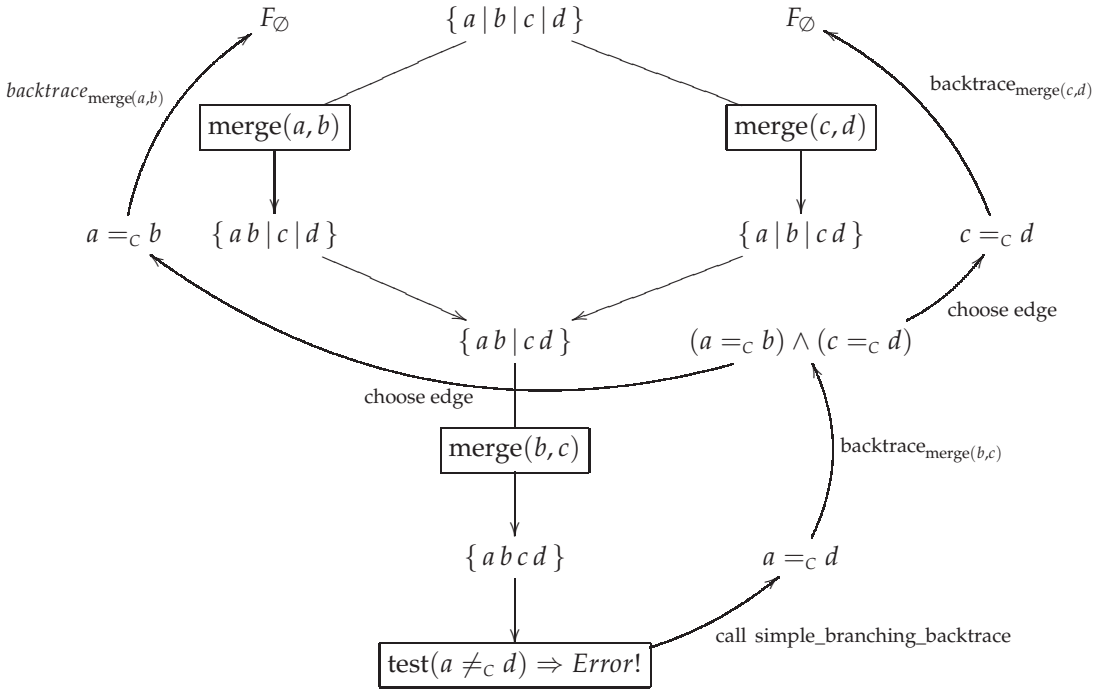


Figure 2.5: Backtracing a program with branches, where no single branch contains the error. The branching_backtrace algorithm returns a backtrace that is a tree.

3. Graph Unfolding

Previously, we excluded programs containing loops from type error backtraces. In this chapter, we will present a method for unrolling loops in a memory efficient fashion, which allows us to backtrace programs with loops using the branching_backtrace algorithm.

3.1. Problem Description

The universe type flow analysis as implemented in MultiJava handles loops and branches with a fixpoint analysis. A loop is executed[1] for as long as the cluster states of nodes inside the loop change. This means that the nodes in the flow graph can possibly be visited several times. Figure 3.1 shows an example for such a situation.

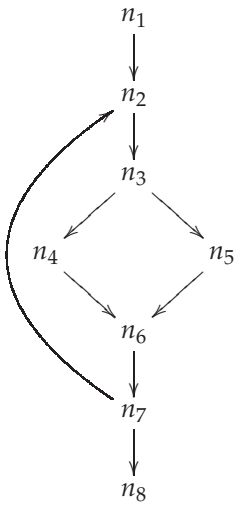


Figure 3.1: a program containing a loop; because of the branching, the loop is possibly executed twice during the flow graph analysis.

The following is a path through the graph:

$$n_1 \rightarrow \boldsymbol{n_2} \rightarrow n_3 \rightarrow n_4 \rightarrow n_6 \rightarrow n_7 \rightarrow \boldsymbol{n_2} \rightarrow n_3 \rightarrow n_5 \rightarrow n_6 \rightarrow n_7 \rightarrow n_8$$

The nodes n_2, n_3, n_6 and n_7 are each visited twice. Since at each node, only the latest cluster state is stored, this path cannot be backtraced: the second time we reach e.g. n_2 in the backtrace, the cluster state with which to evaluate the error formula is no longer accessible.

[1] By execute we mean that the node's transfer functions are called; we're purely within the flow analysis here; the actual program is not run.

3.2. Generations and Edge Traces

We equip each alias matrix representing a cluster state with a value called *generation*. The generation acts as a counter: Whenever a node is visited during the flow graph analysis, and its cluster state would change, we increase the the generation, and store the new cluster state as a separate object. The previous cluster state (indeed, all previous cluster states) remain accessible and can be identified by their generation value. If when a node is revisited, the cluster state does not change, the generation is not incremented.

Note: When we refer to a node's generation, we refer to the node with it's cluster state at the given generation.

By observing the flow analysis process, we can define a new graph, the *unrolled flow graph* where the nodes consist of all unique $\langle n, g \rangle$ tuples where n is a node in the original flow graph and g is a generation, and the edges are the edges followed by the flow graph analysis. We will call such an edge in the unrolled flow graph, which is a tuple $(\langle n_{\text{from}}, g_{\text{from}} \rangle, \langle n_{\text{to}}, g_{\text{to}} \rangle)$ an *edge trace*.

Figure 3.2 shows the unrolled flow graph for the program shown in figure 3.1.

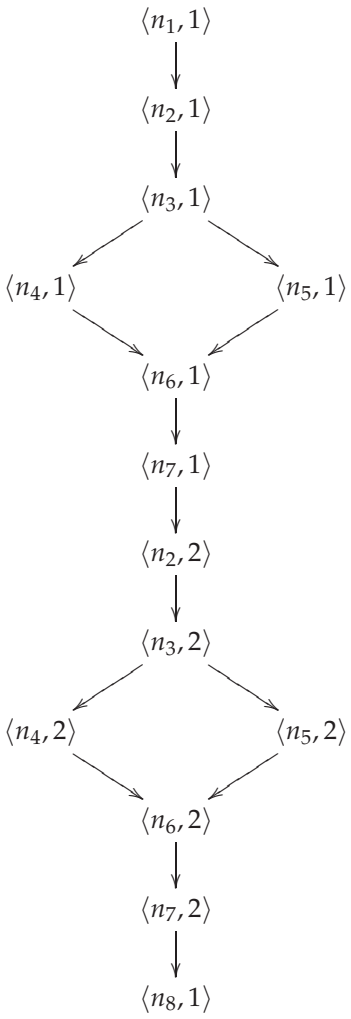


Figure 3.2: The unrolled flow graph for the previous program.

An unrolled flow graph will contain no loops, and since each node $\langle n, g\rangle$ corresponds to exactly one cluster state, the cluster state of n at generation g, the unrolled flow graph satisfies all preconditions for the branching_backtrace algorithm.

3.3. Alias Matrix Compression

Unrolling the flow graph to deal with loops has one drawback: The number of cluster states we need to store for the backtrace can be large, if loops are iterated over several times during the flow graph analysis. However, it is possible to compress all generations of the cluster state of a single node into a single alias matrix. In this section, we will assume that all cluster states are always represented by alias matrices.

The JOIN(a, b) function, which takes two tri-state logic values and returns one tri-state logic value, has been defined in [1] as:

JOIN(a,b)		b		
		TRUE	FALSE	UNKNOWN
a	TRUE	TRUE	UNKNOWN	UNKNOWN
	FALSE	UNKNOWN	FALSE	UNKNOWN
	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN

We observe:

Corollary 3.1. *Let $a_{i+1} :=$ JOIN(a_i, b). a_{i+1} is either equal to a or* UNKNOWN.

The JOIN(a, b) function is used to define the JOIN(s, t) function, which takes as parameter two cluster states with n elements and returns the cluster state u with n elements:

$$u_i := \text{JOIN}(s_i, t_i) \mid i \in [1, n]$$

Theorem 3.1. *As the generation of a cluster state increases, its values may change in a monotonous fashion from* TRUE *or* FALSE *to* UNKNOWN *only.*

Proof. Let the node n in a flow graph be a node which is visited more than once by the flow graph analysis, and let the i'th visit result in the cluster state $s_{\langle n,i\rangle}$. In the $i+1$'th iteration, let the chosen parent node n be m with generation j. The new cluster state at n is defined as follows:

$$s_{\langle n,i+1\rangle} := \text{JOIN}(s_{\langle n,i\rangle}, \text{transfer}_m(s_{\langle m,j\rangle})),$$

where JOIN$(s_{\langle n,i\rangle}, \text{transfer}_m(s_{\langle m,j\rangle}))$ is the function that applies JOIN(a, b) in an element-wise manner to the elements of $s_{\langle n,i\rangle}$ and $\text{transfer}_m(s_{\langle m,j\rangle})$. As per corollary 3.1, each element in $s_{\langle n,i+1\rangle}$ is either equal to the same element in $s_{\langle n,i\rangle}$ or UNKNOWN. □

A series of tri-state logic values $a_1, \ldots, a_n$ that change in a monotonous fashion from TRUE or FALSE to UNKNOWN can be compressed to just the two values a_{previous} and g:

$$\begin{aligned} a_{\text{previous}} &:= a_1 \\ g &:= \begin{cases} i \mid a_i \neq a_{i-1} & \text{if } \exists a_i \neq a_{i-1} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

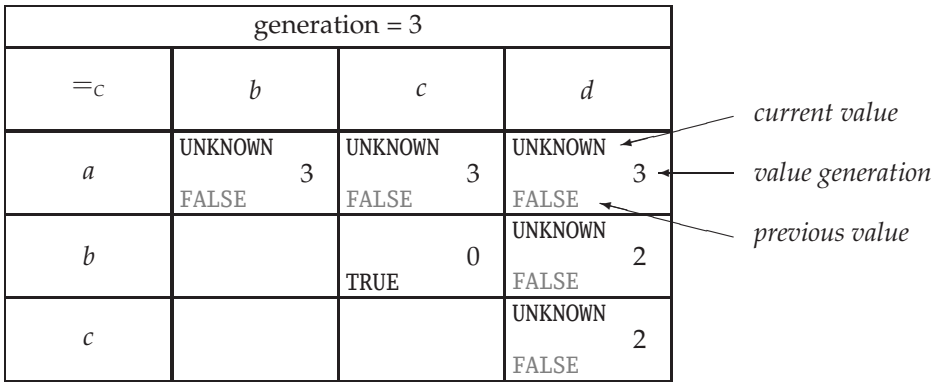
The series can be reconstructed as follows:

$$a_i = \begin{cases} a_{\text{previous}} & \text{if } i < g \vee g = 0, \\ \text{UNKNOWN} & \text{otherwise} \end{cases}$$

By compressing each series of values in a cluster state alias matrix in this way, we can represent an arbitrary number of cluster state generations in a compressed alias matrix. Each element of the compressed alias matrix contains the *previous value* and the *value generation*; the *current value* is implied. To reconstruct a generation i of the cluster state from the compressed alias matrix, we reconstruct the generation i of each value.

Note that the values of any generation of the cluster state can be accessed in constant time, i.e. element access is an $O(1)$ operation. Therefore, using a compressed alias matrix instead of a series of alias matrices does not have any negative impact on the scalability of the algorithm in which it is used, processing time-wise. As for memory-wise scalability, using compressed alias matrices to store all generations of all cluster states has the same scalability as storing just the latest generation with standard alias matrices.

An example showing matrix compression is presented in figure 3.1.



generation = 3			
$=_C$	*b*	*c*	*d*
a	UNKNOWN 3 FALSE	UNKNOWN 3 FALSE	UNKNOWN 3 FALSE
b		0 TRUE	UNKNOWN 2 FALSE
c			UNKNOWN 2 FALSE

generation = 2			
$=_C$	*b*	*c*	*d*
a	UNKNOWN 3 FALSE	UNKNOWN 3 FALSE	UNKNOWN 3 FALSE
b		0 TRUE	UNKNOWN 2 FALSE
c			UNKNOWN 2 FALSE

generation = 1			
$=_C$	*b*	*c*	*d*
a	UNKNOWN 3 FALSE	UNKNOWN 3 FALSE	UNKNOWN 3 FALSE
b		0 TRUE	UNKNOWN 2 FALSE
c			UNKNOWN 2 FALSE

Figure 3.1.: Compressed alias matrix. The generation number inside each box specifies at which the generation the *current value* is used; at earlier generations, the *previous value* is used. The unused value is printed in gray. Note that the data structure is exactly the same in all three generations; what changes is only which values are used.

4. Implementation

4.1. Mapping Type Errors to Error Formulas

This section lists the universe type errors found by MultiJava, and their respective mappings to error formulas. We will refer to the errors by their message names as defined in the class `org.multijava.mjc.CUniverseUniqMessages`. Any program variables they refer to will be appended to the name in parentheses. Every universe type error that launches a backtrace is mapped to an error formula F which evaluates to `TRUE` or `UNKNOWN` at the node and generation where the error is detected.

Most error backtraces are launched from the `org.multijava.mjc.JMethodDeclaration` class, unless otherwise noted. The errors are actually detected by the flow analysis solver however, i.e. a subclass of `org.multijava.universes.uniqueness.Solver`.

In this section, the symbols a and b will refer to variables, while C_x and C_y will refer to cluster identification markers.

The following type errors launch a backtrace:

- `REP_CLUSTER_INCOMPATIBLE_VARIABLE_IS_IS`$(a, \mathrm{C}_x, b, \mathrm{C}_y)$
  Illegal attempt to merge a, which points into C_x, and b, which points into C_y.
  $\Rightarrow F := (a =_C \mathrm{C}_x) \wedge (b =_C \mathrm{C}_y)$
- `REP_CLUSTER_INCOMPATIBLE_VARIABLE_IS_MAY`$(a, \mathrm{C}_x, b, \mathrm{C}_y)$
  Illegal attempt to merge a, which points into C_x, and b, which may point into C_y.
  $\Rightarrow F := (a =_C \mathrm{C}_x) \wedge (b =_C \mathrm{C}_y)$
- `REP_CLUSTER_INCOMPATIBLE_VARIABLE_MAY_IS`$(a, \mathrm{C}_x, b, \mathrm{C}_y)$
  Illegal attempt to merge a, which may point into C_x, and b, which points into C_y.
  $\Rightarrow F := (a =_C \mathrm{C}_x) \wedge (b =_C \mathrm{C}_y)$
- `REP_CLUSTER_INCOMPATIBLE_VARIABLE_MAY_MAY`$(a, \mathrm{C}_x, b, \mathrm{C}_y)$
  Illegal attempt to merge a, which may point into C_x, and b, which may point into C_y.
  $\Rightarrow F := (a =_C \mathrm{C}_x) \wedge (b =_C \mathrm{C}_y)$
- `REP_CLUSTER_INCOMPATIBLE_CLUSTER_IS`$(\mathrm{C}_x, b, \mathrm{C}_y)$
  Illegal attempt to merge C_x with b, which points into C_y.
  $\Rightarrow F := b =_C \mathrm{C}_y$
- `REP_CLUSTER_INCOMPATIBLE_CLUSTER_MAY`$(\mathrm{C}_x, b, \mathrm{C}_y)$
  Illegal attempt to merge C_x with b, which may point into C_y.
  $\Rightarrow F := b =_C \mathrm{C}_y$
- `NON_UNIQUE_CLUSTER_IS_CONSUMED`(a)
  Illegal attempt to transfer the **`this`**-cluster; a points into **`this`**-cluster.
  $\Rightarrow F := a =_C \mathrm{C}_{\texttt{this}}$
- `NON_UNIQUE_CLUSTER_MAY_BE_CONSUMED`(a)
  Illegal attempt to transfer the **`this`**-cluster; a may point into **`this`**-cluster.
  $\Rightarrow F := a =_C \mathrm{C}_{\texttt{this}}$

- VARIABLE_IS_UNUSABLE(a)
  Variable a is unusable.
  $\Rightarrow F := a =_C \mathsf{C}_{\text{unusable}}$
- VARIABLE_MAY_BE_UNUSABLE(a)
  Variable a may be unusable.
  $\Rightarrow F := a =_C \mathsf{C}_{\text{unusable}}$
- DIFFERENT_TYPE_FIELDS_IN_SAME_CLUSTER(a, b)
  The fields a and b are declared to be of different types and should therefore point to different clusters at the end of the method.
  $\Rightarrow F := a =_C b$

The following errors are no longer used by MultiJava. These errors will therefore not result in a backtrace, and are not mapped to an error formula:

- FIELD_IS_UNUSABLE_BEFORE_NON_PURE_PEER_CALL
- FIELD_MAY_UNUSABLE_BEFORE_NON_PURE_PEER_CALL
- FIELD_IS_UNUSABLE_BEFORE_NON_PURE_PEER_CONSTRUCTOR_CALL
- FIELD_MAY_UNUSABLE_BEFORE_NON_PURE_PEER_CONSTRUCTOR_CALL
- FIELD_IS_UNUSABLE_UPON_NON_PURE_EXIT
- FIELD_MAY_UNUSABLE_UPON_NON_PURE_EXIT
- INFERRED_FIELDTYPE_NOT_ASSIGNABLE_TO_DECLARED
- MERGE_TWO_NONTRANSFERABLE_CLUSTERS

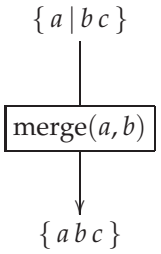
The following errors are type declaration errors or result from implementation limitations, and will not result in a backtrace. They are therefore not mapped to an error formula:

- UNIQ_FORBIDDEN_HERE
- FREE_FORBIDDEN_HERE
- REP_CLUSTER_FORBIDDEN_HERE
- REP_CLUSTER_FIELD_UNKNOWN
- REP_CLUSTER_FIELD_UNKNOWN_IN_CLASS
- REP_CLUSTER_FIELD_NOT_UNIQUE
- ILLEGAL_CAST
- PURE_METHOD_PARAMETER_CHANGED
- PURE_METHOD_RETURN_CHANGED
- PURE_METHOD_PARAMETER_ERROR
- PURE_METHOD_RETURN_ERROR
- NON_FREE_REP_PARAM_OF_NON_PURE_THROUGH_PIVOT_FORBIDDEN
- CONSTRUCTOR_NON_FREE_REP_IN_SIGNATURE_FORBIDDEN
- METHOD_PARAMETER_MISMATCH
- RETURN_TYPE_MISMATCH
- ILLEGAL_ARRAY_WRITING
- ARRAY_WRITING_THROUGH_READONLY

- NO_DYNAMIC_UNIQUENESS_YET
- MODIFIER_BEFORE_LOCAL
- MODIFIER_FOR_LOCAL_OBJECT_CREATION

4.2. Presenting the Backtrace

An error backtrace produces a great deal of information, not all of which is useful to the programmer encountering a universe type error. In general, we add the following information to an error message: For the nodes in the backtrace, we show their operation and parameters, and their entry error formula, i.e. the error formula that matches the node's entry cluster state. For example, a merge statement that looks like this:

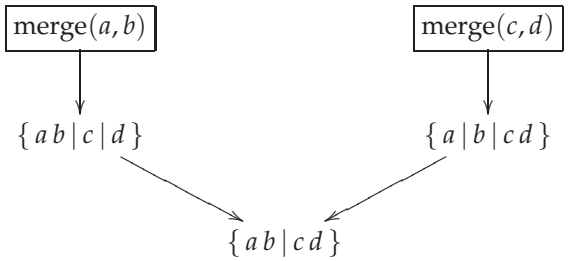


Would result in the following output, assuming the error formula after the merge statement is $a =_C c$:

```
File "Example.java", line 30, merge(field "a", field "b")
    [field "b" = field "c"]
```

Note that the order of the information in the error message is reversed with regard to the program. The point at which the type error was detected will form the first line of the error message, and as we backtrace upwards through the program, output is appended to the error message.

If a join point in the program causes the backtrace to split into branches, we mention explicitly that there is a join point, we show the exit formulas for each branch, then continue the backtrace in each branch. For example, a join that looks like this:



would produce the following segment in the error message, assuming the error formula is $(a =_C b) \wedge (c =_C d)$:

```
JOIN:
/
|     [ field "a" = field "b" ]
|     File "Example.java", line 10, merge(field "a", field "b")
|     []
\
/
|     [ field "c" = field "d" ]
|     File "Example.java", line 12, merge(field "c", field "d")
|     []
\
```

There are two exceptions to this procedure: if at a join point, the backtrace splits into several branches, we don't print any branches which have an empty exit formula. And if a node does not cause a change in the error formula, we usually don't show it in the error message. This behavior can be changed by switching MultiJava into verbose mode using the –v command line parameter. Printing all nodes can be helpful when trying to verify the exact path taken through a program by the backtrace.

There remain two minor issues with the presentation of an error backtrace in the described form. The restriction to plain text means that the wealth of information in a backtrace can make it somewhat difficult to read; debugging tools which create a graphic representation of a backtrace might be helpful here. And the mechanism of expression flattening leads to the appearance of many temporary variables which show up in the backtrace, increasing its size and making it harder to read. An unedited backtrace is shown in appendix B; the temporary variables created by expression flattening show up as *sub expressions*.

4.3. Scalability

It is one goal of this work to implement universe type error backtracking in such a way that it scales well enough to allow error analysis in any real life program, and on commodity hardware. The uniqueness extension of MultiJava with the alias matrix solver is quadratically bounded in memory and processing (time) requirements [1, p.101]. In this chapter, we show that the memory requirements with the additional information required for error backtracking is still quadratically bounded, and that the runtime is quadratically bounded in most cases.

We define the scalability of universe type error backtracking in a given method m in terms of

- s, the size of the method, i.e. the number of nodes in its flow graph,
- f, the number of fields in the class containing m, and
- l, the number of local variables in m.

Note that non-reference fields and variables, i.e. those containing basic Java types such as **`float`** or **`int`**, are not considered for defining f and l.

4.3.1. Memory Requirements

The following information relevant to an universe type error backtrace is stored:

- At each of the s nodes, an alias matrix containing the current cluster and all previous cluster states at s is stored. The alias matrix has $(f + l)$ columns and rows. Each element (i, j) of the alias matrix contains three values: the current value of $(i =_C j)$, the generation at which this value was set, and the previous value.

  The space requirements for all cluster states in a method are therefore $O(s(f + l)^2)$.

  It is possible to reduce space usage for this item by a constant factor: since the alias matrix is symmetric, only one half of the matrix needs to be stored. And because the current value at a given position in the alias matrix is always `UNKNOWN` for any non-zero generation, the current generation value can be made implicit:

$$current\_value_{i,j} = \begin{cases} previous\_value_{i,j} & \text{if } generation_{i,j} = 0 \\ \texttt{UNKNOWN} & \text{otherwise} \end{cases}$$

- Edge traces are stored in an array of linear linked lists, with the array index associating each list of edge traces with an edge id. Whenever an edge is walked during the flow graph analysis, a tuple of the form $(\text{generation}_{\text{source}}, \text{generation}_{\text{destination}})$ is appended to the list thus associated with the edge.

  The number of edges in a method is proportional to its size s. If the method contains no loops, each edge is walked once during the flow analysis, so the number of edge traces is also proportional to s. If the method does contain loops, then the edges inside the loops may be walked several times, until the cluster states at the nodes inside the loop no longer change. Because of monotony, each cluster state value can only change once (see theorem 3.1), so there are at most $(f + l)^2$ loop iterations. A loop can be no bigger than the method that contains in, the number of edge traces is in $O(s(f + l)^2)$. Nested loops cannot cause more edge walks because of the monotony theorem.

- Typically, the error formula will be small, containing only a handful of terms. However, it is possible to construct a program that contains every variable once, so the size of the error formula is in $O(f + l)$.

4.3.2. Processing Requirements

The flow graph analysis was extended in two ways:

- For each edge that is walked during the analysis, create and store an edge trace. All operations involved (looking up the current generation of the source and destination node, looking up the list associated with an edge and appending the edge trace to that list) are constant time operations.
- Storing the previous value and the generation of the current value for each value in a cluster state. This involves copying the current value to the previous value, and copying the current cluster state generation to the value generation. These are constant time generations.

Since all extensions to the flow graph analysis run in constant time, its processing time is still quadratically bounded.

Once an error is detected, a backtrace is started. Many operations are proportional to the size of the error formula, which is in $O(f + l)$.

The following operations are involved in an universe type error backtrace:

- Adding terms to the error formula: since terms are stored in a `HashSet`, it is in $O(1)$.
- Removing terms from an error formula: also in $O(1)$.

- Evaluating an error formula against a given cluster state: For each term in the error formula, we look up one value in the cluster state, then apply the operations in the error formula. The look-up takes constant time, so the evaluation is proportional to the size of the error formula, and therefore in $O(f + l)$.

- Backtracking through break, continue, exit, skip and new statements: Since the backtrace methods of these statements are empty, they are in $O(1)$.

- Backtracking through consume statements: As this is handled like a `MergeStmt`, it is in $O(f + l)$.

- Backtracking through `InvRestoreStmt`: The transfer function() can create as many merges as there are local variables, which each need to be backtraced. The backtrace is therefore in $O(l(f + l))$.

- Backtracking through merge statements: This operation evaluates each term in the error formula. If a term evaluates to `FALSE`, it is replaced with at most two other terms, which takes constant time. The operation is thus in $O(f + l)$.

- Backtracking through move statements: This operation evaluates each term in the error formula. If a term evaluates to `FALSE`, it is replaced with exactly one term, which takes constant time. The operation is thus in $O(f + l)$.

- Backtracking through restore fields statements: Duplicating the cluster state is in $O((f + l)^2)$, as is simulating the transfer function. Backtracking through the up to $f - 1$ merges is in $O(f^2 + l)$, so the whole operation is in $O((f + l)^2)$.

- Looking up an edge trace: Since edge traces are stored in a linear list, and the number of edge traces for an edge is in $O((f + l)^2)$, looking up an edge trace is in $O((f + l)^2)$ too[1].

- Switch to a previous generation of a cluster state: Every value in a cluster state is checked against the wanted generation, and possibly switched to its previous value. This operation is therefore in $O((f + l)^2)$.

The full backtrace in a linear (branch and loop free) program consists of up to s repetitions of the operations looking up an edge trace, following an edge, restoring a cluster state to a previous generation and calling a backtrace function. The processing times of these operations are all in $O((f + l)^2)$, therefore the universe type error backtrace's processing time is in $O(s(f + l)^2)$

Since a branch join point can cause each half of the branch to be backtraced, scalability is worse for programs with branching. It is possible to construct a program with approximately $\frac{s}{3}$ join points, that causes two backtraces at each such point. The number of backtraces launched will be $2^{\frac{s}{3}}$, however as backtraces are launched nearer to the top of the program, they will be shorter. The total number of statement backtraces is approximately $3(2^{\frac{s}{3}+1} - 1)$, so the backtrace's processing time is in $O(2^s(f + l)^2)$.

4.4. Nonfunctional Changes

An attempt was made to modernize and clean up the MultiJava code without changing its behavior:

[1] Using another data structure than a linear linked list would allow for a speed-up here; however we feel that the occurrence that a loop is iterated more than a couple of times is too rare to warrant the added complexity.

- Since some parts of the universe type code already used Java 5-only features, generics paramaters were added to collections throughout the code base in order to increase type safety. In some isolated areas, this was not possible because it would have lead to covariance errors. Where possible, iterator-based loops have been replaced with Java 5's new enhanced for-loop, making the code clearer and more concise.
- The constants used for tri-state logic, `TRUE`, `FALSE` and `UNKNOWN`, were previously values of type **`int`**. To increase type safety and code clarity, their type was changed to `TriValue`, a new `enum` class.
- Similarly, the internals of the `AliasMatrix` implementation were modified for more type safety. The external interface remains unchanged.
- Some of the algorithms used in the `AliasMatrix` class were replaced with more efficient versions. This was only done if the new algorithm stayed true to the idea and spirit of the previous one.
- The flow analysis code contained some parts which apparently were created in earlier versions of the code base, but then fell into disuse. Such parts were removed to improve code clarity.

5. Conclusion and Future Work

5.1. Conclusion

An algorithm has been presented that allows backtracing of universe type errors through any program. We have shown how loops can be unrolled by working with cluster state generations, and how this additional information can be stored efficiently, so that the algorithm's memory requirements are quadratically bound, even when the program contains an arbitrarily complex arrangement of branches and loops. The backtracking algorithm has been implemented in MultiJava and connected to the existing universe type checking code.

5.2. Future Work

We present a list of ideas of how future work may develop the concepts presented in this thesis further.

Technical future work:

- When backtracing a program with branches which rejoin and split several times, it is possible that several paths through the backtrace tree visit the same node more than once with the same error formula. In such a case, the remaining part of the paths will be identical; the backtrace can be sped up by caching the first path.
- At the moment, the universe type checking framework of MultiJava may report errors that are "shadowed" by another error, i.e. that are subsequent errors. Implementing a reachability analysis and suppressing error reports in such cases reduces the amount of error messages given by MultiJava, while the relevant information remains the same and thus stands out more.
- Because of flattening, backtraces are longer and less intuitive to understand than they could be. If the effects of flattening could be at least partially hidden from the programmer, the backtraces would be easier to understand.

Theoretical future work:

- The backtracking algorithms ought to be generalizable to any flow graph framework with per-node monotony.

A. Bibliography

[1] Yoshimi Takano. *Implementing Uniqueness and Ownership Transfer in the Universe Type System*. Master's thesis, ETH Zurich, 2007.
URL: `http://www.sct.ethz.ch/projects/student_docs/Yoshimi_Takano`.

[2] Annetta Schaad. *Inferring Universe annotations in the presence of ownership transfer*. Master's thesis, ETH Zurich, 2007.
URL: `http://sct.ethz.ch/projects/student_docs/Annetta_Schaad/`.

[3] Philip Wadler. *Linear Types Can Change the World!* In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pp. 347–359. North Holland, 1990.

[4] John Hogg. *Islands: aliasing protection in object-oriented languages*. *ACM SIGPLAN Notices*, 26(11):271–285, November 1991. ISSN 0362-1340.

[5] Henry G. Baker. *"Use-once" variables and linear objects: storage management, reflection and multi-threading*. *ACM SIGPLAN Notces*, 30(1):45–52, 1995. ISSN 0362-1340.

[6] Naftaly H. Minsky. *Towards Alias-Free Pointers*. In *ECCOP '96: Proceedings of the 10th European Conference on Object-Oriented Programming*, pp. 189–209. Springer Verlag, 1996. ISBN 3-540-61439-7.

B. Example Backtrace

The following is the full, unredacted backtrace for the example program shown in listing 1.3. Some line breaks have been added because of width constraints.

```
File "Example.java", line 32 error: Field "a" is unusable. [Universes (Uniqueness)]
    Backtrace:
        File "Example.java", line 32, move(variable "l", field "a")
            [cluster "unusable" = field "a"]
        File "Example.java", line 30, merge(sub expression "f", ANY/UNUSABLE)
            [field "a" = sub expression "f"]
        File "Example.java", line 30, move(sub expression "f", field "f")
            [field "f" = field "a"]
        File "Example.java", line 28, merge(sub expression "e", sub expression "d")
            [field "f" = sub expression "e" ^ field "a" = sub expression "d"]
        File "Example.java", line 28, move(sub expression "e", field "e")
            [field "a" = sub expression "d" ^ field "f" = field "e"]
        File "Example.java", line 28, move(sub expression "d", field "d")
            [field "f" = field "e" ^ field "d" = field "a"]
        File "Example.java", line 27, merge(sub expression "c", sub expression "b")
            [field "f" = field "e" ^ field "d" = sub expression "c"
                                   ^ field "a" = sub expression "b"]
        File "Example.java", line 27, move(sub expression "c", field "c")
            [field "f" = field "e" ^ field "a" = sub expression "b"
                                   ^ field "d" = field "c"]
        File "Example.java", line 27, move(sub expression "b", field "b")
            [field "f" = field "e" ^ field "d" = field "c" ^ field "b" = field "a"]
        JOIN:
        /
        |     [field "b" = field "a"]
        | File "Example.java", line 20, merge(sub expression "b",
        |                                      sub expression "a")
        |     [field "b" = sub expression "b" ^ field "a" = sub expression "a"]
        | File "Example.java", line 20, move(sub expression "b", field "b")
        |     [field "a" = sub expression "a"]
        | File "Example.java", line 20, move(sub expression "a", field "a")
        |     []
        \
        /
        |     [field "d" = field "c"]
        | File "Example.java", line 22, merge(sub expression "d",
        |                                      sub expression "c")
        |     [field "d" = sub expression "d" ^ field "c" = sub expression "c"]
        | File "Example.java", line 22, move(sub expression "d", field "d")
        |     [field "c" = sub expression "c"]
        | File "Example.java", line 22, move(sub expression "c", field "c")
        |     []
        \
```

```
/
|     [field "f" = field "e"]
| File "Example.java", line 24, merge(sub expression "f",
|                                     sub expression "e")
|     [field "f" = sub expression "f" ^ field "e" = sub expression "e"]
| File "Example.java", line 24, move(sub expression "f", field "f")
|     [field "e" = sub expression "e"]
| File "Example.java", line 24, move(sub expression "e", field "e")
|     []
\
```