Type Inference for Locality Analysis of Distributed Data Structures

Satish Chandra, IBM Research

Joint work with Ras Bodik (Berkeley), Vijay Saraswat (IBM), and Vivek Sarkar (Rice)

Mysore Park “Chemistry” workshop, Feb 2011
Partitioned Global Address Space (PGAS)

X10, Titanium, UPC, ..
Partitioned Global Address Space

```
int x
T f
T g
```

HERE

T p

T q

T r

q = p.f
r = p.g
r.x
Partitioned Global Address Space

objects don’t migrate, so migrate the computation over there to read a value
Partitioned Global Address Space

\[ \text{HERE} \]

\[ \text{T q} \]

\[ \text{T p} \]

\[ \text{int x} \]

\[ T f \]

\[ T g \]

\[ \text{at (r)xr.x} \]

\[ \text{at (q)xq.x} \]

\[ \text{T r} \]

\[ \text{THERE} \]
Partitioned Global Address Space

HERE

\[ \begin{align*}
T_p & \rightarrow \text{int } x \\
T_f & \rightarrow T_g \\
\end{align*} \]

at (r) r.x

T q

T r

at (r) q.x

THERE
Part I: Type System

Goal: In a distributed program with at statement, ensure that all dereferences are local
Place Types

- \( T\@! \): a pointer to an object of type T located HERE
- \( T\@? \): a pointer to an object of type T located anywhere

- Rule 1: An \( @? \) value cannot be dereferenced
- Rule 2: An \( @? \) value cannot be assigned to a \( @! \) variable

\[
T\@! \ p = \text{new} \ T(); \quad \text{new objects are allocated here}
\]
\[
p.\text{foo}();
\]
\[
T\@? \ r = p.\text{bar}();
\]
\[
\text{int} \ y = r.x;
\]
\[
T\@! \ t = r;
\]
\[
T\@! \ t = (@!) \ r;
\]

Rule 1

Rule 2

Type conversion with a dynamic check
Place Types

• Rule 3. For \texttt{at (x) e}
  – If \( x \) is \(@!\), type check \( e \) as usual
  – If \( x \) is \(@?\), type check \( e \) with \textbf{place shifting}
    • type of \( x \) is \(@!\)
    • type of remaining variables is \(@?\)
  – Return value is always \(@?\)

\begin{verbatim}
T@? r = ...
T@? s = at (r) r.g;  \textcolor{green}{r.g \textit{type checked assuming} r \textit{is} T@!}
int y = at (r) q.x;  \textcolor{red}{q.x \textit{type checked assuming} q \textit{is} T@?}
\end{verbatim}
Enriching the type system

- Existing machinery does not track place equality

```
T@? r;
T@r s = at (r) new T();
at (r) {s.f = ...};
```

\( s.f \) is type checked assuming \( s \) is @!
Tracking information across method calls
(encoding class invariants)

class T {
  S@! u;
  final S@? v;
  S@v w;
  T(S s) {
    v = s; w = v; u = new S();
  }
  m1() { u.foo();}
  m2() {
    at (w) { v.bar(); }
  }
}

Since v is @w, v.bar() is type checked assuming v is @!

Constructor respects the place types of fields

u.foo() is executed at the place of enclosing object
class BoxT(p) {
    final S@? p;
    T@p data;
}

final S@? q = ... BoxT(p=q) @! bt = new BoxT(q);
...
T@q d = bt.data;
at (q) d.access;

Place of a final field is a “property”, and can be remembered in the type of the instance

Place Types
Main Ideas so far …

• Remember in a variable’s type the place where the object lives
  – T@!
  – T@?
  – T@v
  – T(p=q)

• Equality of places

• A place-type-checked program never fails because of a dereference at incorrect place
Example: Evenly Distributed Binary Tree
Example (contd.)

class DarkNode {
    DarkNode@! left;
    DarkNode@! right;

    int count () {
        return left.count() +
        right.count() + 1;
    }
}

class LightNode {
    Node@? left;
    Node@! right;

    int count () {
        int l = at (left) left.count();
        int r = right.count();
        return l + r + 1;
    }
}
Part II: Inference of Place Types
Why Inference?

- Writing place types can be cumbersome
  - Many times the types of intermediate variables are “obvious”
- Help in understanding/migrating existing code
  - Type inference can tell what it believes are the place annotations for a program to run without place errors
- Compiler can use it to eliminate run-time place checks
Intuitions

1. If \( p \) is dereferenced HERE, place of \( p \) must be **here**

2. If \( p \) is dereferenced at (\( q \)), place of \( p \) must be equal to the place of \( q \)

3. If \( p.f \) gets the value of \( r \), ...

\[ \alpha = \text{here} \]
4. If \( p.f \) gets the value of \( r \), but elsewhere in the program:
   - \( s \) gets the value of \( r.g \)
   - \( v \) gets the value of \( p.f \)
   - \( t \) gets the value of \( v.g \)
Expect place of \( s \) and \( t \) to be the same

\[ \alpha = \text{here} \quad \beta = \gamma \]
Inferred Type Expressions
Type Inference Algorithm

- Keeps track of which place variables must be equal
- Based on unification
- Fails when forced to merge unequal places
  - Unequal places originate from `v.location.next()`
  - Also from distributed arrays
There’s Something About Equality

- Hindley-Milner polymorphic type inference is based on structurally the same underlying equality-based constraint system
- FUN, LIST, INT, BOOL, PAIR are “place constants” that cannot be equivalenced with other place constants
Not presented today … but paper available

• Details of the type inference algorithm
• *Place polymorphism*
• Context-sensitivity in inference of program with multiple procedures
• Handling of recursion
• Treatment of distributed arrays
Interaction with the Programmer

- In general, inference would figure out the right place annotation for local variables and fields.

- However, type inference cannot figure out when @? is needed.
  - Might fail when trying to equivalence unequal places.

- A programmer must determine when to supply a @?, so that those variables (or fields) do not create a place equality constraint.
Distributed Arrays

- Arrays have a distribution: map from index to place
- The following is place correct:

  ```java
  T@! [.] a = ...
  
  T@! t = new T();
  a[i] = t;
  }
  
  for each i, the place of the content of a[i] is the same as where the slot is located
  ```

- We augment the type system to support distributed arrays
  - Introduce an indexed place variable (δ, i), where δ is a distribution
Putting it all together: Distributed Hash Table

distribution \texttt{d} = ...

\texttt{Bucket@! [.] buckets = new Bucket[d];}

\texttt{public void put(K key, V val) {}
    int hash = key.hashCode() \% d.size;
    \texttt{at (d[hash]) {}
        Bucket b = buckets[hash];
        while (b != null) {
            if (b.k.equals(key)) {
                atomic {b.v = val; }
                return;
            }
            b = b.next;
        }
        \texttt{b = b.next;}
    }
    \texttt{atomic {}
        buckets[hash]=new Bucket(key,val,buckets[hash]);
    }
};

Place Types
Distributed Hash Table

```java
public void put(K key, V val) {
    int hash = key.hashCode() % d.size;
    at (d[hash]) {
        Bucket b = buckets[hash];
        while (b != null) {
            if (b.k.equals(key)) {
                atomic {b.v = val; }
                return; // from async
            }
            b = b.next;
        }
        atomic {
            buckets[hash]=new Bucket(key,val,buckets[hash]);
        }
    }
}
```

distribution d = …

Bucket[] buckets = new Bucket[d];

class Bucket {
    K k;
    V v;
    final Bucket[] next;
}
...
```
Related Work

• Type systems for PGAS languages
  – Liblit and Aiken [POPL 2000], Zhu and Hendren [TPDS 07]
  – Distinguish between local and remote pointers
• Base constraint system similar to ML-style type inference
  – Lots of results carry over
• Region-based memory management
  – Also based on equality-based constraint system
  – Equivalence criteria is different: which objects have equal live ranges
Summary

• A place type system to prove that a distributed address space program does not dereference a remote reference
• Based on an equality-based constraint system
  – Therefore admits standard type inference machinery
  – Treatment of distributed arrays is new
• Implemented for X10/Java
  – Not in the public release
Implementation in X10 compiler

- An implementation of type inference has been created in the X10 compiler
- Uses polyglot front end
- Bridge to WALA (wala.sf.net)
- Analysis implemented in WALA
  - Potentially useful for other problems that can be modeled using an equality-based constraint system
Distributed Arrays

\[ a[i] = t; \]

\[ a \rightarrow \delta \]

\[ i \]

\[ t \rightarrow (\delta, i) \]

\[ (\delta, 1), (\delta, 2), (\delta, i) \]

\[ (\delta, i) = \text{here} \]

\[ \text{at } (D[i]) \{ a[i] = t \} \]

\[ a \rightarrow D \]

\[ (D, 1), (D, 2) \]

\[ (\delta, i) = \text{here} = (D, i), \text{ therefore } \delta = D. \text{ Cannot unify unequal indices.} \]
Enriching the type system

- Existing machinery does not track place equality

\[ T \Rightarrow r; \]
\[ T \Rightarrow s = \text{at}(r) \text{ new } T(); \]
\[ \text{at}(r) \{ s.f = \ldots \}; \]
\[ s.f \text{ is type checked assuming } s \text{ is } @? \]

- \( T\Rightarrow v \): an object of type \( T \) located at the same place as \( v \) (final, i.e. a constant)
  - Minor change in type checking rules

\[ T \Rightarrow r \ s = \text{at}(r) \text{ new } T(); \]
\[ \text{at}(r) \{ s.f = \ldots \}; \]
\[ s.f \text{ is type checked assuming } s \text{ is } @! \]
Tracking Places Through Data Structures

class BoxT {
    final S@? p;
    T@p data;
}

final S@? q = ...
BoxT@! bt = new BoxT(q);
...
T@? d = bt.data;

at (q) d.access;  d.access is type checked
assuming d is @? ; correlation is lost
Type Inference Algorithm

- Each variable \( p \) is associated with a type variable \( \sigma_p \)
- For each field selector \( f \), let \( f(\sigma_p) = \sigma_{p.f} \) (new)
- Associate each type variable \( \sigma \) with a fresh place variable \( \alpha \)
- Place constraints are enforced by equivalencing place variables \( (\alpha, \beta, \text{here}, ...) \)
- At assignment, unify the lhs and rhs type variables
- Unification of type variables, say \( \sigma_1 \) and \( \sigma_2 \)
  - Equivalence the associate place variables
  - For each applicable field selector \( f \), recursively unify \( f(\sigma_1) \) and \( f(\sigma_2) \)
  - "folding over" to handle recursive fields