Type Systems for Distributed Data Structures *

Ben Liblit
liblit@cs.berkeley.edu

Alexander Aiken
aiken@cs.berkeley.edu

Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720-1776

Abstract

Distributed-memory programs are often written using a global address space: any process can name any memory location on any processor. Some languages completely hide the distinction between local and remote memory, simplifying the programming model at some performance cost. Other languages give the programmer more explicit control, offering better potential performance but sacrificing both soundness and ease of use.

Through a series of progressively richer type systems, we formalize the complex issues surrounding sound computation with explicitly distributed data structures. We then illustrate how type inference can subsume much of this complexity, letting programmers work at whatever level of detail is needed. Experiments conducted with the Titanium programming language show that this can result in easier development and significant performance improvements over manual optimization of local and global memory.

1 Introduction

While there have been many efforts to design distributed, parallel programming languages, none has been completely satisfactory. Many approaches present the illusion of a single shared, global address space. While easy for programmers to understand, this approach hides the real structure of memory, making it difficult to exploit locality of data. In complex applications where local memory accesses may be orders of magnitude faster than remote accesses, this can seriously harm performance, development time, or both.

Another approach is to reveal the full distributed memory hierarchy at the language level. A popular model is to allow a mixture of *global* and *local* pointers: the former span the entire global address space, while the latter only address memory that is physically colocated with a given processor. This supports globally shared data structures while still allowing efficient implementation of algorithms specifically structured for distributed parallel execution [4, 5, 7, 8, 15, 21, *et al*].

Historically, programming languages that expose mutable local and global addresses have been unsound. Designing a sound type system which allows local and global pointers turns out to be a subtle problem. Exposing local/global also places an additional burden on the programmer, who may be forced to attend to the details of memory layout even in sections of code that are not performance critical.

This paper makes three principal contributions:

- Through a progression of sound type systems, we illustrate and clarify the semantic issues surrounding local and global pointers.
- We present a type inference system that is capable of completing a program with inferred local/global annotations, thereby relieving the programmer from managing address spaces in much or all of the code.
- We present experimental results showing that this inference algorithm improves program performance significantly, simplifies development, and does a better job than hand-optimization by humans.

The remainder of this paper is structured as follows. Section 2 offers a primer on the common terminology with which we discuss distributed address spaces and highlights some of the performance costs of simpler models that treat distributed memory as though it were shared memory. In Section 3 we develop a series of small languages and type systems that codify sound computing with distributed mutable data structures. The more expressive systems are also more complex; Section 4 shows how type inference can simplify programming while retaining the full power of the type system. We have applied these principles to the Titanium programming language, and report the results of our experiments in Section 5. Section 6 reviews related work. We conclude in Section 7 by summarizing our findings, and discussing directions for future research.

2 Background

When describing interconnections between allocated blocks of data, we use the term *pointer*, which reinforces the idea that we are discussing very low level operations. Although pointers can implement Standard ML `ref`'s [29] or Java references [20], pointers are more primitive.

Our distributed memory model is an explicit two-level hierarchy with *local* and *global* memory. Local memory is physically colocated with a processor. A system with sixteen processors has sixteen distinct local memories. A *local pointer* encodes an address within one local memory and corresponds to a pointer or memory address in standard languages. Local pointers do not

*This research was supported in part by NASA Grant No. NAG2-1210, Defense Advanced Research Projects Agency contract No. F30602-95-C-0136, National Science Foundation Infrastructure Grant Nos. CDA-9401156 and EIA-9802069, and an NDSEG fellowship. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

```
if (p.processor == MyProcessor)
    result = *p.address;
else
    result = RemoteRead(p.processor, p.address);
```

Figure 1: **Dereferencing a global pointer.** *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.*

	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 global pointers to local data.** *"Function" uses global pointers and requires a function call for every read or write. "Inline" inlines global pointer code directly at the point of use. "Optimized" uses extensive manual optimization and likely represents the theoretical best performance possible for global references. "Narrow" uses simple pointers, and represents a level of performance only possible with hardware support for shared memory.*

travel well; a local address formed on one processor is meaningless elsewhere.

Global memory is the union of all local memories. If we assume that processors are uniquely numbered, then a global pointer encodes a pair $\langle processor, address\rangle$, with a home processor and an address within that processor's local memory. Global pointers have a different representation from local pointers and are more costly to use. Manipulating remote memory may involve special machine instructions, trapping into the operating system, or function calls into a message-passing library. The exact mechanism is irrelevant. What matters is that global and local pointers have different representations and are manipulated using different operations.

While dereferencing a global pointer to another processor's memory can be extremely slow, even a global pointer into local memory generally incurs a performance penalty. As Figure 1 illustrates, dereferencing a global pointer that turns out to be local may entail 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, may make it difficult for an optimizing compiler to improve code using the result of a statically global dereference.

Benchmarking quantifies these concerns. A Split-C [18] benchmark was run using various strategies to implement global pointers. The benchmark, EM3D, repeatedly walks across an irregular bipartite graph performing a simple calculation. We can estimate the cost of global pointers to local data by computing the average time required per edge when all data is stored locally. Table 1 shows times collected on a Thinking Machines CM-5 and partial times collected on a Cray T3D. These findings were originally presented in [26] and [34], respectively.

The benchmark reveals that the performance cost of using global pointers for local data is significant. Even when the code for reading and writing through global pointers 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. Such heroic efforts bring performance to within 13% of simple pointers; the difference is probably due to less effective register use and the increased time to move larger amounts of data around in memory.

Thus, high performance parallel code must acknowledge the distributed nature of memory. Where data structures genuinely span processor boundaries, global pointers are entirely appropriate. But when static information can prove that data is always local, global pointers are needlessly costly.

3 A Progression of Type Systems

We present a suite of three languages and type systems that offer both global and local pointers, illustrating the key soundness issues that arise when manipulating distributed data structures. All three systems have been reduced to essentials to more clearly illuminate the novel issues. These are not languages in which one would program directly. Rather, these languages should be considered as just barely above the level of primitive machine addressing.

Our foremost concern is distributed data, not mobile code. Therefore, none of the languages we describe contains λ expressions, `let` bindings or any other facility for introducing new functions, variables, or closures. Rather, we assume a fixed set of named functions and variables available in an initial environment. Functions are not first-class; function types are not data types, and function names only appear directly applied to arguments. In Section 7 we briefly consider extensions allowing first-class functions; for now, we focus on data.

Similarly, we omit the details of a parallel semantics. A single language construct, the unary *transmission operator*, represents an explicit transfer of information from one processor to another. An expression of the form "`transmit` e" should be read as evaluating expression "e" on one processor, then transmitting the result to a different processor. The result of a `transmit` expression is the value as seen on the receiving processor. This is the only explicit communication primitive; all other data is exchanged implicitly, via global pointers. The presentation here is deliberately somewhat informal. An operational semantics and soundness proof for the most complex type system are presented in the appendix.

The first language contains local and global pointers with arbitrary levels of indirection but without updates. The second language introduces an assignment operator for destructive updates. The third language adds pairs with updatable fields, which model the composite records, objects, or data structures of higher level languages.

3.1 System I: Simple Pointers

Our first language contains integers, local and global pointers, and basic pointer operations. It has neither destructive assignment nor compound data types; these are added in sections 3.2 and 3.3, respectively. Expression and type grammars are given in Figure 2. Figure 3 gives type checking rules. A type environment, A, encapsulates information about externally defined variable and function names.

To discuss pointers and pointer operations, we work with boxed and unboxed values. As is standard, types represent unboxed values unless explicitly boxed. One may take a value's address using the "$\uparrow$" *indirection operator*, so while "5" is a pattern of bits representing five, "$\uparrow 5$" is a local pointer to a memory location holding the value five. We use "`boxed`" to describe pointer types, augmented with a *width qualifier* to distinguish global from local pointers. The "`widen`" operator widens a local pointer to

$$
\begin{array}{rcl}
\mathbb{J} & ::= & \text{integer literals} \\
e & ::= & \mathbb{J} \mid x \mid f\,e \mid \uparrow e \mid \downarrow e \mid \texttt{widen}\ e \mid \texttt{transmit}\ e \\
\tau & ::= & \texttt{int} \mid \texttt{boxed}\,\omega\,\tau \\
\omega & ::= & \texttt{local} \mid \texttt{global}
\end{array}
$$

Figure 2: **Expressions and types I.** *Expressions are given by e, while τ represents expression types.*

$$
\frac{}{A \vdash \mathbb{J} : \texttt{int}} \qquad \frac{A(x) = \tau}{A \vdash x : \tau}
$$

$$
\frac{A(f) = \tau \to \tau' \quad A \vdash e : \tau}{A \vdash f\,e : \tau'}
$$

$$
\frac{A \vdash e : \tau}{A \vdash \uparrow e : \texttt{boxed local}\ \tau}
$$

$$
\frac{A \vdash e : \texttt{boxed local}\ \tau}{A \vdash \downarrow e : \tau}
$$

$$
\frac{A \vdash e : \texttt{boxed global}\ \tau}{A \vdash \downarrow e : \mathit{expand}(\tau)}
$$

$$
\frac{A \vdash e : \texttt{boxed local}\ \tau}{A \vdash \texttt{widen}\ e : \texttt{boxed global}\ \tau}
$$

$$
\frac{A \vdash e : \tau}{A \vdash \texttt{transmit}\ e : \mathit{expand}(\tau)}
$$

Figure 3: **Type checking rules I.**

global. Hence:

$$
\begin{array}{rcl}
5 & : & \texttt{int} \\
\uparrow 5 & : & \texttt{boxed local int} \\
\uparrow\uparrow 5 & : & \texttt{boxed local boxed local int} \\
\texttt{widen}\ \uparrow\uparrow 5 & : & \texttt{boxed global boxed local int}
\end{array}
$$

The "$\downarrow$" *dereferencing operator* retrieves the value addressed by a pointer. Dereferencing a local pointer works as expected, essentially stripping off an outer level of boxing. Dereferencing a global pointer is more subtle.

3.1.1 Implicit Type Expansion

The difficulty with global pointer dereferencing is illustrated in Figure 4. Dotted lines mark local memory boundaries; in this case, we have two processors and therefore two local memories. Processor 1 has constructed a local pointer to a memory location storing the value five. We indicate local pointers using a single arrow. Processor 0 has a variable x of type `boxed global boxed local int`: a global pointer to a local pointer to an integer. We use double arrows to indicate global pointers. A naïve dereference of x would simply extract the local pointer value $\uparrow 5$. However, that local pointer is meaningless in processor 0's local address space. Rather, as the figure suggests, the local pointer addressed by x must be widened, so that $\downarrow x$ is global as well. The new global pointer's home processor is 1, while its address on processor 1 is the same as the address expressed by $\uparrow 5$.

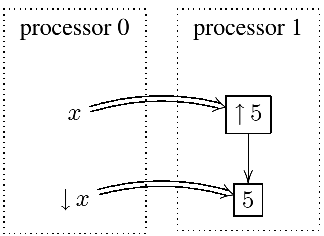

Figure 4: **Situation requiring type expansion.**

$$
\begin{array}{rcl}
\mathit{expand}(\texttt{boxed local}\ \tau) & \triangleq & \texttt{boxed global}\ \tau \\
\mathit{expand}(\tau) & \triangleq & \tau \quad \text{otherwise}
\end{array}
$$

Figure 5: **Type manipulating functions I.**

Widening is only needed when an operation could cause the value of a local pointer to cross processor boundaries. Thus, if y : `boxed global int` is a global pointer to an integer, then $\downarrow y$: `int` is the value of that integer. Similarly, if z : `boxed global boxed global int` is a global pointer to a global pointer to an integer, then $\downarrow z$: `boxed global int` would traverse one level of indirection, yielding a global pointer to an integer. Widening is required when transmitting a local pointer: if $\uparrow 5$ has type `boxed local int`, then `transmit` $\uparrow 5$ must have type `boxed global int`, or else the receiving processor would be left holding a local pointer into the wrong address space. But `transmit` 5 requires no special manipulation, because integers travel safely across processor boundaries.

The *expand* function, used in the final two type rules, is given in Figure 5. It widens local pointers to global, but leaves other types unchanged. Simple though this may seem, real parallel programming languages do not necessarily get this right. Split-C, for example, makes no effort to prevent processors from seeing each other's local pointers. In cases like Figure 4, the programmer is expected to extract the processor number from x and manually combine that with the local pointer at $\downarrow x$ to produce a valid global pointer. Forgetting to do so elicits no warning from the compiler; the program simply contains a wild pointer [17].

3.2 System II: Assignable Pointers

We now extend the language with destructive assignment through pointers. An updated grammar appears in Figure 6. To help support assignment we have also added sequencing.

Given a pointer to some memory location and a compatible value, the new "`:=`" *assignment operator* writes a new value into the pointed-to location, replacing what may have been stored there before. The pointer itself is unchanged; it merely identifies the target of the store operation. This is a more primitive operation than, for example, assignment to an ML `ref`, although ML assignment could be implemented using our primitive plus an extra level of indirection. The key point is that the left hand side of an assignment must always be a pointer, and that the new value is placed in the location to which the pointer refers.

3.2.1 Type Expansion Versus Assignment

Type checking rules for the augmented language are given in Figure 7. As before, the interesting case is a global pointer to local pointer, such as x in Figure 8. Suppose that global pointer x is to receive an assignment, via "x `:=` $\uparrow 6$". The types seem, superficially, to match: x addresses a local pointer to `int`, and $\uparrow 6$

$$
\begin{array}{lcl}
\mathbb{J} & ::= & \text{integer literals} \\
e & ::= & \mathbb{J} \mid x \mid f\,e \mid \uparrow e \mid \downarrow e \mid \texttt{widen}\ e \mid \texttt{transmit}\ e \mid \\
 & & e \,\texttt{;}\, e \mid e \,\texttt{:=}\, e \\
\tau & ::= & \texttt{int} \mid \texttt{boxed}\,\omega\,\tau \\
\omega & ::= & \texttt{local} \mid \texttt{global}
\end{array}
$$

Figure 6: **Expressions and types II.** *Relative to Figure 2, expressions now allow sequencing (* **;** *) and assignment (* **:=** *).*

$$
\frac{}{A \vdash \mathbb{J} : \texttt{int}} \qquad \frac{A(x) = \tau}{A \vdash x : \tau}
$$

$$
\frac{A(f) = \tau \rightarrow \tau' \quad A \vdash e : \tau}{A \vdash f\,e : \tau'}
$$

$$
\frac{A \vdash e : \tau}{A \vdash \uparrow e : \texttt{boxed local}\ \tau}
$$

$$
\frac{A \vdash e : \texttt{boxed local}\ \tau}{A \vdash \downarrow e : \tau}
$$

$$
\frac{A \vdash e : \texttt{boxed global}\ \tau}{A \vdash \downarrow e : \mathit{expand}(\tau)}
$$

$$
\frac{A \vdash e : \texttt{boxed local}\ \tau}{A \vdash \texttt{widen}\ e : \texttt{boxed global}\ \tau}
$$

$$
\frac{A \vdash e : \tau}{A \vdash \texttt{transmit}\ e : \mathit{expand}(\tau)}
$$

....

$$
\frac{A \vdash e : \tau \quad A \vdash e' : \tau'}{A \vdash e \,\texttt{;}\, e' : \tau'}
$$

$$
\frac{A \vdash e : \texttt{boxed local}\ \tau \quad A \vdash e' : \tau}{A \vdash e \,\texttt{:=}\, e' : \tau}
$$

$$
\frac{A \vdash e : \texttt{boxed global}\ \tau \quad A \vdash e' : \tau \quad \mathit{robust}(\tau)}{A \vdash e \,\texttt{:=}\, e' : \tau}
$$

Figure 7: **Type checking rules II.** *Rules above the dotted line are identical to those in Figure 3, while those below the line are new.*

is also a local pointer to `int`. Yet that local pointer would be meaningless if transported from processor 0 across to processor 1. Widening $\uparrow 6$ to global is no solution either, because the box to which x points is typed as local.

In general, then, we must forbid assignment to local pointers

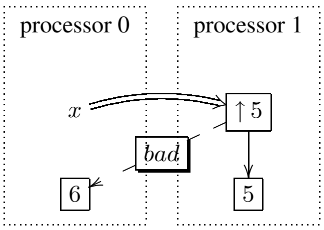

Figure 8: **Situation precluding assignment.**

$$
\begin{array}{rcl}
\mathit{expand}(\texttt{boxed local}\ \tau) & \triangleq & \texttt{boxed global}\ \tau \\
\mathit{expand}(\tau) & \triangleq & \tau \quad \text{otherwise} \\
\\
\mathit{robust}(\texttt{boxed local}\ \tau) & \triangleq & \mathit{false} \\
\mathit{robust}(\tau) & \triangleq & \mathit{true} \quad \text{otherwise}
\end{array}
$$

Figure 9: **Type manipulating functions II.** *The expand function is unchanged from Figure 5. The robust predicate is new.*

across globals. The local pointer value can be read, subject to expansion as seen earlier. But it can never be updated. The core issue is that local pointers cannot travel across processor boundaries, and global pointers use a different and incompatible representation. Figure 9 gives the *robust* predicate that enforces these restrictions. A robust type is one that can safely travel across a global pointer during an assignment. Note that assignment across local pointers requires no such test, as it is always safe providing the source and destination types match.

3.3 System III: Assignable Tuples

Lastly, we enrich the language with tuples. For simplicity, we only permit pairs; general n-tuples contribute nothing novel. The language and type grammars appear in Figure 10. We have added a pair constructor $\langle \_, \_ \rangle$, plus two new operators for decomposing pairs.

Given a valid pointer to a pair, the `@1` and `@2` *pair selection operators* produce offset pointers to the first and second components of the pair. Again, this is more primitive than the `#`n record selection operator from ML, and the two should not be confused. Assuming that ML records are always boxed, ML record selection roughly corresponds to pair selection followed by dereference ($\downarrow$ `@`n). Primitive pair selection alone, without dereference, forms a pointer suitable for assignment, permitting in-place mutation of one component of a pair while leaving the other unchanged. The need for these atypical operators will become more evident in Section 3.3.2.

We have also added a subtyping relation, defined in Figure 11. The subtyping relation allows one to weaken pointer types by promoting certain ρ qualifiers from `valid` to `invalid`. This qualifier subsumption is allowed at the top level or embedded anywhere within a top level pair. However, one cannot change validity qualifiers below a pointer. If this were permitted, then it would be possible for two pointers with different types to alias the same value, which is unsound in the presence of assignment. No implicit changes to the ω qualifier are permitted at all, because this entails a change of representation, and therefore should logically produce a new value.

3.3.1 Consistent Representation of Pairs

As we have seen, when an isolated local pointer moves across processor boundaries, it must be expanded into a global pointer. What about moving an unboxed pair containing a local pointer? One option would be to expand the embedded pointer as before. Thus, $\mathit{expand}(\langle \texttt{boxed local}\ \tau, \texttt{int} \rangle)$ could be defined as $\langle \texttt{boxed global}\ \tau, \texttt{int} \rangle$. However, this means that the expanded pair would have a different representation than the original pair. This approach is very unattractive in any language with named record types (*i.e.*, almost all languages). Suppose the programmer declares `Entry` as a pair $\langle \texttt{boxed local}\ \tau\ , \texttt{int} \rangle$ for some τ. What name would we use for the expanded pair? `Entry` is

$$
\begin{array}{rcl}
\mathbb{J} & ::= & \text{integer literals} \\
e & ::= & \mathbb{J} \mid x \mid f\,e \mid \uparrow e \mid \downarrow e \mid \texttt{widen}\ e \mid \texttt{transmit}\ e \mid \\
 & & e\,;e \mid e\,\texttt{:=}\,e \mid \langle e, e\rangle \mid \texttt{@1}\,e \mid \texttt{@2}\,e \\
\tau & ::= & \texttt{int} \mid \texttt{boxed}\,\omega\,\rho\,\tau \mid \langle \tau, \tau\rangle \\
\omega & ::= & \texttt{local} \mid \texttt{global} \\
\rho & ::= & \texttt{valid} \mid \texttt{invalid}
\end{array}
$$

Figure 10: **Expressions and types III.** *Relative to Figure 6, expressions now allow pair creation ($\langle \_, \_ \rangle$) and selection (@n). Types include pairs, and the pointer types now carry an additional validity qualifier ρ.*

[p]

$$\rho \le \rho \qquad \texttt{valid} \le \texttt{invalid} \qquad \tau \le \tau$$

$$\texttt{boxed}\,\omega\,\rho\,\tau \le \texttt{boxed}\,\omega\,\rho'\,\tau \iff \rho \le \rho'$$

$$\langle \tau_1, \tau_2\rangle \le \langle \tau_1', \tau_2'\rangle \iff \tau_1 \le \tau_1' \wedge \tau_2 \le \tau_2'$$

Figure 11: **Subtyping relation for type system III.**

inappropriate, since the type has changed. Do we synthesize a new name? Assume that the value belongs to some anonymous record type? Any functions that manipulate unboxed `Entry` values cannot properly use the expanded pair, because its representation (and possibly size and component offsets) will have changed. Treating `Entry` as polymorphic in its ω qualifiers would entail either generating multiple copies of code, or else inserting runtime tests wherever polymorphic pointers are used. But code expansion is undesirable and runtime pointer tests are exactly what we wish to avoid.

Thus, we wish to ensure that *expand* never causes a pair to change representation. Local pointers within pairs should remain local, even when copied between processors. Such pointers no longer represent valid memory addresses and must never subsequently be used. We add a new *validity qualifier*, ρ, to mark when an embedded local pointer has been invalidated by movement between processors. Thus, when an unboxed `Entry` is moved across processor boundaries, its embedded local pointer is marked as `invalid`. But the second component of the tuple, an embedded integer, remains accessible. An embedded global pointer would likewise arrive unscathed. Any existing function that manipulates unboxed `Entry` values could still be used, provided that it only accesses the integer, and never touches the (now invalid) local pointer.

Figure 12 presents our final set of type checking rules. The updated *expand* and *robust* functions in Figure 13 complete the picture. A new function, *pop*, is responsible for traversing pairs and invalidating any embedded local pointers. The *robust* predicate, which forbids unsound assignments across global pointers, has been relaxed slightly. Cross-global assignments to valid local pointers are forbidden. But cross-global assignments to invalid local pointers are allowed: if a local pointer is already invalid on the receiving end, one can certainly replace it with a different invalid local pointer. The *robust* and *pop* functions have an important relationship: *robust*(τ) is true if and only if *pop*$(\tau) = \tau$. Intuitively, a value can be assigned across a global pointer if and only if it will not be damaged in transit.

$$\frac{}{A \vdash \mathbb{J} : \texttt{int}} \qquad \frac{A(x) = \tau}{A \vdash x : \tau}$$

$$\frac{A(f) = \tau \to \tau' \quad A \vdash e : \tau}{A \vdash f\,e : \tau'}$$

$$\frac{A \vdash e : \tau}{A \vdash \uparrow e : \texttt{boxed local valid}\ \tau}$$

$$\frac{A \vdash e : \texttt{boxed local valid}\ \tau}{A \vdash \downarrow e : \tau}$$

$$\frac{A \vdash e : \texttt{boxed global valid}\ \tau}{A \vdash \downarrow e : \textit{expand}(\tau)}$$

$$\frac{A \vdash e : \tau}{A \vdash \texttt{transmit}\ e : \textit{expand}(\tau)}$$

$$\frac{A \vdash e : \tau \quad A \vdash e' : \tau'}{A \vdash e\,; e' : \tau'}$$

$$\frac{A \vdash e : \texttt{boxed local valid}\ \tau \quad A \vdash e' : \tau}{A \vdash e\,\texttt{:=}\,e' : \tau}$$

$$\frac{A \vdash e : \texttt{boxed global valid}\ \tau \quad A \vdash e' : \tau \quad \textit{robust}(\tau)}{A \vdash e\,\texttt{:=}\,e' : \tau}$$

....

$$\frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2}{A \vdash \langle e_1, e_2\rangle : \langle \tau_1, \tau_2\rangle}$$

$$\frac{A \vdash e : \texttt{boxed}\,\omega\,\texttt{valid}\,\langle \tau_1, \tau_2\rangle}{A \vdash \texttt{@}n\,e : \texttt{boxed}\,\omega\,\texttt{valid}\,\tau_n}$$

$$\frac{A \vdash e : \tau \quad \tau \le \tau'}{A \vdash e : \tau'}$$

Figure 12: **Type checking rules III.** *Rules above the dotted line are identical to those in Figure 7, or have been changed trivially to support the ρ qualifier. Rules below the line are new.*

$$
\begin{array}{rcl}
\textit{expand}(\texttt{boxed local}\ \rho\,\tau) & \triangleq & \texttt{boxed global}\ \rho\,\tau \\
\textit{expand}(\langle \tau_1, \tau_2\rangle) & \triangleq & \langle \textit{pop}(\tau_1), \textit{pop}(\tau_2)\rangle \\
\textit{expand}(\tau) & \triangleq & \tau \quad \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{pop}(\texttt{boxed local}\ \rho\,\tau) & \triangleq & \texttt{boxed local invalid}\ \tau \\
\textit{pop}(\langle \tau_1, \tau_2\rangle) & \triangleq & \langle \textit{pop}(\tau_1), \textit{pop}(\tau_2)\rangle \\
\textit{pop}(\tau) & \triangleq & \tau \quad \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{robust}(\texttt{boxed local valid}\ \tau) & \triangleq & \mathit{false} \\
\textit{robust}(\langle \tau_1, \tau_2\rangle) & \triangleq & \textit{robust}(\tau_1) \wedge \textit{robust}(\tau_2) \\
\textit{robust}(\tau) & \triangleq & \mathit{true} \quad \text{otherwise}
\end{array}
$$

Figure 13: **Type manipulating functions III.**

3.3.2 Selection Without Dereference

We can now demonstrate why it is important to have pair selection operators that do not also immediately dereference. Suppose that we are given a global pointer to $\langle 4, \langle x, 5\rangle\rangle$, where x is some embedded local pointer. We wish to extract x. If selection is always coupled with dereference, then selecting the second component of the pair would produce the unboxed value $\langle x, 5\rangle$. There is no global pointer associated with this value; we have carried the local pointer x across processors, and can no longer safely use it. Therefore, the *expand* and *pop* functions will have correctly marked x as `invalid`.

However, if selection and dereferencing are distinct operations, we can do better. Given a global pointer to $\langle 4, \langle x, 5\rangle\rangle$, selecting the second component will produce a global pointer to $\langle x, 5\rangle$. Selecting the first component of this yields a global pointer to x. We already know how to use global pointers to local pointers: dereferencing yields a valid global pointer equivalent to `widen` x.

Thus, we find that a sequence of selection operations must not dereference too early. Selection should be treated as simple pointer displacement. When extracting a value deeply embedded in nested pairs, all selection displacements must be applied first, and only then should the final offset pointer be dereferenced.

4 From Checking to Inference

The third system provides address space management, safe pointers, and updatable tuples. This forms a suitable starting point for the design of a realistic language for manipulating distributed mutable data structures. However, it is impractical to expect programmers to systematically annotate programs with `local`, `global`, `valid`, and `invalid` type qualifiers; it is simply too cumbersome and time consuming (see Section 5.1).

Fortunately, the type qualifiers we have described are quite amenable to automatic inference. Figure 14 shows a set of inference rules directly derived from the third type system. One new rule allows implicit coercion of pointers from `local` to `global`. This is allowed at the top level only, both to keep pair types consistent as well as to avoid the well-known soundness problems in allowing distinct aliases of mutable data to have different types. For clarity of presentation, the rules use several abbreviations:

1. Constraints are not explicitly propagated up from subexpressions; assume that the complete constraint set is the simple union of the sets of constraints induced by all subexpressions.

2. A nontrivial rule hypothesis such as

$$e \;:\; \texttt{boxed}\,\omega\,\texttt{valid}\,\tau$$

should be read as an equality constraint

$$e \;:\; \tau_0 \qquad \tau_0 = \texttt{boxed}\,\omega\,\texttt{valid}\,\tau$$

3. All constraint variables are fresh.

The inference rules induce a set of constraints on unknown qualifiers; for example, the operand of any dereference operator is constrained to be qualified as `valid`. Figure 15 shows supporting functions that generate additional constraints. Type qualifier inference requires finding a solution to the set of all constraints induced by a program.

Some constraints generated by the *pop* and *robust* functions have the following general form:

$$\omega_0 = \texttt{global} \implies (\omega = \texttt{global} \vee \rho = \texttt{invalid})$$

These conditional constraints arise whenever data crosses a (possibly global) pointer. For example, when dereferencing a pointer to a pair, if the pointer being dereferenced is global ($\omega_0 = \texttt{global}$), then either a pointer embedded in the pair must also be global ($\omega = \texttt{global}$) or else it must be marked invalid ($\rho = \texttt{invalid}$).

$$\frac{}{A \vdash \mathbb{J} \;:\; \texttt{int}} \qquad \frac{A(x) = \tau}{A \vdash x \;:\; \tau}$$

$$\frac{A(f) = \tau \to \tau' \quad A \vdash e \;:\; \tau}{A \vdash f\,e \;:\; \tau'}$$

$$\frac{A \vdash e \;:\; \tau}{A \vdash \uparrow e \;:\; \texttt{boxed}\,\texttt{local}\,\texttt{valid}\,\tau}$$

$$\frac{A \vdash e \;:\; \texttt{boxed}\,\omega\,\texttt{valid}\,\tau \quad \textit{expand}(\omega, \tau, \tau')}{A \vdash \downarrow e \;:\; \tau'}$$

$$\frac{A \vdash e \;:\; \tau \quad \textit{expand}(\texttt{global}, \tau, \tau')}{A \vdash \texttt{transmit}\; e \;:\; \tau'}$$

$$\frac{A \vdash e \;:\; \tau \quad A \vdash e' \;:\; \tau'}{A \vdash e\,;e' \;:\; \tau'}$$

$$\frac{\begin{array}{c} A \vdash e \;:\; \texttt{boxed}\,\omega\,\texttt{valid}\,\tau \\ A \vdash e' \;:\; \tau \quad \textit{robust}(\omega, \tau) \end{array}}{A \vdash e \,\texttt{:=}\, e' \;:\; \tau}$$

$$\frac{A \vdash e_1 \;:\; \tau_1 \quad A \vdash e_2 \;:\; \tau_2}{A \vdash \langle e_1, e_2\rangle \;:\; \langle \tau_1, \tau_2\rangle}$$

$$\frac{A \vdash e \;:\; \texttt{boxed}\,\omega\,\rho\,\langle \tau_1, \tau_2\rangle}{A \vdash \texttt{@}n\, e \;:\; \texttt{boxed}\,\omega\,\rho\,\tau_n}$$

....

$$\frac{A \vdash e \;:\; \texttt{boxed}\,\texttt{local}\,\rho\,\tau}{A \vdash e \;:\; \texttt{boxed}\,\texttt{global}\,\rho\,\tau}$$

Figure 14: **Type inference rules.** *Rules above the dotted line correspond directly to type checking rules in Figure 12, while the rule below the line is new.*

In general, solving conditional disjunctive constraints is NP-complete, by reduction from satisfiability of boolean formulae in 3-conjunctive normal form. However, we can exploit the particular structure of this inference problem to find a solution more efficiently.

Our goal is to minimize the number of `global` pointers. The conditional disjunctive constraints may leave us with a choice between having a `global valid` pointer and a `local invalid` one. If either would be correct, we will always prefer `local invalid`. Of course, if that pointer is required to be `valid` elsewhere, then `local invalid` is not an option and we must choose `global valid` instead.

The constraints have two important properties. First, the constraints on types can induce constraints on qualifiers, but constraints on qualifiers do not introduce constraints on types. Thus, we can resolve the type constraints to obtain the complete set of qualifier constraints. Second, the conditional qualifier constraints mention only `global`/`local` qualifiers in the antecedents. This observation suggests the following procedure for selecting a best solution of the constraints:

1. Assume that initially we have an unqualified static typing for the program. That is, we know what is a pointer, pair, or integer, but we do not know which pointers are local, global, valid, or invalid.

2. Using the equivalences in Figure 11, expand the type con-

$$\begin{aligned}
\mathit{expand}(\omega_0, \texttt{boxed}\,\omega\;\rho\;\tau, \texttt{boxed}\,\omega'\;\rho'\;\tau') &\triangleq \{\omega_0 \le \omega', \omega \le \omega', \rho = \rho', \tau = \tau'\} \\
\mathit{expand}(\omega_0, \langle \tau_1, \tau_2 \rangle, \langle \tau_1', \tau_2' \rangle) &\triangleq \mathit{pop}(\omega_0, \tau_1, \tau_1') \cup \mathit{pop}(\omega_0, \tau_2, \tau_2') \\
\mathit{expand}(\omega_0, \tau, \tau') &\triangleq \{\tau = \tau'\} \quad \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathit{pop}(\omega_0, \texttt{boxed}\,\omega\;\rho\;\tau, \texttt{boxed}\,\omega'\;\rho'\;\tau') &\triangleq \{\omega_0 = \texttt{global} \Rightarrow (\omega = \texttt{global} \vee \rho' = \texttt{invalid}), \\
&\qquad \omega = \omega', \rho \le \rho', \tau = \tau'\} \\
\mathit{pop}(\omega_0, \langle \tau_1, \tau_2 \rangle, \langle \tau_1', \tau_2' \rangle) &\triangleq \mathit{pop}(\omega_0, \tau_1, \tau_1') \cup \mathit{pop}(\omega_0, \tau_2, \tau_2') \\
\mathit{pop}(\omega_0, \tau, \tau') &\triangleq \{\tau = \tau'\} \quad \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathit{robust}(\omega_0, \texttt{boxed}\,\omega\;\rho\;\tau) &\triangleq \{\omega_0 = \texttt{global} \implies (\omega = \texttt{global} \vee \rho = \texttt{invalid})\} \\
\mathit{robust}(\omega_0, \langle \tau_1, \tau_2 \rangle) &\triangleq \mathit{robust}(\omega_0, \tau_1) \cup \mathit{robust}(\omega_0, \tau_2) \\
\mathit{robust}(\omega_0, \tau) &\triangleq \emptyset \quad \text{otherwise}
\end{aligned}$$

Figure 15: **Constraint generating functions.**

straints $\tau = \tau'$ and $\tau \le \tau'$ to obtain the complete set of qualifier constraints.

3. Solve the unconditional equality and inclusion constraints on ρ variables. Set any ρ variable not required to be `valid` to `invalid`. At this point all ρ variables are resolved.

4. Remove conditional constraints of the form

   $$\omega_0 = \texttt{global} \Rightarrow (\omega = \texttt{global} \vee \texttt{invalid} = \texttt{invalid})$$

   These are always satisfied.

5. Replace conditional constraints of the form

   $$\omega_0 = \texttt{global} \Rightarrow (\omega = \texttt{global} \vee \texttt{valid} = \texttt{invalid})$$

   by simply $\omega_0 \le \omega$.

6. Resolve the conditional and unconditional constraints on ω variables. Set any ω variables not required to be `global` to `local`. Note that the conditional constraints no longer mention ρ variables, so this step cannot introduce an inconsistency. It is easy to show that there is a unique solution minimizing the number of ω variables resolved to `global`. This devolves to graph reachability, computable in time linear with respect to the number of `global` qualifiers in the solution [19, 23].

7. Complete the program by adding a minimal set of explicit `widen` operators wherever the new `local`-to-`global` coercion rule has been used. This is similar to Henglein's *minimal completions* [22], but with neither induced coercions nor projections, and requiring only a linear-time pass across the derivation tree.

We note that setting all possible variables to `global` and `valid` will always produce one legitimate solution to the constraints. Thus, languages that require all pointers to be `global` are safe, albeit overly conservative.

5 Experimental Implementation

5.1 A Practical Need for Sound Inference

Titanium is an experimental language for high-performance parallel computing. Titanium has the syntax and semantics of Java, although it compiles to native machine code rather than virtual machine bytecodes. Titanium extends Java with a distributed global address space, where processes can address, read, and write each other's data across physical machine boundaries [24].

By default, all references in a Titanium program are assumed to be global. This makes it easy to build simple programs that work. It is also a suitable choice for architectures with true shared memory (SMP's), which Titanium also supports. However, when tuning a program for speed, programmers may selectively declare some references as local (*e.g.* within inner loops). If the programmer knows that a large array is always local, a `local` declaration causes the Titanium compiler to produce more efficient code to traverse the local array. The compiler checks explicit `local` qualifiers statically, using rules similar to those presented here. For example, if a method expects a local pointer as a parameter, passing it a global pointer is a simple type error [35].

This design allows programmer to ignore locality issues until the code is running correctly and then add `local` qualifiers to speed things up. However, Titanium does not provide qualifier inference, and experience working with application developers has shown that adding `local` qualifiers by hand is not easy. Arrays of arrays of arrays are bewildering; static type errors are often reported far away from the site of the offending declaration; and the more aggressive one is at adding `local` qualifiers, the harder it is to maintain a valid program in the long run.

Maintenance issues become dominant 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 in Titanium. The Titanium compiler produces native code directly from Sun's Java source code for these packages. Incorporating the standard Java libraries is very desirable: the libraries represent an enormous amount of work that does not need to be repeated.

However, this large body of existing code was written for Java, not Titanium. The three packages comprise sixteen thousand lines of source code without `local` qualifiers. None of this code uses Titanium's cross-processor communication; but in the absence of explicit qualifiers, every variable, field, and method parameter defaults to a global reference. Methods are assumed to return global references, 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 very tedious and would need to be redone with each new release from Sun. Yet without reducing these global references to local, it may be impossible to achieve acceptable performance.

Practical `local` qualification has proven unexpectedly difficult for programmers. Furthermore, formally defining how `local` qualification may be used in a sound manner has been an ongoing source of bugs in the Titanium language design. For these reasons, we have implemented a `local` qualification inference engine, *LQI*, and made it available as an optimization within the Titanium compiler.

5.2 Accommodating Titanium Features

Titanium contains many features not present in the languages presented earlier. However, these may all be handled without difficulty; the core issues of type expansion and pointer validity can be extended to accommodate a realistic language. We briefly describe the highlights.

Titanium is object-oriented, with methods, inheritance, and class- and interface-based polymorphism. A method's actual arguments must match its formals; thus, if a method is observed to receive a global argument in any context, the corresponding formal parameter is constrained to be global within the method body. Titanium permits implicit coercion from local to global, so a method can receive a local argument in one context and a global elsewhere. The local argument is widened at the point of the call.

Native methods, which are implemented by external C code, are treated conservatively. Because the compiler has no access to the implementation, it is never safe to change either the formal parameter types or the return type of a native method. This conservative approach can be taken in any situation where only partial information is available. For example, while the analysis is currently whole-program, it could be made to accommodate separate compilation by forcing conservative analysis at module boundaries.

Inheritance simply induces additional constraints between parent and child classes. A subclass is constrained to use identical types for any fields inherited from its parent. Interfaces and overridden methods are handled in the same manner.

Arrays are treated similarly to references. An array of references is akin to a pointer to an n-tuple of homogeneously-typed pointers. A particularly tricky issue is handling type casts involving arrays. When an array is implicitly cast to `Object`, we forbid changes to any "forgotten" qualifiers below the topmost level of the array type. When an `Object` is dynamically cast back to an array type, we also forbid changes to any "remembered" qualifiers below the topmost level. By holding the qualifiers fixed in both cases, we ensure that any dynamic casts will behave identically in the original and optimized programs. Otherwise, if qualifiers were changed in the array declaration but not the explicit cast, or vice versa, dynamic cast failures would occur where none existed in the original program.

5.3 Local Qualification Inference for Titanium

As implemented in the Titanium compiler, the LQI optimization is slightly less powerful than the inference system presented in Section 4. The initial pass, which identifies references that must remain `valid`, is omitted. Instead, it is assumed that all references must be valid at all times. This is safe, if overly conservative. In some cases, when data is copied across processors but never subsequently used, the validity assumption may force references to be `global` when they could have been `local invalid`.

We have measured the effectiveness of LQI optimization on several numerical kernels and applications. These include:

`cannon` Cannon's algorithm for dense matrix multiplication. We multiply a pair of random 256×256 matrixes.

`lu-fact` LU factorization for dense matrixes. We factor a 1024×1024 element random matrix, partitioned into sixty four 128×128 element blocks. No pivoting is used.

`sample` Sample sort, a distributed sorting algorithm. We sort 2^{20} thirty two bit integer keys, with 64 keys per sample.

`gsrb` The Gauss-Seidel Red Black algorithm for solving elliptic partial differential equations. We solve a 2048×128 element problem, partitioned into four 512×128 element patches across 100 full iterations.

`pps` A parallel solver for elliptic equations with infinite domain boundary conditions, using a two-level domain decomposition approach. We solve a 512×512 element problem partitioned into four 128×128 element patches, with a refinement ratio of 16 between coarse and fine grids.

In all cases, the programs were run in parallel on four nodes of the Berkeley Network of Workstations (*NOW*) [1, 16]. The Titanium runtime system implements cross-processor reads and writes by sending messages from node to node; Active Messages II provides the lightweight fast messaging substrate [28].

Table 2 shows our experimental results. Note that for `cannon` and `lu-fact`, two sets of measurements were taken. The "manual" measurements reflect the code as originally produced by the programmer. In both `cannon` and `lu-fact`, the programmer had already deployed numerous explicit `local` qualifiers in an effort to speed up the code. Thus, the "manual" measurements reflect the additional speedup available from `local` qualification opportunities that the programmer missed, even in these relatively small kernels. The "auto" variants use the same code but with all explicit `local` qualifications removed. These reflect the opposite extreme, where a programmer has relied completely upon LQI.

As one would expect, the manual variants show less relative benefit than their auto counterparts. For `lu-fact`, the programmer has already added so many explicit qualifications as to leave little room for further improvement. However, the same programmer missed a few important spots in `cannon`, even though the entire program is only 180 lines long. LQI was able to discover and optimize these for a 5.7% net speedup.

For both `cannon` and `lu-fact`, manual annotation plus LQI is just slightly faster than LQI alone. Human programmers can add explicit casts that recover `local` qualifiers, but which are only correct due to deep properties of the program that static analysis cannot reveal. This affirms our hypothesis that the best design combines selective manual annotation with aggressive, sound inference.

The measurements as a whole show that improvement varies widely from program to program. In a sense, LQI identifies the portion of a calculation that takes place locally, and optimizes that to run using fast `local` pointers. Thus, the benefit to be gained is directly dependent upon the locality of the underlying algorithms. A program that genuinely uses lots of cross-processor data will harbor few opportunities for `local` qualification. Conversely, an algorithm that has been specifically designed for scalable distributed operation will perform most work locally, and only communicate very rarely. Such algorithms will show larger speedups from LQI, and the relative speedup will become greater when working on increasingly large problems. This is particularly evident in `pps`, a fairly new algorithm that is specifically designed for scalable distributed operation. It performs relatively more local calculations than `gsrb`, but is thereby able to greatly

Benchmark	Effect on Speed			Effect on Code Size		
	Naïve	LQI	Improvement	Naïve	LQI	Improvement
`cannon` manual	53.4 sec	50.3 sec	5.7%	43.5 MB	23.4 MB	46.2%
`cannon` auto	58.1	51.3	13.2%	43.0	23.6	45.2%
`lu-fact` manual	131.4	130.1	< 1.0%	78.1	44.6	42.9%
`lu-fact` auto	227.1	131.3	42.2%	87.4	44.9	48.7%
`sample`	29.2	21.4	26.6%	40.5	20.3	49.8%
`gsrb`	16.0	15.7	1.9%	99.1	64.4	35.0%
`pps`	92.2	40.3	56.3%	545.0	309.8	43.2%

Table 2: **Titanium benchmark performance.**

reduce the amount of cross-processor communication [3]. Because communication is so costly, this gives much better performance in general, and meshes particularly well with LQI, for an impressive speedup. The anecdotal experience of the programmer who wrote `pps` is illuminating. When asked if he had previously put in many explicit `local` qualifiers, he replied "Yes, but apparently not anywhere that it mattered." LQI's analysis is more thorough and 56.3% more effective.

The primary concern of most Titanium programmers is execution speed. However, LQI also makes code smaller. As Titanium is implemented on the NOW, local pointers require many fewer instructions to use. Table 2 shows that LQI makes the benchmarks' code segments 35% to 50% smaller. These sizes exclude code for the standard Java classes, like `String` or `Math`. If the standard classes are included as well, the overall reduction is smaller, from 13% to 18% for a complete executable.

6 Related Work

Nearly one hundred distributed programming languages were identified ten years ago [2], and many more have appeared since. We highlight some representative examples of approaches previously taken to the local/global pointer problem.

Olden adds parallelism to C, focusing on dynamic structures augmented with compiler-directed software caching and migration [10, 11, 32]. All Olden pointers are global, so it is never possible to see an invalid local pointer from another processor's address space. However, pointer operations require four extra instructions to test the processor ID and decode the machine address. Data flow analyses can eliminate some redundant checks, but address decoding always adds one instruction of overhead. The inference described in this paper could complement these analyses, using a faster (unencoded) representation for those pointers that are statically guaranteed to be local.

Emerald also focuses on fine-grained object mobility [25]. While local and global are not distinguished at the source level, selected object fields may be declared as *attached*. Because an object and its transitively attached fields always live in the same address space, the compiler can use fast local addresses to implement attached fields. This is a safe alternative to the techniques presented here, but may require more data motion to keep attached fields colocated as objects migrate.

Cid [30], Split-C, and Titanium explicitly distinguish local and global in the source language. Cid uses a single type for all global pointers, the distributed equivalent of `void *`. Split-C assumes all pointers local unless declared otherwise, while Titanium references default to global. Cid and Split-C make little effort to enforce soundness; while this is consistent with C's low-level approach, the difficulty of distributed debugging compounds the standard issue of wild pointers. Titanium attempts to be as safe Java, and does address some of the issues highlighted in Section 3. However, it does not do so consistently or completely, and one can easily craft unsound expressions. Those remaining holes can now be closed in light of this research.

Compositional C++ [14] also offers explicit local and global pointers. The CC++ language definition states that "a local pointer cannot be accessed through a global pointer." [9] However, it is not clear whether this rule is expected to be enforced at compile time, run time, or not at all. Our experiments with the CC++ compiler reveal that violations of this rule elicit obscure internal error messages from late stages of the compiler, well beyond the point where type checking ought to have approved or rejected such operations. While this may be an improvement over Cid and Split-C's complete lack of static checking, it conservatively forbids many operations that could have been given reasonable, sound semantics. At best, CC++ might be interpreted as statically forbidding roughly the same constructs that we would permit with type expansion.

AC [12] and Unified Parallel C [13] offer alternate models for distributed memory. Each of these languages divides memory into several processor-specific private address spaces plus a single shared space. However, a shared pointer into another processor's private space creates problems akin to those we have previously seen. Such dangerous constructs will pass AC and UPC's type checking rules, but have no defined semantics in either language. AC's creators state that the language "is solidly in the C tradition. Programmers can write efficient programs because they do not have to pay the overhead of software protections." As we have illustrated, efficiency and software protection need not be mutually exclusive; a rich static type system can support both sound language design as well as performance-boosting optimization.

Certain aspects of our approach may be applicable to other models of distributed computing, such as those based on remote procedure calls [6]. Inferred type qualifications might allow specialized marshaling for particular recipients. For example, Java has no global pointers, so when an object is marshaled using Java remote method invocation, all other objects transitively reachable from it must be marshaled as well [33]. Inference of `invalid` qualifiers would let the sender prune this reachability graph if the recipient were known to never traverse certain pointers. Conversely, CORBA objects always reference each other with network-aware handles [31]. Inference of `local` qualifiers could replace some handles with simple local pointers, thereby reducing overhead. In general, any system based on distributed objects may be able to leverage qualification inference to simplify representations of data that never actually span the network.

7 Conclusions and Future Work

Distributed computing environments have distinct notions of local and remote memory. However, explicitly distinguishing between pointer types creates several opportunities for unsoundness. We have described a suite of type systems that clarify these problems and demonstrate how they can be avoided. A simple, asymptotically efficient type inference system can automatically insert an optimal set of qualifiers, reducing the burden on the programmer. Experiments with the Titanium language show that inference can greatly improve performance, particularly for codes specifically designed for scalable distributed execution.

The systems presented here could be enhanced in three important ways. First, the assumption of a two-level memory could be generalized to n levels of partitioned address spaces. This may become important as simple distributed uniprocessors give way to clusters of SMP's, clusters of clusters, and other deep parallel hierarchies. Second, the model should be extended to include mobile code, an area of growing interest. A simple approach may be to require that only *robust* free variables appear in any mobile closure, but more study is needed. Finally, polymorphic analysis of functions could be beneficial. For example, this would let Titanium's LQI automatically produce both `local` and `global` variants of standard container classes like `Vector` or `Hashtable`, for potentially larger improvements to performance.

8 Acknowledgements

Titanium benchmark programs were written by Siu Man Yau, Kar Ming Tang, and Gregory T. Balls. Chris Harrelson adapted the IBANE constraint solver shell for use within the Titanium compiler. Additional support came from members of the Titanium and BANE research groups too numerous to mention, but without whose help this research could not have taken place.

References

[1] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. M. Patterson. Searching for the sorting record: Experiences in tuning NOW-sort. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT-98)*, pages 124–133. ACM Press, 1998.

[2] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, Sept. 1989.

[3] G. T. Balls. *A Finite Difference Domain Decomposition Method Using Local Corrections for the Solution of Poisson's Equation*. PhD thesis, Department of Mechanical Engineering, University of California at Berkeley, 1999.

[4] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 711–718, Philadelphia, 1993. SIAM.

[5] J. Barnes and P. Hut. A hierarchical *O(N* log*N)* force-calculation algorithm. *Nature*, 324(4):446–449, Dec. 1986.

[6] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. In A. L. Ananda and B. Srinivasan, editors, *Distributed Computing Systems: Concepts and Structures*, pages 89–109. IEEE Computer Society Press, 1992.

[7] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, Hilton Head, South Carolina, July 21–24, 1991. SIGACT/SIGARCH.

[8] W. L. Briggs. *A Multigrid Tutorial*. SIAM Books, Philadelphia, 1987.

[9] P. Carlin, K. M. Chandy, and C. Kesselman. The compositional C++ language definition. Technical Report 1993.cs-tr-92-02, California Institute of Technology, Mar. 12, 1993.

[10] M. C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Department of Computer Science, Princeton University, June 1996.

[11] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 29–38, Santa Barbara, California, July 1995. Princeton.

[12] W. W. Carlson and J. 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.

[13] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, May 13 1999.

[14] K. M. Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. *Lecture Notes in Computer Science*, 757:124–144, 1993.

[15] J. Choi, J. Demmel, I. Dhillon, and J. Dongarra. ScaLAPACK: A portable linear algebra library for distributed memory computers — design issues and performance. *Lecture Notes in Computer Science*, 1041, 1996.

[16] D. E. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, and F. Wong. Parallel computing on the Berkeley NOW. In *9th Joint Symposium on Parallel Processing*, Kobe, Japan, 1997.

[17] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, S. Luna, T. von Eicken, and K. Yelick. *Introduction to Split-C*. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley, Apr. 1996. Version 1.0.

[18] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. 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.

[19] J. S. Foster, M. Fähndrich, and A. Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1999.

[20] J. Gosling, B. Joy, and G. Steele. *The Java™ Language Specification*. The Java™ Series. Addison-Wesley, Menlo Park, California, first edition, 1996.

[21] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–348, 1987.

[22] F. Henglein. Dynamic typing. In B. Krieg-Brückner, editor, *Proc. European Symp. on Programming (ESOP), Rennes, France*, pages 233–253. Springer-Verlag, Feb. 1992. Lecture Notes in Computer Science, Vol. 582.

[23] F. Henglein and J. Rehof. The complexity of subtype entailment for simple types. In *Proceedings, Twelth Annual IEEE Symposium on Logic in Computer Science*, pages 352–361, Warsaw, Poland, 29 June–2 July 1997. IEEE Computer Society Press.

[24] P. N. Hilfinger. *Titanium Language Working Sketch*, June 14 1999. Draft version 0.22w.

[25] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb. 1988.

[26] A. Krishnamurthy. Analyses and optimizations for shared address space programs. Ph.D. qualifying examination talk, Nov. 1995.

[27] B. Liblit and A. Aiken. Type systems for distributed data structures. Technical Report CSD-99-1072, University of California, Berkeley, Jan. 25, 2000.

[28] A. Mainwaring and D. Culler. Active message applications programming interface and communication subsystem organization. Technical Report CSD-96-918, University of California, Berkeley, Oct. 14 1996.

[29] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass., 1990.

[30] R. S. Nikhil. Parallel symbolic computing in Cid. *Lecture Notes in Computer Science*, 1068, 1996.

[31] The Object Management Group, Framingham, Massachusetts. *The Common Object Request Broker: Architecture and Specification*, Oct. 1999. Revision 2.3.1.

[32] A. Rogers, M. C. Carlisle, J. 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, Mar. 1995.

[33] Sun. Java remote method invocation specification. Technical report, Sun Microsystems, 1997.

[34] K. Yelick, D. Culler, and J. Demmel. Programming support for clusters of multiprocessors (CLUMPs). Talk presented at Lawrence Livermore National Laboratories, Mar. 1997.

[35] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, Sept. 1998. Special Issue: Java for High-performance Network Computing.

A Operational Semantics and Soundness

In this appendix we prove that the type checking system presented in Section 3.3 is sound with respect to an operational semantics. We focus on the sequential subset of the language, which includes everything except `transmit` expressions. Because the semantic problems with local and global pointers are the representation and movement of pointers between address spaces, dealing with concurrency complicates the semantics while also obscuring the core issues. The language subset we work with is:

$$\begin{array}{rcl} e & ::= & \mathbb{J} \mid x \mid f\ e \mid \uparrow e \mid \downarrow e \mid \texttt{widen}\ e \mid \\ & & e_1\ \texttt{;}\ e_2 \mid e_1 \texttt{:=} e_2 \mid \langle e_1, e_2 \rangle \mid \texttt{@1}\ e \mid \texttt{@2}\ e \end{array}$$

Furthermore, we restrict primitive functions to be mappings from integers to integers. This simplifies the proof without hiding any core issues.

A.1 Semantic Domains

We use the following semantic domains. The treatment of stored pairs is unusual and is explained below.

$$\begin{array}{rcll} M & & & \text{the set of machines} \\ A & & & \text{the set of local addresses} \\ Id & & & \text{the set of identifiers} \\ T & & & \text{the set of all types} \\ G & = & M \times A & \text{global addresses} \\ V & = & \mathbb{J} + A + G + V \times V & \text{values} \\ SV & = & \mathbb{J} + A + G + A \times A & \text{values that can be stored} \\ Store & = & G \rightarrow SV & \\ Fun & = & \mathbb{J} \rightarrow \mathbb{J} & \\ Env & = & Id \rightarrow Fun + V & \end{array}$$

We use the following conventions for naming elements of the semantic domains.

$$\begin{array}{rl} m, m_0, m', \ldots \in M & \text{a machine} \\ v, v_0, v', \ldots \in V & \text{a value} \\ sv, sv_0, sv', \ldots \in SV & \text{a storable value} \\ S, S_0, S', \ldots \in Store & \text{a store} \\ E \in Env & \text{the environment} \\ e, e_0, e', \ldots & \text{a source expression} \\ i, i_0, i', \ldots \in \mathbb{J} & \text{an integer} \\ g, g_0, g', \ldots \in G & \text{a global pointer} \\ a, a_0, a', \ldots \in A & \text{a local pointer} \end{array}$$

In the operational semantics, the use of i, a, or g in a hypothesis should be read as a constraint, not a comment. That is, a hypothesis $e \rightarrow i$ means that e must evaluate to an integer for the rule to be applicable.

We write global addresses as a pair $\langle m, a \rangle$ of machine and local address. Global addresses can be distinguished from pair values $\langle v_1, v_2 \rangle$ by context, as machines cannot be a component of pairs.

A store is a finite function from global addresses to values. When a value is created a new location in the store must be allocated. The function

$$new\ :\ Store \times M \rightarrow A$$

takes a store and a machine m and returns a fresh local address. We also use a shorthand

$$new_n(m, S) \quad = \quad \langle a_1, \ldots, a_n \rangle$$

to simultaneously obtain n distinct fresh addresses in a local memory. By "fresh" we mean that new satisfies:

$$new(m, S) = a \quad \Longrightarrow \quad a \notin dom(\lambda a_0.S(\langle m, a_0 \rangle))$$

In other words, the new address is not already in use on machine m.

Our treatment of pairs is unusual. Unboxed pairs are treated as values, but only pairs of addresses are placed in the store. Because the operations `@1` and `@2` take the addresses of pair components, and because these addresses are then first-class values, we must model the location in the store of the components of the pair as well as the pair itself. This is done most directly by simply storing the two components of the pair at different addresses, rather than more usual solution of representing the entire pair value with a single address. To maintain the knowledge that these two components represent a pair we store the pair of addresses at the address of the pair itself.

For example, consider an unboxed pair consisting of two integers $\langle 5, 6\rangle$. Taking the address $\uparrow\langle 5, 6\rangle$ forces the pair to be placed in the store S. Three new locations on the local machine m are allocated to store the pair:

$$\begin{array}{rcl} S(\langle m, a_1\rangle) & = & \langle a_2, a_3\rangle \\ S(\langle m, a_2\rangle) & = & 5 \\ S(\langle m, a_3\rangle) & = & 6 \end{array}$$

The value of $\uparrow\langle 5, 6\rangle$ is the pair address a_1. Selecting the address of the first field `@1` $\uparrow\langle 5, 6\rangle$ yields the value a_2.

Nested pair values are stored recursively when boxed. Thus the expression $\uparrow\langle\langle 5, 6\rangle, 7\rangle$ allocates five new locations in the local store for the three integers and two pairs:

$$\begin{array}{rcl} S(\langle m, a_0\rangle) & = & \langle a_1, a_4\rangle \\ S(\langle m, a_1\rangle) & = & \langle a_2, a_3\rangle \\ S(\langle m, a_2\rangle) & = & 5 \\ S(\langle m, a_3\rangle) & = & 6 \\ S(\langle m, a_4\rangle) & = & 7 \end{array}$$

In practical language implementations only the "leaf" values 5, 6, and 7 are stored and the knowledge of the grouping of the addresses into pairs is maintained implicitly inside the compiler. The stored pair values are the semantic representation of this compiler knowledge.

Unboxing a nested pair is the inverse of boxing a pair: any stored address pairs are traversed recursively to recreate the unboxed value. In the example just given $\downarrow\uparrow\langle\langle 5, 6\rangle, 7\rangle$ is the value $\langle\langle 5, 6\rangle, 7\rangle$.

A.2 Operational Semantics

Operational rules have the form:

$$m, S_0, E \vdash e \to v, S_1$$

which should be read "on a given machine m in store S_0 and environment E, the expression e evaluates to the value v and produces a new store S_1."

The rules for integer, variable, and function application expressions are simple.

$$\frac{}{m, S, E \vdash i \to i, S} \qquad \frac{E(x) = v \in V}{m, S, E \vdash x \to v, S}$$

$$\frac{\begin{array}{c} m, S_0, E \vdash e \to i, S_1 \\ E(f) = \phi \in Fun \quad \phi(i) = i' \end{array}}{m, S_0, E \vdash f\ e \to i', S_1}$$

The rules for referencing and dereferencing values are more elaborate. We need a number of auxiliary functions. Let $a \cdot \langle b, c\rangle = \langle a, b, c\rangle$ be a tuple append operator. Append may also be applied on the right $\langle b, c\rangle \cdot a = \langle b, c, a\rangle$ and to sets of tuples:

$$a \cdot B \quad = \quad \{a \cdot b \mid b \in B\}$$

A *path* is a tuple with elements appearing in an order described by the regular expression $(\swarrow \mid \searrow)^\star sv$. That is, a path consists of a sequence of $\swarrow$ and $\searrow$, except for the last element which is a storable value. A path describes a sequence of selections within nested pairs (taking either the left or right component) to reach a storable value. We write $t, t_0, t', \ldots$ to denote paths.

A *pure path* is a tuple with elements appearing in an order described by the regular expression $(\swarrow \mid \searrow)^\star$. We write $p, p_0, p', \ldots$ to denote pure paths. Figure 16 defines a number of functions on paths and values.

Taking the address of any value but a pair simply boxes the value by allocating a local address on the current processor and storing the value at that address. As described above, the components of pairs are recursively boxed.

$$\frac{\begin{array}{c} m, S_0, E \vdash e \to v, S_1 \\ \mathit{Paths}(v) = \{p_1, \ldots, p_l, p_{l+1} \cdot sv_{l+1}, \ldots, p_n \cdot sv_n\} \text{ where } p_1 = \langle\rangle \\ \mathit{new}_n(m, S_1) = \{a_1, \ldots, a_n\} \\ sv_i = \langle a_j, a_k\rangle \text{ where } p_i \cdot \swarrow = p_j \text{ and } p_i \cdot \searrow = p_k, \text{ for } 1 \le i \le l \\ S_2 = S_1[\langle m, a_1\rangle \leftarrow sv_1, \ldots, \langle m, a_n\rangle \leftarrow sv_n] \end{array}}{m, S_0, E \vdash \uparrow e \to a_1, S_2}$$

For dereferences there are two cases. For a dereference of a local pointer, we use the auxiliary function *Value* defined in Figure 16 to unbox the value. For a dereference of a global pointer we use auxiliary function *WideValue*, which widens any local pointer appearing at the top level but is otherwise identical to *Value*.

$$\frac{m, S_0, E \vdash e \to a, S_1}{m, S_0, E \vdash \downarrow e \to \mathit{Value}(S_1, \langle m, a\rangle), S_1}$$

$$\frac{m, S_0, E \vdash e \to g, S_1}{m, S_0, E \vdash \downarrow e \to \mathit{WideValue}(S_1, g), S_1}$$

The rules for widening, sequencing, and pairing are straightforward.

$$\frac{m, S_0, E \vdash e \to a, S_1}{m, S_0, E \vdash \texttt{widen}\ e \to \langle m, a\rangle, S_1}$$

$$\frac{\begin{array}{c} m, S_0, E \vdash e_1 \to v_1, S_1 \\ m, S_1, E \vdash e_2 \to v_2, S_2 \end{array}}{m, S_0, E \vdash e_1\ \texttt{;}\ e_2 \to v_2, S_2}$$

$$\frac{\begin{array}{c} m, S_0, E \vdash e_1 \to v_1, S_1 \\ m, S_1, E \vdash e_2 \to v_2, S_2 \end{array}}{m, S_0, E \vdash \langle e_1, e_2\rangle \to \langle v_1, v_2\rangle, S_2}$$

The rule for assignment is complicated by the semantics of assigning into pairs. Assume a is a boxed local pointer to a pair of integers. Then the assignment a `:=` $\langle 1, 2\rangle$ overwrites the two integers of the pair in the store with the integers 1 and 2. This semantics corresponds directly to the structure assignment primitive in the C programming language. The auxiliary functions *LeafAddresses* and *LeafPaths* in Figure 16 provide the mechanism for matching addresses with the values to be assigned. Note that in the case where $S(\langle m, a\rangle)$ and v are not pairs, the sets of leaf addresses and leaf values are just $\{\langle\langle m, a\rangle\rangle\}$ and $\{\langle v\rangle\}$

$$
\begin{array}{rcl}
\mathit{Paths}(v) & = & \begin{cases} \{\langle\rangle\} \cup (\swarrow \cdot \mathit{Paths}(v_1)) \cup (\searrow \cdot \mathit{Paths}(v_2)) & \text{if } v = \langle v_1, v_2\rangle \\ \{\langle v\rangle\} & \text{otherwise} \end{cases} \\
\mathit{LeafPaths}(v) & = & \{x \mid x \in \mathit{Paths}(v) \wedge x = p \cdot sv\} \\
\mathit{LeafAddresses}(S, \langle m, a\rangle) & = & \begin{cases} (\swarrow \cdot \mathit{LeafAddresses}(S, \langle m, a_1\rangle)) & \text{if } S(\langle m, a\rangle) = \langle a_1, a_2\rangle \\ \quad \cup (\searrow \cdot \mathit{LeafAddresses}(S, \langle m, a_2\rangle)) & \\ \{\langle\langle m, a\rangle\rangle\} & \text{otherwise} \end{cases} \\
\mathit{Value}(S, \langle m, a\rangle) & = & \begin{cases} \langle \mathit{Value}(S, \langle m, S(\langle m, a_1\rangle)\rangle), & \text{if } S(\langle m, a\rangle) = \langle a_1, a_2\rangle \\ \quad \mathit{Value}(S, \langle m, S(\langle m, a_2\rangle)\rangle)\rangle & \\ S(\langle m, a\rangle) & \text{otherwise} \end{cases} \\
\mathit{WideValue}(S, \langle m, a\rangle) & = & \begin{cases} \langle m, a'\rangle & \text{if } S(\langle m, a\rangle) = a' \\ \mathit{Value}(S, \langle m, a\rangle) & \text{otherwise} \end{cases}
\end{array}
$$

Figure 16: **Auxiliary functions for boxing, unboxing, and assignment.**

respectively. There are two cases of assignment: one for assigning across a local pointer and one for assigning across a global pointer.

$$
\frac{\begin{array}{c} m, S_0, E \vdash e_1 \rightarrow a, S_1 \\ m, S_1, E \vdash e_2 \rightarrow v, S_2 \\ \mathit{LeafAddresses}(S_2, \langle m, a\rangle) = \{p_1 \cdot g_1, \ldots, p_n \cdot g_n\} \\ \mathit{LeafPaths}(v) = \{p_1 \cdot sv_1, \ldots, p_n \cdot sv_n\} \\ S_3 = S_2[g_1 \leftarrow sv_1, \ldots, g_n \leftarrow sv_n] \end{array}}{m, S_0, E \vdash e_1 \texttt{ := } e_2 \rightarrow v, S_3}
$$

$$
\frac{\begin{array}{c} m, S_0, E \vdash e_1 \rightarrow g, S_1 \\ m, S_1, E \vdash e_2 \rightarrow v, S_2 \\ \mathit{LeafAddresses}(S_2, g) = \{p_1 \cdot g_1, \ldots, p_n \cdot g_n\} \\ \mathit{LeafPaths}(v) = \{p_1 \cdot sv_1, \ldots, p_n \cdot sv_n\} \\ S_3 = S_2[g_1 \leftarrow sv_1, \ldots, g_n \leftarrow sv_n] \end{array}}{m, S_0, E \vdash e_1 \texttt{ := } e_2 \rightarrow v, S_3}
$$

The final four rules implement the `@`n operators, which return the addresses of pair components.

$$
\frac{m, S_0, E \vdash e \rightarrow a, S_1 \qquad S_1(\langle m, a\rangle) = \langle a_1, a_2\rangle}{m, S_0, E \vdash \texttt{@1}\, e \rightarrow a_1, S_1}
$$

$$
\frac{m, S_0, E \vdash e \rightarrow a, S_1 \qquad S_1(\langle m, a\rangle) = \langle a_1, a_2\rangle}{m, S_0, E \vdash \texttt{@2}\, e \rightarrow a_2, S_1}
$$

$$
\frac{m, S_0, E \vdash e \rightarrow \langle m', a\rangle, S_1 \qquad S_1(\langle m', a\rangle) = \langle a_1, a_2\rangle}{m, S_0, E \vdash \texttt{@1}\, e \rightarrow \langle m', a_1\rangle, S_1}
$$

$$
\frac{m, S_0, E \vdash e \rightarrow \langle m', a\rangle, S_1 \qquad S_1(\langle m', a\rangle) = \langle a_1, a_2\rangle}{m, S_0, E \vdash \texttt{@2}\, e \rightarrow \langle m', a_2\rangle, S_1}
$$

A.3 Soundness

Before we can prove type soundness we need to state what representation we expect the values of types to have. Figure 17 defines a predicate *Consistent* that recursively compares a type with a value and a store to check that the value matches requirements of the type. We say that a store S on machine m is *consistent* with value v and type τ if *Consistent*$(m, S, \langle v, \tau\rangle)$ is true. We extend consistency to apply to sets of values and types as well. If U is a set of value/type pairs, then *Consistent*(m, S, U) if and only if *Consistent*(m, S, u) for all $u \in U$.

To prove soundness, there is another issue we must address. Our language allows pointer aliasing, and the language will be unsound if stored pointer values can be given different types by different aliases. In particular,

if $\quad x$: `boxed local valid boxed local invalid` τ

and $\quad y$: `boxed local valid boxed local valid` τ

and x and y happen to refer to the same pointer, then the type system might permit an assignment of an invalid pointer into x, thereby giving y a value that disagrees with its type. The *Consistent* predicate cannot detect this situation; to check this it is necessary to compare all the different typings of each memory address through all of its aliases to ensure they agree.

The function *StoreType* in Figure 18 captures the needed property. A *StoreType* maps mutable locations to types, $\perp$, or $\top$. The ordering of elements is $\perp \leq \tau \leq \top$, with all types τ being incomparable. The least upper bound of two elements is the smallest element that is $\geq$ to both. The least upper bound of two functions is defined point-wise:

$$
(f \sqcup f')(x) \quad = \quad f(x) \sqcup f'(x)
$$

If a store typing st has the property that $st(g) = \top$, then the location g is typed differently by two or more aliases of the location; in this case we say the store typing st is *not uniform*. If there is no g such that $st(g) = \top$ then all of the aliases of all mutable locations agree on the types of those locations: the store typing is *uniform*. Predicate *Uniform* in Figure 18 formalizes this notion.

Data that is immutable need not have the same typing for every alias. *StoreType* does not require the top-level pointer encountered in its traversal of a value to have a uniform view everywhere. This pointer is not itself mutable, only the data it points to is mutable.

Finally, the full notion of soundness we need simultaneously confirms that the execution and type environments also agree. For this purpose it is useful to combine the two environments pairwise, matching each variable's value with its corresponding type:

$$
E \bowtie A \quad = \quad \{\langle E(x), A(x)\rangle \in \mathcal{U} \mid x \in \mathit{dom}(E) \cap \mathit{dom}(A)\}
$$

$$
\begin{array}{rcl}
\mathcal{U} & = & V \times T \\
U & \in & 2^{\mathcal{U}} \\
u, u_0, u', \ldots & \in & \mathcal{U} \\
\\
\textit{Consistent} & : & M \times \textit{Store} \times \mathcal{U} \rightarrow \textit{boolean} \\
\\
\textit{Consistent}(m, S, \langle i, \texttt{int} \rangle) & \Longleftrightarrow & \textit{true} \\
\textit{Consistent}(m, S, \langle a, \texttt{boxed local invalid } \tau \rangle) & \Longleftrightarrow & \textit{true} \\
\textit{Consistent}(m, S, \langle g, \texttt{boxed global invalid } \tau \rangle) & \Longleftrightarrow & \textit{true} \\
\\
\textit{Consistent}(m, S, \langle \langle v_1, v_2 \rangle, \langle \tau_1, \tau_2 \rangle \rangle) & \Longleftrightarrow & \textit{Consistent}(m, S, \langle v_1, \tau_1 \rangle) \\
& \wedge & \textit{Consistent}(m, S, \langle v_2, \tau_2 \rangle) \\
\\
\textit{Consistent}(m, S, \langle a, \texttt{boxed local valid } \tau \rangle) & \Longleftrightarrow & S(\langle m, a \rangle) \text{ is defined} \\
& \wedge & \textit{Consistent}(m, S, \langle S(\langle m, a \rangle), \tau \rangle) \\
& & \text{where } \tau \neq \langle \tau_1, \tau_2 \rangle \\
\\
\textit{Consistent}(m, S, \langle a, \texttt{boxed local valid } \langle \tau_1, \tau_2 \rangle \rangle) & \Longleftrightarrow & S(\langle m, a \rangle) = \langle a_1, a_2 \rangle \\
& \wedge & \textit{Consistent}(m, S, \langle a_1, \texttt{boxed local valid } \tau_1 \rangle) \\
& \wedge & \textit{Consistent}(m, S, \langle a_2, \texttt{boxed local valid } \tau_2 \rangle) \\
\\
\textit{Consistent}(m, S, \langle \langle m', a \rangle, \texttt{boxed global valid } \tau \rangle) & \Longleftrightarrow & S(\langle m', a \rangle) \text{ is defined} \\
& \wedge & \textit{Consistent}(m', S, \langle S(\langle m', a \rangle), \tau \rangle) \\
& & \text{where } \tau \neq \langle \tau_1, \tau_2 \rangle \\
\\
\textit{Consistent}(m, S, \langle \langle m', a \rangle, \texttt{boxed global valid } \langle \tau_1, \tau_2 \rangle \rangle) & \Longleftrightarrow & S(\langle m', a \rangle) = \langle a_1, a_2 \rangle \\
& \wedge & \textit{Consistent}(m, S, \langle \langle m', a_1 \rangle, \texttt{boxed global valid } \tau_1 \rangle) \\
& \wedge & \textit{Consistent}(m, S, \langle \langle m', a_2 \rangle, \texttt{boxed global valid } \tau_2 \rangle) \\
\\
\textit{Consistent}(m, S, U) & \Longleftrightarrow & \bigwedge_{u \in U} \textit{Consistent}(m, S, u)
\end{array}
$$

Figure 17: **Consistent stores.**

For the soundness proof we require that the execution and type environments agree from the outset; that is, $dom(E) = dom(A)$.

Because we do not have any iteration constructs in our small language, all computations are terminating. We can use this fact to sidestep the usual issues with showing type soundness even for infinite computations. We simply show that if an expression has any type then computation never goes wrong, provided the computation is performed in an environment consistent with the typing assumptions.

A.4 Main Soundness Theorem

Theorem 1. Let $A \vdash e : \tau$. Assume that m is a machine, S is a store, and E is an environment such that $dom(E) = dom(A)$. If initially

$$\textit{Uniform}(\textit{StoreType}(m, S, E \bowtie A))$$

then

$$
\begin{array}{rl}
& m, S, E \vdash e \rightarrow v, S' \\
\wedge & \textit{Consistent}(m, S', (E \bowtie A) \cup \{\langle v, \tau \rangle\})
\end{array}
$$

i.e., computation succeeds and ends in a state where all values have types consistent with the store.

The proof is omitted from this summary, but the interested reader can find the complete version in [27].

$$
\begin{array}{rcl}
ST & = & G \to (\tau + \perp + \top) \\
\mathit{StoreType} & : & M \times \mathit{Store} \times \mathcal{U} \to ST \\
\\
\mathit{StoreType}(m, S, \langle i, \mathtt{int} \rangle) & = & \lambda x.\perp \\
\mathit{StoreType}(m, S, \langle a, \mathtt{boxed\ local\ invalid}\ \tau \rangle) & = & \lambda x.\perp \\
\mathit{StoreType}(m, S, \langle \langle m', a \rangle, \mathtt{boxed\ global\ invalid}\ \tau \rangle) & = & \lambda x.\perp \\
\\
\mathit{StoreType}(m, S, \langle \langle v_1, v_2 \rangle, \langle \tau_1, \tau_2 \rangle \rangle) & = & \mathit{StoreType}(m, S, \langle v_1, \tau_1 \rangle) \\
& & \sqcup\ \mathit{StoreType}(m, S, \langle v_2, \tau_2 \rangle) \\
\\
\mathit{StoreType}(m, S, \langle a, \mathtt{boxed\ local\ valid}\ \tau \rangle) & = & \lambda x.\perp [\langle m, a \rangle \leftarrow \tau] \\
& \sqcup & \mathit{StoreType}(m, S, \langle S(\langle m, a \rangle), \tau \rangle) \\
& & \text{where } \tau \neq \langle \tau_1, \tau_2 \rangle \\
\\
\mathit{StoreType}(m, S, \langle a, \mathtt{boxed\ local\ valid}\ \langle \tau_1, \tau_2 \rangle \rangle) & = & \lambda x.\perp [\langle m, a \rangle \leftarrow \langle \tau_1, \tau_2 \rangle] \\
& \sqcup & \mathit{StoreType}(m, S, \langle a_1, \mathtt{boxed\ local\ valid}\ \tau_1 \rangle) \\
& \sqcup & \mathit{StoreType}(m, S, \langle a_2, \mathtt{boxed\ local\ valid}\ \tau_2 \rangle) \\
& & \text{where } S(\langle m, a \rangle) = \langle a_1, a_2 \rangle \\
\\
\mathit{StoreType}(m, S, \langle \langle m', a \rangle, \mathtt{boxed\ global\ valid}\ \tau \rangle) & = & \lambda x.\perp [\langle m', a \rangle \leftarrow \tau] \\
& \sqcup & \mathit{StoreType}(m', S, \langle S(\langle m', a \rangle), \tau \rangle) \\
& & \text{where } \tau \neq \langle \tau_1, \tau_2 \rangle \\
\\
\mathit{StoreType}(m, S, \langle \langle m', a \rangle, \mathtt{boxed\ global\ valid}\ \langle \tau_1, \tau_2 \rangle \rangle) & = & \lambda x.\perp [\langle m', a \rangle \leftarrow \langle \tau_1, \tau_2 \rangle] \\
& \sqcup & \mathit{StoreType}(m, S, \langle \langle m', a_1 \rangle, \mathtt{boxed\ global\ valid}\ \tau_1 \rangle) \\
& \sqcup & \mathit{StoreType}(m, S, \langle \langle m', a_2 \rangle, \mathtt{boxed\ global\ valid}\ \tau_2 \rangle) \\
& & \text{where } S(\langle m', a \rangle) = \langle a_1, a_2 \rangle \\
\\
\mathit{StoreType}(m, S, U) & = & \bigsqcup_{u \in U} \mathit{StoreType}(m, S, u) \\
\\
\mathit{Uniform} & : & ST \to \mathit{boolean} \\
\mathit{Uniform}(st) & \iff & \nexists g.\, st(g) = \top
\end{array}
$$

Figure 18: **Uniform store typings.**