Local Qualification Inference for Titanium

Ben Liblit

EECS Department
University of California, Berkeley
387 Soda Hall #1776
Berkeley, CA 94720-1776
⟨liblit@cs.berkeley.edu⟩

August 26, 1998

Abstract

Titanium is a parallel programming language that presents a single global address space for all concurrent computations. To improve performance on distributed memory architectures, programmers may explicitly qualify selected references as *local* to a single process. This paper proposes a static inference system for automatically applying "**local**" qualifications wherever possible, within the constraints of the Titanium type system. A partial prototype implementation of the analysis has been completed with the help of the BANE analysis toolkit. We describe and evaluate the analysis itself, as well as the implementation strategy by which the analysis has been deployed.

Contents

List of Figures

List of Tables

1 Introduction

Titanium is a parallel programming language for high performance scientific computing. Titanium presents a logically global address space to programmers. However, the components of a Titanium program may be distributed across several distinct computers with no physically shared memory. Titanium combines *wide references* with a message-passing runtime system to present the illusion of shared memory when none actually exists. Wide references are more costly than simple local addresses; in order to offset this cost, programmers may explicitly declare selected references as local to a single computational unit, by adding the "**local**" type qualifier to any field, variable, or formal parameter declaration.

Unfortunately, explicit qualification is an imperfect solution. Programmers may miss many opportunities for local qualification, particularly as data types grow more complex. Also, existing legacy code written in other languages, or targeted at true shared memory machines, may prove too difficult or time-consuming to hand-qualify for maximal performance. For this reason, we wish to automate the process of adding "**local**" qualifications to a body of Titanium code.

The automation strategy uses a static analysis that resembles a simple form of type inference. We have used the Berkeley Analysis Engine (BANE) to implement the analysis. As described by its creators:

> BANE is a toolkit for constructing program analyses such as data-flow and type inference systems. ... BANE is constraint-based, meaning that analyses are formulated as systems of constraints generated from the program text. Constraint resolution (i.e., solving the constraints) computes the desired information.[1]

In this case, the constraints are dictated by the Titanium type system. The "desired information" is the largest set of global references that may be safely requalified as local.

The ultimate purpose of this research is twofold: first, we wish to assess the effectiveness of static analysis at aggressively adding "**local**" qualifications to Titanium programs. Second, we wish to evaluate the broader viability of BANE as a tool for developing other Titanium analyses in the future.

The remainder of this paper is organized as follows. Section 2 describes the Titanium memory model in greater detail and motivates the analysis that follows. Section 3 formalizes the inference rules that drive the analysis. In section 4 we outline the implementation strategy by which BANE and the Titanium compiler have been integrated, and present preliminary results from using the analysis. Section 5 reviews related work, suggests future research directions, and summarizes our conclusions.

2 Background and Motivation

We now describe the Titanium language in greater depth, focusing primarily on Titanium's memory model. We highlight some difficulties that the memory model creates for programmers, and develop the motivation for using automated inference to simplify the programming task. The reader who is already an experienced Titanium programmer may wish to skip ahead to section 2.5, wherein we highlight problems that remain unsolved by the current Titanium memory model.

2.1 Titanium Basics

Titanium uses the syntax and semantics of Java[1] [12], and adds a number of features to support the needs of the scientific computing community. In this paper, we mention only those features pertinent to the analysis at hand; see [25] for a more complete introduction to the language as a whole, or [13] for a draft reference manual.

Two notable Java features are absent from Titanium. First, Titanium has no bytecodes or virtual machine. Titanium programs are compiled to native code and executed directly on the host system. Second, threads are not the dominant mechanism for expressing concurrency. Rather, Titanium programs use a single program, multiple data (SPMD) model of parallelism. In this model, several identical copies of a program are executed simultaneously, each on different data. These multiple concurrent executions, or *Titanium*

[1] "Java" is a registered trademark of Sun Microsystems, Inc.

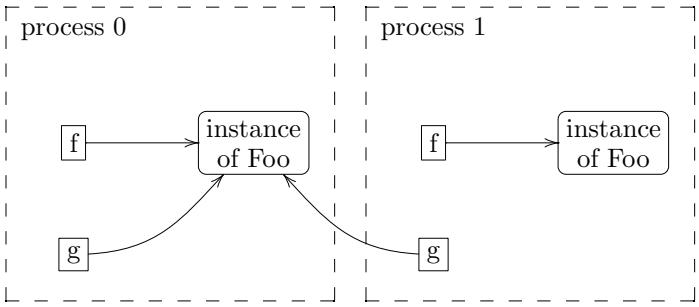

Figure 1: Local allocations. *Two processes are executing, each in its own region. Each process has independently allocated an instance of class Foo in its own local region. Process 0's instance is then broadcast to all processes. In process 1, this broadcast produces a region-spanning reference.*

processes, may take a variety of forms, ranging from lightweight threads with shared memory to truly independent processes executing on physically distinct computers connected by a fast network. Processes may synchronize or share data under explicit programmer control, but otherwise operate independently. This model is well suited to many scientific programming problems, where a generally uniform computation is to be applied across a large body of data.

2.2 Local/Global Memory Model

Titanium is targeted at both shared and distributed memory architectures. The memory model presented to programmers is a hybrid, reflecting this wide variation in supported host platforms.

The address space of a Titanium program is partitioned into a set of distinct local memories, or *regions*. Each Titanium process is associated with a single region, which provides memory to store that process's stack and local variables. Heap allocations requested by a process are also satisfied using memory drawn from the associated region.

When Titanium is running in a distributed memory environment, each region corresponds to the contiguous physical memory available on one host processor. Each Titanium process, then, uses the region local to the host on which it is executing. On the other hand, on a shared memory SMP, one region represents the entire system, and all processors allocate out of this single shared pool.

The Titanium runtime presents programmers with a single global address space, formed by the union of all local memory regions. References (analogous to C pointers) may address memory across region boundaries. Any process may read or write data in any region, at any time, provided that it has a valid reference to that data. Since allocations take place locally, Titanium provides communications primitives that let processes gain access to objects allocated elsewhere. For example, the expression

broadcast E **from** p

is a one-to-many communication. All processes rendezvous at the broadcast expression. The expression E is evaluated only on process p, and the result is then communicated to all processes. If E is a reference to some allocated object, then all processes will receive references to the same single object. The object itself is not copied to all processes; all that is communicated is a reference to the object's location within the global address space. Titanium also provides an "exchange" primitive for many-to-many communication, but we will not describe it further here, except to note that it also produces region-spanning references.

Figure 1 illustrates the notion of local regions and a global address space. Assume a one-to-one mapping between processes and regions. Two processes are executing the following program text:

```
Foo f = new Foo();
Foo g = broadcast f from 0;
```

Each process has its own local variables f and g. Each process independently allocates an instance of class Foo, and stores a reference to the new instance in its own local variable f. The broadcast then communicates the value of process 0's f to all other processes. All local variables g refer to process 0's instance of Foo.

```
if (p.region == MyRegion)
    result = *p.addr;
else
    result = RemoteRead(p.region, p.addr);
```

Figure 2: Dereferencing a wide reference. *Typical C code sequence for dereferencing a wide reference. Because "result" may receive its value from an opaque function call, the compiler is unlikely to be able to effectively optimize any code that uses the resulting value.*

2.3 Wide References: A Mixed Blessing

A global address space provides some immediate benefits to the application programmer. Program development is simplified without the need to consider memory boundaries. Data sharing reduces to a simple matter of passing references around, avoiding the complications of planning out an explicit communication strategy. A global address space maps well onto a shared memory system, making it easier to port existing codes and algorithms. Users who only intend to use Titanium on shared memory hardware can ignore regions entirely.

Unfortunately, in a distributed memory environment, a global address space is a costly illusion. Since the global address space is larger than the address space of any single machine, global references must include more information than just a simple local memory address. These *wide references* are actually pairs, consisting of a region number and a memory address within that region. Dereferencing a wide reference requires several steps, as illustrated in Figure 2.

If the reference addresses memory within the local region, then the corresponding memory address may be used directly as a standard pointer. If the reference addresses memory from a remote region, then some form of network activity is required to issue a remote request for the value in question. The current Titanium distributed memory back end reuses an existing message passing system from Split-C, an earlier language with a similar memory model [11]. Higher-performance runtime systems are currently being studied [17].

Even the fastest messaging system does nothing to reduce the costs of global references to local data, though. Assuming word alignment of structures, a wide reference may consume twice as much memory as a simple pointer. More worrisome than the space costs, though, are the time costs. As Figure 2 illustrates, using a wide reference to local data requires comparing two values, ignoring a branch to the remote fetch clause, dereferencing the local address, and branching to the end of the entire conditional. The presence of a branch, combined with the possibility of a function call, makes it extremely difficult for an optimizing compiler to improve code that uses the result of a dereference. The Titanium compiler generates C code as its output, which is subsequently compiled to native code by a traditional C compiler. It is imperative that the generated C code avoid constructs that confound optimization in the C compiler.

Benchmarking quantifies these concerns. A Split-C benchmark was run using various strategies to implement wide pointers, which directly correspond to Titanium's wide references. The benchmark, EM3D, repeatedly walks across an irregular bipartite graph performing a simple computation. We can estimate the cost of wide references by computing the average time required per edge when all data is stored in the local memory region. In Table 1, we present times collected on a Thinking Machines CM-5 and partial times collected on a Cray T3D. These findings were originally presented in [15] and [24], respectively.

The benchmark reveals that the performance cost of using wide references for local data can be profound. Even when the code for reading and writing through wide references is inlined, the CM-5 shows nearly a 75% slowdown compared with simple pointers. This is largely due to lost opportunities for optimization. Extensive manual optimization included relocating code into the "local" clause of the locality test to avoid a branch. Even such heroic efforts only bring performance to within 13% of simple pointers, probably due to less effective register use and the increased time cost of moving larger amounts of data around in memory.

2.4 Explicit Qualification

A 75% overhead for every reference is unacceptable in high performance computing. Furthermore, one expects that in a well-crafted concurrent program, communication and data sharing will be kept to a strict minimum to maximize parallelism. Most references will never span regions, and programmers can identify those few that do.

	CM-5	T3D
function	2.8 μsec/edge	1.19
inline	2.0	0.71
optimized	1.3	0.66
narrow	1.15	N/A

Table 1: Costs of overusing wide references. *Benchmark program is the Split-C version of EM3D, which repeatedly performs a simple computation across the edges of an irregular bipartite graph. Values reported are the average cost of traversing a single edge, in microseconds. "Function" uses wide pointers and requires a function call for every read or write. "Inline" inlines wide pointer code directly at the point of use. "Optimized" uses extensive manual optimization and likely represents the theoretical best performance possible for wide references. "Narrow" uses simple pointers, and represents a level of performance only possible with true, physically shared memory.*

Titanium provides an explicit declaration qualifier, "**local**", for just this reason. A reference may be qualified as local to indicate that it never refers to data in a remote memory region. This qualification is available when declaring fields, local variables, formal method arguments, method return types, and even the implicit "**this**" parameter to non-static methods. Wherever a named reference is used, the reference may be qualified as local.

Locally qualified references may be implemented using simple, narrow pointers, neatly avoiding the time and space costs described earlier. Furthermore, because the qualification is part of a reference's type, programmers' claims of locality may be statically checked by the compiler. Allocations produce local references, so the result of a "**new**" expression may be stored in a local reference. Global communication primitives like "**broadcast**" produce global values; storing the result of a broadcast into a local reference is a type error, detectable at compile time.

Revisiting the example from Figure 1, local variable "f" always refers within the local region, and could safely be qualified as local. The qualification appears after the type name, so "f" could be redeclared as "Foo **local** f = ... ;". However, local variable "g" holds the result of a broadcast, and so cannot be qualified as local.

Titanium also allows implicit promotion, or *widening*, of references from local to global. If "f" were a local reference and "g" were global, as described above, then the assignment "g = f" would be perfectly legal. This is analogous to using, for example, a String in a context where an Object is expected. Indeed, these two ideas combine orthogonally. In the statement "Object thing = **new** Foo();", the allocation produces a reference of type "Foo **local**", which is promoted to type "Object **local**" and then widened to type "Object".

Narrowing from global to local requires an explicit cast. At runtime, the global reference is checked to see if it addresses memory in the local region. If it does, then a narrow local reference to the same object is produced. If the global reference genuinely does reach into a different region, though, the cast fails and an exception is thrown. Again, the behavior is analogous to a checked downcast, say from Object down to String.

Titanium has no explicit "**global**" qualifier. Global references are assumed by default. This promotes the notion of a global address space, which simplifies application development for reasons stated earlier. The intention is that programmers can add local qualifications as needed to improve performance once the basic program logic is working.

2.5 Problems Remain

Unfortunately, explicit qualification is only a partial solution. It may not be reasonable to expect programmers to qualify their references aggressively enough to achieve maximal performance. Qualification becomes particularly difficult for compound data types, such as arrays. Java arrays are not truly multidimensional; each level of indexing in an array crosses another reference. At each level, then, the programmer has the option of adding local qualification. The situation quickly becomes ridiculous.

For example, a programmer may wish to establish a private table of Foo instances arranged in rows and columns. The table is never shared with other processes; all data lives within the local region. The full

declaration for such a construct would be:

Foo **local** [] **local** [] **local** myTable;

Reading the declaration from right to left, myTable is a local array of local arrays of local Foo's. In general, an n-dimensional array contains $n+1$ opportunities for local qualification, giving the programmer 2^{n+1} possible declarations. Given this large space of choices, it is likely that the typical programmer will miss many opportunities. The problem will be particularly pronounced if the programmer is porting unfamiliar code originally designed for a shared memory architecture.

The same issues apply when dealing with legacy code. Titanium incorporates a large portion of the standard Java class library into its own runtime environment. The complete contents of the java.io, java.lang, and java.util packages are available to Titanium programmers. The Titanium compiler produces native code directly from Sun's Java source code for these packages. This lets Java programmers step up to SPMD programming while still using familiar classes like String, InputStream, and Hashtable.

However, this large body of existing code was written for Java, not Titanium. These three packages comprise some sixteen thousand lines of source code without a single "**local**" qualifier. None of this code uses Titanium's broadcast or exchange communications primitives; but in the absence of explicit qualifiers, every variable, field, and parameter defaults to being a wide, global reference. Every method is assumed to return global references to its callers, making it even more difficult for programmers to use local references in their own code. Manually annotating this large body of legacy Java code would be extraordinarily tedious, and would need to be redone with each new Java Development Kit release from Sun. Yet without reducing these global references to local, it may be impossible to achieve acceptable levels of performance.

Clearly, then, some more automated mechanism is needed. We would like to statically infer local qualifications wherever possible. Only those references that may actually span regions should remain global. This will give us all of the benefits of a global address space while only deploying wide references where they are actually needed.

3 Formalizing the Analysis

We now formalize the analysis that drives local qualification inference. First, we shall describe a small object-oriented language, Tin. Tin has been crafted to resemble Titanium, but has been greatly simplified. We then develop a set of inference rules that allow us to add local qualifications to well-typed Tin programs. Finally, we describe how to extend the Tin inference system to include other Titanium features.

3.1 Tin: A Tiny Titanium

Tin is a small object-oriented language that resembles a simplified Titanium. Tin is class-based, with explicit subclassing that forms a static tree. Classes are collections of named fields and methods, and allow creation of instances.

The local qualification analysis is intended to function as part of a more complete compiler. We can stipulate that analysis take place after type checking and name resolution. Thus, we can assume that any expression has a known static type, and that this type is available to our qualification inference rules. We also have easy access to information about classes and the class hierarchy, so it is reasonable to refer to the superclass of an arbitrary class, or to the superclass method that a given class method may override.

3.1.1 Tin types

A qualified Tin type T is a pair, consisting of a locality qualifier q followed by an underlying class C. A locality qualifier may explicitly require a local or global reference, or the qualifier may be unknown and therefore subject to inference. A class may be a named class declared elsewhere, or it may be an array of some element type:

$$\begin{aligned} T &::= \langle q\ C \rangle \\ q &::= \textbf{local} \mid \textbf{global} \mid \alpha \\ C &::= \mathit{ClassName} \mid \textbf{array}\ T \end{aligned}$$

Recall the example given earlier of a fully local two-dimensional array of Foo. The Titanium type for this construct was "Foo **local** [] **local** [] **local**". The corresponding Tin type would be:

⟨**local array** ⟨**local array** ⟨**local** Foo⟩⟩⟩

Observe that that Titanium types are most legibly read right-to-left, whereas Tin types are most legibly read left-to-right.

Additionally, we define a distinct set of method types:

$$F \quad ::= \quad \langle T_0 \times T_1 \times \cdots \times T_n \rightarrow T_f \rangle$$

The domain of a method type is required to contain at least one type T_0 corresponding to the ubiquitous implicit "**this**" parameter. Remaining domain types, if any, provide the types of explicit formal parameters. Although method types never appear in the text of a Tin program, they will be useful for certain definitions and inference rules that follow.

3.1.2 Tin class declarations

A Tin program consists of some number of class declarations. Each class declaration specifies the class name and an optional superclass. The body of a class declaration consists of a list of zero or more member declarations:

class *ClassName* [**inherits** *ClassName*] {
    memberDecl
    ...
    memberDecl
}

A member declaration is either a field declaration or a method declaration. A field declaration consists of a qualified type T followed by some field name b. A method declaration consists of a qualified return type T, some method name f, declarations for each formal parameter, and a method body e:

$$\begin{aligned} memberDecl &::= fieldDecl \mid methodDecl \\ fieldDecl &::= T\ b; \\ methodDecl &::= T\ q\ f(T\ x, \ldots, T\ x)\ \{\ e;\ \} \end{aligned}$$

Note the extra locality qualifier q that appears between the return type and method name in a method declaration. This qualifier applies to the implicit **this** parameter. The qualifier determines whether **this** is local or global within the body of the method.

3.1.3 Tin expressions

Tin expressions resemble a restricted subset of Titanium. The complete expression grammar is given in Figure 3.

Method invocation is given by production 1. Local variable declarations are allowed by production 2. The declaration syntax is borrowed from functional programming in order to simplify the inference rules that follow. Local variables and formal method parameters may be accessed using production 3. Production 4 provides access to self in the style of Titanium, Java, and C++. Productions 5 and 6 provide field access and array indexing in the typical manner. For simplicity we intentionally ignore the semantic question of what it means to index an array in a language with no integers.

Destructive assignment is governed by productions 7, 8, and 9. Observe that only certain expressions may receive assignments. One may not, for example, assign to the result of an allocation or method invocation. These restrictions are consistent with the Java definition of "named or computed variables"[12], or what C and C++ define as "lvalues"[4, 3].

Sequential evaluation uses the expected syntax in production 10. New instances of a given class may be allocated using the syntax of production 11. Note that the type here is unqualified; no locality qualifier is needed because all allocation is assumed to take place locally. Lastly, production 12 provides global communication.

$$
\begin{array}{rcll}
e & ::= & e.f(e, \ldots, e) & (1) \\
& | & \textbf{let } x : T = e \textbf{ in } e & (2) \\
& | & x & (3) \\
& | & \textbf{this} & (4) \\
& | & e.b & (5) \\
& | & e[e] & (6) \\
& | & x = e & (7) \\
& | & e.b = e & (8) \\
& | & e[e] = e & (9) \\
& | & e \ ; \ e & (10) \\
& | & \textbf{new } C & (11) \\
& | & \textbf{broadcast } e \textbf{ from } e & (12)
\end{array}
$$

Figure 3: Tin expression grammar.

3.2 Type Constraints

We now define several important relations on components of the Tin type system. Using these relations, we develop a simple language of logical assertions about types. In the inference rules that follow in section 3.3, conjunctive sets of these assertions form the constraint systems that we wish to solve.

3.2.1 Relations on qualifiers

The fixed locality qualifiers form a trivial two-point lattice. Since local references are implicitly promoted to global, we axiomatically define global to subsume local:

$$\frac{}{\textbf{local} \leq q} \qquad \frac{}{q \leq \textbf{global}}$$

Qualifier equality is trivial:

$$\frac{}{\textbf{local} = \textbf{local}} \qquad \frac{}{\textbf{global} = \textbf{global}}$$

3.2.2 Relations on qualified types

Subsumption of qualified types takes two forms, *strong* and *weak*. We say that type T is strongly subsumed by type T' if all locality qualifiers within the complete, deep structure of T' exactly match the corresponding qualifiers within T. However, if type T has deeper structure than type T', we place no additional constraints on those unmatched qualifiers. For example, $\langle \textbf{global array } T \rangle$ is strongly subsumed by $\langle \textbf{global } \text{Object} \rangle$ for any T. The converse, however, does not hold. When type T is strongly subsumed by type T', we write "$T \trianglelefteq T'$". The following rules define strong subsumption:

$$\frac{q = q'}{\langle q \ ClassName \rangle \trianglelefteq \langle q' \ C \rangle} \qquad \frac{q = q' \qquad T \trianglelefteq T'}{\langle q \textbf{ array } T \rangle \trianglelefteq \langle q' \textbf{ array } T' \rangle}$$

Weak subsumption allows promotion from **local** to **global**, but only at the topmost level within the type structure. Deeper levels of array nesting must be qualified identically, using the strong subsumption rules described above. Thus, $\langle \textbf{local array } T \rangle$ is weakly subsumed by $\langle \textbf{global } \text{Object} \rangle$ for any T. However, $\langle \textbf{global array } \langle \textbf{local } C \rangle \rangle$ is not weakly subsumed by $\langle \textbf{global array } \langle \textbf{global } C \rangle \rangle$, because qualifier promotion is not permitted below the topmost level. When type T is weakly subsumed by type T', we write

$$
\begin{array}{rcccc}
r & ::= & q \leq q & | & q = q \\
 & | & T \leq T & | & T = T \\
 & | & F \leq F & | & F = F
\end{array}
$$

Figure 4: Tin constraint grammar.

"$T \leq T'$". The following rules define weak subsumption:

$$\frac{q \leq q'}{\langle q\ ClassName\rangle \leq \langle q'\ C\rangle} \qquad \frac{q \leq q' \qquad T \trianglelefteq T'}{\langle q\ \textbf{array}\ T\rangle \leq \langle q'\ \textbf{array}\ T'\rangle}$$

A pair of qualified types are equal if they have identical deep structure and if all corresponding qualifiers are equal:

$$\frac{q = q'}{\langle q\ ClassName\rangle = \langle q'\ ClassName\rangle} \qquad \frac{q = q' \qquad T = T'}{\langle q\ \textbf{array}\ T\rangle = \langle q'\ \textbf{array}\ T'\rangle}$$

3.2.3 Relations on method types

Subsumption of method types obeys the expected form, with contravariant constraints on subsumed formal parameters.

$$\frac{T_f \leq T'_f \qquad \forall i \in [0,n]\,.\,T'_i \leq T_i}{\langle T_0 \times T_1 \times \cdots \times T_n \to T_f\rangle \leq \langle T'_0 \times T'_1 \times \cdots \times T'_n \to T'_f\rangle}$$

Equality of method types is straightforward:

$$\frac{T_f = T'_f \qquad \forall i \in [0,n]\,.\,T'_i = T_i}{\langle T_0 \times T_1 \times \cdots \times T_n \to T_f\rangle = \langle T'_0 \times T'_1 \times \cdots \times T'_n \to T'_f\rangle}$$

3.2.4 Constraint language

Given these equality and subsumption relations, we can formulate a simple language of logical assertions. This language is presented in Figure 4. We allow assertions on qualifiers, qualified types, and method types. Note, however, that any assertion on a pair of qualified types or method types may be decomposed into a finite set of assertions on qualifiers. The constraint solver ultimately manipulates qualifiers alone, with no knowledge of type structure.

3.3 Qualification Inference for Tin

We now present a set of monomorphic qualification inference rules for Tin programs. The rules are applied in the context of a set of constraints. The constraints represent the minimal required relationships among qualifiers and types within a program. Any set of qualifiers and types that satisfies the constraints yields a correct program. For aggressive optimization, we are interested in the unique set of qualifiers and types that uses "**local**" wherever possible while still satisfying the constraint system.

As stipulated above, the analysis is cast as part of a larger compiler, with rich and complete static type information. The static judgments drawn from this information are summarized in Figure 5. In order to simplify the symbology, we assume that a static, context-sensitive type environment is ubiquitously available rather than threading the type environment through and among all of the inference rules. For example, the inference rule for the "**let**" construct does not explicitly create a type binding for the bound variable. Rather, we assume that preceding compiler stages have already managed this. Similarly, the judgment "**this** : T" refers to the static type of **this** in a particular context; **this** may have a completely different static type in a different context.

$b : T$	Field b has static qualified type T.
$x : T$	Local variable or formal parameter x has static qualified type T.
this $: T$	Self parameter has static qualified type T.
$f : F$	Method f has static method type F.

Figure 5: Static judgments.

$\vdash e : T, R$	Expression e has inferred qualified type T under constraints R.
$\vdash \mathit{methodDecl} : R$	Method declaration *methodDecl* type checks under constraints R.

Figure 6: Inferred judgments.

Figure 6 summarizes the new judgments expressed by the inference rules themselves. Inference on expressions yields an inferred qualified type and a corresponding set of constraints. Inference on method declarations yields a set of constraints for the method as a whole. In general, the inference rule for any construct yields a set of constraints that includes the union all subconstructs' constraints plus some finite set of additional constraints particular to the language construct at hand.

3.3.1 Method invocation

Method invocation is straightforward. We require that the dispatching instance be weakly subsumed by the implied "**this**" parameter. We likewise require that each of the actual parameters be weakly subsumed by the corresponding formal parameter. The type of the entire expression is the result type of the method:

$$\frac{\begin{array}{c} f : F \\ \forall i \in [0, n] \,.\, \vdash e_i : T'_i, R_i \\ F = \langle T_0 \times T_1 \times \cdots \times T_n \to T_f \rangle \\ F' = \langle T'_0 \times T'_1 \times \cdots \times T'_n \to T_f \rangle \\ R = \left(\bigcup_{i \in [0,n]} R_i \right) \cup \{F' \leq F\} \end{array}}{\vdash e_0.f(e_1, \ldots, e_n) : T_f, R}$$

3.3.2 Variable declaration and access

Name binding introduces a new program variable into the type environment for the duration of the binding. The initialization expression must be weakly subsumed by the declared type of the new program variable. When the locality qualification is an unknown α, assume that this is a "fresh" constraint variable that does not already appear in any constraint within the constraint system.

$$\frac{\begin{array}{c} \vdash e_0 : T_0, R_0 \\ \vdash e_1 : T_1, R_1 \\ R = R_0 \cup R_1 \cup \{T_0 \leq T\} \end{array}}{\vdash \textbf{let } x : T = e_0 \textbf{ in } e_1 : T_1, R}$$

Access to a local variable or formal parameter simply requires locating its static type as recorded by the compiler's type checking phase. The "**this**" construct is handled identically:

$$\frac{x : T}{\vdash x : T, \emptyset} \qquad \frac{\textbf{this} : T}{\vdash \textbf{this} : T, \emptyset}$$

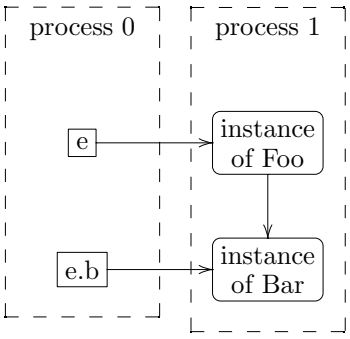

Figure 7: Global objects with local fields. *Suppose that e is of type $\langle$**global** Foo$\rangle$, and that it refers to an instance in a remote region. Suppose that b is a field of type $\langle$**local** Bar$\rangle$. Since b is local, then $e.b$ must be in the same remote region as e. Thus, $e.b$ is also global.*

3.3.3 Field and array access

Field access is subtle. Consider the expression $e.b$. If b was declared as a global field, then $e.b$ is global as well, regardless of the locality qualification of e.

However, if b was declared as a local field, then we must also consider the type of the containing instance e. If e is also local, then we are accessing a local field within a local object, and therefore $e.b$ is local as well.

If b is local and e is global, then we have an interesting situation, presented visually in Figure 7. Such a configuration suggests that e addresses an object from a remote memory region, and that field b within this remote object addresses memory within the same remote region. Thus, if e is global then $e.b$ must be global as well.

In general, then, $e.b$ has the same underlying class as b, but must be qualified as global if either e or b is global. Only when both e and b are local can $e.b$ be local as well. The following inference rule codifies these constraints, where α is a fresh constraint variable:

$$\frac{\begin{array}{cc} \vdash e : T_0, R_0 & T_0 \equiv \langle q_0\ C_0 \rangle \\ b : T_b & T_b \equiv \langle q_b\ C_b \rangle \\ R = R_0 \cup \{q_0 \leq \alpha\} \cup \{q_b \leq \alpha\} & \end{array}}{\vdash e.b : \langle \alpha\ C_b \rangle, R}$$

Array indexing creates an identical set of issues. The inference rule for array access directly parallels the rule for field access presented above, except that we must also include any additional constraints produced by the index expression, where α is again a fresh constraint variable:

$$\frac{\begin{array}{cc} \vdash e_0 : T_0, R_0 & T_0 \equiv \langle q_0\ \text{array}\ T_b \rangle \\ & T_b \equiv \langle q_b\ C_b \rangle \\ \vdash e_1 : T_1, R_1 & \\ R = R_0 \cup R_1 \cup \{q_0 \leq \alpha\} \cup \{q_b \leq \alpha\} & \end{array}}{\vdash e_0[e_1] : \langle \alpha\ C_b \rangle, R}$$

3.3.4 Assignment

Assignments to local variables simply require that type of the value being assigned be weakly subsumed by the type of the recipient:

$$\frac{\begin{array}{c} x : T_x \\ \vdash e : T_e, R_e \\ R = R_e \cup \{T_e \leq T_x\} \end{array}}{\vdash x = e : T_x, R}$$

Assignment to fields is more complex. The issues are similar to those presented earlier for field access. Recall that the special case concerned accessing a local field within a global object. The result of such an

access must always be global. However, consider the consequences of assigning to such a field. If the field access yields a global value, then it appears that one should be able to assign a global value into the field. However, the field is declared local: it is too small to accommodate a wide global reference. Furthermore, the semantics of having a local field dictate that it not cross region boundaries. Thus, we must forbid assignment into local fields within global objects. Phrased differently, when we observe an assignment into a field, if the object containing the field is global, then the field must be global as well. The assigned field must be at least as global as the containing object.

Of course, the field must also be at least as global as the value it is about to receive. This is the same subsumption constraint that applies to simple local variable assignment. Thus we need two constraints, as expressed by the following rule:

$$\frac{\begin{array}{cc} \vdash e_0 : T_0, R_0 & T_0 \equiv \langle q_0\ C_0 \rangle \\ b : T_b & T_b \equiv \langle q_b\ C_b \rangle \\ \vdash e_2 : T_2, R_2 & T_2 \equiv \langle q_2\ C_2 \rangle \\ R = R_0 \cup R_2 \cup \{q_0 \leq q_b\} \cup \{T_2 \leq T_b\} & \end{array}}{\vdash e_0.b = e_2 : T_b, R}$$

Assignment into an indexed array creates an identical set of issues. The inference rule for array assignment directly parallels the rule for field assignment presented above:

$$\frac{\begin{array}{cc} \vdash e_0 : T_0, R_0 & T_0 \equiv \langle q_0 \text{ array } T_b \rangle \\ & T_b \equiv \langle q_b\ C_b \rangle \\ \vdash e_1 : T_1, R_1 & \\ \vdash e_2 : T_2, R_2 & T_2 \equiv \langle q_2\ C_2 \rangle \\ R = R_0 \cup R_1 \cup R_2 \cup \{q_0 \leq q_b\} \cup \{T_2 \leq T_b\} & \end{array}}{\vdash e_0[e_1] = e_2 : T_b, R}$$

3.3.5 Other expression constructs

The remaining inference rules are quite simple.

Sequential evaluation requires that inference be applied to both subexpressions. Borrowing from functional programming, we use the type of the last subexpression as the type of the sequence itself:

$$\frac{\begin{array}{c} \vdash e_0 : T_0, R_0 \\ \vdash e_1 : T_1, R_1 \\ R = R_0 \cup R_1 \end{array}}{\vdash e_0\ ;\ e_1 : T_1, R}$$

Allocation always produces a local reference to the requested underlying class:

$$\frac{}{\vdash \textbf{new}\ C : \langle \textbf{local}\ C \rangle, \emptyset}$$

Broadcasting always yields a global reference with the same underlying class as the expression being transmitted:

$$\frac{\begin{array}{c} \vdash e_0 : \langle q_0\ C_0 \rangle, R_0 \\ \vdash e_1 : \langle q_1\ C_1 \rangle, R_1 \\ R = R_0 \cup R_1 \end{array}}{\vdash \textbf{broadcast}\ e_0\ \textbf{from}\ e_1 : \langle \textbf{global}\ C_0 \rangle, R}$$

3.3.6 Class and method declarations

Inference on a method declaration entails two tasks. First, we must ensure that the inferred type of the method body can be subsumed by the static return type. Second, if the method overrides some superclass

method, we must ensure that the two methods' types match exactly.[2] For any method f, let $super(f)$ be the closest ancestral superclass method overridden by f, or "$\perp$" if no such superclass method exists. Then:

$$\frac{\begin{array}{c} super(f) = \perp \\ f : F \quad F \equiv \langle T_0 \times T_1 \times \cdots \times T_n \rightarrow T_f \rangle \\ \vdash e : T_e, R_e \\ R = R_e \cup \{T_e \leq T_f\} \end{array}}{\vdash T_f \ q \ f(T_0 \ x_0, \ldots, T_n \ x_n) \ \{ \ e; \ \}, R} \qquad \frac{\begin{array}{c} super(f) : F' \\ f : F \quad F \equiv \langle T_0 \times T_1 \times \cdots \times T_n \rightarrow T_f \rangle \\ \vdash e : T_e, R_e \\ R = R_e \cup \{T_e \leq T_f\} \cup \{F = F'\} \end{array}}{\vdash T_f \ q \ f(T_0 \ x_0, \ldots, T_n \ x_n) \ \{ \ e; \ \}, R}$$

Inference upon a class entails inference upon all methods within that class. Inference upon a Tin program entails inference upon all classes within that program. The ultimate constraint system to solve consists of the union of all constraint systems generated for all methods within all classes comprising the complete program.

3.3.7 Solving the constraint system

Given the global constraint system that describes the complete program, we may solve the system for all unknown type qualifiers. Any solution binds all qualifiers to **local** or **global**. The unique *least solution* is the one that binds the greatest number of qualifiers to **local**. This solution represents the most aggressive possible monomorphic use of local references within the program, although a proof of this claim is beyond the scope of this paper.

3.4 Tin to Titanium: Extending the Analysis

The Tin language is but a small subset of Titanium. However, the inference analysis for Tin does address the majority of important issues that arise in a full Titanium analysis. We now sketch how the analysis can be extended to encompass other important Titanium features. These descriptions are intentionally brief, but by now the astute reader should be capable of filling in the missing details.

3.4.1 Simple cases

Tin explicitly distinguishes local, global, and inferred (α) qualification. Titanium provides only explicit local qualification, in the absence of which global qualification is assumed. The Titanium analysis treats all of these implicit global qualifiers as subject to inference. This lets us deploy local qualifiers aggressively in large preexisting code bases, such as the standard Java class library.

Native methods are an exception to this rule. Since the compiler has no access to native code, it cannot simply assume that a native method's formal parameter can be changed from global to local without introducing errors. Similarly, the return value of a native method might be the result of some global communication; inferring a local return type is unsound, since the analysis does not have access to the method's implementation. For this reason, native methods are treated pessimistically. All formal parameters, implicit "**this**" parameters, and return types that are not qualified as local are assumed to be global and not subject to inference. A similar approach could be taken in any situation where only partial information is available. For example, the analysis could accommodate separate compilation by forcing conservative analysis at module interface boundaries.

Titanium's type system includes more than just classes and arrays, but the additional features do not significantly complicate matters. Primitive types have no qualifiers, so primitive expressions simply glide through the analysis without affecting the constraint systems in any way. Immutable classes are similar to normal classes, but immutables do not introduce a qualifier at the topmost level. Both Java and Titanium arrays may be treated like Tin arrays, except that a Titanium array introduces only two qualifiers regardless of its dimensionality.

Interfaces require a simple generalization of the method analysis. We still connect each method to the superclass method that it overrides. However, we also add identical constraints tying each method to any

[2]Instead of requiring equality between overriding method types, we could allow subsumption as defined earlier. However, neither Java nor Titanium grants this flexibility, so we choose to forbid it in Tin as well.

```
Foo    foo     = new Foo();                    // ⊢ foo     : Foo local
Foo []  mine    = { foo };                      // ⊢ mine    : Foo local   [] local
Foo []  yours   = { broadcast foo from 0 };     // ⊢ yours   : Foo global [] local
Object obscure = ... ? mine : yours;           // ⊢ obscure : Object local
Foo []  recover = (Foo []) obscure;             // ⊢ recover : Foo ?????   [] local
```

Figure 8: Problematic use of casts. *Naïve inference produces different qualified types for "mine" and "yours", causing the cast to fail where it would previously have succeeded. One possible conservative treatment would infer qualified type "Foo **global** [] **local**" for all three arrays, and thereby avoid breaking the cast. Remember that Titanium qualified types are most legibly read right-to-left, with "[]" pronounced as "array of".*

interface methods that it implements. This guarantees that each interface method has a consistent calling signature across all implementations.

Most Titanium expressions can be handled without undue difficulty. String literals, for example, are treated as allocations that always return a local string. The conditional expression operator (?:) yields a result that weakly subsumes each of its two alternatives. Structured control flow constructs such as **while** and **if** require no special handling beyond recursive descent into their subcomponents. Most arithmetic operators manipulate and produce primitive types, and thus also require nothing more than recursive descent.

3.4.2 Difficult case: casting

Explicit cast expressions, however, do create a bit of a puzzle, because their purpose is to override the static type system. Casting is badly overloaded: a cast may be used to statically raise or dynamically lower the class of an object; it may be used to statically widen or dynamically narrow the local qualification of a reference; it may be used to guide overloaded method selection or to explicitly document an implicit conversion; it may be used for multiple such purposes simultaneously within a single operation.

Some casts affect only static information, while others are dynamic and can throw exceptions at runtime. Local qualification inference must avoid introducing new runtime exceptions where none existed before, because this would change the observable behavior of the program. Because casts are so overloaded, though, it may be difficult to divine the underlying intent of any single cast. It is correspondingly difficult to avoid changing or corrupting this intent as the analysis changes type qualifiers.

The Titanium code in Figure 8 illustrates a pathological but not uncommon case. In the original code, variables "mine" and "yours" have identical qualified types, and this matches the qualified type of the cast. Thus, the cast always succeeds. After local qualification inference, variables "mine" and "yours" have different qualified types. Thus, the cast fails in one case or the other, no matter what qualified type we infer for the cast expression.

The heart of the problem is the loss of qualifiers when promoting an array to an Object, which leaves us with too little information to properly handle the subsequent recovery of qualifiers in the cast. One possibility would be to use a conservative approach similar to that for native methods. When an array is promoted to Object, all "lost" qualifiers that are not *explicitly* local are constrained to be global. When an Object is cast back to an array, all "recovered" qualifiers are similarly constrained to be global unless already explicitly local. This avoids introducing spurious cast failures at the expense of using local qualification less aggressively than might otherwise be possible. This issue may warrant further investigation in the future.

4 Analysis Implementation

We have partially implemented local qualification inference for the Titanium language. The conservative treatment of arrays proposed in section 3.4.2 has not yet been enacted, but all other language features are fully supported. Also, the analysis currently reports its conclusions textually rather than using them to produce runnable, optimized code. This section describes the implementation approach and presents initial results from running the analysis on Titanium programs.

```
(defnode IfStmtNode (StatementNode) (condition thenPart elsePart)
    "A statement of the form
        if (CONDITION) THENPART else ELSEPART
    ELSEPART may be StmtNode::omitted if absent."
    (emitStatement typecheck single operatorName introduceTemporary makeCfg))
```

Figure 9: Sample AST node type definition. *IfStmtNode is defined as a subclass of the abstract StatementNode, which is itself defined earlier in the same file. The node is to have exactly three child nodes, referred to using accessors named condition, thenPart, and elsePart, also defined earlier. The remainder of the node definition consists of a prose description and a list of supported operations, neither of which is used in this analysis.*

4.1 Design Challenges

The Titanium compiler [25] consists of roughly 38,000 lines of C++ code. The compiler incorporates parsing, type checking and name resolution, several forms of loop optimization, and code generation for a variety of parallel back ends using C as an intermediate language. In the course of type checking and name resolution, the Titanium compiler front end accumulates information about the static properties of a Titanium program. This information is extensive and nontrivial; it is also ever-changing, as the Titanium language itself is still evolving. For these reasons, we wish to reuse as much existing compiler infrastructure as possible when implementing local qualification inference.

The Bane project has been developing a generic, reusable toolkit for constraint-based program analysis [2]. By cleanly separating constraint generation from constraint solving, Bane facilitates rapid prototyping of novel analyses. The core constraint solver engine has been meticulously designed for speed and scalability; building an analysis upon Bane lets researchers benefit from this investment without having to code their own solver for each new analysis.

The first stage in developing a Bane analysis is generally to produce a front end for the language one wishes to study. The analysis then proceeds as a constraint-generating traversal across data structures produced by the front end. As stated above, we wish to reuse existing Titanium compiler technology for this purpose. Unfortunately, integrating these two large software systems is nontrivial: the Titanium compiler is written in C++, whereas Bane is written in Standard ML [16], using the Standard ML of New Jersey compiler and runtime system [5].

Although a C callout interface is being developed for SML/NJ [14], this interface was still rather primitive at the time this research was begun, and was not judged to be suitable for our purposes. Most notably, the interface does not support cyclic data structures, of which the Titanium compiler has many. Instead, the decision was made to more loosely couple the two systems. The Titanium compiler would be modified to serialize the annotated abstract syntax trees (*AST's*) that form its internal program representation. A distinct SML process would parse this representation, reconstituting an equivalent AST in the SML universe. Local qualification inference would then be performed upon this transported AST. The results of the analysis would be serialized and communicated back to the Titanium compiler. The Titanium compiler would incorporate the results of the analysis into its own AST, from which runnable code would be generated.

4.2 Abstract Syntax Tree Serialization

The primary representation of a program within the Titanium compiler is an annotated abstract syntax tree. Nodes of the tree correspond to semantic language features, and are expressed as C++ classes. For example, "IfStmtNode" is the name of a C++ class corresponding to conditional statement nodes in the tree. Each node type has a well-defined set of child nodes, attributes, and operations. An IfStmtNode has three child nodes, corresponding to the condition expression and the two alternative substatement blocks.

The set of AST node types, including their children, attributes, and operations, is defined externally to the core compiler. These definitions use a specialized Lisp-like syntax, an example of which is given in Figure 9. Several filters then use these definitions to generate C++ code which is incorporated into the compiler. This includes such items as class declarations, simple helper and utility functions, and generic traversal routines. The use of a high-level, declarative representation for AST node types greatly simplifies

the task of transporting AST's from C++ to SML. We have composed a new collection of filters that operate on node definitions to produce:

1. C++ code to serialize an AST.
2. ML-Lex code to tokenize a serialized AST stream.
3. ML-Yacc code to parse a serialized AST stream.
4. SML datatype declarations corresponding to AST node types.
5. SML functions for accessing various properties of nodes, such as the type of an expression node or a list of any node's children.

The generated C++ serialization code is incorporated into the Titanium compiler. The ML-Lex and ML-Yacc deserialization code is translated to SML [7, 22], and together with the other generated SML code, forms the SML Titanium front end. Given a serialized AST from the Titanium compiler, the SML Titanium front end reconstitutes an equivalent representation within the SML universe, ready for traversal and Bane constraint generation. Because so much of the front end is generated automatically, the system should be robustly maintainable in the face of future changes to in the Titanium language or Titanium compiler infrastructure.

The serialized representation of an annotated AST uses ASCII text with nesting expressed using a Lisp-like syntax. Such a syntax is extremely easy to parse; furthermore, it is amenable to human examination during testing and debugging. An example of a serialized subtree appears in Figure 10. The code in question is excerpted from the java.io.PrintStream.write(int) method. It encodes the logic that conditionally flushes a buffered output stream upon writing a newline.

Examine the first line of the serialization, which is marked with a double dagger (‡). This line represents the root of the entire subtree. The root is an IfStmtNode, described earlier. The hexadecimal number on that first line is a unique node identifier. Such identifiers are useful when communicating information back to the compiler, or for expressing cyclic structures. The first line also identifies the file name and line number containing the source code to which the node corresponds. This is used for debugging purposes only.

Three subsequent lines are marked with asterisks (*). These correspond to the three child nodes that we have stipulated each IfStmtNode must contain. The first child is a logical "and" expression. The second child is an expression statement consisting of a method call. The third child is the optional "else" clause, which has been omitted in this code and therefore is marked with "@", the serialized analogue of a null pointer.

Three examples of annotations are highlighted with daggers (†). The first and third identify "autoflush" and "out" as fields of the PrintStream class. The second identifies "b" as a formal parameter to the current method. Any other references to the same named entities carry identical declaration annotations, which are used to tie constraints together. Thus, if the field "PrintStream.out" is constrained to be global in any context, these annotations ensure that it is global in all other contexts as well.

4.3 Constraint Generation

Once the SML Titanium front end has reconstructed an annotated AST in the SML universe, local qualification inference may begin. Constraint generation uses two passes over the AST. In the first pass, a structured collection of constraint variables is formed for each declared variable, field, or formal method parameter. A simple reference type requires a single constraint variable; an array requires additional variables depending upon its dimensionality. The collection of constraint variables for each declared entity is associated with that entity's declaration annotation. As described earlier, this allows us to correlate multiple uses of a single entity in different contexts.

The second pass traverses the entire tree bottom-up, inductively accumulating constraints per the inference rules presented earlier. Bane supports several distinct logical constraint calculi, which may be intermixed according to well-structured rules. Local qualification inference uses only the simplest of Bane's constraint languages: that of term unification. With "**local**" and "**global**" as concrete terms, term equality constraints correspond directly to the qualifier equality relation defined in 3.2.1. Terms may also be

```
if (autoflush && (b == '\n'))
    out.flush();
```

(a) Titanium source code

```
‡   (IfStmtNode 0x84f2648 "java/io/PrintStream.java" 75
*       (CandNode 0x84f2478 "java/io/PrintStream.java" 75
            (ObjectFieldAccessNode 0x89b1be0 "java/io/PrintStream.java" 75
                (ThisNode 0x89b1bb0 @
                    (TypeNameNode 0x84f58b8 @
                        (NameNode 0x84f5890 @
                            @
                            "PrintStream"
                            <ClassDecl 0x84f5830 0x84ff2c8>))
                    0)
                (NameNode 0x84f2378 "java/io/PrintStream.java" 75
                    @
                    "autoflush"
†                   <FieldDecl 0x87e6108 0x84f5dd8>))
            (EQNode 0x84f2448 "java/io/PrintStream.java" 75
                (ObjectNode 0x89a6600 "java/io/PrintStream.java" 75
                    (NameNode 0x84f23c8 "java/io/PrintStream.java" 75
                        @
                        "b"
†                       <FormalParameterDecl 0x89a6528 0x84f21d0>))
                (LitNode 0x84f2418 "java/io/PrintStream.java" 75
                    jchar(10))))
*       (ExpressionStmtNode 0x84f25b8 "java/io/PrintStream.java" 76
            (MethodCallNode 0x84f2568 "java/io/PrintStream.java" 76
                (ObjectFieldAccessNode 0x89a6658 "java/io/PrintStream.java" 76
                    (ObjectFieldAccessNode 0x89b1c40 "java/io/PrintStream.java" 76
                        (ThisNode 0x89b1c10 @
                            (TypeNameNode 0x84f58b8 @
                                (NameNode 0x84f5890 @
                                    @
                                    "PrintStream"
                                    <ClassDecl 0x84f5830 0x84ff2c8>))
                            0)
                        (NameNode 0x84f24a8 "java/io/PrintStream.java" 76
                            @
                            "out"
†                           <FieldDecl 0x87e99a0 0x8503998>))
                    (NameNode 0x84f2518 "java/io/PrintStream.java" 76
                        @
                        "flush"
                        <MethodDecl 0x87ec228 0x850afd8 0x0 { }>))
                [
                ]))
*       @)
```

(b) Serialized annotated AST

Figure 10: Sample serialized subtree. *The Titanium code in (a) corresponds to the serialized annotated AST in (b). Special marks (*†‡) highlight certain lines that are referenced in the text; these marks are not part of the serialization proper.*

	library	library + AMR	relative increase
source lines	16,491	18,669	13%
AST nodes	99,195	114,682	16%
AST load time	3:53	4:34	18%
constraints	11,655	15,514	33%
references	8,474	11,461	35%
inferred local	6,681	8,896	33%
local proportion	79%	78%	
analysis time	0:11	0:14	22%

Table 2: Comparative results

constrained to be conditionally equivalent in the manner of [21]. This corresponds to qualifier subsumption. Relations on qualified types and on method types are decomposed into the corresponding relations on qualifiers as suggested in 3.2.4. The aggregate global constraint system may be solved in nearly linear time [20].

4.4 Findings

Eventually, the results of the analysis will be fed back into the Titanium compiler. The unique node identifiers present in the AST serialization will be used to correlate solutions back to AST nodes and types, whose locality qualifiers will be updated to reflect the changes suggested by the analysis. Code generation may then proceed normally. At the moment, this feedback stage has not yet been implemented. Instead, conclusions of the analysis are presented textually for human inspection. Summary statistics are also generated that describe the overall performance and effectiveness of the analysis.

Local qualification inference has been applied to the standard Titanium class library. This includes the standard Java packages java.io, java.lang, and java.util, as well as the ti.lang package which provides certain additional Titanium-specific classes. This represents some sixteen thousand lines of source code without a single explicit **local** qualification. The annotated abstract syntax tree contains 99,195 nodes.

The serialized form of this AST consumes nineteen megabytes, and takes nearly four minutes to deserialize and reassemble in the SML universe. This clearly indicates that the serialization strategy is not viable in the long term. The AST serialization format shown in Figure 10(b) is excessively verbose, and contains considerable redundant information, particularly regarding source files and line numbers. This was helpful during development, but is now a performance liability. Anecdotally, ML-Lex is also known to contain certain inefficiencies that the SML Titanim front end may be stumbling upon. A condensed binary serialization that avoids gratuitous redundancy may help solve both of these problems.

Once the AST has been deserialized and reconstituted, the analysis itself performs quite reasonably. The first pass takes three seconds to preassign constraint variables to declared entities. The second pass requires an additional eight seconds to concurrently generate and solve the constraints. The final constraint system expresses 11,655 relations among locality qualifiers.

The analysis locates 8,474 distinct declared references, each of which is a potential opportunity for local qualification. Of these, 1,793 are ultimately constrained to be global, while the remaining 6,681 are unconstrained and therefore may be optimized to local. Since the standard Java classes perform no Titanium-style global communications, one might be surprised that so many references (21%) were inferred as global. This is primarily due to the many native methods in these classes, as predicted in section 3.4. If a native method is implicitly declared to return a global reference, we cannot infer a local qualification because to do so would break the external implementation of that method. This "globalness" then infects any code that calls a native method, and so on.

The analysis has also been applied to a large scientific program. "AMR" is a full three-dimensional adaptive mesh refinement Poisson solver [25]. AMR has been developed primarily on shared memory machines, and therefore has no explicit local qualifications. Results on AMR are comparable with those on the standard class library, and are summarized in Table 2.

AMR contains sparse internal documentation relative to Sun's Java source. In this sense, AMR code is more dense, which may explain why the AST node count grew more than the source line count. The number of references and constraints grew even more dramatically. This may be due to AMR's extensive use of arrays, which entail an additional reference for each dimension. The relative proportion of references that could be inferred as local decreases only slightly, from 79% to 78%. Although AMR performs numerous global communications, it contains no black-box native methods. Those communications that are present are generally well isolated in the interests of performance.

5 Closing

5.1 Related Work

Classical optimization for distributed parallel programming has focused on extremely regular problems; primarily arrays and computations upon them. Only more recently have researchers turned their attention to the issues created by irregular computations and pointer-intensive data structures. Efforts such as Linda [19], Emerald [10], and Olden [18] have concentrated on runtime cost-reduction mechanisms, such as dynamic migration of data or computation. Prelude [23] uses static program annotations to specify architecture-specific implementation details in otherwise portable parallel codes. This approach is closer in spirit to Titanium's type qualifiers, although the two languages use rather different mechanisms to model and achieve distributed computing.

Orca [8], AC [9], and Split-C [11] share Titanium's notion of local versus remote data, with Orca placing greater emphasis on compiler-driven placement while AC and Split-C give the programmer more direct control. The creators of AC observed that locality qualifications applied to multidimensional arrays and multiply-indirect pointers could lead to declaration syntaxes that are both confusing and baroque. From a language design standpoint, it is interesting to note that both AC and Split-C assume that references are local unless explicitly qualified as global; Titanium uses the opposite approach.

5.2 Conclusions and Future Work

We have described and implemented a static analysis that infers local qualifications on Titanium reference types. The analysis appears to be quite effective in adding qualifications to realistic application code. However, the results of the analysis are not used by the Titanium compiler, so it is difficult to gauge how much the analysis can be expected to boost actual performance. Completing the results feedback portion of the analysis remains the single most significant area for future work.

The Bane analysis engine has both helped and hindered implementation of the analysis. For all of its expressiveness and righteous design, SML remains quite difficult to integrate with larger, preexisting code bases. Integration of SML with the existing Titanium compiler consumed the preponderance of development time for the analysis, and the time required to load an AST dwarfs the time required to analyze it. Yet the Bane engine itself proved quite effective. The analysis is fast, both asymptotically and in practice, and should scale well. Bane's strict distinction between constraint generation and constraint solution forces one to define analyses in a clearer, more formal manner, discouraging *ad hoc* solutions that obscure more than they illuminate.

The analysis as currently described is fairly simple, and one could reasonably argue that a heavyweight tool like Bane is not justified. However, future work may render the analysis more sophisticated, making a high-performance solver like Bane more attractive. One clear area for exploration would be polymorphic analysis of methods. Such an analysis should be particularly effective for many of the standard Java utility classes, like java.util.Vector and java.util.Hashtable, which may store global objects in one context, but local objects in another. Again, it remains to be seen how significant an impact this would have on real programs. The Bane research group is currently developing a more general theory of polymorphic type qualifiers, of which Titanium's "**local**" may be one driving example.

Lastly, we observe that certain Titanium and Java design decisions have a significant impact on the tractability and effectiveness of the analysis. We have highlighted array casts as a source of difficulty, the solution to which may require a more conservative analysis than one might wish. However, we should also

point out that many other aspects of the Titanium language greatly simplify the analysis. For example, the complete lack of references to references (or references to methods) means that we always have good information about the target of an assignment (or destination of a method call). Similarly, Java's safety features allow us to avoid dealing with pointer/integer conversion. A similar analysis for C or C++ would need to be much more conservative and much less complete, or else suffer the added complexity of performing points-to and other analyses. Thus, as others have pointed out [6], we find that safety and analyzability go hand in hand. While Java may seem a strange starting point for high-performance scientific computing, a safe language is ultimately an optimizable one.

5.3 Acknowledgements

This endeavor benefited from the support and interest of many members of the Titanium and BANE research teams. David Gay provided invaluable help navigating through the Titanium compiler, as did Manuel Fähndrich navigating through BANE. The Tin programming language is based on similar work codeveloped with Alex Aiken.

References

[1] Alex Aiken. The BANE Home Page. Available at <http://www.cs.berkeley.edu/Research/Aiken/bane.html>, August 5 1998.

[2] Alex Aiken, Manuel Fähndrich, Jeff Foster, and Zhendong Su. A Toolkit for Constructing Type- and Constraint-Based Program Analyses. In *Proceedings of the Second International Workshop on Types in Compilation (TIC'98)*, 1998.

[3] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *ISO/IEC FDIS 14882 – Programming Languages – C++*, December 1997.

[4] American National Standards Institute, Computer and Business Equipment Manufacturers Association Secretariat, International Organization for Standardization, and International Electrotechnical Commission. *American National Standard for programming languages: C: ANSI/ISO 9899-1990 (revision and redesignation of ANSI X3.159-1989)*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1992. Approved August 3, 1992.

[5] Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[6] Andrew W. Appel. A critique of Standard ML. *Journal of Functional Programming*, 3(4):391–429, October 1993.

[7] Andrew W. Appel, James S. Mattson, and David R. Tarditi. *A lexical analyzer generator for Standard ML*. Department of Computer Science, Princeton University, October 1994.

[8] Henri E. Bal and Frans M. Kaashoek. Object Distribution in Orca using Compile-Time and Run-Time Techniques. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 162–177, October 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number 10.

[9] William W. Carlson and Jesse M. Draper. Distributed data access in AC. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 39–47, Santa Barbara, California, July 1995. IDA Supercomputing Research Center.

[10] Nicholas Carriero, David Gelernter, and Jerry Leichter. Distributed data structures in Linda. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 236–242, St. Petersburg Beach, Florida, January 1986.

[11] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In IEEE, editor, *Proceedings, Supercomputing '93: Portland, Oregon, November 15–19, 1993*, pages 262–273, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1993. IEEE Computer Society Press.

[12] James Gosling, Bill Joy, and Guy Steele. *The Java^{TM} Language Specification*. The Java^{TM} Series. Addison-Wesley, Menlo Park, California, 1996.

[13] Paul N. Hilfinger. Titanium Language Working Sketch. Available at <http://www.cs.berkeley.edu/Research/Projects/titanium/lang-ref.ps>, July 23 1998.

[14] Lorenz Huelsbergen. A Portable C Interface for Standard ML of New Jersey. Technical report, AT&T Bell Laboratories, Room 2C-307, 600 Mountain Ave., Morray Hill, NJ 07974, January 15 1996.

[15] Arvind Krishnamurthy. Analyses and Optimizations for Shared Address Space Programs. Ph.D. qualifying examination talk, November 1995.

[16] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[17] Carleton Miyamoto and Ben Liblit. Themis: Enforcing Titanium Consistency on the NOW. Available at <http://www.cs.berkeley.edu/~liblit/classes/fall97/cs262/themis/>, December 1997. CS262 semester project report.

[18] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A general-purpose programming language. *Software—Practice and Experience*, 21(1):91–118, January 1991.

[19] Anne Rogers, Martin C. Carlisle, John H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[20] Peter Ruzicka and Igor Prívara. An almost linear Robinson unification algorithm. *Acta Informatica*, 27(1):61–71, 1989.

[21] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, Florida, 21–24 January 1996.

[22] David R. Tarditi and Andrew W. Appel. *ML-Yacc User's Manual*. Department of Computer Science, Princeton University, October 6 1994.

[23] W. Weihl, E. Brewer, A. Colbrook, Ch. Dellarocas, W. Hsieh, A. Joseph, C. Waldspurger, and P. Wang. PRELUDE: A system for portable parallel software. *Lecture Notes in Computer Science*, 605:971–??, 1992.

[24] Kathy Yelick, David Culler, and Jim Demmel. Programming Support for Clusters of Multiprocessors (CLUMPs). Talk presented at Lawrence Livermore National Laboratories, March 1997.

[25] Kathy Yelick, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a High-Performance Java Dialect. In *ACM Workshop on Java for High-Performance Network Computing*, pages 1–13, Stanford, California, February 1998. Association for Computing Machinery.