Position Paper: Static Debugging of Programs Using High-Level Concurrency Libraries

Philipp Haller
EPFL, Switzerland
firstname.lastname@epfl.ch

Abstract

High-level concurrency libraries, such as MapReduce and frameworks for fork/join parallelism, are promising tools to make parallelism available to the application developer. Optimizations and intrinsics provided by mainstream VM platforms, as well as programming languages supporting powerful means of abstraction greatly improve the performance and use of such frameworks.

However, debugging programs using high-level concurrency frameworks remains very difficult. In this position paper, we outline research on two approaches to improve static debugging of programs using high-level concurrency frameworks. The first approach provides library-specific annotation checkers that can be optionally plugged into the compiler. The second approach uses lightweight type annotations to adapt error messages involving implementation details of the (high-level) programming interface.

1. Introduction

In this paper we want to focus on two approaches to make parallelism available to the application developer. High-level concurrency libraries, such as MapReduce [9] and fork/join frameworks (*e.g.*, [19]), and embedded domain-specific languages (DSLs) [5, 6, 16]. Even though the two approaches seem to be quite different, some concurrency frameworks can be considered embedded domain-specific languages. For instance, Scala Actors [12] provide a syntax that appears to extend the host language, even though the primitives are implemented as a library in Scala.

For the rest of our discussion we are mostly concerned with the programming interface of embedded DSLs, including their syntax, operations, and types. When only considering their programming interfaces, there is virtually no difference between embedded DSLs and high-level libraries. Therefore, in the following we use the expression *high-level concurrency frameworks* to comprise the techniques for building programming interfaces of both approaches.

There has been substantial progress on support for high-level concurrency libraries, both in terms of optimizations and intrinsics provided by VM platforms, and in terms of high-level programming interfaces. Support for the latter is greatly facilitated by language support for closures, as demonstrated by languages such as X10 [22], Fortress [17], and Scala [21].

On the other hand, debugging support for programs written using high-level concurrency frameworks and embedded DSLs has been lacking. Existing approaches have two important problems.

1. When compiling programs using high-level libraries the error messages produced by the compiler can be cryptic. The main reason is that errors often refer to constructs used to *implement* the programming interface; however, normally the programmer should not be concerned with the implementation of the interface.
2. The compiler of the host language accepts many programs that contain severe errors in terms of how the constructs provided by the high-level library are used.

To illustrate the second problem, consider the following quote taken from the Java API documentation of the jsr166y `ForkJoinPool` (a lightweight execution environment supporting fork/join parallelism):[1]

> The efficiency of ForkJoinTasks stems from a set of restrictions (that are only partially statically enforceable) reflecting their intended use as computational tasks calculating pure functions or operating on purely isolated objects.

The documentation talks about restrictions and correctness criteria for using the `ForkJoinTask` class, which is the fundamental unit for computations that are run inside a `ForkJoinPool`. Two important properties that correct uses of that class must have are: (1) each task must compute a

SPLASH Workshop on Concurrency for the Application Programmer (CAP'10) October 18, 2010, Reno/Tahoe, Nevada, USA.
Copyright © 2010 ACM . . . $10.00

[1] See `http://gee.cs.oswego.edu/dl/concurrency-interest/`.

pure function, and (2) all tasks must operate on *isolated objects*. Programs that do not obey these usage restrictions are likely to have bugs that manifest at run time, such as data races. This is problematic, since debugging and testing concurrent programs is much harder compared to sequential programs, because program executions are typically not easily reproducible because of timing-dependent thread scheduling.

2. Our Approaches

We propose two approaches addressing those issues:

1. Pluggable, library-specific (or DSL-specific) checkers. These checkers can be shipped as part of the library, and optionally loaded when compiling (debug) builds.
2. Lightweight type annotations to customize and adapt (type) error messages emitted by the compiler.

The first approach is more heavyweight, but it has the potential to address both of the problems mentioned above: apart from statically checking library usage restrictions, a checker may post-process (low-level) error messages from the compiler and adapt them before presenting them to the user. In previous work [13] we have demonstrated a compiler plug-in to be used with Scala's actor library for checking actor isolation.

We intend to factor out pluggable type-and-effect systems that can be used to check the safety of several different concurrency libraries. This will reduce the effort required to build, test, and deploy pluggable checkers for specific concurrency libraries. For example, checking that two references point to disjoint object graphs is useful for isolating both actors and deterministic parallel computations (*e.g.*, [18]).

The second approach is not sufficient to check for properties like purity and isolation. On the upside, the approach can be very lightweight: simple annotations placed on types and constructs of the library that should not be visible to the user can be used for adapting error messages of the compiler. Such annotations can already improve the user experience substantially; at the same time the approach places only a small burden on the library author.

In fact, the latter approach is already finding its way into Scala's standard collections library. Scala's collections [20] use *implicit parameters* [8] to support operations on collections that are polymorphic in the type of the resulting collection. These implicit parameters should not be visible to the application developer. However, in Scala 2.8.0, error messages when using collections incorrectly could refer to these implicits. In Scala 2.8.1, a lightweight mechanism has been added to adapt error messages involving implicits: by adding an annotation to the type of the implicit parameter, a custom error message is emitted when no implicit value of that type can be found.

For instance, immutable maps define a `transform` method that applies a function to the key/value pairs stored in the map resulting in a collection containing the transformed values:

```
def transform[C, That]
    (f: (A, B) => C)
    (implicit bf
     : CanBuildFrom[This, (A, C), That])
     : That
```

This function transforms all the values of mappings contained in the current map with function `f`. Here, `This` is the type of the actual map implementation. `That` is the type of the updated map. The implicit parameter ensures that there is a builder factory that can be used to construct a collection of type `That` given a collection of type `This` and elements of type `(A, C)`.

Actual implicit arguments passed to transform should not be visible to the application developer. However, wrong uses of maps may result in the compiler not finding concrete implicit arguments; this would result in confusing error messages. In Scala 2.8.1 error messages involving type `CanBuildFrom` are improved using a type annotation:

```
@implicitNotFound(msg = "Cannot construct a
  collection of type ${To} with elements of
  type ${Elem} based on a collection of
  type ${To}.")
trait CanBuildFrom[-From, -Elem, +To] {
  // ...
}
```

The `implicitNotFound` annotation is understood by the implicit search mechanism in Scala's type checker. Whenever the type checker is unable to determine an implicit argument of type `CanBuildFrom`, the compiler emits the (interpolated) error message specified as the argument of the `implicitNotFound` annotation. Thereby, a low-level implicit-not-found error message is transformed to only mention the types `From`, `Elem`, and `To`, which correspond to types occurring in user programs.

3. Related Work

There is a substantial body of literature investigating compile-time safety checking for parallel and concurrent programs using actors/active objects [7, 25], shared-memory concurrency [1, 3, 4, 18], stream-based programming [2, 10, 24], and other concurrency models [11, 22, 23]. In our approach, we intend to explore static checking for programs that use multiple concurrency libraries.

There has been work on type error debugging in functional languages (see, *e.g.*, [14]), motivated by their sophisticated type systems and type inference. In these approaches, type errors are explained by collecting constraints for correct type assignment globally, and solving these constraints subsequently. While applicable in educational programming environments like Helium [15], global constraint solving is not practical for large-scale software systems. Furthermore, previous approaches are not applicable to object-oriented languages with local type inference like Scala.

Moreover, we intend to integrate type error debugging with library-specific annotation checkers. In addition to annotations on the types and operations of the high-level concurrency framework (as shown in the example in Section 2), this will require interaction between type inference and (pluggable) annotation checking.

4. Conclusion

We believe when using high-level concurrency libraries, new mechanisms are needed to (a) enforce more usage restrictions statically (preventing hard-to-debug concurrency hazards), and (b) make error messages when compiling user programs more useful. Otherwise, powerful frameworks, although providing high performance, will remain tools only for expert programmers. In this paper we have outlined two approaches to improve static debugging of concurrent programs based on high-level libraries and embedded DSLs.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst*, 28(2):207–255, 2006.

[2] J. S. Auerbach, D. F. Bacon, R. Guerraoui, J. H. Spring, and J. Vitek. Flexible task graphs: a unified restricted thread programming model for Java. In *Proceedings of the 2008 ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08)*, pages 1–11, 2008.

[3] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of Java without data races. In *Proceedings of the 2000 ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'00)*, pages 382–400, Oct. 2000.

[4] C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 211–230, 2002.

[5] J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program*, 19(5):509–543, 2009.

[6] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing, 2010.

[7] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal ownership for active objects. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems (APLAS'08)*, pages 139–154. Springer, Dec. 2008.

[8] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA/SPLASH'10*, 2010.

[9] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

[10] R. Ennals, R. Sharp, and A. Mycroft. Linear types for packet processing. In *Proceedings of the 13th European Symposium on Programming (ESOP'04)*, pages 204–218. Springer, 2004.

[11] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proc. EuroSys*, 2006.

[12] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 410(2-3): 202–220, 2009.

[13] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*, pages 354–378. Springer, June 2010.

[14] B. Heeren and J. Hage. Type class directives. In *PADL*, pages 253–267. Springer, 2005.

[15] B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning Haskell. In *Proc. of the ACM Workshop on Haskell*, pages 62–71, 2003.

[16] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In *GPCE*, pages 137–148. ACM, 2008.

[17] G. L. S. Jr. Parallel programming and parallel abstractions in Fortress. In *IEEE PACT*, page 157. IEEE Computer Society, 2005.

[18] R. L. B. Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *OOPSLA*, pages 97–116. ACM, 2009.

[19] D. Lea. A Java fork/join framework. In *Proceedings of the ACM Java Grande Conference*, pages 36–43. ACM, June 2000.

[20] M. Odersky and A. Moors. Fighting bit rot with types (experience report: Scala collections). In R. Kannan and K. N. Kumar, editors, *FSTTCS*, volume 4 of *LIPIcs*, pages 427–451. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2009.

[21] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Press, Mountain View, CA, 2008.

[22] V. A. Saraswat, V. Sarkar, and C. von Praun. X10: concurrent programming for modern architectures. In *Proceedings of the 12th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'07)*, page 271, Mar. 2007.

[23] J. Schäfer and A. Poetzsch-Heffter. JCobox: Generalizing active objects to concurrent components. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*, pages 275–299. Springer, June 2010.

[24] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: High-throughput stream programming in Java. In *Proceedings of the 22nd ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 211–228, Oct. 2007.

[25] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, pages 104–128. Springer, July 2008.