Language-Agnostic Static Deadlock Detection for Futures

Anonymous Author(s)

ABSTRACT

Deadlocks, in which threads wait on each other in a cyclic fashion and can't make progress, have plagued parallel programs for decades. In recent years, as the parallel programming mechanism known as *futures* has gained popularity, interest in preventing deadlocks in programs with futures has increased as well. Various static and dynamic algorithms exist to detect and prevent deadlock in programs with futures, generally by constructing some approximation of the dependency graph of the program but, as far as we are aware, all are specialized to a particular programming language.

A recent paper introduced *graph types*, by which one can statically approximate the dependency graphs of a program in a language-independent fashion. By analyzing the graph type directly instead of the source code, a graph-based program analysis, such as one to detect deadlock, can be made language-independent. Indeed, the paper that proposed graph types also proposed a deadlock detection algorithm. Unfortunately, the algorithm was based on an unproven conjecture which we show to be false. In this paper, we present, and prove sound, a type system for finding possible deadlocks in programs that operates over graph types and can therefore be applied to many different languages. As a proof of concept, we have implemented the algorithm over a subset of the OCaml language extended with built-in futures.

1 INTRODUCTION

The problem of *deadlocks*, in which two or more threads are waiting on each other in a cyclic fashion so none can make progress, has been observed since the early days of parallel and concurrent programming [7]. Many solutions to the problem have been proposed over the years. We can broadly group these into static approaches (e.g. [5, 9, 13, 16, 21]), which detect using either a type system or static analysis on the source code of a program whether the conditions necessary for a deadlock may exist in the program, and dynamic approaches (e.g., [8, 19, 20]) which run along side the program and detect either that the conditions necessary for a deadlock exist at runtime, or that a deadlock has occurred.

Much prior work on deadlock has been focused on cyclic requests for resources (often locks) by coarse-grained system threads, such as pthreads. In more recent years, there has been intense interest in fine-grained parallelism, where large numbers of lightweight threads are scheduled automatically by the runtime system onto system-level threads. A mechanism for fine-grained parallelism that has attracted particular interest recently is the *future* and its closely related cousin the *promise*. A future is spawned to compute a designated piece of work asynchronously with the rest of the program. The handle to the future is then a first-class object that can be stored, passed as an argument to functions, etc. When the result of the asynchronous computation is needed (even in a far-away part of the program), its handle can be "touched" (or "forced"). This operation blocks until the future's computation completes and then returns the result. Since being introduced in Multilisp [11], variants of these mechanisms have made their way into numerous languages, including Cilk [10], Habanero-Java [6], JavaScript, Python, Rust [1], and the latest version of OCaml [17]. Futures can be used for everything from reducing latency in concurrent interactions to implementing asymptotically efficient pipelined data structures [3]. Because of their generality, however, futures can also be used in ways that cause a deadlock.

Even when considering one threading paradigm such as futures, tools for solving the deadlock problem have been proposed for numerous languages and libraries. However, as far as we are aware, virtually all solutions proposed thus far are specific to at least a particular language, if not a particular runtime and/or threading library. This specificity of deadlock analyses to a particular language is odd when one considers that the essence of the deadlock problem for futures, regardless of language, can be boiled down to a graph problem. If we think of the program as a directed graph of dependences between threads, a deadlock in which two futures wait on each other will show up as a cycle in the graph. Indeed, many existing static and dynamic analyses for deadlock work by (implicitly or explicitly) constructing some approximation of the dependency graph. This observation leads to the central question of this paper: is it possible to statically predict deadlocks in programs with futures in a language-agnostic way by analyzing not the program source code but a representation of dependency graphs?

Recent work [14] proposed *graph types* as a way of representing the set of dependency graphs that might result from executing a program. Such a representation is necessary because, especially in fine-grained parallel programs such as those with futures, runtime decisions based on either

Conference'17, July 2017, Washington, DC, USA
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

input values or nondeterminism can affect the structure of the dependency graph. As a result, a dependency graph as described above represents not the program itself but rather a particular execution of the program. The program then corresponds to a (possibly infinite) set of graphs describing the structure of every possible execution. Graph types represent these sets in a finite, compact way, and can be statically assigned to a program by a *graph type system*. Moreover, the graph type representation is not tied to a particular language or parallelism model (although the graph type system, which produces a graph type from source code, is specific to the language). The problem of determining whether a deadlock is possible in a parallel program then reduces to determining whether any graph represented by the program's graph type can contain a cycle. Because graph types can, in principle, represent programs in many different languages, such an analysis over the graph type would lead to a language-agnostic static deadlock detection tool.

Indeed, the initial work on graph types presents a proof-of-concept static deadlock algorithm based on the above idea—after inferring graph types for a program, their tool, called GML for Graph ML (the tool accepts source code in a dialect of the ML language), can optionally run deadlock detection on the resulting graph type. The algorithm in this prior work is not proven sound and relies on a conjecture (admitted as such in the paper) that any cycles that might arise in graphs represented by a graph type can be found by "unrolling" the graph type to a fixed depth and testing a small number of representative graphs for cycles. Unfortunately, as we show in this paper with a general family of counterexamples, that conjecture is false and the deadlock detection algorithm unsound. Moreover, any fixes to the algorithm that might resolve these issues would result in an exponential blowup in the number of graphs that must be checked for cycles.

In this paper, we propose a different static deadlock detection algorithm on graph types, which takes the form of a type system over graph types and does not rely on unrolling the graph type to extract representative graphs. We prove the algorithm sound by showing that any program it determines to be deadlock-free will at runtime obey the *transitive joins* property [19], a condition used in prior work on dynamic deadlock avoidance for futures which has been shown to imply deadlock-freedom. At a high level, the algorithm works by controlling the ownership and use of futures in a graph type, ensuring two properties. First, while the original graph type system has a robust mechanism for determining where futures *may* be spawned, we extend this to determine where futures *must* be spawned, in order to detect situations in which a future handle could be touched without a spawn of the corresponding future. Next, we reject graph types in which it cannot be determined statically that the touch of a future comes "after" (in a well-defined partial order on the program) the spawn, which prevents cycles of futures blocking on each other. We have implemented the algorithm in an extension of GML and show using a number of qualitative examples that it is not overly restrictive.

The rest of the paper proceeds as follows. In Section 2, we introduce the thread model we consider—the language we use for examples is intentionally simple so that it can represent the spectrum of languages for which our techniques can be applied—and the basics of graph types. Next (Section 3), we outline the counterexample to the prior deadlock detection algorithm. In Section 4, we present our algorithm as a type system and prove it sound. In Section 5, we describe our implementation of the algorithm as well as a qualitative evaluation that shows the scope of programs it can prove deadlock-free. Finally, we discuss related work and conclude. For space reasons, details of some of the proofs are omitted from the body of the paper but included in an appendix submitted as supplementary material.

2 PRELIMINARIES

2.1 Language Model

Graph types abstract away details of the programming language and even the exact parallelism constructs, so the algorithm we describe in this paper is applicable to a wide variety of languages with futures. For the purposes of presenting examples, we adopt a simple, imperative language with a built-in type `future[A]` representing a future asynchronously computing a value of type `A`. We distinguish between a *future thread*, or simply *thread*, which is an asynchronous thread performing some computation, and a *future handle*, which is a value of type `future[A]` providing the programmer a means of accessing the result of an associated future thread. When it is clear from context, we will simply use the term *future*. We consider three operations on futures. The constructor `new future[A]()` creates a new future handle which is currently not initialized with a running future thread. This handle can then be used to perform two operations: if `h` is a future handle, then `h.spawn(f)` spawns a new asynchronous future thread to compute the function `f`, and installs the handle to this future into `h`. Calling `h.touch()` waits for the future thread associated with `h` to complete and returns the thread's return value (if no thread is associated with `h` because `spawn` has not yet been called, then `touch()` waits for a thread to be installed, and then waits for it to complete).

As an example, the program in Figure 1 implements a generic parallel recursive divide-and-conquer algorithm (this could be instantiated with Mergesort, Quicksort, Fibonacci, or many other standard algorithms). If the length of an input is greater than some threshold, the input is divided into two halves. A new future is spawned to run the program recursively on the first half, while the second half is computed in

```
function divide_and_conquer (list[A] l):
  if l.length < threshold:
    return base_case(l)
  else:
    (l1, l2) = divide l
    h = new future[B]()
    h.spawn({ divide_and_conquer l1 })
    l2_result = divide_and_conquer l2
    l1_result = h.touch()
    return combine(l1_result, l2_result)
```

Figure 1: Example code for a divide-and-conquer program implemented with futures.

the current thread. The future handle is then touched to get the result of the first half, and the two results are combined.

The combination of futures and an imperative language with mutable state allows for programs with deadlocks. Consider the following program:

```
a = new future[int]();
b = new future[int]();
a.spawn({ return b.touch() })
b.spawn({ return a.touch() })
```

The program declares two futures handles, and then initializes each with a computation that touches the other. Neither future thread can make progress until the other completes, and so this is a classic deadlock. We note that the imperative nature of spawn is crucial for this example. In purely functional programs with futures, the use of futures is constrained to be *structured* [12], which precludes deadlocks; however, many real-world uses of futures are not structured.

2.2 Graphs

We abstractly represent the parallel structure of a program using a directed graph expressing the dependences between threads. We will use metavariables u and variants to refer to vertices of the graph, which represent individual, sequential computations. If u is an ancestor of u', then u must happen before u'. The lack of a path between two computations indicates that they may occur in parallel.

Formally, we represent a graph g as a quadruple (V, E, s, t) of a set V of vertices, a set E of directed edges, a designated "start" vertex s and a designated "end" vertex t. We consider each graph to have a "main" thread that starts at s and ends at t. We use a number of shorthands to build and compose graphs. The notation $\bullet$ represents a graph containing a single vertex. The graph $g_1 \oplus g_2$ represents sequential composition of the two graphs, composing the two main threads together in sequence. The graph $g \swarrow_u$ describes a main thread consisting of one vertex that spawns another thread (e.g., a future thread). The new thread consists of the graph g, post-composed with a new designated "end" vertex u. We add this vertex to give the future a unique name that can be referred to later, such as when another thread wants to touch the future. This touch corresponds to adding an edge from the last vertex of the future thread, which is u, and we write this as ${}^{u}\searrow$. The notations are defined formally in Figure 2, and additionally require that all vertices in the graph are unique.

2.3 Graph Types

The graphs of the previous section represent a record of one execution of a program: while the graph abstracts away from details of how parallel threads are scheduled, if a program makes choices based on unknown input or involves any nondeterminism, the graph still reflects only one possible resolution of these choices. As an example, the graph that results from performing a parallel Quicksort on a sorted list will be quite different from the graph that results from a randomly-ordered list. There is no way to know without running the program exactly how the graph will look.

Graph types [14] compactly represent the set of all possible graphs that might result from running a particular program, and are assigned statically to programs, allowing us to make statements about a program's graph without running it. Like the abstract graphs described above, graph types abstract away details of the language model, and so are an ideal intermediate representation for performing analyses on the structure of a program in a language-agnostic way. In this subsection, we give a brief overview of the graph type notation we need for the rest of the paper, and direct readers to the prior work for a more complete presentation.

The syntax for graph types G is given below:

$$
\begin{array}{lcl}
G & ::= & \bullet \mid G_1 \oplus G_2 \mid G \swarrow_u \mid {}^{u}\searrow \\
& \mid & G_1 \vee G_2 \mid \mu\gamma.G \mid \gamma \mid \nu u.G \mid \Pi\vec{u}_f;\vec{u}_t.G \mid G[\vec{u}_f;\vec{u}_t]
\end{array}
$$

The first row of constructs looks similar to the notation used for building graphs in the previous subsection. Indeed, any graph constructed using the constructs of Figure 2 is also a valid graph type inhabited by only that one graph.

The constructs in the second row allow graph types to reflect a set containing multiple graphs. The graph type $G_1 \vee G_2$ represents the disjunction of two alternatives; for example, if a program might take either branch of a conditional at runtime, its graph might correspond to the if branch or the else branch. The set of graphs represented by this graph type is the union of the graphs represented by G_1 and G_2.

Graph types must also be able to represent unbounded sets of graphs, which generally result from either recursion or iteration in the parallel program. As an example, there is no way to tell statically how many times the `divide_and_conquer` function of Figure 1 will call itself. The graph type for this

$$
\begin{array}{llll}
\bullet & \triangleq & (\{u\}, \emptyset, u, u) & u \text{ fresh} \\
(V_1, E_1, s_1, t_1) \oplus (V_2, E_2, s_2, t_2) & \triangleq & (V_1 \cup V_2, E_1 \cup E_2 \cup \{(t_1, s_2)\}, s_1, t_2) & V_1 \cap V_2 = \emptyset \\
(V, E, s, t) \swarrow_u & \triangleq & (V \cup \{u, u'\}, E \cup \{(u', s), (t, u)\}, u', u') & u' \text{ fresh}, u \notin V \\
{}^{u}\searrow & \triangleq & (\{u'\}, \{(u, u')\}, u', u') & u' \text{ fresh}
\end{array}
$$

Figure 2: Shorthands for combining graphs.

function needs to contain graphs corresponding to any number of recursive calls. This is represented with the recursive graph type $\mu\gamma.G$, which binds a *graph variable* γ inside G. The inner graph type, G, can "call" the entire recursive graph type recursively using γ.

Here, we take a slight diversion to introduce an important point about graph types. Recall from the previous subsection that vertices in a graph must be unique—if there are two vertices u in a graph, then there is no way to know which one is the source of an edge (u, u'). The graph-composition constructs in Figure 2 simply enforce, as a condition of their use, that composing graphs would not duplicate vertex names. In graph types, it is not always clear when a graph type would yield a graph with duplicate vertex names. Consider the following invalid graph type, which we might naively use to represent the parallel divide-and-conquer example:

$$G \triangleq \mu\gamma. \bullet \vee (\gamma \swarrow_u \oplus \gamma \oplus {}^{u}\searrow)$$

The graph type indicates that the program either 1) "bottoms out" to a sequential base case, or 2) spawns a future whose graph is also represented by G using a designated vertex name u, then does another computation represented by G, then touches the future. The problem with this graph type is that finding the set of graphs to which it corresponds requires "unrolling" the recursion, e.g., one such graph is

$$(\bullet \swarrow_u \oplus \bullet \oplus {}^{u}\searrow) \swarrow_u \oplus (\bullet \swarrow_u \oplus \bullet \oplus {}^{u}\searrow) \oplus {}^{u}\searrow$$

which has 3 vertices "named" u.

To avoid duplicating vertex names when unrolling recursion, we need a way to generate fresh vertex names. This is accomplished with the $\nu u.G$ construct, which introduces a vertex variable u within the scope of G. This variable will be instantiated with a unique vertex each time the binding is encountered. The divide-and-conquer example graph could then be expressed correctly as:

$$G \triangleq \mu\gamma.\nu u. \bullet \vee (\gamma \swarrow_u \oplus \gamma \oplus {}^{u}\searrow)$$

To enforce that graph types are used in a way that will not result in graphs with duplicate vertices, prior work equips graph types with a "well-formedness" judgment that takes the form of a type system over graph types (or rather, a "kind" system because graph types are already type-level constructs). In this judgment, vertices that are used to spawn futures are subject to an *affine* restriction, which prevents them from being used more than once. In the next section, we describe how this is accomplished in more detail.

The final two graph type constructs allow graph types to be parameterized by sets of vertices. The graph type $\Pi\vec{u}_f; \vec{u}_t.G$ introduces the variables $\vec{u}_f$ and $\vec{u}_t$ which may be used in G. Both notations represent a comma-separated vector of zero or more vertices; we will use $\emptyset$ if there are no vertices in one vector. The vertices in $\vec{u}_f$ may be used to spawn futures, while the vertices in $\vec{u}_t$ may be used to touch futures. It will become clear when we discuss well-formedness of graph types in the next section why these two sets are separated. The parameters of such a graph type can be instantiated with the application $G[\vec{u}_f; \vec{u}_t]$.

Finally, we discuss formally how to construct a set of graphs from a graph type, a process we have motivated informally above. We refer to this process as *normalization*. Generally, one should not have to normalize graph types in order to use them, but normalization is useful for defining the semantics and soundness of graph types. Specifically, the soundness theorem of the graph type system [14] ensures that any graph that results from executing a program is contained in the normalization of the program's graph type. (We also use normalization in the proof of soundness for the analysis we present in this paper, but normalization is not necessary for actually performing the analysis.) Because graph types (such as the divide-and-conquer example above) can correspond to infinite sets of graphs, we parameterize the normalization function by a natural number n roughly corresponding to how many times. recursive graph types should be unrolled. Figure 3 defines the normalization operation as a function $Norm_G(n)$[1]. Once n reaches zero, normalization returns the empty set. Otherwise, normalization proceeds largely as we have motivated above. A sequential composition $G_1 \oplus G_2$ is normalized by pairwise composing the normalizations of the two subgraphs, disjunctions union their normalizations, and a future $G \swarrow_u$ introduces a spawn of g using vertex u for all g in the normalization of G. The normalization of recursive bindings allows the binding to be unrolled or not; in either case, n is decremented. A "new"

[1]The definition here is slightly different from the presentation in prior work [14]; specifically, the prior presentation returned the singleton graph type $\bullet$ rather than the empty set of graphs as the base case. The definition here is more convenient for our proofs; we have confirmed that the soundness proof of the graph type system is unaffected by this change.

$$
\begin{array}{lcll}
Norm_0(G) & \triangleq & \emptyset & \\
Norm_n(\bullet) & \triangleq & \{\bullet\} & \\
Norm_n(G_1 \otimes G_2) & \triangleq & \{G_1' \otimes G_2' \mid G_1' \in Norm_n(G_1), G_2' \in Norm_n(G_2)) & \\
Norm_n(G_1 \oplus G_2) & \triangleq & \{G_1' \oplus G_2' \mid G_1' \in Norm_n(G_1), G_2' \in Norm_n(G_2)) & \\
Norm_n(G_1 \vee G_2) & \triangleq & Norm_n(G_1) \cup Norm_n(G_2) & \\
Norm_n(G \swarrow_u) & \triangleq & \{G' \swarrow_u \mid G' \in Norm_n(G)\} & \\
Norm_n({}^{u}\searrow) & \triangleq & \{{}^{u}\searrow\} & \\
Norm_n(\mu\gamma.G) & \triangleq & Norm_{n-1}(G[\mu\gamma.G/\gamma]) \cup Norm_{n-1}(\mu\gamma.G) & \\
Norm_n(\nu u.G) & \triangleq & Norm_n(G[u'/u]) & u' \text{ fresh} \\
Norm_n(G[\vec{u}_f;\vec{u}_t]) & \triangleq & Norm_{n-k}(G'[\vec{u}_f/\vec{u}_f'][\vec{u}_t/\vec{u}_t']) & \mathsf{unroll}_k(G) = \Pi\vec{u}_f';\vec{u}_t'.G' \\
Norm_n(G[\vec{u}_f;\vec{u}_t]) & \triangleq & \emptyset & \mathsf{unroll}_n(G) \neq \Pi\vec{u}_f';\vec{u}_t'.G'
\end{array}
$$

Figure 3: Normalization.

binding $\nu u.G$ is normalized by substituting a fresh vertex for u. The normalization of an application unrolls the applied graph type until it is a Π binding (decrementing n by the number of times it needs to be unrolled) and then substitutes the arguments for the parameters.

3 COUNTEREXAMPLE TO CONJECTURE

The original work on graph types [14] proposed and implemented a proof-of-concept deadlock detection algorithm for graph types. The algorithm worked by normalizing the graph type to the minimum level n (that is, computing $Norm_n(G)$) such that every recursive binding in the graph type is unrolled twice. It would then check each of the resulting graphs for cycles[2]. The (purported) soundness of this algorithm depends on a conjecture that if $g \in Norm_m(G)$ for any m and g has a cycle, then there is a graph with a cycle in $Norm_n(G)$, where n is as described above. In this section, we present a counterexample to this conjecture. Consider the graph type

$$\nu u_1, u_2. \bullet \swarrow_{u_2} \oplus G[u_1; u_2]$$

where

$$G \triangleq \mu\gamma.\Pi u_a; u_x.\nu u.\ \bullet \vee ({}^{u_x}\searrow \oplus \bullet \swarrow_{u_a} \oplus \gamma[u; u])$$

This graph type could arise from the following program.

```
function g(future[int] a, x):
  u = new future()
  if (rand () == 0):
    return
  else:
    x.touch()
    a.spawn({ return 42 })
    g (u, u)
    return

function main():
  u1, u2 = new future[int]()
  u2.spawn({ return 42 })
  g(u1, u2)
  return
```

The function g takes two futures, a and x, which it spawns and touches, respectively. At the first call to g, these are instantiated with different futures, but when it is called recursively, both are instantiated with the same future.

If we unroll the recursive binding of G once, we get:

$$\bullet \swarrow_{u_2} \oplus {}^{u_2}\searrow \oplus \bullet \swarrow_{u_1} \oplus \bullet$$

where we take the "else" branch in the first unrolling of G and the "then" branch in the second (this is the only option available that would produce a graph, because taking the "else" branch again would require unrolling the recursion again). Unrolling the recursion a second time gives rise to a graph where we call g recursively with u as both arguments and get the following graph:

$$\bullet \swarrow_{u_2} \oplus {}^{u_2}\searrow \oplus \bullet \swarrow_{u_1} \oplus {}^{u}\searrow \oplus \bullet \swarrow_{u} \oplus$$

This graph has a cycle because u is touched before it is spawned, but this cycle was only detected by unrolling the graph type an extra time.

Furthermore, the problem cannot be fixed by simply unrolling more times (increasing the n value above) and checking more graphs. If we unroll every recursion three times, the following program serves as a counterexample (we have omitted the main function here, which just initializes g)[3]:

[2]Separately, the algorithm checks that the graph type does not allow a vertex to be touched without being spawned, but we focus here on the cycle detection part of the algorithm.

[3]While this example is syntactically valid, we note that if the code is converted to GML's OCaml-like syntax, GML is not able to infer a graph type for the program. This is due to a design decision in GML's handling of polymorphic recursion; the details are beyond the scope of this paper, but the high-level issue is that it may take several iterations of graph inference over a recursive function to arrive at the proper type. In the type inference literature, this is referred to as Mycroft iteration [15]. GML short-cuts this

```
function g(future[int] a, b, x, y):
  u = new future[int]()
  if (rand () == 0):
    return
  else:
    x.touch()
    a.spawn({ return 42 })
    g (b, u, y, u)
    return
```

This version of the program takes two futures to spawn and two to touch. On the recursive call, the second "spawn" future, b, is moved into the first position so it will be spawned on the next iteration, and the second "touch" future, y, is moved into the first "touch" position so it will be touched on the next iteration. The new future u is passed as both the second "spawn" and second "touch" future so it will be both touched and spawned (creating a cycle) on the *following* iteration. For any number n of unrollings, this example can be extended so that the deadlock will not manifest until the $n + 1^{st}$ call to g, and therefore the $n + 1^{st}$ unrolling.

The above counterexample shows that there is no global number n of unrollings such that a deadlock will manifest in the first n unrollings (which would make it possible to soundly detect deadlocks by checking all of the graphs in $Norm_n(G)$ for cycles). It is possible that there exists such an *n for each program.* For the family of counterexamples above, if m is the number of "spawn" and "touch" arguments, n could be set to $m + 1$, as the examples were constructed precisely to manifest a deadlock on the $m + 1^{st}$ unrolling. However, this solution, even if sound, leaves much to be desired in both elegance and efficiency. The latter is easily seen, as the number of graphs in $Norm_n(G)$ is, for most graph types, exponential in n. We therefore take a different approach in designing the algorithm in the next section.

4 A GRAPH TYPE ANALYSIS FOR DEADLOCK DETECTION

In Section 4.1, we present our main result, a type system for detecting whether deadlock is possible in a given program using its graph type. We then prove it correct in Section 4.2.

4.1 Graph Kind System

Our deadlock detection algorithm is a static analysis pass over graph types [14]. That is, we do not depend on source code and do not perform any evaluation (although our soundness proof will involve normalizing graph types, a form of evaluation on graph types). We present the analysis as a type

process by performing graph inference on each recursive function twice. If the type has not reached a fixed point after the second iteration, an error is raised. For reasons that are similar to why this works as a counterexample, the type of this example will not reach a fixed point after two iterations.

(DF:Empty)
$$\frac{}{\Delta; \cdot; \Psi \vdash_{DF} \bullet : *}$$

(DF:Var)
$$\frac{}{\Delta, \gamma : \kappa; \cdot; \Psi \vdash_{DF} \gamma : \kappa}$$

(DF:Seq)
$$\frac{\Delta; \Omega_1; \Psi \vdash_{DF} G_1 : * \qquad \Delta; \Omega_2; \Psi, \Omega_1 \vdash_{DF} G_2 : *}{\Delta; \Omega_1, \Omega_2; \Psi \vdash_{DF} G_1 \oplus G_2 : *}$$

(DF:Or)
$$\frac{\Delta; \Omega; \Psi \vdash_{DF} G_1 : * \qquad \Delta; \Omega; \Psi \vdash_{DF} G_2 : *}{\Delta; \Omega; \Psi \vdash_{DF} G_1 \vee G_2 : *}$$

(DF:RecPi)
$$\frac{\Delta, \gamma : \Pi\vec{u}_f; \vec{u}_t.*; \vec{u}_f; \Psi, \vec{u}_t \vdash_{DF} G : *}{\Delta; \cdot; \Psi \vdash_{DF} \mu\gamma.\Pi\vec{u}_f; \vec{u}_t.G : \Pi\vec{u}_f; \vec{u}_t.*}$$

(DF:Spawn)
$$\frac{\Delta; \Omega; \Psi \vdash_{DF} G : *}{\Delta; \Omega, u; \Psi \vdash_{DF} G \swarrow_u : *}$$

(DF:Touch)
$$\frac{}{\Delta; \cdot; \Psi, u \vdash_{DF} {}^{u}\searrow : *}$$

(DF:New)
$$\frac{\Delta; \Omega, u; \Psi \vdash_{DF} G : * \qquad u \notin \Omega, \Psi}{\Delta; \Omega; \Psi \vdash_{DF} \nu u.G : *}$$

(DF:Pi)
$$\frac{\Delta; \Omega, \vec{u}_f; \Psi, \vec{u}_t \vdash_{DF} G : *}{\Delta; \Omega; \Psi \vdash_{DF} \Pi\vec{u}_f; \vec{u}_t.G : \Pi\vec{u}_f; \vec{u}_t.*}$$

(DF:App)
$$\frac{\Delta; \Omega; \Psi, \vec{u}'_t \vdash_{DF} G : \Pi\vec{u}_f; \vec{u}_t.\kappa}{\Delta; \Omega, \vec{u}'_f; \Psi, \vec{u}'_t \vdash_{DF} G[\vec{u}'_f; \vec{u}'_t] : \kappa}$$

Figure 4: Rules for deadlock avoidance.

system over graph types (since graph types are analogous to types, a better term would be a *kind system*, where kinds are "types of types"). There are two *graph kinds* κ, which may be thought of as the "types of graph types":

$$\kappa ::= * \mid \Pi\vec{u}_f; \vec{u}_t.*$$

The graph kind $*$ represents ordinary graph types; these are graph types that can be directly normalized. The graph kind $\Pi\vec{u}_f; \vec{u}_t.*$ is a graph type with two sets of parameters $\vec{u}_f$ and $\vec{u}_t$; these parameters must be instantiated to produce an ordinary graph type. The deadlock freedom judgment is $\Delta; \Omega; \Psi \vdash_{DF} G : \kappa$, which assigns a graph kind κ to the graph type G. The judgment uses three contexts: Δ contains graph variables γ together with their graph kinds, Ω contains vertex names that may be used for spawning futures, and Ψ contains vertex names that may be touched. Other than the subscript on the turnstile, the deadlock freedom judgment looks quite similar to the well-formedness judgment of Muller [14], which also assigns graph kinds to graph types. That judgment, however, aims to assign a graph kind to all properly formed graph types. It serves mainly to reject graph types that would spawn multiple futures using the same vertex, which would result in meaningless graphs. As such, the spawn context Ω is treated as *affine*, meaning that vertices in this context may be used at most once in the type. The touch context Ψ has no such restriction, as vertices may be touched any number of times.

Our judgment serves a different purpose, in that it seeks to assign a graph kind *only* to graph types that are guaranteed to be deadlock-free. This type system is designed to be conservative, and (as with all static analysis) will reject some safe programs. We seek to prevent two types of deadlocks:

(1) A touch targets a vertex that is never spawned, so the touch will block indefinitely.
(2) Touches and spawns create a cycle in the graph.

Item (1) requires ensuring that vertices that *may* be spawned indeed *are* spawned. It is therefore not enough, as in prior work, to treat the spawn context as affine. Instead, we treat it as *linear*, meaning that vertices in the spawn context must be used *exactly* once. This guarantees that any vertex that may be spawned by a graph type *will* be spawned. As before, there are no affine or linear restrictions on the touch context. However, we take more care in when we add vertices to the touch context: we will add vertices to the touch context only after they are known to have been spawned.

The rules for the deadlock freedom judgment are in Figure 4, and we describe a few of the key points here. Rule DF:Empty indicates that the single-node graph is well-kinded, but only under an empty spawn context; if there are any vertices in the spawn context, this would violate linearity as they are not spawned by the graph type. Rule DF:Var handles graph variables which are found in the context Δ. Again, the spawn context must be empty. Rule DF:Seq handles sequential composition of two graph types. The spawn context is split (nondeterministically) into two pieces Ω_1 and Ω_2. As is typical in linear and affine type systems, this must constitute a disjoint splitting of the spawn context. We type G_1 with the spawn context Ω_1. Recall that this means that G_1 *must* spawn *all* vertices in Ω_1. It is therefore safe to add the vertices from Ω_1 to the touch context when analyzing G_2—we know that all of these vertices will have already been spawned. It is worth noting that rule DF:Or does not split the spawn context—only one of G_1 and G_2 will actually be executed, and so both may spawn the same set of vertices (indeed, because of linearity, both *must* spawn the same vertices). Rule DF:New introduces the new vertex into the spawn context, but not the touch context (it will only be added to the touch context after being spawned). These are the important features of the type system for ensuring deadlock freedom; the remaining rules are largely unchanged from the original graph kinding judgment and we describe them here only briefly. Rule DF:RecPi handles recursive parameterized graph types, which arise from recursive functions. The parameters are added to the appropriate contexts when checking the body. The outer spawn context must be empty, because it is not safe for linear resources (vertices) to be captured in a recursive binding, where they may be duplicated. This restriction is not needed in DF:Pi, which checks graph types that accept parameters but do not recur. Rules DF:Spawn and DF:Touch require u to be in the appropriate context. Finally, DF:App requires the vertex arguments to be in the appropriate contexts and removes the spawn arguments from the spawn context.

4.2 Soundness Proof

We now prove that a graph type that is declared to be deadlock-free by the analysis of the previous subsection (that is, one that is well-kinded) does not admit deadlocks. To do this, we show that any graph contained in the normalization of such a graph type obeys the *transitive joins* property [19], which implies deadlock freedom. In short, the transitive joins (TJ) property relies on a "permission to join" relation $<$, which is the transitive closure of the following two properties:

(1) If a spawns b, then a may touch b ($a < b$).
(2) If when a spawns b, a may touch c, then b also has permission to touch c ($b < c$).

It is shown that $<$ establishes a total order on threads, preventing the creation of cycles in the graph.

Preliminaries on Transitive Joins. We now go into more detail on the formal definitions surrounding transitive joins, which we will need in our proof. For more information, the reader is directed to the original presentation [19]. A program execution is abstracted as a *trace* t, which records a sequence of *actions* α. There are three types of actions: the initialization of the main thread a, written $init(a)$; the thread a spawning b, written $fork(a, b)$; and a touching b, written $join(a, b)$. We write the concatenation of two threads as $t_1; t_2$. We write $\cdot$ for the empty trace, and note that $t; \cdot = \cdot; t = t$.

The "permission-to-join" relation depends on the history of spawn operations, and so it is defined inductively over traces with the judgment $t \vdash a < b$, defined as follows:

$$\text{(TJ-LEFT)}\ \frac{t \vdash c \leq a}{t; fork(a, b) \vdash c < b} \qquad \text{(TJ-RIGHT)}\ \frac{t \vdash a < c}{t; fork(a, b) \vdash b < c} \qquad \text{(TJ-MONO)}\ \frac{t_1 \vdash a < b}{t_1; t_2 \vdash a < b}$$

We may also write $a \leq b$ to mean that $a = b$ or $a < b$. A trace is *TJ-valid* if it begins with the initialization of the main thread and all subsequent touches obey the permission-to-join relation. The judgment $t : A$ indicates that t is a TJ-valid trace with the set A of thread names. This set is added to by *fork* actions in the inductive definition of the judgment:

$$\frac{}{\bullet \leadsto_a \cdot}\ (\textsc{Tr:Empty}) \qquad \frac{g_1 \leadsto_a t_1 \quad g_2 \leadsto_a t_2}{g_1 \oplus g_2 \leadsto_a t_1; t_2}\ (\textsc{Tr:Seq})$$

$$\frac{g \leadsto_u t}{g \swarrow_u \leadsto_a \mathit{fork}(a,u); t}\ (\textsc{Tr:Spawn}) \qquad \frac{}{{}^{u}\searrow \leadsto_a \mathit{join}(a,u)}\ (\textsc{Tr:Touch})$$

Figure 5: Rules for producing traces.

$$\frac{}{\mathit{init}(a) : \{a\}}\ (\textsc{valid-init}) \qquad \frac{t : A \quad a \in A \quad b \notin A}{t; \mathit{fork}(a,b) : A \cup \{b\}}\ (\textsc{valid-fork}) \qquad \frac{t : A \quad t \vdash a < b}{t; \mathit{join}(a,b) : A}\ (\textsc{valid-join})$$

Well-formed graphs are TJ-valid. To connect our notation for graphs to transitive joins, we must define a way to produce traces from graphs. We write $g \leadsto_a t$ to mean that a graph whose main thread is named a produces the trace t. The rules for this judgment are defined in Figure 5. Spawns and touches are recorded appropriately. When a new thread is spawned using a vertex u, we reuse u as the name of the new thread and recursively compute the trace corresponding to the new thread by deriving $g \leadsto_u t$ (note that the "main" thread of this derivation has now changed to u). To produce a trace from the sequential composition of two graphs, we sequentially compose the traces resulting from the two graphs. Note that t will never contain an *init* action, so to produce a (potentially) valid trace, we would take $\mathit{init}(a); t$.

We now turn our attention to proving the main result of the section, which is that if a graph is in the normalization of a well-kinded (according to the rules of Figure 4) graph type, then the trace produced from the graph is TJ-valid. The proof uses the following technical lemma, which says that substituting graphs for graph variables or vertices for vertex variables in well-kinded graph types results in well-kinded graph types. Similar results have been shown for the original graph type well-formedness judgment [14], but we show them here for our deadlock-freedom judgment. The full proof is available in the supplementary appendix.

Lemma 1. (1) *If* $\cdot; \Omega, \vec{u}_f; \Psi, \vec{u}_t \vdash_{DF} G : \kappa$ *then*

$$\cdot; \Omega, \vec{u}'_f; \Psi, \vec{u}'_t \vdash_{DF} G[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t] : \kappa$$

and the height of this derivation is no larger than the height of the original typing derivation.

(2) *If* $\gamma : \kappa'; \Omega; \Psi \vdash_{DF} G : \kappa$ *and* $\cdot; \cdot; \Psi \vdash_{DF} G' : \kappa'$ *then* $\cdot; \Omega; \Psi \vdash_{DF} G[G'/\gamma] : \kappa$.

Proof.

(1) By induction on the derivation of $\cdot; \Omega, \vec{u}_f; \Psi, \vec{u}_t \vdash_{DF} G : \kappa$.

(2) By induction on the derivation of $\gamma : \kappa'; \Omega; \Psi \vdash_{DF} G : \kappa$

□

The heavy lifting for our main theorem is done by Lemma 2, which proves a stronger result. The lemma allows us to focus on a part of the graph and the corresponding part of the resulting trace. In the statement of the lemma, the trace generated up until this point is t_0 and is assumed to be well-formed with the set A_0 of vertices. We furthermore assume that we do not have permission to spawn any of the vertices in A_0 (that is, $A_0 \cap \Omega = \emptyset$), because this would result in spawning a vertex twice. We also assume that Ψ does indeed represent the set of vertices we have permission to touch based on the current trace t_0 (that is, for all $b \in \Psi$, we have $t_0 \vdash a < b$). Under these assumptions, the resulting trace $t_0; t$ is TJ-valid and its set of threads consists of A_0 plus the vertices in Ω (which must have been spawned), plus a set of fresh vertex names that will not conflict with any other names. Finally, the new trace gives permission to touch any newly-spawned vertices (i.e., those in Ω).

Lemma 2. *Suppose* $\cdot; \Omega; \Psi \vdash_{DF} G : *$, *and* $g \in \mathit{Norm}_n(G)$ *for some* n. *Let* $t_0 : A_0$ *be a TJ-valid trace such that* $A_0 \cap \Omega = \emptyset$ *and for all* $b \in \Psi$, *we have* $t_0 \vdash a < b$. *If* $g \leadsto_a t$, *then* $t_0; t : A$ *is TJ-valid and* $A = \Omega \cup A_0 \cup A_f$ *where all vertices in* A_f *are fresh, and for all* $b \in \Omega$, *we have* $t_0; t \vdash a < b$.

Proof. By lexicographic induction on n and the derivation of $\cdot; \Omega; \Psi \vdash_{DF} G : *$. If $n = 0$, then $\mathit{Norm}_n(G) = \emptyset$, which contradicts $g \in \mathit{Norm}_n(G)$. So, suppose $n > 0$ and proceed by induction on the derivation.

We prove some representative cases here. Proofs for the remaining cases are available in the supplementary appendix.

- DF:Seq. Then $G = G_1 \oplus G_2$ and $g = g_1 \oplus g_2$ where $g_1 \in \mathit{Norm}_n(G_1)$ and $g_2 \in \mathit{Norm}_n(G_2)$ and $\Delta; \Omega_1; \Psi \vdash_{DF} G_1 : *$ and $\Delta; \Omega_2; \Psi, \Omega_1 \vdash_{DF} G_2 : *$. We have $A_0 \cap \Omega_1, \Omega_2 = \emptyset$ and for all $b \in \Psi$, $t_0 \vdash a < b$. By inversion, $t = t_1; t_2$ and $g_1 \leadsto_a t_1$ and $g_2 \leadsto_a t_2$. By induction, $t_0; t_1 : A_1$ is TJ-valid and $A_1 = \Omega_1 \cup A_0 \cup A_{f1}$ where all vertices in A_{f1} are fresh, and for all $b \in \Omega_1$, we have $t_0; t_1 \vdash a < b$. We have $\Omega_1 \cap \Omega_2 = \emptyset$, so $A_1 \cap \Omega_2 = \emptyset$. For all $b \in \Psi, \Omega_1$, we have $t_0; t_1 \vdash a < b$. By induction on the second premise, we have $t_0; t_1; t_2 : A$ is TJ-valid where $A = \Omega_2 \cup A_1 \cup A_f = \Omega_1, \Omega_2 \cup A_0 \cup A_f$ where all vertices in A_f are fresh, and for all $b \in \Omega_2$, we have $t_0; t_1; t_2 \vdash a < b$. Combining this with the above and monotonicity of $<$, for all $b \in \Omega_1, \Omega_2$, we have $t_0; t_1; t_2 \vdash a < b$.
- DF:Spawn. Then $G = G_1 \swarrow_u$ and $\Delta; \Omega_1; \Psi \vdash G_1 : *$ where $\Omega = \Omega_1, u$, and $g = g_1 \swarrow_u$, where $g_1 \in \mathit{Norm}_n(G_1)$. By inversion, $t = \mathit{fork}(a,u); t_1$ where $g_1 \leadsto_a t_1$. We have $A_0 \cap \Omega_1 = \emptyset$. By valid-fork, we have $t_0; \mathit{fork}(a,u)$:

$A_0 \cup \{u\}$ is TJ-valid and $(A_0 \cup \{u\}) \cap \Omega_1 = \emptyset$. By induction using $t_0; fork(a, u)$ as the trace, $t_0; t_1 : A$ is TJ-valid and $A = \Omega_1 \cup A_0 \cup \{u\} \cup A_f = \Omega \cup A_0 \cup A_f$ where all vertices in A_f are fresh and for all $b \in \Omega_1$, we have $t_0; t \vdash a < b$. For all $b \in \Omega$, if $b \in \Omega_1$, then $t_0; t \vdash a < b$ from above. If $b = u$, then $t_0; t \vdash a < b$ by TJ-LEFT.

- DF:App. Then $G = G_1[\vec{u}'_f; \vec{u}'_t]$ and $\Omega = \Omega_1, \vec{u}'_f$ and

$$\cdot; \Omega_1; \Psi \vdash_{DF} G_1 : \Pi\vec{u}_f; \vec{u}_t.*$$

By inversion, either

(1) $G_1 = \Pi\vec{u}_f; \vec{u}_t.G_2$ and $g \in Norm_n(G_2[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t])$ and $\cdot; \Omega_1, \vec{u}_f; \Psi, \vec{u}_t \vdash_{DF} G_2 : *$ or

(2) $G_1 = \mu\gamma.\Pi\vec{u}_f; \vec{u}_t.G_2$ and

$$g \in Norm_{n-1}(G_2[\mu\gamma.\Pi\vec{u}_f; \vec{u}_t.G_2/\gamma][\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t])$$

and $\gamma : \Pi\vec{u}_f; \vec{u}_t.*; \cdot; \Psi, \vec{u}_t \vdash_{DF} G_2 : *$.

Proceed in these two cases.

(1) By Lemma 1, $\cdot; \Omega; \Psi \vdash_{DF} G_2[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t] : *$. The result follows by induction.

(2) By Lemma 1,

$$\cdot; \cdot; \Psi \vdash_{DF} G_2[\mu\gamma.\Pi\emptyset; \vec{u}_t.G_2/\gamma][\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t] : *$$

The result follows by induction, decreasing on n.

□

The main theorem simply instantiates the lemma with appropriate initial conditions: Ω and Ψ are empty, and the trace generated so far is simply $init(a)$, where a is a designated name for the main thread.

Theorem 1. *Suppose $\cdot; \cdot; \cdot \vdash_{DF} G : *$, and $g \in Norm_n(G)$ for some n. If $g \rightsquigarrow_a t$, then $init(a); t : A$ is TJ-valid.*

Proof. This is a direct result of Lemma 2, because $init(a) : \{a\}$ is TJ-valid by VALID-INIT, and $\{a\} \cap \cdot = \emptyset$. □

5 IMPLEMENTATION AND EVALUATION

We implemented the deadlock analysis, based on the rules in Section 4, in OCaml as an extension of GML [14], a tool for inferring graph types from source programs in a large subset of OCaml (extended with futures as a built-in type). In particular, the language subset accepted by GML includes OCaml-style mutable references and is sufficient to express all of the examples in this paper (except the extended counterexample in Section 3, which as described in the footnote, cannot be inferred by GML). After GML infers graph types for the program, the user can request that one function or the entire program be checked for deadlocks, in which case our analysis extracts the corresponding graph type from the graph-annotated output of GML and runs our algorithm on it. It is relatively straightforward to turn the rules of Figure 4 into a type-checking algorithm because the rules are *syntax-directed*, that is, it is clear from the syntax of the graph type being checked which rule should be applied. Before presenting our evaluation of the implementation, we describe one additional optimization that improves the precision of the algorithm on some examples.

New pushing. Consider the graph type below.

$$\mu\gamma.\nu u.\bullet \vee (\gamma \swarrow_u \oplus \gamma \oplus {}^u\searrow)$$

This graph type corresponds to many common divide-and-conquer parallel algorithms, e.g. Figure 1. However, as shown, it is not well-formed according to the rules of Figure 4. The reason is that the vertex u is placed into the spawn context for both branches of the $\vee$, but the left branch (corresponding to the base case of the algorithm) does not use this vertex, violating linearity. However, the graph above is semantically equivalent to this one:

$$\mu\gamma.\bullet \vee (\nu u.\gamma \swarrow_u \oplus \gamma \oplus {}^u\searrow)$$

where we have simply moved the "new" binding inside the recursive case of the graph type, and so the base case is no longer in the scope of this binding. However, GML will always produce the first graph type because, for efficiency reasons, it only inserts "new" bindings at the top of function bodies. In order to reduce false positives for graph types produced by GML, we introduce a procedure we call "new pushing", which pushes "new" bindings through a graph type to the smallest scope possible, and apply this transformation to graph types before checking them for deadlocks.

Precision comparison. In order to show the flexibility and precision of our algorithm, we ran the implementation on four example programs, with and without deadlocks:

(1) *Fibonacci*: An example from Muller [14] that computes the 8^{th} Fibonacci number in parallel by spawning (in parallel) 8 threads to compute the first 8 Fibonacci numbers; threads 3–8 touch the previous two threads and sum their results.
(2) *FibDL*: The Fibonacci program from above but with one of the touches altered to create a cycle.
(3) *Pipeline*: The motivating example of GML, which performs a pipelined map over a list of inputs.
(4) *Counterex.*: The second counterexample of Section 3.

For Counterex., to avoid the subtlety discussed in Section 3, rather than run a source program through GML, we hand-coded the AST for the graph type of the counterexample and ran our deadlock detection algorithm on this directly. Because the contribution of this paper is the deadlock detection algorithm, which already operates on ASTs for graph types, no part of our algorithm is bypassed.

Table 1 lists the examples and (in column 2) whether or not the example has a deadlock. The third column indicates

Table 1: Example programs comparing the precision of our deadlock detector with prior work.

Program	DL?	Does analysis give correct answer?		
		Ours	GML [14]	Known Joins [8]
Fibonacci	No	✓	✓	✗
FibDL	Yes	✓	✓	✓
Pipeline	No	✓	✓	✓
Counterex.	Yes	✓	✗	✓

that our algorithm gives the correct answer in each case (i.e., correctly identifies Fibonacci and Pipeline as deadlock-free and FibDL and Counterex. as having deadlocks). The next column shows the same results for GML [14], which is shown to be unsound by the counterexample. We also compare to Known Joins (KJ) [8], a weaker version of the Transitive Joins property which also guarantees deadlock-freedom but is overly pessimistic in some cases and, for example, is not able to show the deadlock-freedom of the Fibonacci example. We manually applied the rules of KJ to determine whether each example would be considered valid by KJ at runtime.

We make two important caveats about this evaluation. First, it is difficult to make an apples-to-apples comparison between static and dynamic analyses. While we show in Section 4 that any program guaranteed deadlock-free by our algorithm will have the transitive joins property, the reverse is not true, and cannot be true for any static analysis. Determining whether a program will have a dynamic property (such as deadlock, known joins, or transitive joins) at runtime using a static analysis is undecidable by reduction to the halting problem, so there will naturally be some programs that are valid under transitive joins (and known joins) but cannot be guaranteed so by our static analysis. A more precise characterization of the false positive profile of our algorithm is an area for future work. We also note that, while a quantitative evaluation is outside the scope of this paper, the deadlock detection algorithm finishes in under 1ms on a commodity desktop for all four examples.

6 RELATED WORK

Numerous solutions to the problem of deadlock have been proposed since 1971 when Coffman et al. [7] neatly characterized the problem and categorized potential solutions. The classes of solutions they propose are (1) *prevent* deadlocks statically by detecting whether the conditions to allow them are present in source code, (2) *avoid* deadlocks at runtime by detecting whether the conditions to allow them have arisen dynamically and (3) *detect* at runtime whether a deadlock has occurred, and ideally recover from the situation. Dynamic techniques (2 and 3) are far too numerous to survey here, so we focus on the most closely related ones. The *known joins* property [8] restricts threads to join on, or touch, futures spawned by an ancestor in the thread hierarchy. Known joins is, however, fairly restrictive and was later extended to *transitive joins* [19], which extends the "permission-to-join" relation of known joins with transitivity. In doing so, it establishes a total order on threads at runtime, in a way similar to work on SP-order [2, 22] has been used for runtime data race detection. We have shown that programs identified by our algorithm as deadlock-free obey the transitive joins property and are therefore indeed deadlock-free. We have also shown (in Section 5) that our program can identify as deadlock-free programs that known joins cannot. Voss and Sarkar [20] present a dynamic deadlock detection algorithm (class 3 above) for *promises*, a mechanism related to futures for which they identify analogues of the two deadlock situations we prevent in futures (cycles and waits on promises that will never be completed). Their semantics requires tracking an *owner* for each promise and detects if a promise is unowned or if the ownership relation is cyclic.

Static techniques fall into two broad categories: type systems for controlling ownership of resources, and dataflow analyses. Our work falls into the former, but operates at the level of graph types rather than source programs. Boyapati et al. [5] also proposed a type system for ownership of locks that prevents deadlock. A similar ownership type system prevents data races in Rust [1]. Vasconcelos et al. [18] present a type system for a typed assembly language that prevents deadlocks but requires annotating locks with an ordering. Most dataflow analyses for deadlock (e.g., [9, 13, 16, 21]) track relations between threads and usage of resources, in some sense building an approximation of a dependency graph. Boudol [4] proposes an approach that mixes static and dynamic techniques: a type system guarantees that programs can be safely run using a "prudent" operational semantics that makes deadlocks impossible by construction.

7 CONCLUSION

We have proposed a static algorithm for predicting deadlock. The analysis is based on *graph types*, a language-independent representation of the set of dependency graphs that might result from a given program, and so in principle can be extended to many paradigms and languages. We have shown the soundness of the algorithm by reduction to *transitive joins*, a condition that is known to imply deadlock freedom. We have implemented a prototype of the analysis on top of GML, a graph type inference tool for a subset of the OCaml language, and shown that it can effectively detect deadlocks in a variety of examples. This work shows the promise of graph types for the development of language-agnostic static analyses for parallel programs, which we hope can be applied in the future to other problems such as race detection.

REFERENCES

[1] [n.d.]. The Rust language. https://www.rust-lang.org. Accessed: 2023-07-07.

[2] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-Fly Maintenance of Series-Parallel Relationships in Fork-Join Multithreaded Programs *(SPAA '04)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/1007912.1007933

[3] Guy E. Blelloch and Margaret Reid-Miller. 1997. Pipelining with Futures. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures* (Newport, Rhode Island, USA) *(SPAA '97)*. Association for Computing Machinery, New York, NY, USA, 249–259. https://doi.org/10.1145/258492.258517

[4] Gérard Boudol. 2009. A Deadlock-Free Semantics for Shared Memory Concurrency. In *Theoretical Aspects of Computing - ICTAC 2009*, Martin Leucker and Carroll Morgan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 140–154.

[5] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Seattle, Washington, USA) *(OOPSLA '02)*. Association for Computing Machinery, New York, NY, USA, 211–230. https://doi.org/10.1145/582419.582440

[6] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java* (Kongens Lyngby, Denmark) *(PPPJ '11)*. Association for Computing Machinery, New York, NY, USA, 51–61. https://doi.org/10.1145/2093157.2093165

[7] E. G. Coffman, M. Elphick, and A. Shoshani. 1971. System Deadlocks. *ACM Comput. Surv.* 3, 2 (Jun 1971), 67–78. https://doi.org/10.1145/356586.356588

[8] Tiago Cogumbreiro, Rishi Surendran, Francisco Martins, Vivek Sarkar, Vasco T. Vasconcelos, and Max Grossman. 2017. Deadlock Avoidance in Parallel Programs with Futures: Why Parallel Tasks Should Not Wait for Strangers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 103 (Oct. 2017), 26 pages. https://doi.org/10.1145/3143359

[9] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

[10] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 212–223. https://doi.org/10.1145/277650.277725

[11] Robert H. Halstead. 1985. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* 7 (1985), 501–538. https://doi.org/10.1145/4472.4478

[12] Maurice Herlihy and Zhiyu Liu. 2014. Well-Structured Futures and Cache Locality. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 155–166. https://doi.org/10.1145/2555243.2555257

[13] Stephen P. Masticola. 1993. *Static Detection of Deadlocks in Polynomial Time.* Ph.D. Dissertation. USA. UMI Order No. GAX93-33428.

[14] Stefan K. Muller. 2022. Static Prediction of Parallel Computation Graphs. *Proc. ACM Program. Lang.* 6, POPL, Article 46 (Jan 2022), 31 pages. https://doi.org/10.1145/3498708

[15] Alan Mycroft. 1984. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, M. Paul and B. Robinet (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 217–228. https://doi.org/10.1007/3-540-12925-1_41

[16] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. 2009. Effective static deadlock detection. In *2009 IEEE 31st International Conference on Software Engineering*. 386–396. https://doi.org/10.1109/ICSE.2009.5070538

[17] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 113:1–113:30. https://doi.org/10.1145/3408995

[18] Vasco T. Vasconcelos, Francisco Martins, and Tiago Cogumbreiro. 2010. Type Inference for Deadlock Detection in a Multithreaded Polymorphic Typed Assembly Language. *Electronic Proceedings in Theoretical Computer Science* 17 (feb 2010), 95–109. https://doi.org/10.4204/eptcs.17.8

[19] Caleb Voss, Tiago Cogumbreiro, and Vivek Sarkar. 2019. Transitive Joins: A Sound and Efficient Online Deadlock-Avoidance Policy *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 378–390. https://doi.org/10.1145/3293883.3295724

[20] Caleb Voss and Vivek Sarkar. 2021. An Ownership Policy and Deadlock Detector for Promises. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 348–361. https://doi.org/10.1145/3437801.3441616

[21] Amy Williams, William Thies, and Michael D. Ernst. 2005. Static Deadlock Detection for Java Libraries. In *ECOOP 2005 - Object-Oriented Programming*, Andrew P. Black (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 602–629.

[22] Yifan Xu, Kyle Singer, and I-Ting Angelina Lee. 2020. Parallel Determinacy Race Detection for Futures. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 217–231. https://doi.org/10.1145/3332466.3374536