

Problems with the Titanium Type System for Alignment of Collectives*

Amir Kamil, Titanium Group, UC Berkeley

February 14, 2006

1 Introduction

In a single-program, multiple data (SPMD) program, collective operations must be properly aligned for the program to execute correctly. Improper alignment can result in deadlock, or worse, corrupted data. Most SPMD languages make no attempt to ensure alignment, relying on the user to do so manually.

The Titanium language [2] is unique in that it attempts to prove alignment using a static type system [1]. The type system, commonly referred to as “**single**” after the main keyword it uses, has been implemented in the Titanium compiler for many years. Through our extensive experience with the system, we have encountered many unforeseen issues. In this paper, we discuss these problems and possible solutions to them.

2 Exceptions and single

There are two major problems with respect to **single** and exceptions. The first is the treatment of unchecked exceptions in single analysis, and the second is the definition of universal exceptions.

2.1 Unchecked Exceptions

In single analysis, the Titanium compiler only considers those exceptions that are declared by the program. In Java, however, declaring unchecked exceptions is optional. Thus, in the following code, the call to method `foo()` results in a compiler error while the call to `bar()` doesn't, even though the two methods are equivalent.

```
public static void foo() throws Error {
    throw new Error();
}
public static void bar() {
    throw new Error();
}
public static void main(String[] args) {
    if (Ti.thisProc() == 0) {
        foo(); // compiler error - non-single termination
        bar(); // no error
    }
    Ti.barrier();
}
```

The fact that the call to `bar()` is allowed is a hole in the analysis – at runtime, process 0 will skip over the barrier due to the thrown `Error`, while the rest of the processes won't. One way to solve this problem is to do an interprocedural analysis to determine exactly what exceptions can be generated by each call, add

*This document is accurate with respect to version 2.19 of the Titanium language reference and version 3.147 of the compiler.

declarations for unchecked exceptions, and force the program to catch them. However, programmers already complain about the fact that they have to catch unchecked exceptions that are declared, and this will only make `single` more inconvenient for them. In addition, it is unlikely that the interprocedural analysis can be done in the presence of partial compilation.

The code above will not actually deadlock at runtime – since the `Error` is not caught by the program, it will propagate out to the command line, killing the program. This suggests an alternative to forcing the programmer to catch every unchecked exception: allow unchecked exceptions to be thrown arbitrarily, as long as they are either caught before a collective operation occurs or are never caught.

2.2 Universal Exceptions

The language reference states:

A termination T (exception of type t , a break, continue or return) of a statement S is universal if either all processes P that start S terminate abruptly with termination T or no process P that starts S terminates abruptly with termination T

This implies that the exception generated in `main()` below is universal with respect to the `try` block, since all processes that enter it throw the exception, while that in `main2()` is not.

```
public static void main(String[] args) {
    if (Ti.thisProc() == 0) {
        try {
            foo();
        } catch (Exception single) {
        }
    }
}
public static void main2(String[] args) {
    try {
        if (Ti.thisProc() == 0) {
            foo();
        }
    } catch (Exception single) {
    }
}
static void foo() {
    throw new RuntimeException();
}
```

Thus the exception will be caught in `main()` but not in `main2()`. Not only is this very confusing, given the similarity of the code, but it also poses difficult engineering problems. In both cases, the exception is generated at the same point, in method `foo()`, so universality cannot be determined at the throw point. The only way to determine whether or not it is universal at the catch point is to examine every block of code entered between `try` block and the throw point and determine whether or not all processes entered each block. It is possible to do so without any runtime communication, but the only solution we came up with involves incrementing a “singleness” counter on entering a block and decrementing it on exiting. This must be done for all blocks and would impose significant runtime costs even when exceptions aren’t being thrown or caught. In addition, the solution requires doing stack unwinding for propagating exceptions, instead of long jumps.

An alternative solution is to change the definition of universal exceptions.

2.3 New Rules

We came up with a new set of rules that addresses both problems above. They are as follows:

1. Introduce a new throw statement


```
single throw <expr>;
```

 that produces a universal exception. This statement has global effects.
2. Universal exceptions are specified in a `throws` or `catch` clause by following the exception type with `single`, as they are now.
3. The regular throw statement


```
throw <expr>;
```

 always produces a non-universal exception.
4. Catch clauses can freely mix universal and non-universal exceptions – rule 4 in §9.4.3 will be removed. A universal catch clause will only catch universal exceptions, and a non-universal clause will catch both universal and non-universal exceptions. Similarly, rule 2 in §9.4.4 will also be removed.
5. A new rule will be added to §9.4.3:

If the main statement of a `try` has global effects, then non-universal unchecked exceptions may not be specified in its `catch` clauses. It is an error if one of its non-universal `catch` clauses catches an unchecked exception at runtime (for example, by specifying `Throwable`).

We are in the process of implementing the new rules.

3 Arrays and Polymorphism of `single`

The Titanium type system treats the `single` qualifier on method parameters, method returns, and instance fields polymorphically, depending on use. A method with the signature

```
static int single foo(int single x);
```

may be applied to both `single` and non-`single` arguments, returning `int single` for the former and `int` for the latter. Similarly, given the type definition

```
class Int {
  int single val;
  int single intValue() {
    return val;
  }
}
```

and variables `x` of type `Int single` and `y` of type `Int`, `x.val` and `x.value()` return `int single` while `y.val` and `y.value()` return `int`.

When it comes to array types, however, `single` is only polymorphic at the top-level. Thus the method

```
static int single bar(int[] single x);
```

can be applied to arrays of type `int[]` and `int[] single`, but the method

```
static int single baz(int single[] single x);
```

can only be applied to arrays of type `int single[] single`.

This makes it impossible to use `single` to specify coherence of array-based data structures. Consider two definitions of integer vectors.

```
class IntVector1 {
  int[] single values;
  int single length() {
    return values.length; // OK
  }
}
```

```

    int single elementAt(int single i) {
        return values[i]; // type error -- values[i] is non-single
    }
}

class IntVector2 {
    int single[] single values;
    int single length() {
        return values.length; // OK
    }
    int single elementAt(int single i) {
        return values[i]; // OK
    }
}

```

The first definition, `IntVector1`, can be used polymorphically, but it contains a type error since the `single` qualifier at the top-level of an array only specifies that the array has the same length on all processes and says nothing of their elements. The second definition, `IntVector2` is correctly typed, but cannot be used polymorphically.

This limitation is particularly relevant when it comes to the `String` class. The contents of a `String single` are not guaranteed to be coherent across all processes.

One possible solution to this problem is to introduce a new explicit qualifier for polymorphic `single`, such as `polysingle`, that can be applied at all levels of an array type. Then the integer vector could be written as below.

```

class IntVector3 {
    int polysingle[] polysingle values;
    int polysingle length() {
        return values.length; // OK
    }
    int polysingle elementAt(int single i) {
        return values[i]; // OK
    }
}

```

4 Casts to single

The Titanium type system allows arbitrary casts to `single`, which go unchecked at runtime. This is used for convenience by some users to circumvent the type system, but it is also used to get around the limitations discussed in §3. For example, a polymorphic integer parsing function could be written as follows.

```

static int single parseInt(String single s) {
    int single result = 0;
    for (int single i = 0; i < s.length(); i++) {
        result = 10 * result + (int single) s.charAt(i);
    }
    return result;
}

```

This procedure requires a cast to `single` in order to function as desired. The cast is unsafe, however, since the contents of a `String single` are not guaranteed to be coherent.

Unfortunately, in a case like this, it is unclear how to check the cast at runtime. This is because `parseInt()` is polymorphic – it may be called by all processes, in which case the cast can be checked using communication, but it may also be called by a subset of the processes, in which case communication cannot be done.

A solution to the array polymorphism would make the cast unnecessary in such a case, and the type system could be patched in two ways:

1. Make casting to `single` illegal.
2. Make casting to `single` a global operation. Then it would always be possible to do communication to check the cast. The check could be disabled when bounds checking is off.

Casts to `single` are often done when parsing program arguments, and the second solution above would still allow these casts.

5 Subset Collectives

The Titanium language reference contains a proposal of a `partition` construct that divides the set of processes into teams. Collective operations affect only members of the team that execute them. In the below code, only those processes whose IDs are less than 3 will execute the barrier.

```
static single void foo() {
    partition {
        Ti.thisProc() < 3 => Ti.barrier();
    }
}
```

Since `single` is used to ensure alignment of collective operations, it only implies coherence across all processors of a given team. This can cause problems if a `single` variable is updated within a `partition` statement. Consider the code below.

```
static single void bar() {
    partition {
        Ti.thisProc() < 3 => setX(1);
    }
    if (x == 0) {
        Ti.barrier(); // misaligned barrier
    }
}
static int single x = 0;
static single void setX(int single y) {
    x = y;
}
```

In the code above, a subset of the processes modify a `single` variable. Since collectives are over a team, the code is correctly typed. However, it will deadlock, since the barrier after the `partition` is executed by all the processes, which now have incoherent values of `x`.

More complicated examples can be constructed using objects and `single` instance variables. It is unclear how the type system can be modified to disallow them.

6 Conclusion

While `single` has been useful in ensuring safety of Titanium programs, it has given rise to a number of issues that were not anticipated by its designers. Most of these issues can be resolved by proper tweaking of the type system, but it is not clear how to reconcile `single` with the `partition` construct. Perhaps a better solution would be to redesign the type system from scratch, given the experience we now have, instead of trying to patch each hole individually.

References

- [1] D. Gay. *Barrier Inference*. PhD thesis, University of California, Berkeley, May 1998.
- [2] P. N. Hilfinger, et al. Titanium language reference manual, version 2.19. Technical Report UCB/EECS-2005-15, November 2005.