Locality Reasoning of Multithreaded Programs through Type Inference

Haitao Steve Zhu Yu David Liu
SUNY Binghamton
Binghamton NY 13902, USA
{hzhu1,davidL}@cs.binghamton.edu

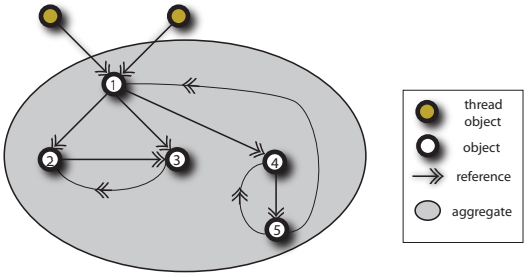
Abstract

In multithreaded object-oriented programs, locality refers to the dynamic scoping relation among threads and objects in a potentially shared-memory context. This paper calls for type-theoretic investigations into two forms of locality – *thread locality* and *aggregate locality* – and proposes a unified type inference algorithm to reason about them over unannotated real-world Java programs.

1. Introduction

The property of *locality* plays an essential role in understanding, designing, verifying, optimizing, and debugging multithreaded programs. The term is perhaps most commonly used in the context of *thread locality* – a memory location is thread-local iff it is accessed by one and only one thread throughout all possible executions of the program. The benefit of reasoning about thread locality is long recognized. For example, it simplifies the reasoning of race condition freedom and atomicity, because access to thread-local data is trivially thread-safe. Since runtime monitoring via locks or transactions is unnecessary over thread-local data, precisely identifying thread-local data further improves system performance, and reduces the occurrence of liveness bugs such as deadlocks and livelocks. Numerous program analyses and language designs exist for analyzing and enforcing thread locality.

When a memory location cannot be thread-local, does that mean we can only pessimistically label it "escaped," and no further insight can be gained? We believe at least one more important flavor of locality exists: *aggregate locality*. Here, an *aggregate* can be intuitively viewed as a "shared cluster of data": any object within the aggregate is accessed by more than one thread, but all accesses must go through some "root" of an aggregate to which the aforementioned object is local. We illustrate this notion as below:



Compared with the established property of thread locality, reasoning about aggregate locality over real-world unannotated Java-like programs is lesser explored. Reports on the number of thread-local objects in standard multithreaded benchmarks (*e.g.* `xalan` or `hsqldb` in Dacapo) have long appeared, but simple questions such as "how many non-thread-local objects in `xalan` are aggregate-local?" or "how many aggregates are there in `xalan`?" remain elusive. Make no mistake: the benefit of reasoning about aggregagte locality is long known to the language designers and software designers. For example, when a cluster of data is transferred from one thread to another, aggregate locality significantly simplifies the reasoning of thread safety by shifting the focus to the aggregate root: it is sufficient to only monitor the root at run time, or only reason about the uniqueness of its aliases [3]. For software developers, the hierarchical structure of aggregates – if recoverable through intuitive visualizations – offers vivid clues in selecting the appropriate granularity of locks [2].

2. Locality as Type Ineference

In this paper, we briefly report a type inference algorithm – named LG – that precisely reasons about thread locality and aggregate locality in one unified decision procedure. Given an unannotated Java program, the algorithm not only answers how many (statically differentiable) objects are thread-local, but also how non-thread-local objects are organized in aggregates. For example, applying the algorithm to the sharing-intensive benchmark `puzzle` [4] yields a *locality tree* as rendered in Fig. 1 – with the root on the very left – where each node represents an object (the colored ones are threads) and each edge from parent `x` to child `y` denotes object `y` "is local to" – *i.e.* never accessed outside the scope set by – object `x`. In this representation, the algorithm guarantees the correctness of the following statements:

- the `Vector#124` object and the `Vector#253` object are thread-local to the `SolverTask#var89` thread. and the `JGFPuzzleConcurrentBench#var12` object and its descendants are thread-local to the `Main` thread;
- `Puzzle#14` can be potentially accessed by more than one thread, and it serves as the root of an aggregate including 9 objects– itself and its decedents – and the aggregate is nested in the sense that `PuzzlePosition#23` is the root of an aggregate of size 6.

At its heart, LG is a form of polymorphic type inference of ownership types [1, 7]. Indeed, the representation in Fig. 1 is the inferred counterpart of a well-known relation enforced by ownership types – *ownership trees*. As the example above demonstrates, both thread locality and aggregate locality can be intuitively derived from the ownership relation. We now elaborate on the unique role LG plays, by relating LG in the contexts of ownership type inference and points-to analysis, respectively.

LG in the Context of Ownership Type Inference Locality precision and soundness are challenging requirements when existing ownership type inference solutions are applied to our problem domain.

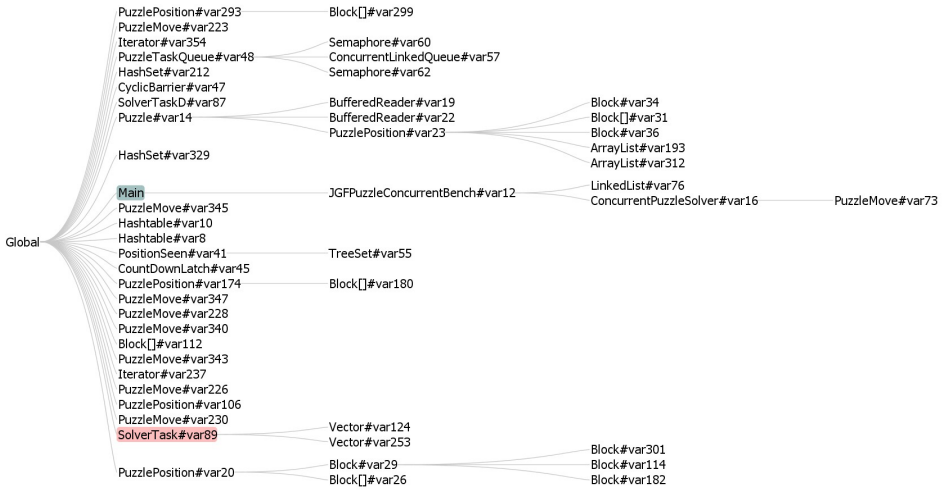


Figure 1. A Locality Tree Example

To elucidate the challenge of locality precision, let us revisit the aggregate rooted at `Puzzle#var14`. Observe that an alternative solution exists where the 3-level hierarchy of this aggregate may be collapsed to 2, *i.e.* making all 8 descendants of `Puzzle#var14` its direct children. This alternative solution is clearly less precise *w.r.t.* aggregate locality as it disregards that `PuzzlePosition#var23` may form a scope of its own. Worse, yet another solution could collapse all objects on the entire tree as the direct children of `Global`, *de facto* implying all objects are "shared." Existing approaches often solve this problem by requiring programmer annotations to express preference, or narrowing the scope *e.g.* by only inferring whether Java **private** field declarations correspond with ownership. These solutions do not naturally translate to the design goal of LG, *i.e.* inferring the all-to-all ownership relation over all objects of an unannotated Java program. LG tackles this challenging problem by reducing ownership type inference as a linear constraint solving problem – inspired by a prior work [5] – and expressing the preference for locality through novel uses of objective functions in linear programming. For example, the more desirable sub-tree for aggregate `PuzzlePosition#var23` would the one where the depth of each node is as high as possible.

Another promising direction for ownership inference is to use dynamic analyses. These approaches have the benefit of being precise, and their usefulness in understanding large real-world programs has been successfully validated. Their main shortcoming is unsoundness, which happens to be bad news for *multithreaded* programs since there are often a very large number of interleaving scenarios to consider, and hence numerous traces to analyze. We speculate advanced testing technologies can perhaps mitigate this problem, but unsoundness still would disqualify the analysis results from being applied in correctness-oriented settings, such as optimizing programs with guaranteed race condition freedom.

LG in the Context of Points-to Analysis As demonstrated by escape analyses and some non-type-based ownership/container program analyses, properties such as thread/aggregate locality may alternatively be reasoned about via points-to analysis. These approaches conceptually share the philosophy of first computing a points-to graph, and then finding subgraphs that satisfy a particular graph-theoretic invariant. Constructing scalable points-to analysis with high approximation precision is an active branch of research. In addition, when such an approach is used in reasoning about concurrent properties, as soundness is known to be dependent of a variety of design choices *e.g.* may-alias vs. must-alias analyses, flow-sensitive vs. flow-insensitive analyses, *etc*.

Along this line, LG can be viewed as a "custom-made" analysis where the points-to graph is only implicitly encoded in type constraints and never explicitly computed. In other words, the points-to graph contains *more* information that what locality reasoning *needs*, and LG computes "just enough" and avoids the rest of the unnecessary (points-to graph) computation. The resulting implementation is a highly scalable algorithm. For example, we are able to analyze `hsqldb` – a benchmark known to be resistant to context-sensitive analyses [8] – with a precision analogous to 5-object sensitivity [6]. In that (extremely) expensive setting, the inference algorithm can approximate around 4000 instances and terminate in around 16 minutes on an Intel Core Duo 2.53G Hz with 4GB RAM.

Reducing locality inference to an ownership type inference further simplifies the analysis of the algorithm correctness, since type soundness trivially entails the soundness of locality reasoning.

References

[1] CLARKE, D. *Object Ownership and Containment.* PhD thesis, University of New South Wales, July 2001.

[2] GRAY, J. N., LORIE, R. A., PUTZOLU, G. R., AND TRAIGER, I. L. Readings in database systems (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994, ch. Granularity of locks and degrees of consistency in a shared data base, pp. 181–208.

[3] HALLER, P., AND ODERSKY, M. Capabilities for uniqueness and borrowing. In *ECOOP'10* (2010), pp. 354–378.

[4] LIU, Y. D., LU, X., AND SMITH, S. F. Coqa: Concurrent objects with quantized atomicity. In *CC'08* (March 2008).

[5] LIU, Y. D., AND SMITH, S. Pedigree types. In *4th International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)* (July 2008), pp. 63–71.

[6] MILANOVA, A., ROUNTEV, A., AND RYDER, B. G. Parameterized object sensitivity for points-to analysis for java. *TOSEM 14* (January 2005), 1–41.

[7] NOBLE, J., POTTER, J., AND VITEK, J. Flexible alias protection. In *ECOOP'98* (Brussels, Belgium, July 1998).

[8] SMARAGDAKIS, Y., BRAVENBOER, M., AND LHOTÁK, O. Pick your contexts well: understanding object-sensitivity. In *POPL '11* (2011), pp. 17–30.