Local Qualification Inference for Titanium

Ben Liblit

UC Berkeley

CS-263 / CS-265

Spring 1998

Overview

- Introduction to Titanium
- Introduction to BANE
- Formulating the Analysis
- Implementation Strategy
- Conclusions

Introduction to Titanium

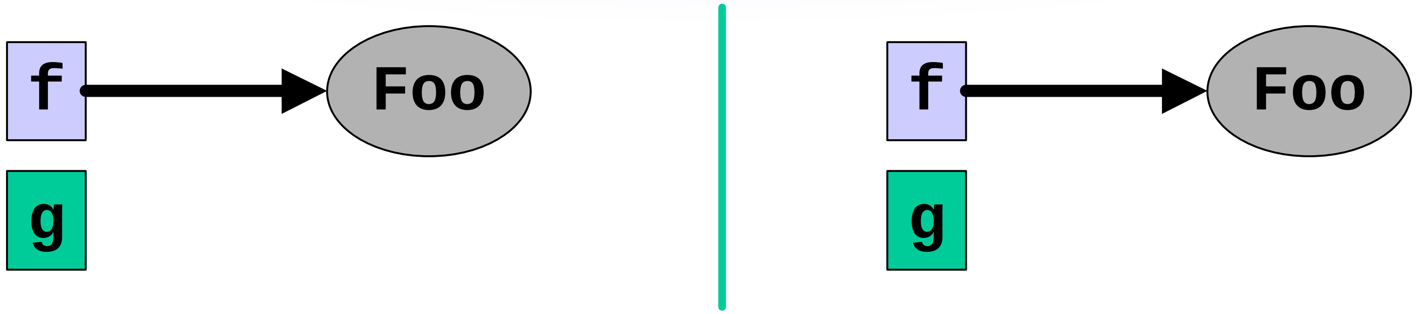
- Titanium: a new language for high-performance scientific computing.
- Syntax & semantics derived from Java, but compiled to native code.
- Explicit, SPMD parallelism.
- Targeted at both shared- and distributed-memory architectures.

Titanium Memory Model

- Each processor has a local stack & heap.

```
Foo f, g;
```

Titanium Memory Model



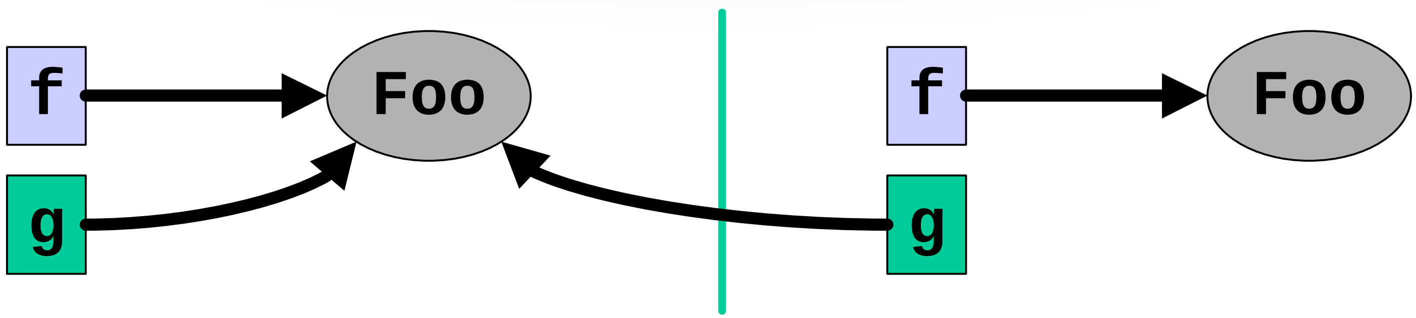
- Each processor has a local stack & heap.

```
Foo f, g;
```

- Allocation takes place locally.

```
f = new Foo();
```

Titanium Memory Model



- Each processor has a local stack & heap.

```
Foo f, g;
```

- Allocation takes place locally.

```
f = new Foo();
```

- Language primitives allow sharing of data.

```
g = broadcast f from 0;
```

Distributed Memory: Creating an Illusion

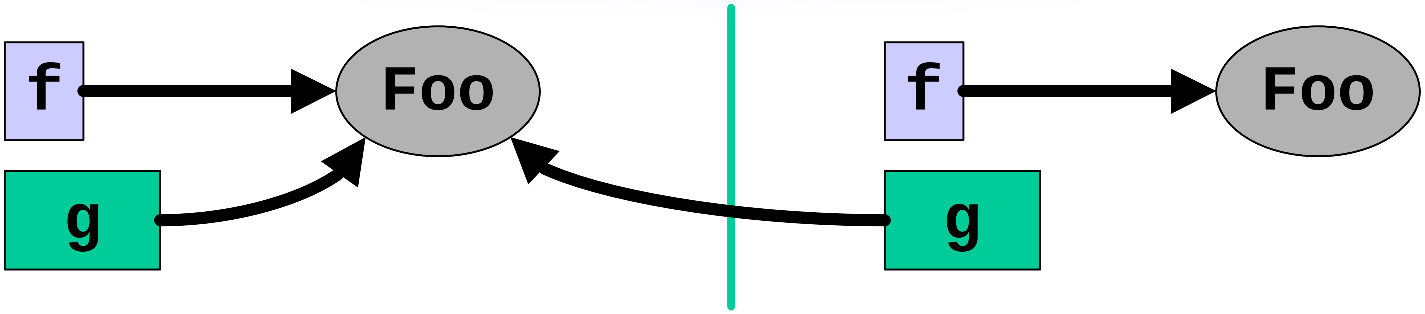
- References are free to point anywhere.
- Use “wide” pointers: `<proc, addr>`.
- Add runtime checks and messaging:

```
if (p.proc == MyProc)
 result = *(p.addr);
else
 result = RemoteRead(p.proc, p.addr);
```

Why This is Unacceptable

- Even local dereferences must go through a conditional test and branch.
- Conditional assignment from a function call confounds many traditional optimizations.
- Many references are *always* be local, and programmers know which ones.

Solution: Explicit Qualification



- Explicitly declare selected references as “`local`”.

```
Foo local f = new Foo;
Foo g = broadcast f from 0;
```

- Allocations produce local values.
- Broadcasts & exchanges produce global values.

Widening and Narrowing

- Local references implicitly widen to global.

```
Foo local f = new Foo();
Foo g = f;
```

- Narrowing global to local must be explicit, and is checked at runtime.

```
Foo g = broadcast …;
Foo local bad = g;
Foo local ok  = (Foo local) g;
```

Better, But Not Good Enough

- Compiler can check programmers’ claims.
- But programmers may miss opportunities, particularly in complex data types.

```
Foo local [] local [] local grid;
```

- Also, how do we handle legacy code?
  - Minimal Java runtime: 16,000 lines without a local qualifier anywhere in sight.
  - Titanium benchmarks written for SMP’s.

Enter BANE: The Berkeley Analysis Engine

- BANE is a generic program analysis tool based on constraint systems.
  - Feed in a set of constraints; pull out a least solution that satisfies them all.
- For this analysis, the “least solution” will add “**`local`**” wherever possible.
- The “constraints” will prevent us from violating the type system.

Formulating the Problem

- Define a lattice { local, global }, where local < global.
- Each declared reference corresponds to an unknown value on this lattice.

"Foo x"	$\leftrightarrow$	unknown x
"Foo [] a"	$\leftrightarrow$	unknowns $<a_0, a_1>$

- Apply constraints based on program, guided by Titanium's type rules.

A Simple Example: Assignments

- Source program:

```
Foo x, y, z;

y = new Foo();
z = broadcast …;

x = y;
x = z;
```

- Constraint system:

unknowns { x, y, z }

$y \geq$ local
$z \geq$ global

$x \geq y$
$x \geq z$

More Interesting: Method Invocation

- Source program:

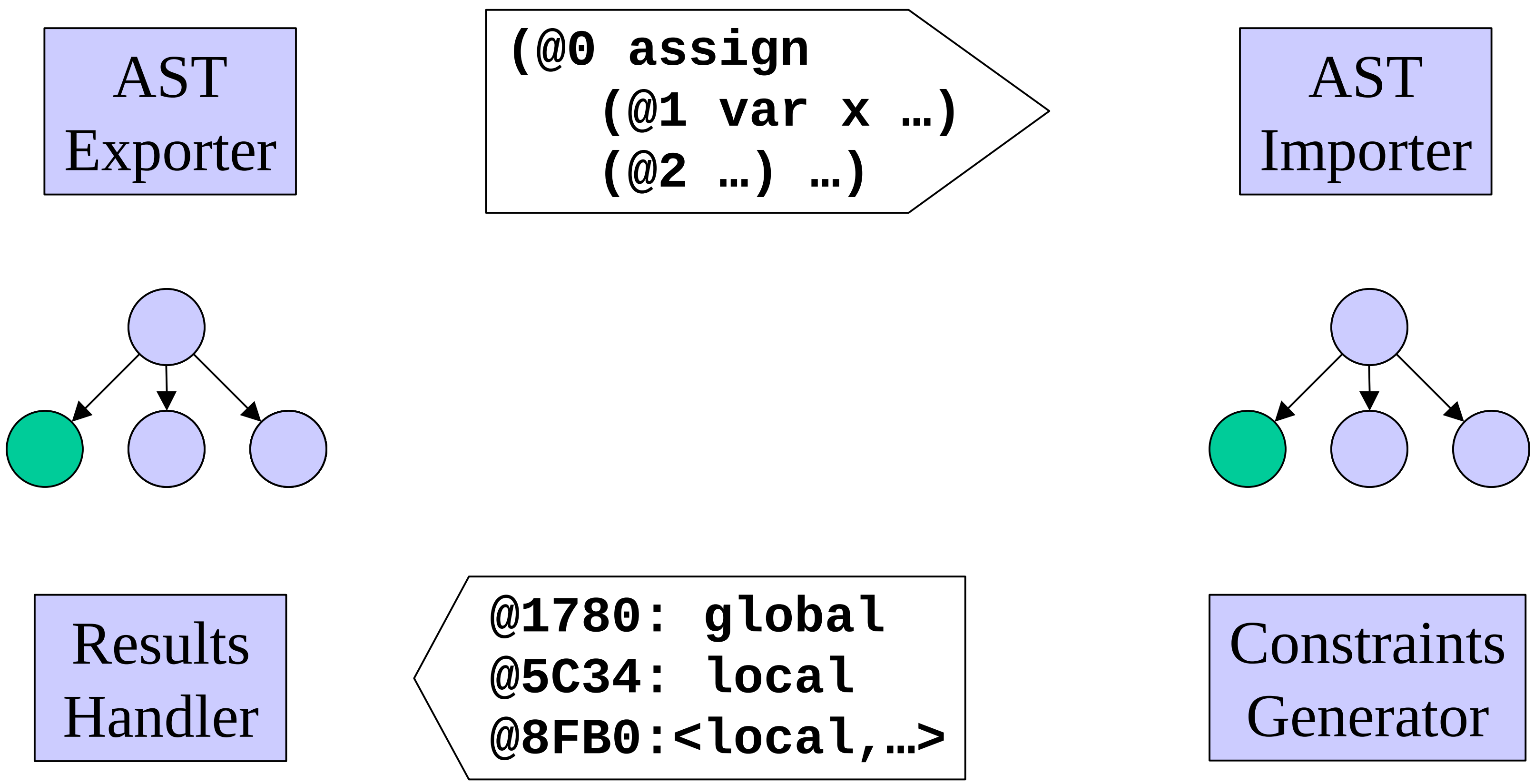
```
Foo x;
String s;

x = broadcast …;
s = x.toString();

Sys.out.print(s);
```

- Final constraints include:
  - $x \geq$ global
  - $x.toString = Foo.toString$
  - $x.toString.this \geq x$
  - $s \geq x.toString.result$
  - $Sys.out.print.arg \geq s$

Implementation Strategy

- Existing Titanium compiler has information we need about types, names, declarations…
- Titanium compiler written in C++
- BANE written in SML
- SML-to-C calling interface too primitive

Solution: Serialization



Preliminary Results: Integration Works, but Badly

- Successfully analyzed Titanium runtime library, including java.{io, lang, util}.
- 16,000 lines of code.
- 99,200 AST nodes.
- 19 megabyte serialized AST dump.
- Four minutes to load AST into SML.
- Clearly, more work is needed here.

Preliminary Results: Analysis Looks Promising

- 8,500 unknowns.
- 11,700 binary constraints.
- Complete analysis in eleven seconds.
- 79% automatically localized.
- 21% remain global
  - pessimistic assumptions about native methods
- Java semantics are a big win!

Preliminary Results: Adaptive Mesh Refinement

- 15,000 additional AST nodes. (+16%)
- 3,000 additional unknowns. (+35%)
- Two seconds longer to solve. (+20%)
- Globals increase from 21% to 22% of total.

Future Work

- Annotate native methods.
- Feed results back into Titanium compiler.
  - Benchmark performance speedup.
  - Estimate precision of results.
- Improved integration strategy.
- More sophisticated analyses.
  - Polymorphic analysis for methods.
  - Incorporate profiling feedback.