Static Prediction of Parallel Computation Graphs

STEFAN K. MULLER, Illinois Institute of Technology, USA

Many algorithms for analyzing parallel programs, for example to detect deadlocks or data races or to calculate the execution cost, are based on a model variously known as a *cost graph*, *computation graph* or *dependency graph*, which captures the parallel structure of threads in a program. In modern parallel programs, computation graphs are highly dynamic and depend greatly on the program inputs and execution details. As such, most analyses that use these graphs are either dynamic analyses or are specialized static analyses that gather a subset of dependency information for a specific purpose.

This paper introduces *graph types*, which compactly represent all of the graphs that could arise from program execution. Graph types are inferred from a parallel program using a *graph type system* and inference algorithm, which we present drawing on ideas from Hindley-Milner type inference, affine logic and region type systems. We have implemented the inference algorithm over a subset of OCaml, extended with parallelism primitives, and we demonstrate how graph types can be used to accelerate the development of new graph-based static analyses by presenting proof-of-concept analyses for deadlock detection and cost analysis.

CCS Concepts: • **Software and its engineering** → **Parallel programming languages**; Automated static analysis; • **Theory of computation** → *Linear logic*.

Additional Key Words and Phrases: parallel programs, graph types, cost graphs, computation graphs, type systems, type inference

ACM Reference Format:
Stefan K. Muller. 2022. Static Prediction of Parallel Computation Graphs. *Proc. ACM Program. Lang.* 6, POPL, Article 46 (January 2022), 36 pages. https://doi.org/10.1145/3498708

1 INTRODUCTION

Recent trends in hardware have led to increased interest in reasoning about and developing parallel software. On the reasoning side, many techniques for studying parallel programs center around an idea variously called a *dependency graph*, *computation graph* or *cost graph*, which abstracts away details of program semantics to show the parallel structure of the program at a high level: in the model we consider in this paper, vertices in a graph represent threads in a program and edges between vertices signify control dependencies between threads. In work spanning several decades, these graphs have been used as a basis for studies of, to name a few examples, deadlock [Cogumbreiro et al. 2018], data races [Banerjee et al. 2006], priority inversions [Babaoğlu et al. 1993] and runtime cost [Blelloch and Greiner 1995, 1996]. In many cases, these properties can be easily "read off" of a suitable representation the graph (for example, a deadlock corresponds to a cycle in the graph). On the development side, most newer languages and systems for developing parallel programs have built on the paradigm of *implicit parallelism*, in which a programmer specifies opportunities for parallelism using high-level abstractions and the language runtime does the work of scheduling. Implicit parallelism is hardly a new idea—it dates back to systems like Id [Arvind and Gostelow 1978] and Multilisp [Halstead 1985]—but is now present in some form in many mainstream languages.

Author's address: Stefan K. Muller, smuller2@iit.edu, Illinois Institute of Technology, USA.

2475-1421/2022/1-ART46
https://doi.org/10.1145/3498708

In implicitly parallel programs, parallelism is *dynamic*: new threads are created and synchronized at runtime based on the execution of the program. As a result, the number and structure of threads are not fixed ahead of time but can and generally do change in response to program inputs. Because computation graphs for implicit parallel programs are inherently dynamic, many graph-based analyses for properties like the ones mentioned above are also dynamic. These analyses track information about dependencies between threads as a program executes. To keep overhead manageable, they will generally track only a subset of the available dependency information, sometimes sacrificing precision for speed. Of course, these dynamic analyses also suffer from the same theoretical issues as dynamic analyses in general: for example, a dynamic deadlock detector will generally only be able to determine if a deadlock can or will occur for a particular execution and can say little or nothing about the behavior of the program in general, on any set of inputs.

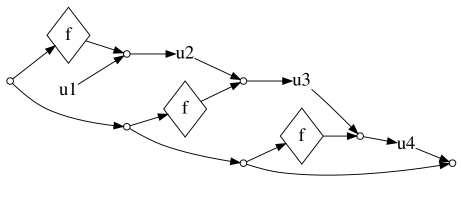
Graph-based static analyses for the same properties, which can be sound and do not suffer from runtime overhead, exist as well, but are tricky to implement because of the dynamic nature of computation graphs. The designer of a dynamic analysis can take the simple, idealized graph-based algorithm (e.g., look for a cycle in the graph to detect a deadlock), track enough dependency information to be able to detect the required property and check the graph dynamically as it's built. Static analyses, on the other hand, traverse a program and construct approximations of the relevant dependency information; the algorithm designer must decide what information to track, and adapt the simple graph-based algorithm to operate over this information.

In this paper, we take a step toward combining the best of both approaches by constructing compact, static representations of the set of computation graphs that may arise from a program at runtime. We call these representations *graph types* by analogy to static types, which may be thought of as compact static approximations of the set of *values* a program might produce at runtime. Graph types capture the dependency information required for a wide range of graph-based parallel program analyses, and so one can design a new static analysis by taking the simple graph-based formulation of the desired algorithm (e.g. look for a cycle in the graph to detect a deadlock) and adapt it to "read off" the property from the graph type rather than an actual graph. The analysis designer no longer needs to build a static analysis to traverse the program and collect dependency information, as long as the graph type of the program is available.

In order to produce graph types from parallel programs, this paper presents a *graph type system* for a simple parallel calculus, as well as an implementation of an inference algorithm for graph types that operates over a sizable subset of OCaml. Inferring graph types is made difficult by more general forms of implicit parallelism, notably *futures*, which allow asynchronous computations to be treated as first-class values in a program. As an example, Figure 1a shows a graph for a program that applies an expensive function f to every element in a list in a parallel pipelined fashion. In the graph, u_1 through u_4 represent futures created at runtime; these vertices must be given unique names when the future is created and these names must be available later when the futures are used, possibly far away in the program. To track vertex names, we adapt notions from region type systems [Tofte and Talpin 1997]. Graph types face an additional problem not handled by region type systems: well-formed graphs cannot reuse vertex names. We ensure unicity of names by placing an affine restriction on vertex names. All of this is done with *no additional burden on the programmer*: all required annotations are automatically inferred by our implementation.

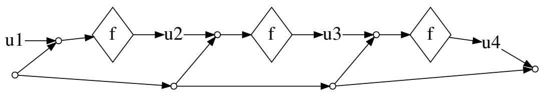
In addition to accelerating the development of new static analyses, the graph types inferred by our implementation can provide the programmer with insight into a program. Figure 1b shows an almost imperceptibly different implementation of the pipeline program which results in substantially less parallelism. Noticing this implementation error in the code would require a great deal of experience and domain knowledge. However, the difference is quite striking in the visualizations of the two

```
let rec pipe (fut, l) =
...pipe (future (f h + (touch fut)), t)
```



a

```
let rec pipe (fut, l) =
...pipe (future ((touch fut) + f h), t)
```



b

Fig. 1. Partial code and graph visualization for a pipelined program (left) and an incorrect implementation of it (right). The visualizations are produced automatically by our implementation.

graph types, which can be *produced automatically by our implementation*: the graph on the right shows all three invocations of f occurring sequentially.

This work subsumes and concretizes techniques already used in the parallelism theory community, where it is common to see dynamic semantics that build a graph as they evaluate a program. These are particularly common in the cost models community, where they are known as *cost semantics*. It is common to use these models to do a sort of static meta-reasoning over the set of graphs that can be produced by any execution of the program. For example, to assure ourselves that a program cannot have a deadlock, we can use meta-level techniques to prove that all possible evaluations of a cost semantic-like model result in acyclic graphs. These reasoning techniques are formalized anew for every property of interest (deadlock, cost, etc.). We note that this is exactly the sort of reasoning that is reified, once and for all properties, by graph types: if, under the kind of cost semantics described above, program e could execute to produce a graph g, then g is described by the graph type of e. The formal version of this statement is, in fact, the soundness theorem of our graph type system, and we state and prove it using a traditional cost semantics of the form described above.

In summary, the contributions of this paper include

- *Graph types*, a compact representation for sets of possible computation graphs.
- A graph type system for a simple parallel calculus λ^G, which computes graph types from annotated parallel programs.
- A proof, using a more traditional cost semantics, that any graph arising from execution of a λ^G program is described by its graph type.
- An algorithm for inferring the annotations required for the graph type system of λ^G from unannotated input programs.
- An implementation of the inference algorithm in a compiler front-end that accepts *unannotated* OCaml-like parallel programs and infers, and optionally visualizes, graph types.
- Proof-of-concept implementations of several applications of graph types, including deadlock detection, cost analysis and program optimization.

The remainder of the paper is organized as follows. In Section 2, we introduce the notations used throughout the paper. Section 3 overviews our methodology on a restricted set of parallelism constructs, to introduce the techniques without their full complexity. In Sections 4 and Section 5, we introduce the full syntax and semantics for graph types, the λ^G calculus and the graph type system. Section 6 presents the graph type inference algorithm, which we implement and evaluate, including by developing new graph-based static analyses over graph types, in Section 7. Finally, Sections 8–10 discuss related and future work, and conclude. An appendix available as supplementary material on the ACM Digital Library contains proofs and additional details.

$$\begin{array}{lcll}
\bullet & \triangleq & (\{u\}, \emptyset, u, u) & u \text{ fresh} \\
(V_1, E_1, s_1, t_1) \oplus (V_2, E_2, s_2, t_2) & \triangleq & (V_1 \cup V_2, E_1 \cup E_2 \cup \{(t_1, s_2)\}, s_1, t_2) & V_1 \cap V_2 = \emptyset \\
(V_1, E_1, s_1, t_1) \otimes (V_2, E_2, s_2, t_2) & \triangleq & (V_1 \cup V_2 \cup \{u_1, u_2\}, & u_1, u_2 \text{ fresh,} \\
 & & \;E_1 \cup E_2 \cup \{(u_1, s_1), (u_1, s_2), (t_1, u_2), (t_2, u_2)\}, u_1, u_2) & V_1 \cap V_2 = \emptyset \\
(V, E, s, t) \swarrow_u & \triangleq & (V \cup \{u, u'\}, E \cup \{(u', s), (t, u)\}, u', u') & u' \text{ fresh}, u \notin V \\
{}^u\searrow & \triangleq & (\{u'\}, \{(u, u')\}, u', u') & u' \text{ fresh}
\end{array}$$

Fig. 2. Shorthands for combining graphs.

2 PRELIMINARIES

We first review the notations and parallelism mechanisms used in the remainder of the paper.

2.1 Graph Notation

A *computation graph* or simply *graph* is used to represent the parallel structure of a program. Vertices of a graph (for which we will use metavariables u and variants) represent sequential computations, and (directed) edges between vertices represent dependences. If there is a path from a vertex u_1 to a vertex u_2, then u_1 must complete before u_2 begins. If there is no path between two vertices, then the computations represented by those two vertices may be executed in parallel.

We represent a graph as a quadruple (V, E, s, t) containing the sets of vertices V and edges E. We will represent a directed edge from u_1 to u_2 as the pair (u_1, u_2). The graph quadruple also contains a designated "source" vertex s which has no ancestors in the graph, and a designated "sink" vertex t which has no descendants in the graph. The vertices s and t represent the start and end of the "main" thread of the graph. Note that a graph may have sink vertices other than t (as a program may spawn parallel threads with which it never synchronizes), but s will be its only source vertex.

We write a graph consisting of a single vertex (which corresponds to a fully sequential computation) as $\bullet$, and use conventional shorthands [Blelloch and Greiner 1995, 1996] for combining subgraphs into larger graphs. These shorthands are defined in Figure 2. For two graphs g_1 and g_2, $g_1 \oplus g_2$ denotes the *sequential composition* g_1 followed by g_2 and $g_1 \otimes g_2$ denotes the *parallel composition* allowing the computations represented by the two graphs to occur in parallel. Sequential composition adds an edge from the sink of g_1 to the source of g_2, thus composing the two "main" threads. One special case is not listed in the figure: because we are primarily interested in the parallel structure of a program, we will treat $\bullet \oplus \bullet$ (which is still a sequential computation) as equivalent to $\bullet$. Parallel composition results in a graph whose "main" thread consists of two newly created vertices: u_1 has the sources of g_1 and g_2 as children and u_2 has the sinks as parents. Left composition [Spoonhower 2009] $g \swarrow_u$ indicates that the graph g should run asynchronously with the "main" thread of the graph. The "main" thread of the resulting graph consists of only a new vertex u', with an edge to the source of g. The indicated vertex u is installed as the sink of g, so that another thread can easily locate and synchronize with g. This is done with the operation ${}^u\searrow$, which again creates a new vertex u' in the "main" thread and installs a synchronization edge (u, u').

It is important to note that all vertices in a graph must have unique identifiers. This is enforced by the operations in Figure 2 by requiring that vertex sets of combined graphs be disjoint and that the vertex of a left composition not already appear in the graph. Both requirements cause no loss of generality, as vertices of a graph can be freely renamed, but this renaming must be done consistently (e.g., all uses of u in the graph must be updated if u is renamed). In most related treatments of graph notation, such renamings are performed implicitly, and so it may seem that we are belaboring this point. However, as we will see, much of the complexity of graph types and of the techniques in the paper arises out of the necessity to keep vertex names distinct.

```
let rec fib n =
  if n <= 1 then 1
  else
    let (a, b) = par (fib (n - 1), fib (n - 2))
    in a + b
```

```
let rec fib n =
  if n <= 1 then 1
  else
    let fut = future (fib (n - 1)) in
    let b = fib (n - 2)
    in (touch fut) + b

```

Fig. 3. Two versions of the Fibonacci program.

2.2 Parallelism Mechanisms

In this paper, we will consider two mechanisms of expressing parallelism. The first is called *fork-join* or *nested* parallelism. In the syntax of our paper, fork-join parallelism is introduced with the construct `par (e1, e2)` where `e1` and `e2` are arbitrary expressions. This operation spawns two threads to execute `e1` and `e2` in parallel. When both expressions evaluate to values, a pair of the two values is returned. Many common parallel algorithms, such as the broad and useful class of *divide and conquer* algorithms, can be expressed using fork-join parallelism. One example of a fork-join parallel algorithm, which has become the "Hello World" of parallel programs, is the inefficient recursive Fibonacci function, in which the two recursive calls are performed in parallel. The code for this example is in the left side of Figure 3.

Another common parallelism mechanism is *futures*. The operation `future e` spawns a thread, or future, to compute `e`, which runs in parallel with the current thread, and returns a handle to this future. The handle can be used to *touch* or *force* the future. For this, we will use the syntax `touch e`, which evaluates `e` down to a future handle, waits for the corresponding future to complete, and returns the value returned by the future. Futures are a very useful and general parallelism mechanism, and indeed are powerful enough to encode fork-join parallelism in a straightforward way. For example, on the right side of Figure 3, we rewrite the above Fibonacci program using a future to compute `fib (n - 1)`. The other recursive call is performed in parallel in the main thread. The main thread then uses the construct `touch` to get the result of the asynchronous computation before adding the two results.

3 GRAPH TYPES FOR FORK-JOIN PROGRAMS

Before introducing the full complexity of our language and analysis, we present an overview of the approach by working through the approach of the paper on a simple calculus λ^G_{FJ} that allows only fork-join programs. This section will thus demonstrate several of the important concepts and contributions we introduce in the paper, particularly graph types and the type-and-effect system for assigning graph types to programs, without the additional complexity necessary to support more general forms of parallelism. Because what is presented in this section is a strict subset of the full presentation to come in the following sections, we will not prove any metatheoretic results in this section, but defer these to the presentation of the full systems.

3.1 Graph Types

The first contribution of this paper is a compact notation for representing families of graphs that could arise from a cost semantics. We call these representations *graph types*, by analogy to how types represent families of values that could arise from execution of a program. The syntax for graph types for λ^G_{FJ}, shown in Figure 4, reuses much of the notation for graphs introduced in

$$G \quad ::= \quad \gamma \mid \bullet \mid G \oplus G \mid G \otimes G \mid G \vee G \mid \mu\gamma.G$$

Fig. 4. Partial syntax for graph types.

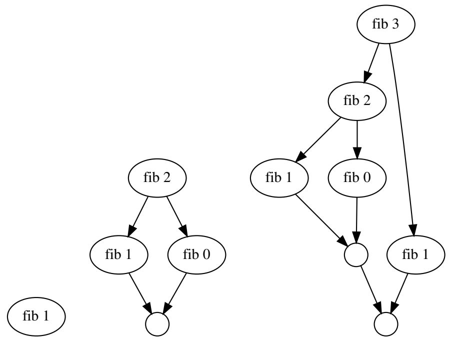


Fig. 5. Graphs for the Fibonacci program for $n = 0$ through $n = 3$ (left to right).

Section 2, with the same meaning. The syntax also includes two new notations, $G \vee G$ to denote two alternative possible graph types, and the notation $\mu\gamma.G$ for recursive graph types.

We illustrate the use of both of these connectives using the fork-join Fibonacci example of Section 2. Figure 5 shows the graphs for $n = 1$ through $n = 3$. For readability, some vertices are labeled with the call to `fib` they represent. The progression of graphs shows a pattern that matches the recursive structure of the code: the graph for `fib n` consists of either a single vertex (if the "then" branch of the conditional is taken) or two parallel recursive instances of the graph for `fib n`.

If we wish to compactly represent all of the graphs that could result from a call to `fib`, we will need some sort of recursion. Abusing notation liberally, we could define the set of graphs that could result from calls to `fib` with the following recursive definition:

$$\mathit{Graphs}(\texttt{fib}) \triangleq \{\bullet\} \cup \{g_1 \otimes g_2 \mid g_1, g_2 \in \mathit{Graphs}(\texttt{fib})\}$$

We introduce the connective $\vee$ to indicate two possible graphs (in the above example, we could write $\bullet \vee G_1 \otimes G_2$ to indicate that the graph is either a singleton or a parallel composition). We also introduce recursive graph types, borrowing notation from recursive types. The graph type $\mu\gamma.G$ denotes a recursive graph type. It binds the graph type variable γ inside G; occurrences of γ inside G indicate a recursive occurrence of the whole graph. Using this new notation, we could define the graph type for the Fibonacci program as follows:

$$G_{\texttt{fib}} \triangleq \mu\gamma.(\bullet \vee (\gamma \otimes \gamma))$$

3.2 Normalization: From Graph Types to Graphs

We define a procedure for enumerating sets of graphs that belong to a graph type. We call this procedure *normalization*, and will use it, among other purposes, to validate the graph type system. Any recursive graph type will correspond to an infinite set of graphs, because the recursion in the graph type can be unrolled an unbounded number of times[1]. We avoid infinite sets by introducing an index parameter, which bounds the number of times recursive graph types will be unrolled.

The normalization procedure for fork-join graph types is shown in Figure 6, and produces a set of graphs using the notation of Figure 2 (recall that the notations $\bullet$, $\oplus$ and $\otimes$ are used for both graph types and graphs; on the left hand side of the definitions, they refer to the graph type operations and on the right hand side, they refer to the graph operations). When the integer index is

[1]In terminating programs, these infinite graphs cannot result from actual executions because the runtime recursion will eventually reach the base case; the graph type notation deliberately abstracts away this detail.

$$\begin{array}{lcl}
Norm_0(G) & \triangleq & \{\bullet\} \\
Norm_n(\bullet) & \triangleq & \{\bullet\} \\
Norm_n(G_1 \otimes G_2) & \triangleq & \{g'_1 \otimes g'_2 \mid g'_1 \in Norm_n(G_1), g'_2 \in Norm_n(G_2)\} \\
Norm_n(G_1 \oplus G_2) & \triangleq & \{g'_1 \oplus g'_2 \mid g'_1 \in Norm_n(G_1), g'_2 \in Norm_n(G_2)\} \\
Norm_n(G_1 \vee G_2) & \triangleq & Norm_n(G_1) \cup Norm_n(G_2) \\
Norm_n(\mu\gamma.G) & \triangleq & Norm_{n-1}(G[\mu\gamma.G/\gamma]) \cup Norm_{n-1}(\mu\gamma.G)
\end{array}$$

Fig. 6. Normalization for fork-join graph types.

0, normalization returns only the singleton graph: this ensures that unrolling recursive graphs will eventually "bottom out" to a single vertex. Sequential and parallel compositions of graph types G_1 and G_2 are normalized by computing the normalizations of both graph types and then pairwise composing the resulting sets. The alternation $G_1 \vee G_2$ corresponds to a graph type that could be either G_1 or G_2, so the normalization of this graph type is the union of the normalizations of G_1 and G_2. The normalization of a recursive graph type at index n includes all of the graphs with the recursion unrolled 1 through n times. To normalize a recursive graph type, we substitute the recursive graph type for the variable γ in the body, and decrement the index. The normalization also includes all graphs with fewer than n unrollings (e.g., in the 3-way unrolling of $G_{\texttt{fib}}$, we allow some recursive branches to terminate early, which does in fact occur in the program.) To this end, we include $Norm_{n-1}(\mu\gamma.G)$ in the normalization. As an example, we normalize $G_{\texttt{fib}}$ from above, using $A \times B$ as shorthand for $\{g'_1 \otimes g'_2 \mid g'_1 \in A, g'_2 \in B\}$.

$$\begin{array}{lcl}
Norm_1(G_{\texttt{fib}}) & = & \{\bullet\} \\
Norm_2(G_{\texttt{fib}}) & = & Norm_1(\bullet \vee G_{\texttt{fib}} \otimes G_{\texttt{fib}}) \cup Norm_1(G_{\texttt{fib}}) \\
 & = & \{\bullet\} \cup (Norm_1(G_{\texttt{fib}}) \times Norm_1(G_{\texttt{fib}})) \cup Norm_1(G_{\texttt{fib}}) \quad = \quad \{\bullet, \bullet \otimes \bullet\} \\
Norm_3(G_{\texttt{fib}}) & = & Norm_2(\bullet \vee G_{\texttt{fib}} \otimes G_{\texttt{fib}}) \cup Norm_2(G_{\texttt{fib}}) \\
 & = & Norm_2(G_{\texttt{fib}}) \times Norm_2(G_{\texttt{fib}}) \cup Norm_2(G_{\texttt{fib}}) \\
 & = & \{\bullet, \bullet \otimes \bullet\} \times \{\bullet, \bullet \otimes \bullet\} \cup \{\bullet, \bullet \otimes \bullet\} \\
 & = & \{\bullet, \bullet \otimes \bullet, \bullet \otimes (\bullet \otimes \bullet), (\bullet \otimes \bullet) \otimes \bullet, (\bullet \otimes \bullet) \otimes (\bullet \otimes \bullet)\}
\end{array}$$

The above set contains the three graphs displayed in Figure 5, as well as others (e.g., $(\bullet\otimes\bullet)\otimes(\bullet\otimes\bullet)$) that are possible according to the graph type but will not arise at runtime.

3.3 Language and Graph Type System

Figure 8 presents the syntax of λ^G_{FJ}. The calculus is essentially a simply-typed lambda calculus with a base type of unit (with value $\langle\rangle$), functions, pairs and sum types Function abstractions fun $f\ x = e$ are explicitly named, rather than anonymous, and recursive by default. We will use a type-and-effect system to assign both standard types and graph types to expressions, and so, as in most presentations of effect systems, we annotate function types with the "latent effect" of the function, in this case, a graph type corresponding to the parallelism of the function body itself. For example, if we extend λ^G_{FJ} with integers, we would express the type of the Fibonacci program described above as $\mathsf{int} \xrightarrow{\mu\gamma.(\bullet\vee(\gamma\otimes\gamma))} \mathsf{int}$, indicating that the function accepts and returns an integer, and results in a graph described by the graph type written over the arrow.

The typing judgment for the type system of λ^G_{FJ} is $\Gamma; \Delta \vdash e : \tau \mid G$. The typing judgment uses two contexts: Γ, as usual, maps variables to their types. The second context, Δ, lists graph type variables γ. The judgment indicates that the expression e has type τ and executing it will produce a graph corresponding to G.

(S:Var)
$$\frac{}{\Gamma, x : \tau; \Delta \vdash x : \tau \mid \bullet}$$

(S:Unit)
$$\frac{}{\Gamma; \Delta \vdash \langle\rangle : \mathsf{unit} \mid \bullet}$$

(S:Fun)
$$\frac{\Gamma, f : \tau_1 \xrightarrow{\gamma} \tau_2, x : \tau_1; \Delta, \gamma \vdash e : \tau_2 \mid G \qquad \gamma \text{ fresh}}{\Gamma; \Delta \vdash \mathsf{fun}\ f\ x = e : \tau_1 \xrightarrow{\mu\gamma.G} \tau_2 \mid \bullet}$$

(S:App)
$$\frac{\Gamma; \Delta \vdash e_1 : \tau_1 \xrightarrow{G_3} \tau_2 \mid G_1 \qquad \Gamma; \Delta \vdash e_2 : \tau_1 \mid G_2}{\Gamma; \Delta \vdash e_1\ e_2 : \tau_2 \mid G_1 \oplus G_2 \oplus G_3}$$

(S:Par)
$$\frac{\Gamma; \Delta \vdash e_1 : \tau_1 \mid G_1 \qquad \Gamma; \Delta \vdash e_2 : \tau_2 \mid G_2}{\Gamma; \Delta \vdash \mathsf{par}(e_1, e_2) : \tau_1 \times \tau_2 \mid G_1 \otimes G_2}$$

(S:Case)
$$\frac{\Gamma; \Delta \vdash e_1 : \tau_1 + \tau_2 \mid G_1 \qquad \Gamma, x : \tau_1; \Delta \vdash e_2 : \tau' \mid G_2 \qquad \Gamma, y : \tau_2; \Delta \vdash e_3 : \tau' \mid G_3}{\Gamma; \Delta \vdash \mathsf{case}\ e_1\ \{x.e_2; y.e_3\} : \tau' \mid G_1 \oplus (G_2 \vee G_3)}$$

Fig. 7. Selected rules for the graph type system of λ^G (fork-join subset only).

$$\begin{array}{lcl}
\tau & ::= & \mathsf{unit} \mid \tau \xrightarrow{G} \tau \mid \tau \times \tau \mid \tau + \tau \\
e & ::= & x \mid \langle\rangle \mid \mathsf{fun}\ f\ x = e \mid e\ e \mid (e, e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{par}(e, e) \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e\ \{x.e; y.e\}
\end{array}$$

Fig. 8. Syntax for λ^G_{FJ}.

The interesting rules for this judgment appear in Figure 7. Variables and unit values result in singleton cost graphs, and have the usual typing properties. When typing a function fun $f\ x = e$, because functions are recursive, we must add f to the context with an appropriate type, which must also include a graph type. A recursive function will yield a recursive graph type, so we assume the form $\mu\gamma.G$ for the graph type of the function. Recursive instances of f within the body then yield recursive instances of the function graph, so we assign f the graph type γ. We also add γ to the graph type variable context and, as usual, the argument x to the context with its type τ_1. If, under this context, the body of the function has the result type τ_2 and yields a graph of graph type G, we conclude the final type $\tau_1 \xrightarrow{\mu\gamma.G} \tau_2$. Functions are values, and so *defining* a function yields only a singleton graph $\bullet$.

The graph contained in a function type only comes into play when the function is applied. At an application $e_1\ e_2$, the function e_1 is required to have a function type $\tau_1 \xrightarrow{G_3} \tau_2$. Because e_1 might not be a function literal or variable, we also account for the fact that evaluating e_1 might itself result in a graph G_1. The argument e_2 must have the argument type τ_1; evaluating it results in a graph G_2. The result has type τ_2. The graph type of the application sequences together the graphs corresponding to evaluating e_1, evaluating e_2, and evaluating the function.

An expression par(e_1, e_2) has a type which is the pair of the types of the two subexpressions; as the two expressions are evaluated in parallel, the resulting graph is the parallel composition of the two subgraphs. Finally, in a pattern match case $e_1\ \{x.e_2; y.e_3\}$, we first evaluate the expression e_1, which must have a sum type. The two branches must have the same type τ', which is also the result type of the entire case expression. After evaluating e_1 and producing a graph corresponding to G_1, we will execute only one of the two branches (either e_2 with graph G_2 or e_3 with G_3). This is indicated with the graph type $G_2 \vee G_3$.

$$\text{(C:App)}\ \frac{e_1 \Downarrow \mathsf{fun}\ f\ x = e \mid g_1 \qquad e_2 \Downarrow v \mid g_2 \qquad e[v/x][\mathsf{fun}\ f\ x = e/f] \Downarrow v' \mid g'}{e_1\ e_2 \Downarrow v' \mid g_1 \oplus g_2 \oplus g'} \qquad \text{(C:Par)}\ \frac{e_1 \Downarrow v_1 \mid g_1 \qquad e_2 \Downarrow v_2 \mid g_2}{\mathsf{par}(e_1, e_2) \Downarrow (v_1, v_2) \mid g_1 \otimes g_2}$$

$$\text{(C:CaseL)}\ \frac{e_1 \Downarrow \mathsf{inl}\ v \mid g_1 \qquad e_2[v/x] \Downarrow v' \mid g_2}{\mathsf{case}\ e_1\ \{x.e_2; y.e_3\} \Downarrow v' \mid g_1 \oplus g_2} \qquad \text{(C:CaseR)}\ \frac{e_1 \Downarrow \mathsf{inr}\ v \mid g_1 \qquad e_3[v/y] \Downarrow v' \mid g_2}{\mathsf{case}\ e_1\ \{x.e_2; y.e_3\} \Downarrow v' \mid g_1 \oplus g_2}$$

Fig. 9. Selected cost semantics rules for the fork-join subset of λ^G.

3.4 Cost Semantics and Soundness

We validate the graph type system by defining a *cost semantics* for λ^G_{FJ}, which evaluates the program to produce both a value and a graph corresponding to the execution. The graph type system is sound if the produced graph is always in the normalization of the predicted graph type. The cost semantics judgment $e \Downarrow v \mid g$ states that e evaluates to value v and produces graph g. Selected rules for the cost semantics are shown in Figure 9. The other rules are straightforward and/or result in singleton cost graphs. Evaluating an application $e_1\ e_2$ evaluates e_1 to a function value and e_2 to a value, then substitutes the argument for the formal parameter and the function itself for f, and composes the three resulting cost graphs. Evaluating a par expression produces a pair of the resulting values and the parallel composition of the two graphs. When evaluating a case expression, we evaluate e_1 to either $\mathsf{inl}\ v$ or $\mathsf{inr}\ v$, at which point the corresponding rule substitutes v into the correct branch and continues evaluating to obtain the final value. Only the selected branch is evaluated and contributes to the final cost graph, which is sequentially composed with the graph that resulted from evaluation of e_1.

Theorem 1 is the key soundness result for the graph type system of λ^G_{FJ}. It states that if an expression e has type τ and graph type G in an empty context and evaluates to value v producing graph g, then g is properly represented by G (that is, it is in the normalization of G for some index n). The theorem also encompasses a more standard preservation result, that the type of the final result v is τ. Because v is a value, its associated graph is trivial.

Theorem 1. *If* $\cdot; \cdot \vdash e : \tau \mid G$ *and* $e \Downarrow v \mid g$, *then* $\cdot; \cdot \vdash v : \tau \mid \bullet$ *and* $g \in Norm_n(G)$ *for some* $n \in \mathbb{N}$.

This result is encompassed by the equivalent theorem presented later along with the full development, so we will not give the proof here.

4 GRAPH TYPES

This section extends the definition and formalisms of graph types to futures. As in the previous section, we will motivate the additional features using example programs. Recall the future-based implementation of the Fibonacci function from Section 2. Using the notation for graph types introduced above, and reusing the notations for sequential composition, left composition and edge addition, we might be tempted to write the graph type for the future-based Fibonacci program as:

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

However, this definition has an important problem: if we were to unroll the recursion to generate sets of actual graphs, the same vertex u would be used in each recursive instance, which would violate the requirement that vertices in a graph be unique. To solve this problem, we introduce the

```
let rec pipeline (f, fut, l) =
  match l with
  | [] -> touch fut
  | h::t ->
     let fprefix = future (f h + (touch fut)) in
     pipeline (f, fprefix, t)

```

Fig. 10. Code for the pipeline program.

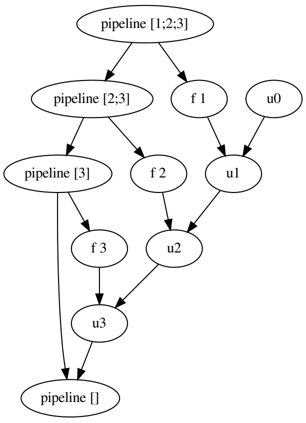


Fig. 11. Graph for the pipeline program with $l = [1; 2; 3]$.

notation $\nu u.G$; this binds the new vertex name u, which may be freely renamed by the standard rules of alpha conversion (the notation goes with the convenient mnemonic "new"). Like other binding constructs, this also induces a notion of scope: the vertex u is not bound, and may not be used, outside G. When unrolling the type definition to give a standard graph during normalization, we will instantiate u with a fresh vertex name, leading to the desired behavior. A correct graph type for the Fibonacci program is then:

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

As another example, consider the program in Figure 10, which takes a function f and a list of integers $l_0 \dots l_n$ and computes $f(l_0) + \cdots + f(l_n)$ in a pipelined fashion: all of the $f(l_i)$ are computed in parallel, and then the results are added together. The function takes a future as an additional argument: in the (recursive) call `pipeline (f, fut,` $l_i \dots l_n$`)`, `fut` is a future that is computing $f(l_0) + \cdots + f(l_{i-1})$ (that is, the sum of the prefix of the desired list). A top-level call of the function would simply pass in `future 0`.

If the list is empty, the function simply touches the provided future (which will, by construction, return the desired sum). Otherwise, it spawns a new future that, in parallel with the rest of the computation, applies `f` to the head of the list before demanding the sum of the prefix and adding them together. The handle to this future is then (immediately, without blocking) passed to the recursive invocation of `pipeline` on the rest of the list. When called on a list, the function runs down the entire list and generates n futures in a tight loop. Each future computes f on an element on the list before touching the previous future. The graph for `pipeline (f, fut, [1; 2; 3])` is shown in Figure 11, where `fut` is a future with sink vertex u_0 (strictly speaking, this is not valid as a complete program graph because u_0 is never spawned and is therefore an additional source). Representing this graph as a graph type requires an additional feature: each recursive instance of the graph must take as an "argument" the vertex of the future it should touch (for example, the outermost instance of the graph must know to touch u_0, the next instance must know to touch u_1, and so on). We therefore allow graph types to abstract over vertices: the graph type $\Pi\vec{u}_f; \vec{u}_t.G$ represents a graph that abstracts over two sequences of vertices, $\vec{u}_f$ and $\vec{u}_t$, and may spawn futures using the vertices in $\vec{u}_f$ and touch the vertices in $\vec{u}_t$ (the two sequences are permitted to overlap and, indeed, $\vec{u}_f$ is generally a subset of $\vec{u}_t$; the reasoning for keeping them separate will become

$$G \quad ::= \quad \gamma \mid \bullet \mid G \oplus G \mid G \otimes G \mid G \vee G \mid \mu\gamma.G \mid G \swarrow_u \mid {}^u\searrow \mid \nu u.G \mid \Pi\vec{u};\vec{u}.G \mid G[\vec{u};\vec{u}]$$

Fig. 12. Full syntax for graph types.

$$\begin{array}{lcll}
Norm_n(G \swarrow_u) & \triangleq & \{G' \swarrow_u \mid G' \in Norm_n(G)\} & \\
Norm_n({}^u\searrow) & \triangleq & \{{}^u\searrow\} & \\
Norm_n(\nu u.G) & \triangleq & Norm_n(G[u'/u]) & u' \text{ fresh} \\
Norm_n(G[\vec{u}_f;\vec{u}_t]) & \triangleq & Norm_{n-k}(G'[\vec{u}_f/\vec{u}'_f][\vec{u}_t/\vec{u}'_t]) & \mathsf{unroll}_k(G) = \Pi\vec{u}'_f;\vec{u}'_t.G' \\
Norm_n(G[\vec{u}_f;\vec{u}_t]) & \triangleq & \bullet & \mathsf{unroll}_n(G) \neq \Pi\vec{u}'_f;\vec{u}'_t.G'
\end{array}$$

Fig. 13. Additional cases of normalization for futures.

clear in the next section). The graph type $G[\vec{u}'_f;\vec{u}'_t]$ applies G to two sequences of vertices. We treat $(\Pi\vec{u}_f;\vec{u}_t.G)[\vec{u}'_f;\vec{u}'_t]$ as equal to $G[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t]$ (that is, G with the vertices $\vec{u}'_f$ substituted pointwise for $\vec{u}_f$ and similar for $\vec{u}'_t$ and $\vec{u}_t$). The graph type for the pipeline program can be written

$$G_{\texttt{pipeline}} \triangleq \mu\gamma.\ \Pi\emptyset;u.\ \nu u'.\ ({}^u\searrow \vee ((G_f \oplus {}^u\searrow) \swarrow_{u'} \oplus \gamma[\emptyset;u']))$$

where $\emptyset$ is an empty sequence of vertices. The graph type indicates that each recursive instance takes a vertex u, binds a new vertex u', and then touches u either immediately or after spawning a future using u' to perform f (whose graph is represented as G_f). In the recursive case, the recursive instance of the graph, γ, is passed the vertex corresponding to the new future, u'.

4.1 Full Graph Type Syntax and Normalization

Figure 12 shows the full syntax for graph types, with the additions motivated above. Figure 13 gives the additional cases of the definition of normalization. Normalizing graph types of the forms $G \swarrow_u$ and ${}^u\searrow$, which borrow syntax from their graph notation counterparts, is straightforward. Normalizing a new vertex binding $\nu u.G$ generates a fresh vertex and substitutes it for the bound vertex. Finally, to normalize an application $G[\vec{u}_f;\vec{u}_t]$, we first unroll any recursive bindings in G using the notation $\mathsf{unroll}_k(G)$:

$$\begin{array}{lcll}
\mathsf{unroll}_{n+1}(\mu\gamma.G) & \triangleq & \mathsf{unroll}_n(G[\mu\gamma.G/\gamma]) & \\
\mathsf{unroll}_n(G) & \triangleq & G & n = 0 \vee G \neq \mu\gamma.G'
\end{array}$$

Unrolling is performed until a Π graph type is exposed, at which point the β-reduction is performed by substituting the vertex sequences pointwise. If we had to unroll k recursive bindings, we only continue to normalize the remaining graph $n-k$ additional steps. Graph types that have a Π binding as their head cannot be normalized, but we will never need to normalize such a graph type, as they will always be applied eventually.

Lemma 1 lists several properties of the normalization procedure that will be useful in the remainder of the paper. The first three allow us to move more easily between graph types and graphs contained in their normalizations. The fourth states that normalization is preserved by substitution. The fifth shows that normalization is monotonic: normalizing a graph type for additional steps only adds graphs to the set.

Lemma 1.

(1) *If* $g_1 \in Norm_n(G_1)$ *and* $g_2 \in Norm_n(G_2)$, *then* $g_1 \oplus g_2 \in Norm_n(G_1 \oplus G_2)$.
(2) *If* $g_1 \in Norm_n(G_1)$ *and* $g_2 \in Norm_n(G_2)$, *then* $g_1 \otimes g_2 \in Norm_n(G_1 \otimes G_2)$.

$$\text{(DW:Empty)}\ \frac{}{\Delta;\Omega;\Psi \vdash \bullet : *} \qquad \text{(DW:Var)}\ \frac{}{\Delta,\gamma:\kappa;\Omega;\Psi \vdash \gamma : \kappa} \qquad \text{(DW:Seq)}\ \frac{\Delta;\Omega_1;\Psi \vdash G_1 : * \qquad \Delta;\Omega_2;\Psi \vdash G_2 : *}{\Delta;\Omega_1,\Omega_2;\Psi \vdash G_1 \oplus G_2 : *}$$

$$\text{(DW:Par)}\ \frac{\Delta;\Omega_1;\Psi \vdash G_1 : * \qquad \Delta;\Omega_2;\Psi \vdash G_2 : *}{\Delta;\Omega_1,\Omega_2;\Psi \vdash G_1 \otimes G_2 : *} \qquad \text{(DW:Or)}\ \frac{\Delta;\Omega;\Psi \vdash G_1 : * \qquad \Delta;\Omega;\Psi \vdash G_2 : *}{\Delta;\Omega;\Psi \vdash G_1 \vee G_2 : *}$$

$$\text{(DW:RecPi)}\ \frac{\Delta,\gamma:\Pi\vec{u}_f;\vec{u}_t.*;\vec{u}_f;\Psi,\vec{u}_t \vdash G : *}{\Delta;\Omega;\Psi \vdash \mu\gamma.\Pi\vec{u}_f;\vec{u}_t.G : \Pi\vec{u}_f;\vec{u}_t.*} \qquad \text{(DW:Spawn)}\ \frac{\Delta;\Omega;\Psi \vdash G : *}{\Delta;\Omega,u;\Psi \vdash G \swarrow_u : *} \qquad \text{(DW:Touch)}\ \frac{}{\Delta;\Omega;\Psi,u \vdash {}^{u}\searrow : *}$$

$$\text{(DW:New)}\ \frac{\Delta;\Omega,u;\Psi,u \vdash G : * \qquad u \notin \Omega,\Psi}{\Delta;\Omega;\Psi \vdash \nu u.G : *} \qquad \text{(DW:Pi)}\ \frac{\Delta;\Omega,\vec{u}_f;\Psi,\vec{u}_t \vdash G : *}{\Delta;\Omega;\Psi \vdash \Pi\vec{u}_f;\vec{u}_t.G : \Pi\vec{u}_f;\vec{u}_t.*} \qquad \text{(DW:App)}\ \frac{\Delta;\Omega;\Psi,\vec{u}'_t \vdash G : \Pi\vec{u}_f;\vec{u}_t.\kappa}{\Delta;\Omega,\vec{u}'_f;\Psi,\vec{u}'_t \vdash G[\vec{u}'_f;\vec{u}'_t] : \kappa}$$

Fig. 14. Rules for graph type formation.

(3) *If $g \in Norm_n(G)$ then $g \swarrow_u \in Norm_n(G \swarrow_u)$.*
(4) *If $g \in Norm_n(G)$ then $g[u'/u] \in Norm_n(G[u'/u])$.*
(5) *If $g \in Norm_n(G)$ and $m \geq n$, then $g \in Norm_m(G)$.*

Cases (1)-(3) are immediate from the definition of normalization. Case (4) is a straightforward structural induction on G. Case (5) is a lexicographic induction on G and m: the key insight is that $Norm_{m-1}(\mu\gamma.G)$ is contained in $Norm_m(\mu\gamma.G)$.

4.2 Graph Type Formation

Adding explicit vertex names to the graph type syntax now opens the possibility of syntactically valid but problematic graph types. For example, we would want to disallow the graph type $\bullet \swarrow_u \oplus \bullet \swarrow_u \oplus {}^{u}\searrow$ because its normalization (which, due to the reuse of syntax, is exactly $\{\bullet \swarrow_u \oplus \bullet \swarrow_u \oplus {}^{u}\searrow\}$) contains an ill-formed graph: the vertex u is used twice and it is therefore impossible to determine which instance of u should be the source of the edge added by the touch ${}^{u}\searrow$. To avoid this, we ensure that vertex names are unique when producing graph types from expressions, and declare invalid any graph types with duplicate vertex names.

The judgment $\Delta;\Omega;\Psi \vdash G : \kappa$ indicates that a graph type is semantically valid. As before, Δ contains bound graph type variables. In addition, it now maps these variables to the *kind* of the graph. Graph kinds indicate whether a graph type is a Π, and if so, its arities. The graph kind $*$ indicates a graph type that does not have a Π at the outermost level.

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

In order to restrict the use of vertex names, we assume that vertex names are drawn from a global "pool" of vertex names (although new vertex names can be added by Π and ν bindings). We use two contexts to store these "available" vertex names. The context Ω includes the vertices that are available to be used in spawns $G \swarrow_u$. In order to ensure that each vertex name is attached to only one future, this context is affine—we permit weakening (additional vertex names can be added to Ω and not used to spawn futures) but not contraction. The context Ψ includes vertices that may be used in touches ${}^{u}\searrow$ and is unrestricted, as vertices can be touched any number of times.

The rules for the graph type formation judgment are presented in Figure 14. Most of the rules are relatively straightforward. In rules DW:Seq and DW:Par, we explicitly split the affine context Ω so that vertex names may be used in spawns in at most one of the two subgraphs. It is *not* necessary to split the context in rule DW:Or: because only one of the two subgraphs will actually appear in a normalization of this graph type, it is safe to reuse spawned vertices between the two. Indeed, it is sometimes necessary to do so: consider the expression `touch (if e then future e1 else future e2)`. We can assign this expression the graph type $(G_1 \swarrow_u \vee G_2 \swarrow_u) \oplus {}^u\searrow$ whereas if we had to assign the two branches different vertices, we wouldn't be able to assign a single graph type to the touch.

For simplicity, we assume that all recursive $\mu\gamma.G$ bindings immediately contain a nested $\Pi\vec{u}_f;\vec{u}_t.G'$ (this does not cause any loss of generality, as $\vec{u}_f$ and $\vec{u}_t$ may both be $\emptyset$) and that G' is then not immediately a Π or μ. All of the graph types used in our graph type system naturally adhere to this form, so this restriction is not burdensome. Under these assumptions, rule DW:RecPi adds the variable γ with the appropriate Π kind to the context Δ and adds the vertex sequence arguments to the appropriate contexts. The graph G may capture names from Ψ but not Ω, as this might allow them to be duplicated. Vertex bindings $\Pi\vec{u}_f;\vec{u}_t.G$ are also permitted to appear on their own without recursion, provided G has kind $*$; these are handled by DW:Pi.

Rules DW:Spawn and DW:Touch require the named vertex to be in the appropriate context. Rule DW:New adds the new vertex to both contexts provided it is not currently in either context (which can be ensured through α-conversion). Finally, DW:App requires the vertex arguments to be present in the appropriate contexts, and removes the vertices $\vec{u}'_f$ from the affine context.

We now show formally that well-formed graph types (according to the rules of Figure 14) can be normalized into sets of meaningful graphs. We refer to this notion as "normalizability" and define it formally below. Recall that graphs of kind $\Pi\vec{u}_f;\vec{u}_t.\kappa$ cannot be normalized directly, so we define such a graph to be normalizable if when unrolled and applied to appropriate arguments, the result can be normalized. This completes the induction hypothesis for the result we show below.

Definition 1. A graph $G : \kappa$ is *normalizable* at n if

- $\kappa = *$ and for all $g \in Norm_n(G)$, the graph g is well-defined.
- $\kappa = \Pi\vec{u}_f;\vec{u}_t.*$ and $\mathsf{unroll}_k(G) = \Pi\vec{u}_f;\vec{u}_t.G'$ for some k, and for all $\vec{u}'_f, \vec{u}'_t$ of appropriate arities, we have that $G'[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t]$ is normalizable at n.

The proof that well-formed graph types are normalizable relies on the fact, stated in Lemma 2 and proven in the appendix, that any vertices generated by normalization are fresh; otherwise vertices in normalized graphs are "free" in the graph type. We define $\mathsf{FF}(G)$ to be the vertex names used in spawns ($G' \swarrow_u$) inside G which are not bound in G. We also use the shorthand $Verts(g) = V$ when $g = (V, E, s, t)$.

Lemma 2. *For all G and all $n \geq 0$ and all $g \in Norm_n(G)$, $Verts(g) \subset \mathsf{FF}(G) \cup V$ where V is a set of freshly-chosen vertices.*

Theorem 2. *For all $n \geq 0$,*

(1) *If $\cdot;\Omega;\Psi \vdash G : \kappa$ then $\mathsf{FF}(G) \subset \Omega$ and G is normalizable at n.*
(2) *If $\gamma;\Omega;\Psi \vdash G : \kappa$ then $\mathsf{FF}(G) \subset \Omega$ and $G[\mu\gamma.\Pi\vec{u}_f;\vec{u}_t.G/\gamma]$ is normalizable at n.*

The proof is by lexicographic induction on n and the appropriate type formation derivation. The induction hypothesis on n is used in the DW:Var case, where we must reason that $\mu\gamma.\Pi\vec{u}_f;\vec{u}_t.G$ is normalizable. The full proof is in the appendix.

$$\begin{array}{llll}
\textit{Types} & \tau & ::= & \mathsf{unit} \mid \Pi\vec{u};\vec{u}.\tau \xrightarrow{G} \tau \mid \tau \times \tau \mid \tau + \tau \mid \tau\ \mathsf{future}[u] \\
\textit{Expressions} & e & ::= & x \mid \langle\rangle \mid \mathsf{fun}[\vec{u};\vec{u}]\ f\ x = e \mid e[\vec{u};\vec{u}]\ e \mid (e,e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{par}(e,e) \mid \\
 & & & \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e\ \{x.e; y.e\} \mid \mathsf{future}[u]\ e \mid \mathsf{touch}\ e \mid \mathsf{new}\ u.e
\end{array}$$

Fig. 15. Syntax of λ^G.

5 A GRAPH-ANNOTATED CORE CALCULUS

In this section, we present the full language λ^G and its graph type system, which we use to infer graph types for programs. To aid in the computation of graph types, we annotate λ^G programs with vertex names for futures: spawns of futures are annotated with the vertex to be used to spawn this future, and the type of futures, τ future$[u]$, is similarly annotated with the sink vertex of the future. New vertex names can be bound using a "new" construct analogous to the $\nu u.G$ binding for graph types. Additional annotations on function definitions and applications allow for a form of future-polymorphism, which will be explained below. Note that we *do not* expect these annotations to be inserted by programmers: Section 6 will discuss how these annotations can be automatically inferred and inserted into unannotated source programs. The problem of inferring these annotations is largely orthogonal to finding a graph type from an annotated program, so we present the graph type system over the annotated language out of a separation of concerns.

5.1 Syntax and Type Formation

The full syntax of λ^G is shown in Figure 15. Types are the same as for λ^G_{FJ}, with the addition of a type of futures. The type of functions has also been modified to account for a form of polymorphism which will be explained below. Expressions include the expressions of λ^G_{FJ}, as well as constructs to spawn and touch futures, and the expression new $u.e$, which binds the new vertex name u in e.

Functions are now polymorphic over the sets of vertices that the body of the function may spawn or touch: the function fun$[\vec{u}_f;\vec{u}_t]\ f\ x = e$ may spawn futures using vertices in $\vec{u}_f$ and may touch futures using vertices in $\vec{u}_t$. This allows functions to accept and return values of future type. For example, consider the function mytouch, which essentially replicates the functionality of touch.

$$\mathsf{fun\ mytouch}\ f = \mathsf{touch}\ f$$

We could assign the type mytouch : u future$[\tau] \xrightarrow{\overset{u}{\searrow}}$ int. However, this is not general enough: it only allows `mytouch` to be applied to the future spawned using vertex u! Instead, we make `mytouch` polymorphic in the vertex u and assign it the type

$$\mathsf{mytouch} : \Pi\emptyset; u.u\ \mathsf{future}[\tau] \xrightarrow{\Pi\emptyset;u.\ \overset{u}{\searrow}} \mathsf{int}$$

Note that the Π binder appears in both the function type and the graph type. We would annotate the definition of mytouch accordingly, with u as a parameter (similar to a type parameter in some presentations of parametric polymorphism):

$$\mathsf{fun}[\emptyset; u]\ \mathsf{mytouch}\ f = \mathsf{touch}\ f$$

If we wish to apply mytouch to the future expression future$[u']\ e : \tau$ future$[u']$, we would do so by instantiating the parameter u to u' using the syntax

$$\mathsf{mytouch}[\emptyset; u']\ (\mathsf{future}[u']\ e)$$

(S:Fun)

$$\frac{\Gamma, f : \Pi\vec{u}_f;\vec{u}_t.\tau_1 \xrightarrow{\gamma} \tau_2, x : \tau_1; \Delta, \gamma : \Pi\vec{u}_f;\vec{u}_t.*;\vec{u}_f;\Psi,\vec{u}_t \vdash e : \tau_2 \mid G \qquad \gamma \text{ fresh} \qquad \Delta;\Psi,\vec{u}_t \vdash \tau_1 \text{ ok} \qquad \Delta;\Psi,\vec{u}_t \vdash \tau_2 \text{ ok}}{\Gamma;\Delta;\Omega;\Psi \vdash \mathsf{fun}[\vec{u}_f;\vec{u}_t]\ f\ x = e : \Pi\vec{u}_f;\vec{u}_t.\tau_1 \xrightarrow{\mu\gamma.\Pi\vec{u}_f;\vec{u}_t.G} \tau_2 \mid \bullet}$$

(S:App)

$$\frac{\Gamma;\Delta;\Omega_1;\Psi,\vec{u}_t' \vdash e_1 : \Pi\vec{u}_f;\vec{u}_t.\tau_1 \xrightarrow{G_3} \tau_2 \mid G_1 \qquad \Gamma;\Delta;\Omega_2;\Psi,\vec{u}_t' \vdash e_2 : \tau_1[\vec{u}_t'/\vec{u}_t] \mid G_2}{\Gamma;\Delta;\Omega_1,\Omega_2,\vec{u}_f';\Psi,\vec{u}_t' \vdash e_1[\vec{u}_f';\vec{u}_t']\ e_2 : \tau_2[\vec{u}_t'/\vec{u}_t] \mid G_1 \oplus G_2 \oplus \mathsf{unroll}(G_3)[\vec{u}_f';\vec{u}_t']}$$

(S:Pair)

$$\frac{\Gamma;\Delta;\Omega_1;\Psi \vdash e_1 : \tau_1 \mid G_1 \qquad \Gamma;\Delta;\Omega_2;\Psi \vdash e_2 : \tau_2 \mid G_2}{\Gamma;\Delta;\Omega_1,\Omega_2;\Psi \vdash (e_1,e_2) : \tau_1 \times \tau_2 \mid G_1 \oplus G_2}$$

(S:Fst)

$$\frac{\Gamma;\Delta;\Omega;\Psi \vdash e : \tau_1 \times \tau_2 \mid G}{\Gamma;\Delta;\Omega;\Psi \vdash \mathsf{fst}\ e : \tau_1 \mid G}$$

(S:Snd)

$$\frac{\Gamma;\Delta;\Omega;\Psi \vdash e : \tau_1 \times \tau_2 \mid G}{\Gamma;\Delta;\Omega;\Psi \vdash \mathsf{snd}\ e : \tau_2 \mid G}$$

(S:InL)

$$\frac{\Gamma;\Delta;\Omega;\Psi \vdash e : \tau_1 \mid G}{\Gamma;\Delta;\Omega;\Psi \vdash \mathsf{inl}\ e : \tau_1 + \tau_2 \mid G}$$

(S:InR)

$$\frac{\Gamma;\Delta;\Omega;\Psi \vdash e : \tau_2 \mid G}{\Gamma;\Delta;\Omega;\Psi \vdash \mathsf{inr}\ e : \tau_1 + \tau_2 \mid G}$$

(S:Case)

$$\frac{\Gamma;\Delta;\Omega_1;\Psi \vdash e_1 : \tau_1 + \tau_2 \mid G_1 \qquad \Gamma, x : \tau_1;\Delta;\Omega_2;\Psi \vdash e_2 : \tau' \mid G_2 \qquad \Gamma, y : \tau_2;\Delta;\Omega_2;\Psi \vdash e_3 : \tau' \mid G_3}{\Gamma;\Delta;\Omega_1,\Omega_2;\Psi \vdash \mathsf{case}\ e_1\ \{x.e_2; y.e_3\} : \tau' \mid G_1 \oplus (G_2 \vee G_3)}$$

(S:Future)

$$\frac{\Gamma;\Delta;\Omega;\Psi \vdash e : \tau \mid G}{\Gamma;\Delta;\Omega,u;\Psi,u \vdash \mathsf{future}[u]\ e : \tau\ \mathsf{future}[u] \mid G \swarrow_u}$$

(S:Touch)

$$\frac{\Gamma;\Delta;\Omega;\Psi,u \vdash e : \tau\ \mathsf{future}[u] \mid G}{\Gamma;\Delta;\Omega;\Psi,u \vdash \mathsf{touch}\ e : \tau \mid G \oplus {}^{u}\searrow}$$

(S:New)

$$\frac{\Gamma;\Delta;\Omega,u;\Psi,u \vdash e : \tau \mid G \qquad u \notin \mathsf{FV}(\tau) \qquad u \notin \Omega,\Psi}{\Gamma;\Delta;\Omega;\Psi \vdash \mathsf{new}\ u.e : \tau \mid \nu u.G}$$

Fig. 16. Graph type system for λ^G.

As we did for graph types, we will now present rules defining what it means for a type to be well-formed. The judgment is $\Delta;\Psi \vdash \tau$ ok. Note that this judgment does not include the affine context of vertices "available" for spawns: types are fundamentally predictions about the *value* of an expression, and values do not spawn futures (future handles are themselves values, however, and so types must include the vertex names for already-spawned futures). The full rules for the judgment are included in the supplementary material. The interesting rule is the one for function types, which requires that the graph on the arrow is well-formed:

$$\frac{\Delta;\Psi,\vec{u}_t \vdash \tau_1 \text{ ok} \qquad \Delta;\Psi,\vec{u}_t \vdash \tau_2 \text{ ok} \qquad \Delta;\cdot;\Psi \vdash G : \Pi\vec{u}_f;\vec{u}_t.*}{\Delta;\Psi \vdash \Pi\vec{u}_f;\vec{u}_t.\tau_1 \xrightarrow{G} \tau_2 \text{ ok}}$$

5.2 Graph Type System

We are now ready to present the rules for the graph type system, which produces types and graph types from annotated λ^G expression. The rules are shown in Figure 16. The judgment now includes the vertex contexts Ω and Ψ but is otherwise the same as the graph typing judgment for the fork-join language. The most interesting rules are S:Fun and S:App. For a function $\mathsf{fun}[\vec{u}_f; \vec{u}_t]\ f\ x = e$, we will infer a type $\Pi\vec{u}_f; \vec{u}_t.\tau_1 \xrightarrow{\mu\gamma.\Pi\vec{u}_f;\vec{u}_t.G} \tau_2$. As before, we add f to the context with a type of this shape, but using the recursive instance γ instead of the full graph. We also add γ to the context, with its kind, $\Pi\vec{u}_f; \vec{u}_t.*$. Placing the Π binding inside the μ binding like this and thus assigning γ a Π kind allows for vertex-polymorphic recursion, which is crucial. Recall the `pipeline` example above. Without polymorphic recursion, each recursive instance of `pipeline` would need to be passed the same future. The body of the function is analyzed using only $\vec{u}_f$ as the affine context; vertices from the context outside the function cannot be captured by the function body, as using the function in multiple places would then allow these vertices to be duplicated. The outer context Ψ, however, may be used freely inside the function body in addition to vertices bound in $\vec{u}_t$. Practically speaking, this means that any vertex spawned by a function body must be either passed as a parameter to the function or bound by a new inside the body. If the vertex needs to be available outside the function body (e.g., if the function returns the future), it must be passed as a parameter.

Applications $e_1\,[\vec{u}'_f; \vec{u}'_t]\ e_2$ now include pairs of vertex sequences as parameters. The affine context must include $\vec{u}'_f$; the remainder of the affine context is split between e_1 and e_2. The unrestricted vertex context must include $\vec{u}'_t$, but the entirety of this context, including $\vec{u}'_t$, is available to both e_1 and e_2. We perform the appropriate substitutions on both the argument and return types. Finally, we must instantiate the graph of the function body G_3, with the appropriate vertices (from rule S:Fun, we know that G_3 will be of Π kind). We first unroll G_3 to try to expose a Π binding and then perform the instantiation. If $G_3 = \gamma$, this will result in the graph type $\gamma[\vec{u}'_f; \vec{u}'_t]$, while if it is actually a Π (under a recursive binding), we will perform the appropriate β-reduction.

Rules S:Future and S:Touch simply require the vertices to be in the appropriate contexts. The rule for new $u.e$ adds u to both contexts. Additionally, it requires that u is does not appear free in the type of e. This restriction prevents a spawned future from escaping the scope of its vertex.

Lemma 3 presents a number of results showing that typing or formation are preserved under substitution: values may be substituted into expressions (part 1), graphs may be substituted into graphs (part 2), types (3) and the typing of expressions (4), and vertices may be substituted into graphs (5), types (6) and expressions (7).

Lemma 3.

(1) *If* $\Gamma, x : \tau'; \Delta; \Omega; \Psi \vdash e : \tau \mid G$ *and* $\cdot; \cdot; \Omega; \Psi \vdash v : \tau' \mid \bullet$ *then* $\Gamma; \Delta; \Omega; \Psi \vdash e[v/x] : \tau \mid G$.
(2) *If* $\Delta, \gamma : \kappa'; \Omega; \Psi \vdash G : \kappa$ *and* $\Delta; \Omega; \Psi \vdash G' : \kappa'$ *then* $\Delta; \Omega; \Psi \vdash G[G'/\gamma] : \kappa$.
(3) *If* $\Delta, \gamma : \kappa; \Psi \vdash \tau$ ok *and* $\Delta; \Omega; \Psi \vdash G : \kappa$ *then* $\Delta; \Psi \vdash \tau[G/\gamma]$ ok.
(4) *If* $\Gamma; \Delta, \gamma : \kappa; \Omega; \Psi \vdash e : \tau \mid G$ *and* $\Delta; \Omega; \Psi \vdash G' : \kappa$ *then* $\Gamma; \Delta; \Omega; \Psi \vdash e : \tau[G'/\gamma] \mid G[G'/\gamma]$.
(5) *If* $\Delta; \Omega, \vec{u}_f; \Psi, \vec{u}_t \vdash G : \kappa$ *then* $\Delta; \Omega, \vec{u}'_f; \Psi, \vec{u}'_t \vdash G[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t] : \kappa$.
(6) *If* $\Delta; \Psi, \vec{u}_t \vdash \tau$ ok *then* $\Delta; \Psi, \vec{u}'_t \vdash \tau[\vec{u}'_t/\vec{u}_t]$ ok.
(7) *If* $\Gamma; \Delta; \Omega, \vec{u}_f; \Psi, \vec{u}_t \vdash e : \tau \mid G$ *and* $\vec{u}'_f \cap \Omega, \vec{u}_f = \emptyset$ *then* $\Gamma; \Delta; \Omega, \vec{u}'_f; \Psi, \vec{u}'_t \vdash e[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t] : \tau[\vec{u}'_t/\vec{u}_t] \mid G[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t]$.

The proofs are straightforward inductions, and are available in the supplementary material.

We can now show that the various static semantics are consistent: If an expression has type τ and graph type G, then τ and G are well-formed. The judgment $\Delta; \Psi \vdash \Gamma$ is the natural extension of type formation to contexts: a context is well-formed if all of the types in it are.

Lemma 4. *If* $\Delta; \Psi \vdash \Gamma$ *and* $\Gamma; \Delta; \Omega; \Psi \vdash e : \tau \mid G$ *then* $\Delta; \Psi \vdash \tau$ ok *and* $\Delta; \Omega; \Psi \vdash G : *$.

Proof. By induction on the derivation of $\Gamma; \Delta; \Omega; \Psi \vdash e : \tau \mid G$. □

$$\text{(C:Unit)}\quad \frac{}{\langle\rangle \Downarrow \langle\rangle \mid \bullet} \qquad \text{(C:Fun)}\quad \frac{}{\mathsf{fun}[\vec{u}_f;\vec{u}_t]\ f\ x = e \Downarrow \mathsf{fun}[\vec{u}_f;\vec{u}_t]\ f\ x = e \mid \bullet}$$

$$\text{(C:App)}\quad \frac{e_1 \Downarrow \mathsf{fun}[\vec{u}_f;\vec{u}_t]\ f\ x = e \mid g_1 \qquad e_2 \Downarrow v \mid g_2 \qquad e[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t][v/x][\mathsf{fun}[\vec{u}_f;\vec{u}_t]\ f\ x = e/f] \Downarrow v' \mid g'}{e_1[\vec{u}'_f;\vec{u}'_t]\ e_2 \Downarrow v' \mid g_1 \oplus g_2 \oplus g'}$$

$$\text{(C:Pair)}\quad \frac{e_1 \Downarrow v_1 \mid g_1 \qquad e_2 \Downarrow v_2 \mid g_2}{(e_1, e_2) \Downarrow (v_1, v_2) \mid g_1 \oplus g_2} \qquad \text{(C:Fst)}\quad \frac{e \Downarrow (v_1, v_2) \mid g}{\mathsf{fst}\ e \Downarrow v_1 \mid g}$$

$$\text{(C:InL)}\quad \frac{e \Downarrow v \mid g}{\mathsf{inl}\ e \Downarrow \mathsf{inl}\ v \mid g} \qquad \text{(C:Future)}\quad \frac{e \Downarrow v \mid g}{\mathsf{future}[u]\ e \Downarrow \mathsf{future}[u]\ v \mid g \swarrow_u}$$

$$\text{(C:Touch)}\quad \frac{e \Downarrow \mathsf{future}[u]\ v \mid g}{\mathsf{touch}\ e \Downarrow v \mid g \oplus {}^{u}\searrow} \qquad \text{(C:New)}\quad \frac{e \Downarrow v \mid g \qquad u' \text{ fresh}}{\mathsf{new}\ u.e \Downarrow v \mid g[u'/u]}$$

Fig. 17. Cost Semantics for λ^G.

5.3 Cost Semantics and Soundness

Figure 17 presents the cost semantics for the full λ^G calculus. The judgment $e \Downarrow v \mid g$ is the same as in Figure 9 for λ^G_{FJ}. We omit rules C:Par, C:CaseL and C:CaseR, which are unchanged from Figure 9. We also omit rules C:Snd and C:InR, which are analogous to C:Fst and C:InL, respectively. The interesting additions are rules C:Future, C:Touch and C:New. For an expression $\mathsf{future}[u]\ e$, if e evaluates to v and has graph g, we represent the spawning of the future by left-composing g using vertex u. The future evaluates to a *future value* that simply stores the result v. An expression touch e first evaluates e down to a future value $\mathsf{future}[u]\ v$, which gives both the value that the touch should return and the vertex that should be touched. The graph that results from evaluating e is then sequentially composed with a touch of u. We evaluate an expression new $u.e$ by first evaluating e. For all instances of u in the resulting graph, we substitute a fresh vertex u' to preserve the "newness" of u in the final graph. Note that we do not need to substitute into the final value v: the typing rule S:New ensures that u does not appear in v.

We can now show the analogue of Theorem 1 for λ^G. As before, we show that if an expression e has a graph type G and evaluates with graph g, then g is in the normalization of G.

Theorem 3. *If $\cdot;\cdot;\Omega;\Psi \vdash e : \tau \mid G$ and $e \Downarrow v \mid g$, then $\cdot;\cdot;\Omega;\Psi \vdash v : \tau \mid \bullet$ and $g \in Norm_n(G)$ for some $n \in \mathbb{N}$.*

Proof. By induction on the derivation of $e \Downarrow v \mid g$. The interesting case is C:App. In this case, $e = e_1[\vec{u}'_f;\vec{u}'_t]e_2$ and $e_1 \Downarrow \mathsf{fun}[\vec{u}_f;\vec{u}_t]\ f\ x = e_0 \mid g_1$ and $e_2 \Downarrow v \mid g_2$ and

$$e_0[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t][v/x][\mathsf{fun}[\vec{u}_f;\vec{u}_t]\ f\ x = e_0/f] \Downarrow v' \mid g_3$$

By inversion on S:App, $\cdot;\cdot;\Omega_1;\Psi \vdash e_1 : \Pi\vec{u}_f;\vec{u}_t.\tau_1 \xrightarrow{G_3} \tau_2 \mid G_1$ and $\cdot;\cdot;\Omega_2;\Psi \vdash e_2 : \tau_1 \mid G_2$ where $\Omega = \Omega_1, \Omega_2, \vec{u}'_f$.

By induction, $\cdot;\cdot;\Omega_1;\Psi \vdash \mathsf{fun}[\vec{u}_f;\vec{u}_t]\ f\ x = e_0 : \Pi\vec{u}_f;\vec{u}_t.\tau_1 \xrightarrow{G_3} \tau_2 \mid \bullet$ and $\cdot;\cdot;\Omega_2;\Psi \vdash v : \tau_1 \mid \bullet$ and $g_1 \in Norm_{n_1}(G_1)$ and $g_2 \in Norm_{n_2}(G_2)$.

By inversion on S:Fun, $f : \Pi\vec{u}_f;\vec{u}_t.\tau_1 \xrightarrow{\gamma} \tau_2, x : \tau_1; \gamma : \Pi\vec{u}_f;\vec{u}_t.*; \vec{u}_f; \Psi, \vec{u}_t \vdash e_0 : \tau_2 \mid G$ and $G_3 = \mu\gamma.\Pi\vec{u}_f;\vec{u}_t.G$. By Lemma 3,

$$\cdot;\cdot;\vec{u}'_f;\Psi,\vec{u}'_t \vdash e_0[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t][v/x][\mathsf{fun}[\vec{u}_f;\vec{u}_t]\ f\ x = e_0/f] : \tau_2[\vec{u}'_t/\vec{u}_t] \mid G[\Pi\vec{u}_f;\vec{u}_t.G/\gamma][\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t]$$

By induction, $\cdot;\cdot;\vec{u}'_f;\Psi,\vec{u}'_t \vdash v' : \tau_2[\vec{u}'_t/\vec{u}_t] \mid \bullet$ and $g_3 \in \mathit{Norm}_{n_3}(G[\Pi\vec{u}_f;\vec{u}_t.G/\gamma][\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t])$. By Lemma 1, we have

$$\begin{array}{rcl} g_1 \oplus g_2 \oplus g_3 & \in & \mathit{Norm}_{\max(n_1,n_2,n_3)}(G_1 \oplus G_2 \oplus G[\Pi\vec{u}_f;\vec{u}_t.G/\gamma][\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t]) \\ & = & \mathit{Norm}_{\max(n_1,n_2,n_3)}(G_1 \oplus G_2 \oplus \mathsf{unroll}(G_3)[\vec{u}'_f;\vec{u}'_t]) \end{array}$$

□

6 VERTEX ANNOTATION INFERENCE

The graph type system for λ^G assumes that programs have been annotated with names of vertices. Recall that this requires three types of additions and annotations to ordinary source programs:

(1) Future creation operations `future e` must be annotated with a vertex name; this vertex becomes the sink of the created future in the graph.
(2) Function definitions must include two sets of vertices $\vec{u}_f$ and $\vec{u}_t$ for use in the function body; $\vec{u}_f$ must include all vertices used to spawn futures in the function body, as functions may not capture such vertices from the environment. The other set, $\vec{u}_t$, allows the function to accept arguments of future type and be polymorphic over the vertices used in the future. Applications of functions must instantiate these sets of vertices.
(3) Any new future names that must be created at runtime (e.g., in the body of a recursive function) must be bound in a "new" construct $\mathsf{new}\ u.e$.

We observed in the introduction (and further discuss in Section 9) an analogy between region type systems and λ^G; a similar analogy exists between region inference and inferring the above annotations. Our vertex inference algorithm is inspired by the region inference algorithm of Tofte and Birkedal [1998], which in turn builds on Hindley-Milner type inference [Hindley 1969; Milner 1978]. In the discussion of our vertex inference algorithm, we will therefore focus on aspects that are new or different from region inference and Hindley-Milner. A more detailed discussion of the differences between the setting of this paper and region systems appears in Section 9.

Inference of the vertex annotations occurs in one pass over the source program, at the same time as, and in concert with, standard Hindley-Milner type inference. Because vertex annotations appear in the types of futures, and vertex bindings appear in the types of functions (as binders of the form $\Pi\vec{u}_f;\vec{u}_t.\tau$), unification performed as part of type inference will naturally propagate and unify vertex annotations correctly. Unifying vertex annotations results in equivalence classes of vertices; vertices that have been made equivalent by unification are treated identically throughout the remainder of our algorithm.

When typing a spawn `future e`, our algorithm creates a new vertex name to use in the annotation of the expression and in the type of the new future. The remaining interesting points of the inference algorithm occur when handling function bodies. We follow these steps in inferring a function definition `fun x -> e` (or a let-bound non-recursive function; recursive functions are more complex and described below). We use $\mathsf{FF}(e)$ to refer to the future names spawned in e and $\mathsf{FVS}(\tau)$ to refer to the free vertex names in τ (this extends naturally to $\mathsf{FVS}(\Gamma)$).

(1) Create a fresh type variable for the argument x and add it to the context.
(2) Perform type inference on the body e to get an annotated function body e', infer a result type τ_2 and possibly narrow down the argument type τ_1.
(3) Let $\vec{u}_n \triangleq \mathsf{FF}(e) \setminus \mathsf{FVS}(\tau_2) \setminus \mathsf{FVS}(\Gamma)$. This is the set of vertices that are newly used as spawns in the function body and do not escape the function body, so may be bound as "new" inside the function. Let $e'' \triangleq \mathsf{new}\ \vec{u}_n.e'$.
(4) Let $\vec{u}_t \triangleq \mathsf{FVS}(\tau_1)$. This is the set of vertices over which the function must be polymorphic.

(5) Let $\vec{u}_f \triangleq \mathsf{FF}(e'')$. This is the set of vertices that are newly used as spawns in the function body and not bound in the "new". They must therefore be passed as parameters.
(6) Return the annotated function definition $\mathsf{fun}[\vec{u}_f; \vec{u}_t]\ x \to e''$.

At an application $e_1\ e_2$, we perform type inference on the two subexpressions. If the inferred type for e_1 is of the form $\Pi \vec{u}_f; \vec{u}_t.\tau_1 \xrightarrow{G} \tau_2$, then as we did when processing a future, we create two sets of fresh new vertices with which to instantiate $\vec{u}_f$ and $\vec{u}_t$. If these new sets are $\vec{u}'_f$ and $\vec{u}'_t$, we generate the annotated application $e_1[\vec{u}'_f; \vec{u}'_t]\ e_2$. If the type of e_1 has not yet been inferred (which might happen if e_1 is a variable), we infer the type $\Pi\emptyset; \vec{u}_t.\tau_1 \xrightarrow{\gamma} \tau_2$ where $\vec{u}_t = \mathsf{FF}(\tau_1)$ and generate the annotated application $e_1[\emptyset; \vec{u}'_t]\ e_2$. If e_2 is a future, this allows the function e_1 to be a vertex-polymorphic function accepting the future argument. Note that this means that a variable of function type (e.g., a functional argument of a higher-order function) will not be assigned a Π type, and therefore not be able to accept or return arguments containing futures, unless the type of the argument is known to contain a future (that is, $\mathsf{FF}(\tau_1) \neq \emptyset$) at the time of application. The impact of this limitation is fairly minimal in practice and is discussed in the appendix.

Recursive functions require additional care. As in traditional type inference, when analyzing the body of a recursive function f, we create a new type variable to represent the type of the function, add it to the context, run the procedure for non-recursive functions above and then unify the "guessed" type with the inferred type of the function. The type of f now has the "refined" form $\Pi \vec{u}_f; \vec{u}_t.\tau_1 \xrightarrow{G} \tau_2$, and so recursive invocations of f in the body will require annotations, e.g., $e_1[\vec{u}'_f; \vec{u}'_t]\ e_2$, as discussed in the previous paragraph. We therefore add the new, "refined," type of f to the context and run the inference procedure on the body of f *again* to produce these annotations and perform the relevant unifications on them. In standard type inference with polymorphic recursion, this iteration of performing inference on the function body could go on forever [Mycroft 1984] if the type of f grows on each iteration. However, in our case, the new vertex annotations on functions may substitute new vertices into the type of f but will not cause it to grow, and we can stop after the second iteration.

As an example, we will walk through inferring the vertex annotations for the Pipeline program of Section 4. Because the `pipeline` function is recursive, we begin by guessing a type `'a` for it and proceed to perform inference on the body. At `touch fut` on line 3, we infer that `fut` has the type $\text{'b future}[u_0]$, where u_0 is a freshly created name. When we later encounter `touch fut` on line 5, we unify 'b with int. When analyzing the application `f h`, we infer a type $\text{'c} \xrightarrow{\gamma_0} \text{'d}$ for `f`, where γ_0 is a freshly created graph variable (as discussed above, we do not infer a Π type for `f` because `h` is not yet known to—and indeed doesn't—have a future type). We generate a new vertex u_1 with which to annotate the `future` on line 5. The type of `fprefix` is then inferred to be $\text{int future}[u_1]$. By the time we encounter the recursive call, we will have inferred the type $(\text{'c} \xrightarrow{\gamma_0} \text{int}) \times \text{int future}[u_1] \times \text{'c list}$ for the argument to the recursive call. Because this has a free vertex u_1, we will infer the type

$$\Pi\emptyset; u_1.(\text{'c} \xrightarrow{\gamma_0} \text{int}) \times \text{int future}[u_1] \times \text{'c list} \xrightarrow{\gamma_1} \text{int}$$

for `pipeline` and annotate the application to be `pipeline[∅;`u_1`] (f, fprefix, t)`. The function body now has a free vertex u_1 used as a spawn. This vertex does not appear in the return type of the function, so we add a new u_1. binding around the body of the function. There are no remaining free spawn vertices, so $\vec{u}_f$ for the function is empty. To compute $\vec{u}_t$ for the function, we look at the type of the argument `(f, fut, l)`, which we have now inferred to be $(\text{'c} \xrightarrow{\gamma_0} \text{int}) \times \text{int future}[u_0] \times \text{'c list}$. We set $\vec{u}_t$ to be the free vertices of this type, which is just u_0. We then iterate this procedure again because `pipeline` is recursive, but this will not produce any additional annotations because $\vec{u}_f = \emptyset$. This gives us the fully annotated function.

```
let rec pipeline[∅;u₀] (f, fut, l) =
  new u₁. match l with
         | [] -> touch fut
         | h::t ->
            let fprefix = future[u₁] (f h + (touch fut)) in
            pipeline[∅;u₁] (f, fprefix, t)
```

7 IMPLEMENTATION AND CASE STUDIES

We implemented the graph type inference algorithm in a tool we call GML, which takes a source program written in a subset of OCaml, extended with `future` and `touch` as primitives, and reports the inferred types, including graph types for functions, of all declarations in the source file. It is also able to produce visualizations of the computation graphs of functions. We describe GML, including extensions to the theory that are supported by the implementation, in Section 7.1. In Section 7.2, we discuss a number of microbenchmarks and one large example program on which we tested GML.

We also consider two program analyses that are typically performed, implicitly or explicitly, on a graph: deadlock detection and cost analysis. Definitions of both analyses in terms of actual graphs g are readily available in the literature. Algorithms for performing these analyses directly over graph types G are a potentially rich area of future work and are outside the scope of this paper, but we report in Sections 7.3 and 7.4 on proof-of-concept implementations of these analyses using our implementations of graph type inference and normalization.

In Section 7.5, we also report on one program *transformation* aided by the graph, which we perform directly using the graph types, with no normalization. The GML input language, like many mainstream languages, includes only futures and not `par`. However, runtime scheduling can be done much more efficiently for fork-join parallelism than for futures. It may therefore beneficial for a compiler to be able to automatically find parts of a program that can be compiled using fork-join. We implement a compiler pass that uses graph type information to find `future`/`touch` pairs that can be replaced with a `par` and convert them accordingly.

7.1 Implementation

GML was written in OCaml and consists primarily of a parser (generated with `ocamllex` and `ocamlyacc`) and the type inference algorithm. At present, GML is a stand-alone tool that only performs graph type inference/checking, as well as the analyses described below. To keep the design simple, it is not implemented as part of OCaml or another compiler, although we hope to pursue such an implementation in the future. We simultaneously infer vertex annotations using the algorithm of Section 6, standard types using Hindley-Milner type inference, and graph types; graph type inference closely follows the rules of Figure 16, which are fairly algorithmic given a vertex-annotated expression. The user does not need to specify type annotations, graph type annotations or vertex annotations (indeed, there is not even syntax for graph types or vertex annotations): all annotations are inferred by GML. GML is not a compiler, as it stops after generating an annotated abstract syntax tree, but programs accepted by our front-end could be compiled using standard OCaml compilers (with libraries for parallelism); efficient compilation and execution of parallel programs is a separate and rich research area that is outside the scope of this paper.

GML accepts input programs written in a sizable subset of OCaml, with many features that are not included in the λ^G calculus including n-ary tuples, lists and limited forms of pattern matching. We do not yet support all of the syntactic sugar in OCaml, including full pattern matching or multi-argument functions (multiple arguments can be simulated in GML using tuple arguments or currying). The biggest extension of the theory supported by GML is mutable references. As in OCaml, references have type `t ref` for any type `t`, including futures. The syntax `ref e` creates a new reference initialized to the value of `e`; the value stored in a future can be retrieved using the syntax `!e` and destructively updated using the syntax `e := e`. We also provide a primitive `futref`, a reference to an as-yet-unspecified future. This allows programs to declare, and access, futures which have not yet been spawned, which is a common idiom in imperative implementations of futures and is useful for applications such as dynamic programming. For how much expressive power they add to the language, mutable references require few extensions to the theory or to vertex annotation inference. The typing rules for references simply propagate types and graphs in the expected ways; dereferencing a reference does not produce a new graph beyond evaluating the dereferenced expression as only values are stored in references.

$$\text{(S:Ref)}\quad \frac{\Gamma; \Delta; \Omega; \Psi \vdash e : \tau \mid G}{\Gamma; \Delta; \Omega; \Psi \vdash \mathsf{ref}\ e : \tau\ \mathsf{ref} \mid G} \qquad \text{(S:Deref)}\quad \frac{\Gamma; \Delta; \Omega; \Psi \vdash e : \tau\ \mathsf{ref} \mid G}{\Gamma; \Delta; \Omega; \Psi \vdash !e : \tau \mid G} \qquad \text{(S:Assign)}\quad \frac{\Gamma; \Delta; \Omega_1; \Psi \vdash e_1 : \tau\ \mathsf{ref} \mid G_1 \quad \Gamma; \Delta; \Omega_2; \Psi \vdash e_2 : \tau \mid G_2}{\Gamma; \Delta; \Omega_1, \Omega_2; \Psi \vdash e_1 := e_2 : \tau \mid G_1 \oplus G_2}$$

Extending the rest of the machinery of the paper to include references would require tracking a store through the operational semantics; this type of cost semantics is doable (see, e.g., [Muller et al. 2020]) but complicates the presentation and so references were left out for the sake of simplicity. Type inference for references proceeds similarly to inference for other types[2].

After performing inference, GML outputs the type of each top-level declaration in the file. Because the ASCII representations of graph types can get quite verbose, GML can also output DOT file representations of the graph types, on which we then run GraphViz [Gansner and North 2000] to generate a visual representation of the graph type. Figure 18a shows the compact visualization of the graph type for `pipeline` above. Note that the visualization includes Π, μ and ν bindings, and displays both alternatives of a $G_1 \vee G_2$ type (as "alt1" and "alt2" in the figure). GML also includes the ability to perform normalization on the graph types and output standard graph representations for easier visualization. Figure 18b shows the pipeline graph type unrolled three times. By default, when unrolling a graph type $G_1 \vee G_2$ n times, GML takes the larger alternative (by number of graph type connectives) the first $n-1$ times and the smaller one on the last iteration.

Many properties of a parallel program can be inferred readily by the human eye when looking at a visualization of one or both forms above (the concrete graph type visualization of Figure 18a or the unrolling of Figure 18b). For example, it is straightforward to tell from such a graph, even when unrolled just once or twice as in Figure 18b, the amount of parallelism in an algorithm. As an example, consider the following slight variation on one line of the pipeline program:

```
let fprefix = future ((touch fut) + f h) in
```

We have swapped the two addends. In GML (unlike in OCaml), function arguments are evaluated left-to-right, so this waits on `fut` *before* computing `f h`, rather than allowing the future to proceed in parallel. This small change to the code eliminates the pipelining effect and, indeed, most of the parallelism in the program; this fact is not easy to see in the code but is readily visible in Figure 18c, the visualization of the pipeline program with the above substitution: all of the instances of "`g2", which represents the calls to `f`, are in a path and therefore occur in sequence.

[2]As always, the combination of mutation and polymorphism causes some problems, which we solve using an OCaml-style value restriction: we only generalize type variables for syntactic values; in particular, we do not generalize the type of `futref` to `'a future ref`.

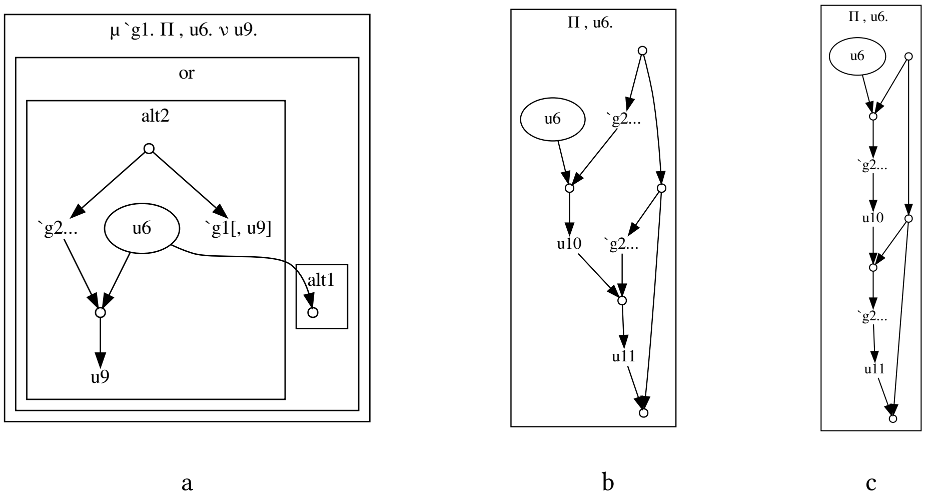


Fig. 18. Two visual representations of the pipeline program, and one of a slight alteration to the source code that drastically decreases parallelism. All three visualizations were produced by GML.

Table 1. Microbenchmarks, grouped by purpose.

Benchmark purpose	#
Standard parallel benchmarks (e.g., Fibonacci, Quicksort)	3
Adversarial/complex tests of vertex and type inference	11
Tests of deadlock detector (9 with deadlocks, 4 without)	13
Adversarial tests for fork-join conversion	4
Total	31

7.2 Benchmarks

We developed a number of benchmarks on which to test graph type inference, as well as the analyses and transformations described later in the section. Some were microbenchmarks, largely used as regression tests or adversarial tests of corner cases for type inference and the analyses. We also implemented one larger case study, a concurrent web server, to demonstrate the expressivity and scaling of the implementation.

7.2.1 Microbenchmarks. The microbenchmark suite consists of 31 example programs of varying sizes and purposes. Most tested a particular behavior of graph type inference or of deadlock detection or fork-join conversion (both described later in the section.) The type/purpose and number of microenchmarks are summarized in Table 1.

7.2.2 Case Study: Web Server. As a larger proof-of-concept example to show that GML allows suitably expressive input programs and can scale, we implemented a variation on the web server example of Muller et al. [2018]. Our version of the program consists of a foreground loop that accepts incoming HTTP connections and a background loop that periodically optimizes a cache of frequently-requested pages. Both the foreground and background tasks involve non-trivial parallelism. The foreground loop spawns a new future to handle every new connection, so that each

Table 2. Benchmark sizes and type inference times.

Benchmark	LoC	Time (ms)
Quicksort	50	1.1
Webserver	156	10.6
Webserver	200	16.3
Webserver	262	16.8
Webserver	362	24.6

client can be served in parallel. The asynchronous interaction that this allows is a major benefit of futures over fork-join programming. The function to handle HTTP requests accepts the request, and looks up the page in a cache maintained in a global mutable reference. If the page is in not in the cache, it is loaded from disk. All requested pages are recorded in a buffer also stored as global mutable reference[3].

The background thread maintains the cache by tracking a histogram of pages with the number of times they have been requested. It periodically traverses the buffer of new requests, updates the histogram and uses a parallel implementation of Quicksort to sort the pages by number of requests. The cache is updated with the top N most-requested pages, where N is a global constant.

For the purposes of type inference, we replace low-level system library calls, mostly those dealing with network operations, with no-ops; this does not impact the graph type inference as these networking system calls are sequential. GML is able to correctly infer useful graph types for all of the top-level functions in the program. As is the case with many static analyses for difficult-to-evaluate properties, correctness was assessed by manually inspecting the program to determine the correct graph type.

Although performance is not a major result of this paper, we tested varying versions of the benchmark of different sizes (by implementing more or less functionality within the benchmark itself that might otherwise be delegated to library code) to determine the effect on type inference time. The results, for 3 versions of the webserver and the smaller Quicksort benchmark (which forms part of the webserver code as well) are summarized in Table 2. The table shows the (non-blank) lines of code for each benchmark, as well as time taken for graph type inference on a commodity desktop. A better-engineered implementation and a more robust scaling evaluation are outside the scope of this paper, but the proof-of-concept implementation shows well-behaved scaling even on the largest (362-line) version of the webserver. This at least shows that graph type inference is fast enough to be performed as a normal part of compilation. This is to be expected, as graph type checking, like type checking, is naturally compositional: after a type is inferred for a function, its size no longer impacts the checking of other code as only its type is used. The types of functions are larger because they capture more information about the function (its parallel dependencies) but in most real-world code such as the web server, parallel behavior is confined to a few "main" functions, so this is not a large scaling problem in practice.

7.3 Application: Deadlock Detection

As discussed above, a deadlock corresponds to a cycle in the computation graph. Therefore, if a function has graph type G and all $g \in Norm_n(G)$ are acyclic for all $n \geq 0$, then the function is

[3]In an implementation to executed on an actual parallel platform, these accesses to global state would need to be protected by some form of synchronization provided by the underlying parallelism implementation; we are only concerned with the inference of the graph types and so the implementation details of parallelism and synchronization are out of scope.

guaranteed not to deadlock. Cycle detection on a graph can be performed in linear time in the number of vertices and edges using a depth-first search, so finding a deadlock in a particular graph is straightforward. We cannot, of course, search an infinite number of normalized graphs. However, we conjecture that it suffices to normalize until every recursive graph type has been unrolled at least once. At this point, further recursive applications of graph types will use either the same vertices as previous applications or newly-created vertices, which will not in either case create new cycles. We have implemented a prototype deadlock detector based on this conjecture; it normalizes the graph type until every recursive binding has been unrolled at least once, then returns the resulting set of normalized graphs. Each graph is then checked for cycles.

We also consider it to be a deadlock if a `touch` targets a future that is never spawned; this corresponds to an edge in a graph whose source does not exist in the graph. Our deadlock detector checks the graph type directly for this possibility, without normalizing. The algorithm traverses the graph type, tracking two sets of vertices: 1) vertices that are definitely spawned in the graph and 2) vertices that may be touched without being spawned. If set (2) is nonempty at the end of the traversal, we report a possible deadlock.

We tested the deadlock detector on a number of example programs with and without deadlocks, including some synthetic, adversarial examples with difficult-to-detect deadlocks, a version of the webserver case study with a subtle deadlock (as well as the original, deadlock-free version), and a motivating example from prior work on dynamic deadlock detection [Cogumbreiro et al. 2017]. Our deadlock detector was correct on all tested examples, where the ground truth was determined by manual inspection of the code. One interesting example is a program that uses a dynamic programming-style algorithm to compute the 8^{th} Fibonacci number. The algorithm spawns 8 futures which are placed in global mutable references: future n computes $fib(n)$ by touching futures $n-1$ and $n-2$. The futures themselves are spawned in parallel in a tree-like, nested-parallel function (the main thread splits into two, each of those threads splits into two, each of those splits into two and each of the 8 resulting threads creates a future and writes it into the array). A slight programming error (e.g., instead touching futures $n-1$ and $n+1$) could result in a deadlock, which our detector successfully reports. On the correct version of the program, the detector correctly reports no deadlock.

7.4 Application: Cost Analysis

The cost of a parallel program is typically reported in terms of its *work* (the number of vertices in the graph, which corresponds to the total amount of computation) and *span* (the longest path in the graph, which must be executed sequentially). Work divided by span is often used as a measure of the amount of parallelism in an algorithm. When reporting the parallelism of a program or algorithm, both quantities are generally reported asymptotically in the input size, recognizing that a program can produce a family of graphs of varying sizes depending on the input.

Determining loop bounds and program running time based on program inputs is a complex task and has been the subject of a great deal of research (e.g., [Blazy et al. 2013; Ermedahl et al. 2007; Hofmann and Jost 2003; Li and Malik 1997]), which we do not aim to replicate or improve upon here. We can, however, use graph types to say something about the asymptotic work and span of a graph in terms of the number of loop iterations (the techniques of the above-cited literature can then be used to translate between program inputs and number of loop iterations). Let $W(g)$ and $S(g)$ be the work and span of a graph, respectively. As a simple case, assume a program has a graph type G with a single recursive binding. Let $w(n) = \max\{W(g) \mid g \in Norm_n(G)\}$ and $s(n) = \max\{S(g) \mid g \in Norm_n(G)\}$ The work of the program is then $O(w(n))$ and the span is $O(s(n))$, where n is the number of recursive iterations of the program (or, alternatively, the number of unrollings of the recursive binding of the graph type). The work and span functions can be expanded into multivariate functions for graph types with multiple recursive bindings.

As a proof of concept, we implemented an extension to GML that unrolls the graph of a program various numbers of times and attempts to determine the complexity class (currently just linear, quadratic or exponential) of the work and span in terms of the number of unrollings. This implementation is able to correctly determine, for example:

- Exponential work and linear span for Fibonacci.
- Quadratic work and linear span for the Pipeline example, where the function f is also linear.
- Quadratic work and span for the accidentally de-parallelized Pipeline example (Figure 18c)

As with the deadlock detector, the ground truth works and spans for the above examples were determined manually (aided by known work and span for well-known algorithms like Fibonacci and parallel Quicksort).

7.5 Application: Fork-Join Transformation

Many parallel applications, such as the web server example of Section 7.2.2, mix fork-join code (e.g., parallel Quicksort) with more complex uses of futures (e.g., the pipelined `buildcache`) function. Many languages, including the input language of GML, provide only futures as a parallelism construct, since it is easier in the early stages of compilation to handle a small number of language constructs, and futures strictly subsume fork-join. However, in work-stealing schedulers, which are common in parallel language runtimes, fork-join code exposes valuable optimizations. It is therefore beneficial for the code actually executed to use a different mechanism for purely fork-join code, such as the `par` construct of Section 3 (by purely fork-join, we mean that expressions within a `par` do not make general use of futures, though they may have nested uses of `par`). We show in this subsection how graph types can be used by a compiler to automatically transform code that uses only futures into code that uses `par` for purely fork-join parts of the program.

The general idea of the transformation algorithm is as follows. We maintain a stack, called the *spine*, of futures spawned by the current thread. For each future, we track its sink vertex, the code that the future executes, and the code executed in the current thread since the spawn. If the spawned futures are thought of as forming a tree, each element of the spine contains a branch and the piece of the trunk between that branch and the next. An entry of the spine also indicates if that future contains only fork-join code. The first entry in the spine contains a special entry with just the code in the "trunk" before the furst future is spawned. On a touch, if the touched future is at the bottom of the stack (was the last one spawned) *and* contains only fork-join code, then we are able to make a `par` out of the code of the future and the trunk. Otherwise, the current spine is not purely fork-join and we reconstruct it into a single trunk and continue. A formalization of this process is available in the supplementary material.

8 DISCUSSION AND FUTURE WORK

Two exciting areas for future work are using graph types to implement static analyses, and enhancing the expressiveness of graph types themselves. This section briefly explores both areas.

Development of Static Analyses. In addition to the proof-of-concept analyses prsented in this paper, we believe that many more, and more elegant or efficient, analyses can be developed that operate directly over graph types. The reason is that a central component of many static analyses for parallel programs is extracting a graph; graph types abstract out this information. Thus, we believe graph types provide two major benefits to the development of static analyses: 1) developers of new analyses need not write code to extract graph information from source programs if this information is already available in a graph type, and 2) while this paper discusses graph types and inference only for an OCaml-like language, these techniques can be applied in many other languages; an analysis that works over graph types rather than source code can then be immediately applied to any language for which graph types are implemented.

The development of our proof-of-concept deadlock and cost analyses can serve as a "recipe" for porting other static analyses to work on graph types or developing other static analyses that work directly on graph types. The basic idea is to formulate (part of) the analysis as a graph problem—for example, deadlock detection is equivalent to determining if there is a cycle in the graph—and then investigate how to solve the graph problem from the graph type. In our deadlock detection case, this involves unrolling the graph type to some extent as some cycles may not be visible in the bare graph type; it remains to be seen whether a deadlock detector can operate directly on the graph type without unrolling. Other questions can be answered simply by reading off the graph type.

One such example not discussed in this paper is race detection. The heart of a race detection algorithm is a may-happen-in-parallel analysis (a race, then, is a write and a read or write to the same location that may happen in parallel). May-happen-in-parallel analyses exist for many languages but are fairly heavyweight to design and implement. Checking if two program points may happen in parallel in a graph type simply involves tracking what vertices in the graph represent the two program points (this shouldn't require unrolling as the graph type corresponds directly to the source program) and performing a reachability query on the graph (vertices u and v may happen in parallel if there is no path from u to v or vice-versa).

Limitations and Future Work. While graph types, as presented here, can represent and have been implemented over a large enough subset of OCaml to encode interesting examples, some uses of futures remain out of reach and the graph type system itself can be extended in many interesting ways to support still more general forms of parallelism. As one example, to support recursive data structures (such as lists) containing futures, types would need a way to generate new vertex names (for example, $\tau\ \mathsf{future\ list} \triangleq \mu t.\mathsf{unit} + \nu u.(\tau\ \mathsf{future}[u] \times t)$). As another, dynamic programming algorithms can be expressed in an elegant way using arrays of futures; to produce graph types for such algorithms in general would require reasoning about array indices in the graph types, using some form of dependent types. We intend to explore such extensions in the future, while also further exploring the power of graph types as presented in this work.

9 RELATED WORK

Graphs and Cost Semantics. The idea of using a graph as an abstract representation of a parallel program goes back to at least the 1960s [Karp and Miller 1966; Rodriguez Bezos 1969]. The work we build on most directly is in the community of *cost semantics* for parallel programs, in which a dynamic semantics for a parallel language is augmented to track cost information in the form of a graph, often called a *cost graph* or *cost DAG* (for Directed Acyclic Graph, as in programs that do not deadlock, the graph will be acyclic). Cost semantics were originally developed to reason about the complexity of sequential algorithms [Rosendahl 1989; Sands 1990]. Our cost semantics notation is most closely based on the work of Blelloch and Greiner [1995, 1996], who were in turn inspired by the work of Hudak and Anderson [1987], which used partially ordered multisets to represent dependencies in much the same way as DAGs. This early work focused on fork-join or nested parallelism and was later extended to handle futures by Spoonhower [2009] using the "left composition" and edge addition operations we also use in our graphs and graph types. While we focus on the work on cost semantics because much of it tends to be more theoretical in nature, papers in other areas have also presented dynamic semantics augmented with some form of graph. For example, Cogumbreiro et al. [2017] present such a semantics in the context of deadlock detection.

In this work, we are primarily interested in control dependencies that arise from the use of fork-join constructs and futures. Other notions of dependency graphs can capture other forms of data and control dependencies (in sequential and parallel or concurrent programs) and can be used for various other program analyses (e.g., [Korel 1987]). Analyzing these dependencies, statically or dynamically, is the main idea behind *program slicing* [Weiser 1984].

Static Graph Analysis. There is little existing work on representing and analyzing computation or dependency graphs for parallel programs. Most of the existing work focuses on coarser-grained parallelism, rather than the fine-grained implicit parallelism provided by fork-join and futures. For example, Chen et al. [2002] developed a static dependency analysis for Ada 95, which includes constructs for coarse-grained parallelism. In a series of papers, Cheng et al. proposed several general graph formalisms for representing a wide variety of control and data dependencies in sequential and parallel programs (e.g., [Cheng 1993]) and developed an automated tool for computing and visualizing these graphs from programs in several languages with coarse-grained parallelism including Ada, C and Pascal [Kasahara et al. 1995]. Their tool even includes analyses built on top of the graph representation, including a deadlock detector. Because of the granularity considered, none of the above work need consider the level of dynamism of the graphs we handle, and none has or needs features similar to ours for compact representation, e.g., recursive and polymorphic graph types: the produced graph is a single, concrete graph.

Other Graph-Based Analyses. Although they do not present a formal operational semantics, theoretical models of properties such as priority inversions [Babaoğlu et al. 1993] and data races [Banerjee et al. 2006] have included models that use dependency information in ways similar to a dependency graph. Practical dynamic analyses for, for example, deadlock (e.g. [Cogumbreiro et al. 2018]) and data races (e.g. [Banerjee et al. 2006]) track this dependency information in a variety of ways, such as by using vector clocks [Lamport 1978]. As tracking and analyzing all of the necessary information at runtime can be quite expensive, much work has focused on only tracking some subset of it [Flanagan and Freund 2009; Utterback et al. 2019, 2016; Xu et al. 2020]. Other work on deadlock detection, such as Known Joins (KJ) [Cogumbreiro et al. 2017] and Transitive Joins (TJ) [Voss et al. 2019] has traded precision for overhead by tracking certain dependency patterns that *can* lead to deadlock. These patterns are easier to track at runtime, leading to low overhead. However, both KJ and TJ disallow many deadlock-free programs, including the dynamic-programming Fibonacci example of Section 7.3, which our analysis correctly reports to be deadlock-free.

In general, dynamic analyses such as these track dependency information for a particular run of a program. Some of them are specific to a particular *schedule* (that is, they may report that an execution is free of deadlocks or races when another run of the program, with different sets of runtime decisions made by the scheduler would result in a deadlock or race). Even those analyses that are agnostic to the schedule are specific to a particular *input* and in general cannot declare a program free of deadlocks or races for all inputs in the same way that a static analysis can.

Other Static Analyses. Static analyses also exist for many of the discussed features of parallel programs. Because many of these features are fundamentally questions of dependency, these analyses in general perform some static approximation of dependency information, but do not explicitly generate or reason about representations of the full dependency graph in the way that this paper does. For example, many static analyses for deadlocks (e.g., [Boyapati et al. 2002; Engler and Ashcraft 2003]) operate on more explicitly threaded code (e.g., code that uses `pthreads` with mutexes) and tracks ownership of locks by threads in order to detect cycles. Navabi et al. [2008] develop a static analysis of the dependency structure of programs with futures, in order to ensure consistency with a sequential semantics (that is, the dependency pattern of the program could arise from a sequential program without futures, a hallmark of reasoning about parallel programs). Nandivada et al. [2013] analyze data dependencies in X10 code to determine whether various optimizations and refactorings are legal; this is similar in nature to our fork-join transformation (in the supplementary material) but the dependency patterns required are less complex, and X10 includes only async-finish parallelism, which is less general than futures.

Hoffmann and Shao [2015] analyze fork-join programs to obtain upper bounds on their work and span in terms of the input sizes. This analysis maintains some information about the structure of the cost graph, but only locally in order to compose the work and span of subgraphs properly. The information maintained is not general enough to handle futures. Static cost analyses like theirs are in many ways complementary to our cost analysis of Section 7.4: their analysis can compute the relationship between loop iterations and input sizes which, together with the dependency information from our analysis, can give the actual work and span bounds for very complex programs.

Our analysis captures a superset of the dependency information tracked by the above analyses, which are each specialized to a particular application. Using our analysis, implementers of new static analyses for parallel programs can simply implement their analyses over the graph type and not need to worry about extracting the necessary dependency information from a source program.

Effect Types, Region Systems and Region Inference. Our graph type system can be considered a form of *type and effect* system. These systems were originally proposed by Lucassen and Gifford [Gifford and Lucassen 1986; Lucassen 1987] to track imperative side effects that occur during evaluation, and later used by Jouvelot and Gifford [1989] to track control flow. One application of effect systems that was particularly influential on this paper is *region type systems*, particularly that of Tofte and Talpin [1997]. These systems give a type discipline for region-based memory management, in which allocations are grouped into *regions*, simplifying memory management. The type system uses effects to track what regions must be live (not deallocated) for certain computations to be meaningful. Like vertices in our graph type system, regions are explicitly named and generated at runtime; several of the technical devices of annotating vertices on spawns (analogous to annotating regions on allocations) and for passing vertices to functions are inspired by Tofte and Talpin's system. Tofte and Birkedal [1998] proposed the first efficient algorithm for inferring region annotations, and our vertex algorithm is in turn inspired by their inference algorithm. There are several important distinctions between our system and region systems, however. The most notable is that, unlike vertices, regions may be used by multiple allocations; indeed, the region inference algorithm often intentionally combines allocations into the same region. Our affine discipline for vertex names is thus new with respect to region type systems, and our inference algorithm must be more careful to ensure that every future creation is annotated with a unique vertex. We have necessarily omitted details of region-based memory management and the related literature; the interested reader can find more information and additional citations in the excellent survey by Henglein et al. [2005].

10 CONCLUSION

We have presented *graph types*, a general and expressive notation for representing sets of computation graphs of a parallel program. Our *graph type system* is able to infer graph types from annotated parallel programs; these annotations need not be added by the programmer but can be inferred by our *vertex annotation inference* algorithm. We have implemented all of the above for a reasonably expressive subset of OCaml, extended with parallelism primitives, in our tool GML. As demonstrated in several proof-of-concept implementations, graph types show some promise in easing the development and implementation of new static analyses for parallel programs.

ACKNOWLEDGMENTS

The author would like to thank the anonymous reviewers, as well as Umut Acar, Jatin Arora, Guy Blelloch, and Sam Westrick for helpful discussions at various points in this work. This work was partially supported by the National Science Foundation under grant CCF-2107289.

REFERENCES

Arvind and K. P. Gostelow. 1978. *The Id Report: An Asychronous Language and Computing Machine.* Technical Report TR-114. Department of Information and Computer Science, University of California, Irvine.

Özalp Babaoğlu, Keith Marzullo, and Fred B. Schneider. 1993. A Formalization of Priority Inversion. *Real-Time Systems* 5, 4 (1993), 285–303. https://doi.org/10.1007/BF01088832

Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. 2006. A Theory of Data Race Detection. In *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging* (Portland, Maine, USA) *(PADTAD '06)*. Association for Computing Machinery, New York, NY, USA, 69–78. https://doi.org/10.1145/1147403.1147416

Sandrine Blazy, André Maroneze, and David Pichardie. 2013. Formal Verification of Loop Bound Estimation for WCET Analysis. In *Revised Selected Papers of the 5th International Conference on Verified Software: Theories, Tools, Experiments - Volume 8164* (Menlo Park, CA, USA) *(VSTTE 2013)*. Springer-Verlag, Berlin, Heidelberg, 281–303. https://doi.org/10.1007/978-3-642-54108-7_15

Guy Blelloch and John Greiner. 1995. Parallelism in Sequential Functional Languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA) *(FPCA '95)*. Association for Computing Machinery, New York, NY, USA, 226–237. https://doi.org/10.1145/224164.224210

Guy E. Blelloch and John Greiner. 1996. A Provable Time and Space Efficient Implementation of NESL. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming* (Philadelphia, Pennsylvania, USA) *(ICFP '96)*. Association for Computing Machinery, New York, NY, USA, 213–225. https://doi.org/10.1145/232627.232650

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

Zhenqiang Chen, Baowen Xu, Jianjun Zhao, and Hongji Yang. 2002. Static Dependency Analysis for Concurrent Ada 95 Programs. In *Reliable Software Technologies — Ada-Europe 2002*, Johann Blieberger and Alfred Strohmeier (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 219–230.

Jingde Cheng. 1993. Process dependence net of distributed programs and its applications in development of distributed systems. In *Proceedings of 1993 IEEE 17th International Computer Software and Applications Conference COMPSAC '93*. 231–240. https://doi.org/10.1109/CMPSAC.1993.404187

Tiago Cogumbreiro, Raymond Hu, Francisco Martins, and Nobuko Yoshida. 2018. Dynamic Deadlock Verification for General Barrier Synchronisation. *ACM Trans. Program. Lang. Syst.* 41, 1, Article 1 (Dec. 2018), 38 pages. https://doi.org/10.1145/3229060

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

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

Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. 2007. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis. 6 (01 2007).

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

Emden R. Gansner and Stephen C. North. 2000. An Open Graph Visualization System and Its Applications to Software Engineering. *Softw. Pract. Exper.* 30, 11 (Sept. 2000), 1203–1233.

David K. Gifford and John M. Lucassen. 1986. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) *(LFP '86)*. Association for Computing Machinery, New York, NY, USA, 28–38. https://doi.org/10.1145/319838.319848

Robert H. Halstead. 1985. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* 7 (1985), 501–538.

Fritz Henglein, Henning Makholm, and Henning Niss. 2005. Effect Types and Region-Based Memory Management. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Cambridge, Massachusetts, Chapter 3, 87–135.

J. Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60. http://www.jstor.org/stable/1995158

Jan Hoffmann and Zhong Shao. 2015. Automatic Static Cost Analysis for Parallel Programs. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 132–157.

Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th Symposium on Principles of Programming Languages (POPL'03)*. 185–197.

Paul Hudak and Steven Anderson. 1987. Pomset Interpretations of Parallel Functional Programs. In *Proceedings of the Functional Programming Languages and Computer Architecture*. Springer-Verlag, Berlin, Heidelberg, 234–256.

Pierre Jouvelot and David K. Gifford. 1989. Reasoning about Continuations with Control Effects. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (Portland, Oregon, USA) *(PLDI '89)*. Association for Computing Machinery, New York, NY, USA, 218–226. https://doi.org/10.1145/73141.74837

Richard M. Karp and Rayamond E. Miller. 1966. Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing. *SIAM J. Appl. Math.* 14, 6 (1966), 1390–1411. https://doi.org/10.1137/0114108

Y. Kasahara, Y. Nomura, M. Kamachi, J. Cheng, and K. Ushijima. 1995. An integrated support environment for distributed software development based on unified program representations. In *Proceedings 1995 Asia Pacific Software Engineering Conference*. 254–263. https://doi.org/10.1109/APSEC.1995.496974

Bogdan Korel. 1987. The program dependence graph in static program testing. *Inform. Process. Lett.* 24, 2 (1987), 103–108. https://doi.org/10.1016/0020-0190(87)90102-5

Leslie Lamport. 1978. Time, Clocks and the Ordering of Events in a Distributed System. *Commun. ACM* 21 (July 1978), 558–565.

Yau-Tsun Steven Li and Sharad Malik. 1997. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16, 12 (1997), 1477–1487.

John M. Lucassen. 1987. *Types and Effects toward the Integration of Functional and Imperative Programming*. PhD thesis. Massachusetts Institute of Technology, Cambridge, Massachusetts. Available as Technical Report MIT-LCS-TR-408.

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Stefan K. Muller, Umut A. Acar, and Robert Harper. 2018. Competitive Parallelism: Getting Your Priorities Right. *Proc. ACM Program. Lang.* 2, ICFP, Article 95 (July 2018), 30 pages.

Stefan K. Muller, Kyle Singer, Noah Goldstein, Umut A. Acar, Kunal Agrawal, and I-Ting Angelina Lee. 2020. Responsive Parallelism with Futures and State. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 577–591. https://doi.org/10.1145/3385412.3386013

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.

V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. 2013. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Trans. Program. Lang. Syst.* 35, 1, Article 3 (April 2013), 48 pages. https://doi.org/10.1145/2450136.2450138

Armand Navabi, Xiangyu Zhang, and Suresh Jagannathan. 2008. Quasi-Static Scheduling for Safe Futures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Salt Lake City, UT, USA) *(PPoPP '08)*. Association for Computing Machinery, New York, NY, USA, 23–32. https://doi.org/10.1145/1345206.1345212

Jorge E Rodriguez Bezos. 1969. *A Graph Model for Parallel Computations*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, Massachusetts.

Mads Rosendahl. 1989. Automatic complexity analysis. In *FPCA '89: Functional Programming Languages and Computer Architecture*. ACM, 144–156.

David Sands. 1990. Complexity Analysis for a Lazy Higher-Order Language. In *ESOP '90: Proceedings of the 3rd European Symposium on Programming*. Springer-Verlag, London, UK, 361–376.

Daniel Spoonhower. 2009. *Scheduling Deterministic Parallel Programs*. Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, USA.

Mads Tofte and Lars Birkedal. 1998. A Region Inference Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (July 1998), 724–767. https://doi.org/10.1145/291891.291894

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (1997), 109–176. https://doi.org/10.1006/inco.1996.2613

Robert Utterback, Kunal Agrawal, Jeremy Fineman, and I-Ting Angelina Lee. 2019. Efficient Race Detection with Futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 340–354. https://doi.org/10.1145/3293883.3295732

(T:Unit)
$$\frac{}{\Delta; \Psi \vdash \mathsf{unit\ ok}}$$

(T:Fun)
$$\frac{\Delta; \Psi, \vec{u}_t \vdash \tau_1 \text{ ok} \qquad \Delta; \Psi, \vec{u}_t \vdash \tau_2 \text{ ok} \qquad \Delta; \cdot; \Psi \vdash G : \Pi \vec{u}_f; \vec{u}_t.*}{\Delta; \Psi \vdash \Pi \vec{u}_f; \vec{u}_t.\tau_1 \xrightarrow{G} \tau_2 \text{ ok}}$$

(T:Sum)
$$\frac{\Delta; \Psi \vdash \tau_1 \text{ ok} \qquad \Delta; \Psi \vdash \tau_2 \text{ ok}}{\Delta; \Psi \vdash \tau_1 + \tau_2 \text{ ok}}$$

(T:Prod)
$$\frac{\Delta; \Psi \vdash \tau_1 \text{ ok} \qquad \Delta; \Psi \vdash \tau_2 \text{ ok}}{\Delta; \Psi \vdash \tau_1 \times \tau_2 \text{ ok}}$$

(T:Fut)
$$\frac{\Delta; \Psi, u \vdash \tau \text{ ok}}{\Delta; \Psi, u \vdash \tau\ \mathsf{future}[u] \text{ ok}}$$

Fig. 19. Type formation rules.

Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. 2016. Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures* (Pacific Grove, California, USA) *(SPAA '16)*. Association for Computing Machinery, New York, NY, USA, 83–94. https://doi.org/10.1145/2935764.2935801

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

Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

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

A TYPE FORMATION

As we did for graph types, we present rules defining what it means for a type to be well-formed. The judgment is $\Delta; \Psi \vdash \tau$ ok. Note that this judgment does not include the affine context of vertices "available" for spawns: types are fundamentally predictions about the *value* of an expression, and values do not spawn futures (future handles are themselves values, however, and so types must include the vertex names for already-spawned futures). The rules for the judgment are shown in Figure 19, and primarily require that graphs contained on any arrow are well-formed.

B PROOF OF LEMMA 3

Proof. We show the interesting cases for all parts except (1), which is standard.

(1) By induction on the derivation of $\Gamma, x : \tau'; \Delta; \Omega; \Psi \vdash e : \tau \mid G$.
(2) By induction on the derivation of $\Delta, \gamma : \kappa'; \Omega; \Psi \vdash G : \kappa$.
   - DW:RecPi. Then $G = \mu\gamma'.\Pi\vec{u}_f; \vec{u}_t.G''$ and $\Delta, \gamma : \kappa', \gamma' : \Pi\vec{u}_f; \vec{u}_t.*; \vec{u}_f; \Psi, \vec{u}_t \vdash G'' : *$. By exchange and induction, $\Delta, \gamma' : \Pi\vec{u}_f; \vec{u}_t.*; \vec{u}_f; \Psi, \vec{u}_t \vdash G''[G'/\gamma] : *$. Apply rule DW:RecPi.
(3) By induction on the derivation of $\Delta, \gamma : \kappa; \Psi \vdash \tau$ ok.
   - T:Fun Then $\tau = \Pi\vec{u}_f; \vec{u}_t.\tau_1 \xrightarrow{G'} \tau_2$. By induction, $\Delta; \Psi \vdash \tau_1[G/\gamma]$ ok and $\Delta; \Psi \vdash \tau_2[G/\gamma]$ ok. By part 2, $\Delta; \cdot; \Psi \vdash G'[G/\gamma] : \Pi\vec{u}_f; \vec{u}_t.*$. Apply rule T:Fun.
(4) By induction on the derivation of $\Gamma; \Delta, \gamma : \kappa; \Omega; \Psi \vdash e : \tau \mid G$.
   - S:Fun. By induction, $\Gamma, f : \Pi\vec{u}_f; \vec{u}_t.\tau_1 \xrightarrow{\gamma'} \tau_2, x : \tau_1; \Delta, \gamma' : \Pi\vec{u}_f; \vec{u}_t.*; \vec{u}_f; \Psi, \vec{u}_t \vdash e : \tau_2[G'/\gamma] \mid G[G'/\gamma]$. By part 3, $\Delta; \Psi, \vec{u}_t \vdash \tau_1[G'/\gamma]$ ok and $\Delta; \Psi, \vec{u}_t \vdash \tau_2[G'/\gamma]$ ok. Apply rule S:Fun.
   - S:App. By induction,

$$\Gamma; \Delta; \Omega_1; \Psi, \vec{u}_t' \vdash e_1 : \Pi\vec{u}_f; \vec{u}_t.\tau_1 \xrightarrow{G_3} \tau_2[G'/\gamma] \mid G_1[G'/\gamma]$$

and

$$\Gamma; \Delta; \Omega_2; \Psi, \vec{u}'_t \vdash e_2 : \tau_1[\vec{u}'_t/\vec{u}_t][G'/\gamma] \mid G_2[G'/\gamma]$$

We have

$$\Pi\vec{u}_f;\vec{u}_t.\tau_1 \xrightarrow{G_3} \tau_2[G'/\gamma] = \Pi\vec{u}_f;\vec{u}_t.\tau_1[G'/\gamma] \xrightarrow{G_3[G'/\gamma]} \tau_2[G'/\gamma]$$

and $\tau_2[G'/\gamma][\vec{u}'_t/\vec{u}_t] = \tau_2[\vec{u}'_t/\vec{u}_t][G'/\gamma]$. We have

$$\mathsf{unroll}(G_3[G'/\gamma])[\vec{u}'_f;\vec{u}'_t] = \mathsf{unroll}(G_3)[\vec{u}'_f;\vec{u}'_t][G'/\gamma]$$

Apply rule S:App.

(5) By induction on the derivation of $\Delta; \Omega, \vec{u}_f; \Psi, \vec{u}_t \vdash G : \kappa$.

- DW:Pi. Then $G = \Pi\vec{u}''_f;\vec{u}''_t.G'$ and $\Delta; \Omega, \vec{u}_f, \vec{u}''_f; \Psi, \vec{u}_t, \vec{u}''_t \vdash G' : *$. Without loss of generality, assume that $\vec{u}''_f$ is disjoint from both $\vec{u}_f$ and $\vec{u}'_f$ and that $\vec{u}''_t$ is disjoint from both $\vec{u}_t$ and $\vec{u}'_t$. By induction, $\Delta; \Omega, \vec{u}'_f, \vec{u}''_f; \Psi, \vec{u}'_t, \vec{u}''_t \vdash G[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t] : *$. Apply rule DW:Pi.
- DW:App. Then $G = G'[\vec{u}'''_f;\vec{u}'''_t]$ and $\Delta; \Omega, \vec{u}_f; \Psi, \vec{u}_t, \vec{u}'''_t \vdash G' : \Pi\vec{u}''_f;\vec{u}''_t.\kappa$. Without loss of generality, assume that $\vec{u}''_f$ is disjoint from both $\vec{u}_f$ and $\vec{u}'_f$ and that $\vec{u}''_t$ is disjoint from both $\vec{u}_t$ and $\vec{u}'_t$. By induction, $\Delta; \Omega, \vec{u}'_f; \Psi, \vec{u}'_t, \vec{u}^{\circ}_t \vdash G[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t] : \Pi\vec{u}''_f;\vec{u}''_t.\kappa$, where $\vec{u}^{\circ}_t = \vec{u}'''_t[\vec{u}'_t/\vec{u}_t]$. We have $G[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t] = (G'[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t])[\vec{u}^{\circ}_f;\vec{u}^{\circ}_t]$ where $\vec{u}^{\circ}_f = \vec{u}'''_f[\vec{u}'_f/\vec{u}_f]$. Apply DW:App.

(6) By induction on the derivation of $\Delta; \Psi, \vec{u}_t \vdash \tau$ ok.

- T:Fun. Then $\tau = \Pi\vec{u}''_f;\vec{u}''_t.\tau_1 \xrightarrow{G} \tau_2$ and $\Delta; \Psi, \vec{u}_t, \vec{u}''_t \vdash \tau_1$ ok and $\Delta; \Psi, \vec{u}_t, \vec{u}''_t \vdash \tau_2$ ok and $\Delta; \cdot; \Psi, \vec{u}_t \vdash G : \Pi\vec{u}''_f;\vec{u}''_t.*$, Without loss of generality, assume $\vec{u}''_t$ is disjoint from $\vec{u}_t$ and $\vec{u}'_t$. By induction, $\Delta; \Psi, \vec{u}'_t, \vec{u}''_t \vdash \tau_1[\vec{u}'_t/\vec{u}_t]$ ok and $\Delta; \Psi, \vec{u}_t, \vec{u}''_t \vdash \tau_2[\vec{u}'_t/\vec{u}_t]$ ok. By part 5, $\Delta; \cdot; \Psi, \vec{u}'_t \vdash G[\vec{u}'_t/\vec{u}_t] : \Pi\vec{u}''_f;\vec{u}''_t.*$. Apply rule T:Fun.

(7) By induction on the derivation of $\Gamma; \Delta; \Omega, \vec{u}_f; \Psi, \vec{u}_t \vdash e : \tau \mid G$.

- S:Fun. Then $e = \mathsf{fun}[\vec{u}_f;\vec{u}_t]\ f\ x = e'$ and

$$\Gamma, f : \Pi\vec{u}''_f;\vec{u}''_t.\tau_1 \xrightarrow{\gamma} \tau_2, x : \tau_1; \Delta, \gamma : \Pi\vec{u}''_f;\vec{u}''_t.*; \vec{u}''_f; \Psi, \vec{u}_t, \vec{u}''_t \vdash e' : \tau_2 \mid G'$$

and $\Delta; \Psi, \vec{u}_t, \vec{u}''_t \vdash \tau_1$ ok and $\Delta; \Psi, \vec{u}_t, \vec{u}''_t \vdash \tau_2$ ok. Without loss of generality, assume that $\vec{u}''_f$ is disjoint from both $\vec{u}_f$ and $\vec{u}'_f$ and that $\vec{u}''_t$ is disjoint from both $\vec{u}_t$ and $\vec{u}'_t$. We have $e[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t] = \mathsf{fun}[\vec{u}''_f;\vec{u}''_t]\ f\ x = e'[\vec{u}'_t/\vec{u}_t]$. By induction, $\Gamma, f : \Pi\vec{u}''_f;\vec{u}''_t.\tau_1 \xrightarrow{\gamma} \tau_2, x : \tau_1; \Delta, \gamma : \Pi\vec{u}''_f;\vec{u}''_t.*; \vec{u}''_f; \Psi, \vec{u}'_t, \vec{u}''_t \vdash \mathsf{fun}[\vec{u}''_f;\vec{u}''_t]\ f\ x = e'[\vec{u}'_t/\vec{u}_t] : \tau_2[\vec{u}'_t/\vec{u}_t] \mid G'[\vec{u}'_t/\vec{u}_t]$. By part 6, $\Delta; \Psi, \vec{u}'_t, \vec{u}''_t \vdash \tau_1[\vec{u}'_t/\vec{u}_t]$ ok and $\Delta; \Psi, \vec{u}'_t, \vec{u}''_t \vdash \tau_2[\vec{u}'_t/\vec{u}_t]$ ok. We have $\tau[\vec{u}'_t/\vec{u}_t] = \Pi\vec{u}''_f;\vec{u}''_t.\tau_1[\vec{u}'_t/\vec{u}_t] \xrightarrow{\mu\gamma.\Pi\vec{u}_f;\vec{u}_t.G'[\vec{u}'_t/\vec{u}_t]} \tau_[\vec{u}'_t/\vec{u}_t]$. Apply rule S:Fun.
- S:App. Then $e = e_1[\vec{u}'''_f;\vec{u}'''_t]\ e_2$ and $\tau = \tau_2[\vec{u}'''_t/\vec{u}''_t]$ and

$$\Gamma; \Delta; \Omega_1, \vec{u}_{f1}; \Psi, \vec{u}_t, \vec{u}'''_t \vdash e_1 : \Pi\vec{u}''_f;\vec{u}''_t.\tau_1 \xrightarrow{G_3} \tau_2 \mid G_1$$

and

$$\Gamma; \Delta; \Omega_2, \vec{u}_{f2}; \Psi, \vec{u}_t, \vec{u}'''_t \vdash e_2 : \tau_1[\vec{u}'''_t/\vec{u}''_t] \mid G_2$$

where $\vec{u}_f = \vec{u}_{f1}, \vec{u}_{f2}$. Let $\vec{u}'_{f1}$ and $\vec{u}'_{f2}$ be the equivalent projections of $\vec{u}'_f$. Without loss of generality, assume that $\vec{u}''_f$ is disjoint from both $\vec{u}_f$ and $\vec{u}'_f$ and that $\vec{u}''_t$ is disjoint from both $\vec{u}_t$ and $\vec{u}'_t$. By induction,

$$\Gamma; \Delta; \Omega_1, \vec{u}'_{f1}; \Psi, \vec{u}'_t, \vec{u}'''_t \vdash e_1[\vec{u}'_{f1}/\vec{u}_{f1}][\vec{u}'_t/\vec{u}_t] : \Pi\vec{u}''_f;\vec{u}''_t.\tau_1[\vec{u}'_t/\vec{u}_t] \xrightarrow{G_3[\vec{u}'_t/\vec{u}_t]} \tau_2[\vec{u}'_t/\vec{u}_t] \mid G_1[\vec{u}'_{f1}/\vec{u}_{f1}][\vec{u}'_t/\vec{u}_t]$$

and

$$\Gamma; \Delta; \Omega_2, \vec{u}'_{f2}; \Psi, \vec{u}'_t, \vec{u}'''_t \vdash e_2[\vec{u}'_{f2}/\vec{u}_{f2}][\vec{u}'_t/\vec{u}_t] : \tau_1[\vec{u}'''_t/\vec{u}''_t][\vec{u}'_t/\vec{u}_t] \mid G_2[\vec{u}'_{f2}/\vec{u}_{f2}][\vec{u}'_t/\vec{u}_t]$$

We have $\tau_1[\vec{u}'''_t/\vec{u}''_t][\vec{u}'_t/\vec{u}_t] = \tau_1[\vec{u}'_t/\vec{u}_t][\vec{u}^\circ_t/\vec{u}''_t]$ where $\vec{u}^\circ_t = \vec{u}'''_t[\vec{u}'_t/\vec{u}_t]$. We have $\tau[\vec{u}'_t/\vec{u}_t] = \tau_2[\vec{u}'_t/\vec{u}_t][\vec{u}^\circ_t/\vec{u}''_t]$ and

$$e[\vec{u}'_t/\vec{u}_t] = (e_1[\vec{u}'_{f1}/\vec{u}_{f1}][\vec{u}'_t/\vec{u}_t])[\vec{u}^\circ_f; \vec{u}^\circ_t]\ e_2[\vec{u}'_{f2}/\vec{u}_{f2}][\vec{u}'_t/\vec{u}_t]$$

where $\vec{u}^\circ_f = \vec{u}'''_f[\vec{u}'_f/\vec{u}_f]$. We have

$$\begin{array}{rl} & G[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t] \\ = & G_1[\vec{u}'_{f1}/\vec{u}_{f1}][\vec{u}'_t/\vec{u}_t] \oplus G_2[\vec{u}'_{f2}/\vec{u}_{f2}][\vec{u}'_t/\vec{u}_t] \oplus (\mathsf{unroll}(G_3)[\vec{u}'''_f; \vec{u}'''_t])[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t] \\ = & G_1[\vec{u}'_{f1}/\vec{u}_{f1}][\vec{u}'_t/\vec{u}_t] \oplus G_2[\vec{u}'_{f2}/\vec{u}_{f2}][\vec{u}'_t/\vec{u}_t] \oplus \mathsf{unroll}(G_3[\vec{u}'_t/\vec{u}_t])[\vec{u}^\circ_f; \vec{u}^\circ_t] \end{array}$$

Apply rule S:App.

- S:New. Then $e = \text{new } u.e'$ and $\Gamma; \Delta; \Omega, \vec{u}_f, u; \Psi, \vec{u}_t, u \vdash e : \tau \mid G'$ where $G = \nu u.G'$ and $u \notin \mathsf{FV}(\tau)$ and $u \notin \Omega, \vec{u}_f, \Psi, \vec{u}_t$. Without loss of generality, assume $u \notin \vec{u}'_f, \vec{u}'_t$. We have $e[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t] = \text{new } u.e'[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t]$ and $G[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t] = \nu u.e'[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t]$. By induction, $\Gamma; \Delta; \Omega, \vec{u}'_f, u; \Psi, \vec{u}'_t, u \vdash e'[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t] : \tau[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t] \mid G[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t]$.

□

C PROOF OF LEMMA 4

Proof. By induction on the derivation of $\Gamma; \Delta; \Omega; \Psi \vdash e : \tau \mid G$.
We prove some representative cases.

- S:Fun. Then $\Delta; \Psi, \vec{u}_t \vdash \tau_1$ ok and $\Delta; \Psi, \vec{u}_t \vdash \tau_2$ ok and $\Gamma, f : \Pi \vec{u}_f; \vec{u}_t.\tau_1 \xrightarrow{\gamma} \tau_2, x : \tau_1; \Delta, \gamma : \Pi \vec{u}_f; \vec{u}_t.\kappa; \vec{u}_f; \Psi, \vec{u}_t \vdash e : \tau_2 \mid G$. By induction, $\Delta, \gamma : \Pi \vec{u}_f; \vec{u}_t.\kappa; \vec{u}_f; \Psi, \vec{u}_t \vdash G : *$. By rule DW:RecPi, $\Delta; \cdot; \Psi \vdash \mu\gamma.\Pi \vec{u}_3; \vec{u}_t.G : \Pi \vec{u}_f; \vec{u}_t.*$. Apply rule T:Fun.
- S:App. Then $\Gamma; \Delta; \Omega_1; \Psi, \vec{u}'_t \vdash e_1 : \Pi \vec{u}_f; \vec{u}_t.\tau_1 \xrightarrow{G_3} \tau_2 \mid G_1$ and $\Gamma; \Delta; \Omega_2; \Psi, \vec{u}'_t \vdash e_2 : \tau_1 \mid G_2$. By induction, $\Delta; \Psi, \vec{u}'_t \vdash \Pi \vec{u}_f; \vec{u}_t.\tau_1 \xrightarrow{G_3} \tau_2$ ok and $\Delta; \Omega_1; \Psi, \vec{u}'_t \vdash G_1 : *$ and $\Delta; \Psi, \vec{u}'_t \vdash \tau_1$ ok and $\Delta; \Omega_2; \Psi, \vec{u}'_t \vdash G_2 : *$. By inversion on T:Fun, $\Delta; \Psi, \vec{u}_t \vdash \tau_2$ ok and $\Delta; \cdot; \Psi \vdash G_3 : \Pi \vec{u}_f; \vec{u}_t.*$. By DW:App, $\Delta; \vec{u}'_f; \Psi, \vec{u}'_t \vdash \mathsf{unroll}(G_3)[\vec{u}'_f; \vec{u}'_t] : *$. Apply DW:Seq.
- S:Case. Then $\Gamma; \Delta; \Omega_1; \Psi \vdash e_1 : \tau_1 + \tau_2 \mid G_1$ and $\Gamma, x : \tau_1; \Delta; \Omega_2; \Psi \vdash e_2 : \tau' \mid G_2$ and $\Gamma, y : \tau_2; \Delta; \Omega_2; \Psi \vdash e_3 : \tau' \mid G_3$. By induction, $\Delta; \Omega_1; \Psi \vdash G_1 : *$ and $\Delta; \Psi \vdash \tau'$ ok and $\Delta; \Omega_2; \Psi \vdash G_2 : *$ and $\Delta; \Omega_2; \Psi \vdash G_3 : *$ By DW:Or, $\Delta; \Omega_2; \Psi \vdash G_2 \vee G_3 : *$. Apply DW:Seq.
- S:Future. Then $\Gamma; \Delta; \Omega; \Psi \vdash e : \tau \mid G$. By induction, $\Delta; \Psi \vdash \tau$ ok and $\Delta; \Omega; \Psi \vdash G : *$. By T:Fut, $\Delta; \Psi, u \vdash \tau$ future$[u]$ ok. By DW:Spawn, $\Delta; \Omega, u; \Psi, u \vdash G \swarrow_u : *$.
- S:Touch. Then $\Gamma; \Delta; \Omega; \Psi, u \vdash e : \tau$ future$[u] \mid G$. By induction, $\Delta; \Psi, u \vdash \tau$ future$[u]$ ok and $\Delta; \Omega; \Psi, u \vdash G : *$. By inversion on T:Fut, $\Delta; \Psi, u \vdash \tau$ ok. By DW:Touch, $\Delta; \Omega; \Psi, u \vdash {}^{u}\searrow : *$. Apply DW:Seq.
- S:NewF. Then $\Gamma; \Delta; \Omega, u; \Psi, u \vdash e : \tau \mid G$ and $u \notin \mathsf{FV}(\tau)$. By induction, $\Delta; \Psi, u \vdash \tau$ ok, so $\Delta; \Psi \vdash \tau$ ok, and $\Delta; \Omega, u; \Psi, u \vdash G : *$. Apply DW:New.

□

D PROOF OF THEOREM 3

Proof. By induction on the derivation of $e \Downarrow v \mid g$.

- C:App. Then $e = e_1[\vec{u}'_f; \vec{u}'_t]e_2$ and $e_1 \Downarrow \mathsf{fun}[\vec{u}_f; \vec{u}_t]\ f\ x = e_0 \mid g_1$ and $e_2 \Downarrow v \mid g_2$ and

$$e_0[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t][v/x][\mathsf{fun}[\vec{u}_f; \vec{u}_t]\ f\ x = e_0/f] \Downarrow v' \mid g_3$$

By inversion on S:App, $\cdot;\cdot;\Omega_1;\Psi \vdash e_1 : \Pi\vec{u}_f;\vec{u}_t.\tau_1 \xrightarrow{G_3} \tau_2 \mid G_1$ and $\cdot;\cdot;\Omega_2;\Psi \vdash e_2 : \tau_1 \mid G_2$ where $\Omega = \Omega_1, \Omega_2, \vec{u}'_f$.

By induction, $\cdot;\cdot;\Omega_1;\Psi \vdash \mathsf{fun}[\vec{u}_f;\vec{u}_t]\ f\ x = e_0 : \Pi\vec{u}_f;\vec{u}_t.\tau_1 \xrightarrow{G_3} \tau_2 \mid \bullet$ and $\cdot;\cdot;\Omega_2;\Psi \vdash v : \tau_1 \mid \bullet$ and $g_1 \in \mathit{Norm}_{n_1}(G_1)$ and $g_2 \in \mathit{Norm}_{n_2}(G_2)$.

By inversion on S:Fun, $f : \Pi\vec{u}_f;\vec{u}_t.\tau_1 \xrightarrow{\gamma} \tau_2, x : \tau_1; \gamma : \Pi\vec{u}_f;\vec{u}_t.*;\vec{u}_f;\Psi,\vec{u}_t \vdash e_0 : \tau_2 \mid G$ and $G_3 = \mu\gamma.\Pi\vec{u}_f;\vec{u}_t.G$. By Lemma 3,

$$\cdot;\cdot;\vec{u}'_f;\Psi,\vec{u}'_t \vdash e_0[\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t][v/x][\mathsf{fun}[\vec{u}_f;\vec{u}_t]\,f\,x = e_0/f] : \tau_2[\vec{u}'_t/\vec{u}_t] \mid G[\Pi\vec{u}_f;\vec{u}_t.G/\gamma][\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t]$$

By induction, $\cdot;\cdot;\vec{u}'_f;\Psi,\vec{u}'_t \vdash v' : \tau_2[\vec{u}'_t/\vec{u}_t] \mid \bullet$ and $g_3 \in \mathit{Norm}_{n_3}(G[\Pi\vec{u}_f;\vec{u}_t.G/\gamma][\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t])$. By Lemma 1, we have

$$\begin{array}{rcl} g_1 \oplus g_2 \oplus g_3 & \in & \mathit{Norm}_{\max(n_1,n_2,n_3)}(G_1 \oplus G_2 \oplus G[\Pi\vec{u}_f;\vec{u}_t.G/\gamma][\vec{u}'_f/\vec{u}_f][\vec{u}'_t/\vec{u}_t]) \\ & = & \mathit{Norm}_{\max(n_1,n_2,n_3)}(G_1 \oplus G_2 \oplus \mathsf{unroll}(G_3)[\vec{u}'_f;\vec{u}'_t]) \end{array}$$

- C:Pair. Then $e = (e_1, e_2)$ and $v = (v_1, v_2)$ and $g = g_1 \oplus g_2$ and $e_1 \Downarrow v_1 \mid g_1$ and $e_2 \Downarrow v_2 \mid g_2$. By inversion on S:Pair, $\cdot;\cdot;\Omega_1;\Psi \vdash e_1 : \tau_1 \mid G_1$ and $\cdot;\cdot;\Omega_2;\Psi \vdash e_2 : \tau_2 \mid G_2$ and $\tau = \tau_1 \times \tau_2$ and $G = G_1 \oplus G_2$. By induction, $\cdot;\cdot;\Omega_1;\Psi \vdash v_1 : \tau_1 \mid \bullet$ and $\cdot;\cdot;\Omega_2;\Psi \vdash v_2 : \tau_2 \mid \bullet$ and $g_1 \in \mathit{Norm}_{n_1}(G_1)$ and $g_2 \in \mathit{Norm}_{n_2}(G_2)$. By rule S:Pair, we have $\cdot;\cdot;\Omega_1;\Psi \vdash e : \tau \mid \bullet$. By Lemma 1, we have $g \in \mathit{Norm}_{\max(n_1,n_2)}(G)$.
- C:CaseL. Then $e = \mathsf{case}\ e_1\ \{x.e_2; y.e_3\}$ and $g = g_1 \oplus g_2$ and $e_1 \Downarrow \mathsf{inl}\ v' \mid g_1$ and $e_2[v'/x] \Downarrow v \mid g_2$. By inversion on S:Case, $\cdot;\cdot;\Omega_1;\Psi \vdash e_1 : \tau_1 + \tau_2 \mid G_1$ and $x : \tau_1;\cdot;\Omega_2;\Psi \vdash e_2 : \tau \mid G_2$ and $G = G_1 \oplus (G_2 \vee G_3)$. By induction, $\cdot;\cdot;\Omega_1;\Psi \vdash \mathsf{inl}\ v' : \tau_1 + \tau_2 \mid \bullet$ and $g_1 \in \mathit{Norm}_{n_1}(G_1)$. By inversion on S:InL, $\cdot;\Gamma;\Omega_1;\Psi \vdash v' : \tau_1 \mid \bullet$. By Lemma 3, $\cdot;\cdot;\Omega_2;\Psi \vdash e_2[v'/x] : \tau \mid G_2$. By induction, and $\cdot;\cdot;\Omega_2;\Psi \vdash v : \tau \mid \bullet$ and $g_2 \in \mathit{Norm}_{n_2}(G_2)$. By Lemma 1, $g \in \mathit{Norm}_{\max(n_1,n_2)}(G)$.
- C:Future. Then $e = \mathsf{future}[u]\ e'$ and $g = g' \swarrow_u$ and $e' \Downarrow v \mid g'$. By inversion on S:Future, we have $\cdot;\cdot;\Omega';\Psi \vdash e' : \tau \mid G'$ where $\Omega = \Omega, u$ and $G = G' \swarrow_u$. By induction, $\cdot;\cdot;\Omega';\Psi \vdash v : \tau \mid \bullet$ and $g' \in \mathit{Norm}_n(G')$. By the definition of normalization, we have $g \in \mathit{Norm}_n(G)$.
- C:Touch. Then $e = \mathsf{touch}\ e'$ and $g = g' \oplus {}^u\!\searrow$ and $e' \Downarrow \mathsf{future}[u]\ v \mid g'$. By inversion on S:Touch, $\cdot;\cdot;\Omega;\Psi \vdash e' : \tau\ \mathsf{future}[u] \mid G'$ where $G = G' \oplus {}^u\!\searrow$. By induction, $\cdot;\cdot;\Omega;\Psi \vdash v : \tau\ \mathsf{future}[u] \mid \bullet$ and $g' \in \mathit{Norm}_n(G')$. By the definition of normalization, we have ${}^u\!\searrow \in \mathit{Norm}_n({}^u\!\searrow)$, so by Lemma 1, $g \in \mathit{Norm}_n(G)$.
- C:New. Then $e = \mathsf{new}\ u.e'$ and $g = g'[u'/u]$ and $e \Downarrow v \mid g'$. By inversion on S:New, $\cdot;\cdot;\Omega,u;\Psi,u \vdash e : \tau \mid G'$ and $u \notin \mathsf{FV}(\tau)$ and $G = \nu u.G'$. By induction, $\cdot;\cdot;\Omega,u;\Psi,u \vdash v : \tau \mid \bullet$ and $g' \in \mathit{Norm}_n(G')$. We have $\mathit{Norm}_n(G) = \mathit{Norm}_n(G'[u'/u])$. By Lemma 1, $g \in \mathit{Norm}_n(G)$.

□

E LIMITATIONS OF THE INFERENCE ALGORITHM

The only major limitation we are aware of on the programs for which our algorithm can infer annotations is the one mentioned above: if variable (e.g., a function argument) f has function type, its inferred type when applied will be $\Pi\emptyset;\vec{u}_t.\tau_1 \xrightarrow{\gamma} \tau_2$, where $\vec{u}_t$ is the set of vertices currently known to be free in the argument. For example, in Figure 20a, the function `mytouch` will not be assigned a Π type because its argument `ft` would not yet have been given a future type. This will result in a type error as written, but can easily be fixed with one type annotation (Figure 20b). Inferring $\emptyset$ as the vertices spawned by the function is a more fundamental (and necessary) limitation. For example, in Figure 20c, `myspawn` takes a function and returns a future, but there is no way to know locally whether it spawns and returns a future that runs the function, returns an already-spawned future it has captured, spawns two futures and returns one depending on the result of the function, etc.

```
let f mytouch =
  let callmt ft =
    mytouch ft
  in
  let fut = future 42 in
  callmt fut
```

a

```
let f mytouch =
  let callmt (ft: 'a future) =
    mytouch ft
  in
  let fut = future 42 in
  callmt fut
```

b

```
let f myspawn =
  let fut =
    myspawn (fun _ -> 42)
  in
  touch fut
```

c

Fig. 20. Programs with functional arguments that accept or return futures.

Therefore, there is very little we can algorithmically infer about function-type variables that return futures (either about the annotated return type or the function's graph type). Fortunately, in our experience, this situation rarely occurs in realistic programs.

F FORMALIZATION OF FORK-JOIN TRANSFORMATION

The transformation algorithm is defined formally in Figure 21 using the judgment $\Gamma; S \vdash \epsilon \rightsquigarrow \_, \_ \triangleright (\_, \epsilon')$ meaning that, under context Γ and spine S, the code ϵ transforms into the code ϵ'. If $\_, \_ \triangleright (\_, \epsilon') = \_, \_ \blacktriangleright (\_, \epsilon')$, then the code is purely fork-join (this fact is used when spawning a new future to know whether the code of the future is purely fork-join). Rules P:EndPure and P:EndNotPure apply when we have reached the end of the code; they reconstruct the spine into a piece of code that is marked as pure if and only if the spine is empty (that is, if there are no outstanding futures that were not fork-join). Rules P:TouchPar applies when we can create a `par`: the newly created `par` is added to the spine when we recursively process the rest of the code. If the touch does not target the last future spawned or that future is not purely fork-join, P:TouchNotPar reconstructs the spine into a piece of code that is passed as the new spine when we recursively process the rest of the code. Regardless of whether the rest of the code is purely fork-join, P:TouchNotPar markes the output as not pure. Finally, rule P:Future adds a newly spawned future to the spine. The rule recursively transforms the code in the future before adding it to the spine; if the code in the future is purely fork-join, it is added as a pure spine entry.

The "Reconstruct" and "MkPar" operations are also defined in Figure 21. The operation $\mathit{Reconstruct}(S)$ traverses the spine S, turning the entries back into the code that spawns the futures, intermixed with the code along the "trunk". The operation $\mathit{MkPar}(\epsilon_1, \epsilon_2, x)$ makes a `par` out of a future and a piece of the trunk, where x is the variable that received the value of the future. After making the actual `par`, x should get the left component of the resulting pair; the other component goes into a fresh temporary variable t_1. The right component of the `par` needs to return any variables defined in the trunk ϵ_2, as these could be used in the future. These are collected into a tuple, returned into t_1; this temporary variable is then deconstructed using the original variable names, thus leaving those variables bound to the expected value in the continuation of the trunk.

(P:EndPure)

$$\frac{}{\Gamma; \_, \_ \triangleright (\_, \epsilon) \vdash (x, e, \tau) :: [] \rightsquigarrow \_, \_ \blacktriangleright (\_, \epsilon :: (x, e, \tau) :: [])}$$

(P:EndNotPure)

$$\frac{S \neq \_, \_ \triangleright (\_, \cdot)}{\Gamma; S \vdash (x, e, \tau) :: [] \rightsquigarrow \_, \_ \triangleright (\_, \mathit{Reconstruct}(S) :: (x, e, \tau) :: [])}$$

(P:TouchPar)

$$\frac{\Gamma; u', x_2 \triangleright (\epsilon_3, \mathit{MkPar}(\epsilon_1, \epsilon_2, y) :: \epsilon_4) :: S \vdash \epsilon \rightsquigarrow \_, \_ \triangleright (\_, \epsilon') \qquad \Gamma(x) = u \ \mathsf{future}[\tau]}{\Gamma; u, x_1 \blacktriangleright (\epsilon_1, \epsilon_2) :: u', x_2 \triangleright (\epsilon_3, \epsilon_4) :: S \vdash (y, \mathsf{touch}\ x, \tau) :: \epsilon \rightsquigarrow \_, \_ \triangleright (\_, \epsilon')}$$

(P:TouchNotPar)

$$\frac{\begin{array}{c}\Gamma; \_, \_ \triangleright (\_, \mathit{Reconstruct}(u, x_2 \triangleright (\epsilon_1, (y, \mathsf{touch}\ x, \tau)\epsilon_2 ::) :: S)) \vdash \epsilon \rightsquigarrow \_, \_ \triangleright (\_, \epsilon') \\ \Gamma(x) = u' \ \mathsf{future}[\tau']\end{array}}{\Gamma; u, x_1 \triangleright (\epsilon_1, \epsilon_2) :: S \vdash (y, \mathsf{touch}\ x, \tau) :: \epsilon \rightsquigarrow \_, \_ \triangleright (\_, \epsilon')}$$

(P:Future)

$$\frac{\Gamma; \_, \_ \triangleright (\_, []) \vdash \epsilon_1 \rightsquigarrow \_, \_ \triangleright (\_, \epsilon_1') \qquad \Gamma; u, x \triangleright (\epsilon_1', []) :: S \vdash \epsilon_2 \rightsquigarrow \_, \_ \triangleright (\_, \epsilon')}{\Gamma; S \vdash (x, \mathsf{future}[u]\ \epsilon_1, \tau) :: \epsilon_2 \rightsquigarrow \_, \_ \triangleright (\_, \epsilon')}$$

$$\frac{}{\mathit{Reconstruct}(\_, \_ \triangleright (\_, \epsilon)) = \_, \_ \triangleright (\_, \epsilon)}$$

$$\frac{\mathit{Reconstruct}(S) = \_, \_ \triangleright (\_, \epsilon)}{\mathit{Reconstruct}(u, x \triangleright (\epsilon_1, \epsilon_2) :: S) = \_, \_ \triangleright (\_, (x, \mathsf{future}[u]\ \epsilon_1,) :: \epsilon_2 :: \epsilon)}$$

$$\frac{\mathsf{FV}(\epsilon_2) = (x_1, \ldots, x_n) \qquad t_1}{\mathit{MkPar}(\epsilon_1, \epsilon_2, y) = ((y, t_1), \mathsf{par}(\epsilon_1, t_2 :: (x_1, \ldots, x_n))) :: ((x_1, \ldots, x_n), t_1) :: []}$$

Fig. 21. Selected rules for transforming fork-join code into pars.