Pipelines and Beyond: Graph Types for ADTs with Futures

FRANCIS RINALDI, Illinois Institute of Technology, USA
JUNE WUNDER, Boston University, USA
ARTHUR AZEVEDO DE AMORIM, Rochester Institute of Technology, USA
STEFAN K. MULLER, Illinois Institute of Technology, USA

Parallel programs are frequently modeled as *dependency* or *cost* graphs, which can be used to detect various bugs, or simply to visualize the parallel structure of the code. However, such graphs reflect just one particular execution and are typically constructed in a *post-hoc* manner. *Graph types*, which were introduced recently to mitigate this problem, can be assigned statically to a program by a type system and compactly represent the family of all graphs that could result from the program.

Unfortunately, prior work is restricted in its treatment of *futures*, an increasingly common and especially dynamic form of parallelism. In short, each instance of a future must be statically paired with a vertex name. Previously, this led to the restriction that futures could not be placed in collections or be used to construct data structures. Doing so is not a niche exercise: such structures form the basis of numerous algorithms that use forms of pipelining to achieve performance not attainable without futures. All but the most limited of these examples are out of reach of prior graph type systems.

In this paper, we propose a graph type system that allows for almost arbitrary combinations of futures and recursive data types. We do so by indexing datatypes with a type-level *vertex structure*, a codata structure that supplies unique vertex names to the futures in a data structure. We prove the soundness of the system in a parallel core calculus annotated with vertex structures and associated operations. Although the calculus is annotated, this is merely for convenience in defining the type system. We prove that it is possible to annotate arbitrary recursive types with vertex structures, and show using a prototype inference engine that these annotations can be inferred from OCaml-like source code for several complex parallel algorithms.

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, futures, pipelining, affine type system

ACM Reference Format:
Francis Rinaldi, june wunder, Arthur Azevedo de Amorim, and Stefan K. Muller. 2024. Pipelines and Beyond: Graph Types for ADTs with Futures. *Proc. ACM Program. Lang.* 8, POPL, Article 17 (January 2024), 30 pages. https://doi.org/10.1145/3632859

1 INTRODUCTION

Decades of work on reasoning about parallel programs have focused on *computation* or *cost graphs*, directed graphs that represent the dependencies of threads. Computation graphs are a convenient target for analysis because they abstract away details of the program, language, and even the parallelism features that were used, while still capturing enough information about the

Authors' addresses: Francis Rinaldi, frinaldi@hawk.iit.edu, Illinois Institute of Technology, Chicago, Illinois, USA; june wunder, jwunder@bu.edu, Boston University, Boston, Massachusetts, USA; Arthur Azevedo de Amorim, aaavcs@rit.edu, Rochester Institute of Technology, Rochester, New York, USA; Stefan K. Muller, smuller2@iit.edu, Illinois Institute of Technology, Chicago, Illinois, USA.

ACM 2475-1421/2024/1-ART17
https://doi.org/10.1145/3632859

relationships between threads to perform many useful analyses. For example, computation graphs have been used to study deadlock [Cogumbreiro et al. 2018], data races [Banerjee et al. 2006], priority inversions [Babaoğlu et al. 1993] and evaluation cost [Blelloch and Greiner 1995, 1996].

To analyze such properties, it is desirable to calculate the computation graph of a program statically, at compile time or analysis time. Doing so is often possible in languages and threading libraries for *coarse-grained parallelism*, such as `pthreads`, where thread creation and synchronization are expensive and rare. Much recent interest in parallel programming, however, has been in the area of *fine-grained parallelism*, in which threads are created cheaply and eagerly, often based on runtime conditions. For example, a program might fork at each level of a divide-and-conquer algorithm, or a web server might spawn a new thread to handle every incoming request asynchronously. Reasoning statically about the dependency structure of fine-grained parallel programs is difficult because of the highly dynamic nature of thread creation and synchronization in these programs.

This difficulty is compounded when programs use *futures* and related abstractions for fine-grained parallelism, which are becoming increasingly popular and have been made available in Python, Scala, Rust, and the most recent release of OCaml [Sivaramakrishnan et al. 2020], among other languages. Essentially, a future is a first-class handle to an asynchronous computation. The result of the computation can be demanded via a *force* or *touch* operation, which blocks if the result is not yet available. Because futures run in separate threads, we can model each future as its own vertex u in the computation graph of a program. Edges leading into u track the intermediate results used to compute the future, and when we touch the future, we add an edge from u to the thread where the touch happened. Futures may be passed around a program arbitrarily and end up being touched in a very different part of the program from where it was spawned, leading to great power and flexibility but also complex computation graphs which are difficult to reason about.

To address the difficulty of predicting parallel dependences in fine-grained parallel programs, especially those with futures, Muller [2022] introduced the notion of *graph types*, which statically overapproximate the set of computation graphs that might result from running a program. A *graph type system* statically assigns graph types to programs, and its soundness theorem ensures that the actual computation graph resulting from any execution of a well-typed program is described by the program's graph type.

Much of the complexity of the graph type system centers around futures. Because futures can be touched in an entirely different part of the program from where they are created, each future type is annotated with a distinguished *vertex name*, so that the graph type system can refer to the correct vertex when tracking the dependencies of touch operations. (Explicit vertex names are not needed in simpler parallelism models such as fork-join, because it is clear what thread is being synchronized.) To avoid tracking spurious dependencies, the graph type system ensures that each vertex name is associated with at most one future during execution. More precisely, when spawning a new future, the graph type system annotates the type of the result with a fresh vertex name, which is tracked in a separate affine context to prevent reuse.

This treatment of futures leads to a significant limitation in prior work: it is difficult or impossible to build useful data structures containing futures. Even an expression as simple as `[future e1; future e2]` (a list containing two new futures) cannot be assigned a type. The reason is that the two elements of this list must have types $\tau\ \mathsf{future}[u_1]$ and $\tau\ \mathsf{future}[u_2]$, respectively, where u_1 and u_2 are distinct vertex names and $\tau\ \mathsf{future}[u]$ is the type of a future returning a value of type τ with the vertex named u—these two elements can't be placed in a list because prior work supports only homogeneous lists. Although this example is simple and artificial, much of the power of futures, as opposed to more limited parallelism models such as fork-join, comes from the ability to program with data structures that contain an unbounded number of futures, such as lists and trees. As examples, Blelloch and Reid-Miller [1997] describe a number of algorithms and data structures that use futures in complex ways to pipeline computations, resulting in asymptotic improvements over the best known fork-join implementations. These programming idioms exercise the full complexity of futures, motivating the need for techniques to reason statically about the computation graphs of these programs.

In this paper, we develop a graph type system, and accompanying inference algorithm, that can handle complex data structures using futures. As a motivating example, consider a function that produces a pipeline of increasingly precise approximations of π. This could be, for example, the first stage in a graphics or simulation pipeline. We wish to compute the approximations asynchronously so that earlier approximations can be used while later ones are still being computed. Figure 1 shows two possible implementations of such a function. The implementation on the left produces a list of futures with the intermediate results. The function list_pi takes a number k and a *future* a, which computes the $(k-1)$st approximation. Each iteration of list_pi spawns a new future to compute the kth term of the Gregory series multiplied by 4, adds it to the running total being computed by a, and adds the new future (which is completing the new running total) to a list, then calls list_pi recursively to compute the remaining terms. To illustrate a use of this structure, the `main` function takes the second approximation from the list. Note that list_pi, as written, doesn't terminate.

Because the function list_pi produces a list of futures, it cannot be given a graph type under prior work [Muller 2022].[1] This is a shame, because its computation graph would have revealed a subtle but fatal bug: despite the futures, there is no real asynchrony or pipelining because almost the entire list of approximations (which, in this example, is infinite) must be constructed before the program proceeds. This can be seen in the visualization on the left side of Figure 2, which is produced automatically by our implementation from the inferred graph type. In the figure, vertices in the graph, notated with either a text label or a small circle, represent pieces of computation. The vertices with labels like $n1 \bullet 1$ are the final vertices of a future, and these labels are the vertex names assigned to the future. The reason for these particular labels will become clear later in the paper. Edges represent dependences: an edge out of a labeled vertex indicates a touch of the corresponding future, and other edges represent sequential dependences within a thread or the spawning of a future. A path of edges in the graph therefore represents a chain of sequential dependences and two vertices with no path between them indicate opportunities for parallelism. Long paths indicate a lack of parallelism.

The figure shows a visualization of the graph type of the program, with the recursion of `list_pi` unrolled a fixed number of times to make the recursive structure visually clear. A vertex labeled ... indicates a recursive call that has been elided because of the cutoff on number of unrollings. The vertex representing the `touch` operation in `main` is circled in red: we can see that there is a long chain of dependences on the critical path to reach this operation, which means the operation will be significantly delayed when running the program. Indeed, the topmost ... appears on the critical path, indicating a potentially (and, in this case, actually) infinite critical path.

The second implementation in Figure 1 instead uses a new data structure `'a pipe` which resembles a lazy list: the head of the list is computed eagerly and may be used immediately but the tail of the list is computed asynchronously in a future. The function `pipeline_pi` takes the running total `a` (now as an actual float, rather than a future) and `k`. It adds the `k`th approximation to the running total, then returns the new running total as well as a future to call `pipeline_pi` recursively to compute the remainder of the pipeline. This is reminiscent of the "producer" example of Blelloch and Reid-Miller [1997]. As we can see from the visualization on the right side of Figure 2, the graphs corresponding to `pipeline_pi` exhibit much more parallelism than the previous version. Here, the `touch` operation

[1]Actually, Muller [2022] does discuss a similar pipelining example in his system; cf. Figure 10 and Section 6. However, that example is expressible precisely because it does not accumulate the intermediate results in a list.

```
let rec list_pi (a, k) : float future list =
  let a' =
     future ((-1.0) ** (k +. 1.0)
             *. 4.0 /. (2. *. k -. 1.0)
             +. touch a)
  in
  a'::(list_pi (a', k +. 1))

let main () =
  touch (hd (tl (list_pi (0.0, 1.0))))
```

```
type 'a pipe = Pipe of 'a * 'a pipe future

let rec pipeline_pi (a, k) : float pipe =
  let a' = a +. (-1.0) ** (k +. 1.0)
           *. 4.0 /. (2. *. k -. 1.0)
  in
  Pipe (a', future (pipeline_pi (a', k +. 1.)))

let main () =
  let Pipe (_, f1) = pipeline_pi (0.0, 1.0) in
  let Pipe (pi2, _) = touch f1 in pi2
```

Fig. 1. Two implementations of a function that iteratively computes π with futures.

in `main` (again circled in red) occurs in parallel with the computation of the remainder of the list and there are only a small, finite number of operations on its critical path.

In this paper, we present a graph type system that can statically compute graph types for the above pipelining examples (and many more), thus allowing us to detect and repair the parallelism bugs we discussed. We present the type system in $\lambda^{G\mu}$, a core calculus containing both futures and recursive types. The key to our approach is parameterizing recursive data structures involving futures with a source of fresh vertex names called a *vertex structure* (or VS, for short). Conceptually, one can think of a VS as a separate structure of the same shape as the program's recursive data structure, containing unique vertex names. For example, both the `float future list` and the `float pipe` of Figure 1 would be parameterized by a stream of vertices. The two functions `list_pi` and `pipeline_pi` would take this vertex stream, let's call it U, as an implicit parameter and, at each iteration, use the next vertex in the stream (fst U) to spawn the new future and pass the rest of the stream (snd U) to the recursive call. As a result, the returned list (resp., pipe) will be "zipped" together with the vertex stream in the sense that the first future in the list (resp., pipe) will use the first vertex of the stream, and so on. In this way, we need not "unroll" the vertex structure at compile time: the types will refer to projections of a VS parameter.[2] Vertex structures are not limited to streams: in general, a VS can be an infinite (corecursive) tree with arbitrary branching patterns. We show that this allows us to construct a VS corresponding to arbitrary recursive data structures.

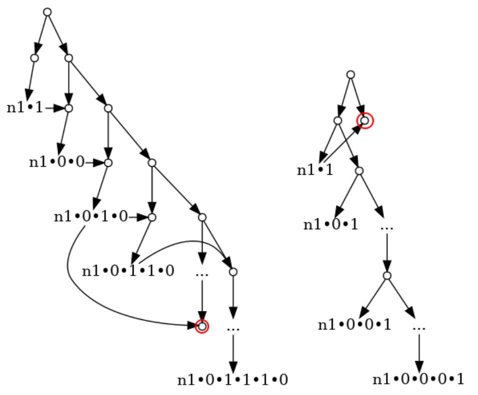

Fig. 2. Visualizations for `list_pi` (left) and `pipeline_pi` (right) showing the differences in parallelization strategies. In both figures, the node corresponding to the touch operation in `main` is circled in red for emphasis.

Recall that we require vertex names to be unique. The vertices contained by a vertex structure are all unique, but ensuring that each vertex is used at most once is non-trivial and requires numerous extensions to the graph type system. One source of complexity is that types can perform significant

[2]We do, however, unroll a VS when unrolling the corresponding graph types, e.g., to create the visualizations in Figure 2. In the figure, $n1$ is the "root" of a VS and the notations that follow are projections of this VS.

computation on vertex structures. As an example, as we discussed above, the same vertex u cannot be used as a name for two futures. In the presence of computation on VSs, this is not a simple restriction to enforce syntactically: if U is a vertex structure, then under a reasonable semantics for vertex structures, $\mathsf{fst}\ (\mathsf{fst}\ U, \mathsf{snd}\ U)$ and $\mathsf{fst}\ U$ refer to the same vertex name and therefore cannot both be used to spawn futures.

The $\lambda^{G\mu}$ calculus assumes that data structures are annotated with vertex structures and makes explicit many of the manipulations of VSs described above. However, $\lambda^{G\mu}$ should be seen as an intermediate representation—an inference algorithm can infer all necessary annotations from unannotated code in a high-level source language. As a proof of concept, we extend GML [Muller 2022], a graph type checker for a subset of OCaml (but which did not previously support lists containing futures, or any sort of user-defined algebraic data types) with support for user-defined algebraic data types containing futures. Our graph type checker is able to infer annotations and produce graph types from the examples in Figure 1, as well as all of the other examples contained in this paper, with no additional annotations or programmer burden, as well as to produce visualizations of their graph types. As shown in the example above, such visualizations can allow programmers to identify errors in the parallelization of their code, and can also be used to reason about parallel complexity and other features. Prior work [Muller 2022] also explains how graph types can be used to aid other analyses, such as deadlock detection. A formal presentation of the inference algorithm is out of the scope of this paper, but much of the challenge in our extension is constructing the vertex structure corresponding to an arbitrary user-defined algebraic data type. We describe this process formally and prove some metatheoretic results about it.

In sum, our contributions are:

- $\lambda^{G\mu}$, a parallel calculus with a graph type system supporting recursive data types (Section 3).
- A soundness result for $\lambda^{G\mu}$, guaranteeing that the graph type of a program correctly describes the computation graph that arises when running the program (Section 4).
- An algorithm for inferring the shape of a vertex structure that will provide the necessary vertex names for an arbitrary recursive data structure, and results showing (among other things) that such a VS exists for any valid recursive data type (Section 5).
- A prototype implementation[3] of graph type inference for an OCaml-like source language, including OCaml-style user-defined algebraic data types mixed with futures (Section 6).

Due to space limitations, we defer some of the technical details and many of the proofs to the full version of the paper [Rinaldi et al. 2024]. We begin with an overview of graph types as well as a high-level description of our extensions.

2 OVERVIEW

We begin with an overview of graph types but refer the interested reader to the original paper [Muller 2022] for a more thorough presentation; we indicate using footnotes where we diverge from that paper's presentation. Our motivating example is a parallel implementation of Quicksort using futures (Figure 3). The code is supplemented with annotations in gray that are inserted during type inference and used in the formal presentation of $\lambda^{G\mu}$, but are not written in actual code; these annotations will be explained later, in Section 3. The implementation returns immediately in the case of an empty list. On a non-empty list, the first element is selected as a pivot and used to partition the list using a sequential function partition, whose implementation we omit. A future is spawned to sort the first list recursively while the second list is sorted in the main thread. When the second list is sorted, we touch the future to retrieve the sorted first list, and append the lists.

[3]source available at `https://github.com/junewunder/gml-popl24/`

The type of the function in $\lambda^{G\mu}$ is given below the code; the type indicates that qsort accepts and returns an `'a list`. As is common in presentations of type-and-effect systems, we write an annotation over the arrow indicating the effects performed by running the function. In this case, the "effect" is the *graph type* of the function; that is, a graph type describing the family of computation graphs which might arise from executing qsort. The prefix $\mu\gamma.$ indicates that G binds a recursive instance of itself as γ—this notation is taken from standard presentations of recursive types. The body of G is a disjunction of two families of graphs, indicated by the $\vee$ symbol. This notation appears when the code executes a conditional or pattern match and indicates two possible families of graphs: $G_1 \vee G_2$ indicates that the graph can take a form indicated by either G_1 or G_2. In the example, the first graph type, $\bullet$, indicates a sequential computation and corresponds to executing the base case. The second graph type corresponds to the recursive case, and indicates that a future is spawned. In order to refer to this future later in the graph type, futures are assigned unique names. By convention, these names are assumed to refer to a vertex "attached" to the computation graph of a future as a final vertex. We will refer to this vertex as the "sink" vertex of the future, borrowing a term from graph theory because, until the future is touched, it has no outgoing edges. The notation new $u{:}\blacklozenge.$, which appears in corresponding locations in the graph type G and as an annotation in the code, indicates binding a new *vertex variable* u which locally refers to a new, fresh vertex name; $\blacklozenge$ is the type of this variable and means that u refers to a single vertex.[4] When the annotated program is evaluated (we do this only to prove soundness; the vertex name annotations have no runtime meaning in the actual program) or the graph type is unrolled (e.g., to produce the visualizations of Figure 2), u will be instantiated with a new, fresh vertex name.[5]

The sequential composition of two graph types is denoted $G_1 \oplus G_2$, indicating that the program performs a computation described by G_1 followed by one described by G_2. In our example graph type, the graph type corresponding to the recursive case is the sequential composition of three operations. The graph type $\gamma \swarrow_u$ indicates that u is the sink of a future whose graph is described by the graph type γ (which, recall, is a recursive instance of G corresponding to a recursive call to qsort). In general, $G \swarrow_u$ indicates a future whose computation graph can be described by G and whose sink vertex is given the name u. In $\lambda^{G\mu}$, spawns using the `future` keyword are also annotated with the vertex that is used; this annotation is shown in gray in the code. The spawn in the graph type is then sequentially composed with another instance of γ for the other recursive call, and finally a touch of the future whose sink is u (a touch of vertex u is denoted ${}^{u}\searrow$).

```
let rec qsort (l: 'a list) : 'a list =
  match l with
  | [] -> []
  | p::t ->
    let (lt, ge) = partition p t in
    new u:♦.
    let future_sort_lt = future[u] (qsort lt) in
    let sort_ge = qsort ge in
    let sort_lt = touch future_sort_lt in
    sort_lt @ [p] @ sort_ge
```

$$\begin{array}{lcl} \text{qsort} & : & \text{'a list} \xrightarrow{G} \text{'a list} \\ \text{where} & & \\ \quad G & = & \mu\gamma.[\bullet \vee (\text{new } u{:}\blacklozenge.\gamma \swarrow_u \oplus \gamma \oplus {}^{u}\searrow)] \end{array}$$

Fig. 3. Code and types for parallel-recursive Quicksort using futures. Code annotations in gray are shown for convenience; these are not written by the programmer.

[4]This type annotation is not used or needed in prior work, where only single vertices are bound. We introduce it here for consistency with the syntax used in the rest of this paper.

[5]In the terminology of this paper, it will actually be instantiated with a new vertex structure of type $\blacklozenge$.

```
let pipeline_pi2[u:♦×♦;_:1] () =
  future[fst u]
    (3.1, future[snd u] 3.14)

let use_pi () =
  new u:♦×♦.
  let (pi1, pi2_fut) =
    touch (pipeline_pi2 ())
  in touch pi2_fut
```

$$\begin{aligned} \text{pipeline\_pi2} &: \Pi[u{:}\blacklozenge\times\blacklozenge;\_{:}1].\text{unit} \xrightarrow{G} \tau\ \text{future}[\text{fst } u] \\ \text{use\_pi} &: \text{unit} \xrightarrow{G'} \text{float} \end{aligned}$$

where

$$\begin{aligned} \tau &= \text{float} \times \text{float future}[\text{snd } u] \\ G &= (\bullet \swarrow_{\text{snd } u}) \swarrow_{\text{fst } u} \\ G' &= \text{new } u'{:}\blacklozenge\times\blacklozenge.G[u'/u] \oplus {}^{\text{fst } u}\!\searrow \oplus {}^{\text{snd } u}\!\searrow \end{aligned}$$

Fig. 4. A function that iteratively approximates π twice in a pipelined manner

Note that the vertex u in the Quicksort example exists only within the scope of the binding and so, in particular, cannot be allowed to escape the scope. If futures are, e.g., returned from a function, the vertices for those futures must be created outside and passed as parameters to the function. As an example, take the pipeline_pi2 function in Figure 4, which returns a future. This function is similar to the analogous function of Section 1, but limited to two approximations of π. The vertex parameter is made explicit in the $\lambda^{G\mu}$ annotations, and also appears in the type of the function (shown on the right side of the figure) as a Π binding. This construct in a graph type binds two parameters. Both parameters stand for *vertex structures* (VSs), type-level (co)data structures containing vertices, and both are annotated with *vertex structure types* indicating their shapes. The first parameter, u, will contain the vertices the function may use to spawn futures. In the case of `pipeline_pi2`, it is annotated with VS type $\blacklozenge \times \blacklozenge$, indicating a pair of vertices (recall that $\blacklozenge$ is a VS type representing a single vertex).[6] The second parameter contains the vertices the function may touch; in the case of `pipeline_pi2`, it is empty as indicated by the unit VS type 1.[7]

The function pipeline_pi2 returns a future (spawned using the first component of the vertex structure u) that produces a pair of a float and another future, spawned with the second component. Note that the types of futures explicitly indicate the vertices with which the future was spawned. As in the Quicksort code, these vertices also appear as annotations on the `future` keyword which are inferred during type checking. Finally, the graph type G of the function body shows that the function spawns a future using the vertex fst u, which in turn spawns a future using the vertex snd u, which finally does not spawn further threads.

The code in Figure 4 also shows a function that calls pipeline_pi2 and touches the two futures. The graph type of this function binds a new vertex structure for the two futures, which no longer need to escape the function. As in Quicksort, the new vertex structure is bound using a binding of the form new $u : \mathcal{U}.G$, where the vertex structure type is now the product $\blacklozenge \times \blacklozenge$ instead of $\blacklozenge$. The call to pipeline_pi2 instantiating the bound vertex structure variable u with the new VS u' is indicated by substituting u' for u in G.

Before proceeding, we make one additional note about the graph type system. We have referred to u and similar as *unique* vertex names—each vertex name can be used to spawn a future at most once, otherwise the resulting graph will be ambiguous (if two futures have u as a sink vertex, there is no way to know to which future a touch ${}^{u}\!\searrow$ refers). The graph type system (both ours and that of prior work) enforce this using an affine type system that restricts the use of vertex names.

[6]Note that prior work had similar notation for vertex parameters but, as it did not have vertex structures, allowed Πs to bind arbitrary-length vectors of vertex parameters. The introduction of vertex structures in the present work, which we will use later to more substantial effect, also simplifies this notation and makes it more uniform.

[7]The theory of $\lambda^{G\mu}$ does not include the VS type 1 to keep the calculus minimal, but it is included in our implementation and would be a straightforward addition to the calculus.

```
type 'a pipe = Pipe of 'a * 'a pipe future

let rec pipeline_pi[u:vstream;_:1] (a, k) =
  let a' = a +. (-1.0) ** (k +. 1.0)
           *. 4.0 /. (2. *. k -. 1.0)
  in
  Pipe (a', future[fst u]
        (pipeline_pi[snd u] (a', k +. 1.)))

let rec nth[_:1;u:vstream]
  ((pipe, n) : 'a pipe[u] * int) =
  let Pipe (a, f) = pipe in
  if n <= 0 then a
  else nth[snd u] (touch f, n - 1)

let main () =
  new u:vstream.
  nth[u] (pipeline_pi[u] (0.0, 1.0)) 1000
```

$$\begin{array}{rcl} \text{pipeline\_pi} & : & \Pi[u\!:\!\text{vstream};\_\!:\!1]. \\ & & \quad \text{float} \times \text{float} \xrightarrow{G} \text{float pipe}[u] \\ \text{nth} & : & \Pi[\_\!:\!1;u\!:\!\text{vstream}]. \\ & & \quad \text{float pipe}[u] \times \text{int} \xrightarrow{G'} \text{float} \\ \text{main} & : & \text{unit} \xrightarrow{G''} \text{float} \end{array}$$

where

$$\begin{array}{rcl} G & = & \mu\gamma.\Pi[u\!:\!\text{vstream};\_\!:\!1].(\gamma[\text{snd } u]) \swarrow_{\text{fst } u} \\ G' & = & \mu\gamma.\Pi[\_\!:\!1;u\!:\!\text{vstream}]. \bullet \vee^{\text{fst } u} \searrow \oplus \gamma[\text{snd } u] \\ G'' & = & \text{new } u\!:\!\text{vstream}.G[u;\langle\rangle] \oplus G'[\langle\rangle;u] \end{array}$$

$$\text{vstream} = \nu t.\blacklozenge \times t$$

Fig. 5. Code and types for a function that iteratively approximates π indefinitely in a pipelined manner.

Thus far, we have discussed examples that are within the capabilities of prior work. Now suppose we wish to generalize pipeline_pi2 to continue producing iterative approximations indefinitely. The code in Figure 5 does this, producing a value of type float pipe, also defined in the figure, which is a recursive type containing an approximation and a future to continue the pipeline. This is reminiscent of the "producer" example of Blelloch and Reid-Miller [1997], who use futures in this general pattern to construct a wide variety of pipelined data structures. As in the Introduction, each iteration computes the kth term in the approximation, adds it to a running total a, and returns the new running total as well as a future to call pipeline_pi recursively to compute the k + 1st term.

Useful instances of the recursive data type ′a pipe cannot be typed with the existing graph type system, because doing so would require an infinite sequence of new vertex names and a way of associating each future in the pipeline with successive vertex names. One (incorrect but illustrative) approach would be to instantiate each future in the type with a fresh vertex name using, for example, an existential. The pipe type would then be annotated as follows:

```
type 'a pipe = Pipe of 'a * pipe future[∃u : ♦.u]
```

This is still not useful, however, because it doesn't allow any vertex name to escape the scope of the single future type, just as the vertex in the qsort function was confined to the qsort function. The type τ future$[\exists u : \blacklozenge.u]$ tells us that the future is spawned with *some* vertex, but gives no information about *which*, an untenable loss of precision when we try to touch this future and add an edge to its vertex. As an illustration of this loss of precision, consider the following program, and suppose we wish to use its graph type to check for deadlocks (simply put, a program may deadlock if its graph type can unroll to a cyclic graph):

```
let rec f n =
  if n <= 0 then [future (fun () -> 0)]
  else
    let l' = f (n - 1) in
    (future (fun () -> touch (List.hd l')))::l'
```

Each future in the list contains a function that touches the following future in the list. This is a fairly clear structure and a visualization or suitable analysis of the graph type produced by our system could show that the program is deadlock-free. However, if the type of the output list were given as (unit $\xrightarrow{\overset{\exists u.u}{\searrow}}$ int) future$[\exists u.u]$, the most precise thing that could be said about this list is that it is a list of thunks under futures, each of which touches any future in the list (or, indeed, without further information, any future in the *program*), including itself. Thus, a sound deadlock detector would have to conclude that the program might deadlock.

As a more precise solution to the problem of generating unique vertex names for elements in a data structure, we introduce *vertex structures*, mentioned above, which we allow to be (co)recursive and thus serve as the source or collection of vertex names we need. In the code annotations and the type of pipeline_pi on the right side of the figure, the function takes a vertex structure parameter of vertex structure type vstream, which is defined in the lower right side of the figure to be a corecursive type of an infinite list or stream of vertices. The return type of the function is float pipe$[u]$, where the recursive pipe type is now parameterized by a vertex structure. This vertex structure is threaded through the recursive structure of the pipeline data type such that successive futures in the data type are associated with corresponding vertices from u. The details of this are technical and so we defer them, as well as the formal presentation of recursive data types in $\lambda^{G\mu}$, to the next section. As in pipeline_pi2, the first future uses the vertex fst u, which appears as an annotation in the code and on the graph type (G contains $\swarrow_{\mathsf{fst}\ u}$, indicating a spawn of fst u).[8] However, now the function calls itself recursively to generate the rest of the pipeline. Because the function takes a vertex parameter, this recursive call must instantiate the vertex parameter with a vstream, and it does so with the tail of the stream, snd u. This appears in the graph type as $\gamma[\mathsf{snd}\ u]$.

We complete this overview with a demonstration of how the pipeline can be consumed, which shows how vertex structures link individual futures to their touches. The nth function in Figure 5 consumes a pipeline recursively, returning the n^{th} value. It also takes a parameter u of vertex structure type vstream, but this time as the second parameter, because the function uses these vertices to touch futures and does not spawn futures. The use of u as the parameter to the pipe data type indicates that vertices for futures in the pipeline will be drawn from the stream u, which is enough information to infer in the graph type G' that the touch ${}^{\mathsf{fst}\ u}\searrow$ targets the first vertex of u. The function then calls itself recursively with the tail of the vertex stream, which also appears in the recursive instantiation of γ in the graph type. Finally, main calls nth with the π pipeline. As we have seen before, the vertex structure u is bound here so that its scope covers its uses both for spawns (in pipeline_pi) and for touches (in nth). The calls to both functions instantiate the vertex structure parameter with the same vertex structure u, linking the spawns and touches in the graph types. The graph type for main composes the graph types of the producer and the consumer and links the spawns and touches by instantiating both graph types with the same vertex structure.

3 GRAPH TYPES WITH VERTEX STRUCTURES

This section provides a formal presentation of $\lambda^{G\mu}$, whose syntax is given in Figure 6. In the remainder of this section, we describe the features of the language in detail, focusing on the main novelties of $\lambda^{G\mu}$ compared to prior work: *vertex structures* (VSs) and *recursive types*.

[8]We treat corecursive vertex structure types as equi(co)recursive, so no unrolling is needed.

$$
\begin{array}{llcl}
\textit{Vertex Structures} & U & ::= & u \mid (U,U) \mid \mathsf{fst}\ U \mid \mathsf{snd}\ U \\
\textit{Vertex Structure Types} & \mathcal{U} & ::= & \blacklozenge \mid \mathcal{U}^a \times \mathcal{U}^a \mid t \mid \nu t.\mathcal{U} \\
\textit{Availability} & a & ::= & \blacksquare \mid \square \\
\textit{Graph Types} & G & ::= & \gamma \mid \bullet \mid G \oplus G \mid G \vee G \mid \mu\gamma.G \mid G \swarrow_U \mid \overset{U}{\searrow} \mid \\
& & & \Pi[u{:}\mathcal{U}; u{:}\mathcal{U}].G \mid G[U;U] \mid \mathsf{new}\ u{:}\mathcal{U}.G \\
\textit{Graph Kinds} & \kappa_G & ::= & * \mid \Pi[u_f{:}\mathcal{U}_f; u_t{:}\mathcal{U}_t].\kappa_G \\
\textit{Kinds} & \kappa & ::= & \mathsf{Ty} \mid \mathcal{U} \rightarrow \mathsf{Ty} \\
\textit{Type Constructors} & c & ::= & \mathsf{unit} \mid \Pi[u{:}\mathcal{U}; u{:}\mathcal{U}].c \xrightarrow{G} c \mid c \times c \mid c + c \mid c\ \mathsf{future}[U] \mid \alpha \mid \\
& & & \mu(\alpha; u{:}\mathcal{U}.c; U) \mid \lambda u{:}\mathcal{U}.c \mid c\ U \\
\textit{Expressions} & e & ::= & x \mid \langle\rangle \mid \mathsf{fun}[u;u]\ f\ x = e \mid e[U;U]\ e \mid (e,e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \\
& & & \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e\ \{x.e; y.e\} \mid \mathsf{roll}\ e \mid \mathsf{unroll}\ e \mid \\
& & & \mathsf{future}[U]\ e \mid \mathsf{touch}\ e \mid \mathsf{new}\ u{:}\mathcal{U}.e
\end{array}
$$

Fig. 6. Syntax of $\lambda^{G\mu}$.

3.1 Vertex Structures and Their Types

Vertex Structures (U) contain vertices that represent futures in computation graphs. As shown in Figure 6, VSs appear in annotations within expressions, type constructors, and graph types; these annotations are not inserted into real code by programmers, but are filled in during type inference.

Vertex structures are classified with VS types ($\mathcal{U}$). The VS type $\blacklozenge$ represents a single vertex, and only VSs of type $\blacklozenge$ can be used to name futures. The product type $\mathcal{U}_1^{a_1} \times \mathcal{U}_2^{a_2}$ represents pairs of VSs in $\mathcal{U}_1$ and $\mathcal{U}_2$. The availability annotations a_1 and a_2 indicate whether the corresponding component is available ($\blacksquare$) or unavailable ($\square$) for spawning new futures; their use is inspired by record types in the Cogent language [O'Connor et al. 2021]. The need for availability will become clearer later, when we discuss the type system. Finally, we can also form corecursive VS types $\nu t.\mathcal{U}$, which we will use to generate graph types that require a potentially unbounded number of vertices.

Figure 7 presents the rules for assigning VS types to VSs. The judgment $\Omega; \Psi \vdash U : \mathcal{U}$ denotes that the VS U has VS type $\mathcal{U}$, where Ω and Ψ are contexts that map VS variables to their types. Vertices, and thus VSs, are treated in an affine manner to ensure that any vertex is used at most once to spawn a future—this affine treatment leads to the use of two contexts. The first, Ω, is an affine context storing vertices that may be used to spawn futures and the second, Ψ, is an unrestricted context for vertices that may be used to touch futures (we may touch a vertex any number of times). Because we wish to be able to touch any vertex we spawn, the set of variables in Ω will always be a subset of that in Ψ. A VS variable is well-typed if it is in either Ω or Ψ, and we assume that Ω does not contain multiple mappings for the same variable.

VSs can be variables (u),[9] pairs, and projections. As seen in the rules U:Fst and U:Snd, only available components can be projected. For example, if u has VS type $\mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{\square}$, then fst u is safe to use, but snd u is not. Rule U:Subtype is a subsumption rule for the subtyping relation on VS types, denoted $\mathcal{U}' \sqsubseteq \mathcal{U}$ and defined in Figure 8. We allow three forms of subtyping: first (UT:Corec1 and UT:Corec2), we can freely roll and unroll corecursive VS types. Second (UT:ProdLeft and UT:ProdRight), it is safe to take an available component and treat it as unavailable. Third, the types of unavailable components of VSs may be changed at will, which is safe since those sides can never be used.

[9]Unlike the original presentation [Muller 2022], u refers to a variable instead of a vertex.

$$\text{(U:OmegaVar)}\ \frac{}{\Omega, u : \mathcal{U}; \Psi \vdash u : \mathcal{U}} \qquad \text{(U:PsiVar)}\ \frac{}{\Omega; \Psi, u : \mathcal{U} \vdash u : \mathcal{U}} \qquad \text{(U:Pair)}\ \frac{\Omega \rightsquigarrow \Omega_1 \boxplus \Omega_2 \qquad \Omega_1; \Psi \vdash U_1 : \mathcal{U}_1 \qquad \Omega_2; \Psi \vdash U_2 : \mathcal{U}_2}{\Omega; \Psi \vdash (U_1, U_2) : \mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{\blacksquare}}$$

$$\text{(U:OnlyLeftPair)}\ \frac{\Omega; \Psi \vdash U_1 : \mathcal{U}_1 \qquad \cdot; \Psi' \vdash U_2 : \mathcal{U}_2}{\Omega; \Psi \vdash (U_1, U_2) : \mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{\square}} \qquad \text{(U:OnlyRightPair)}\ \frac{\cdot; \Psi' \vdash U_1 : \mathcal{U}_1 \qquad \Omega; \Psi \vdash U_2 : \mathcal{U}_2}{\Omega; \Psi \vdash (U_1, U_2) : \mathcal{U}_1^{\square} \times \mathcal{U}_2^{\blacksquare}} \qquad \text{(U:Fst)}\ \frac{\Omega; \Psi \vdash U : \mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{a}}{\Omega; \Psi \vdash \mathsf{fst}\ U : \mathcal{U}_1}$$

$$\text{(U:Snd)}\ \frac{\Omega; \Psi \vdash U : \mathcal{U}_1^{a} \times \mathcal{U}_2^{\blacksquare}}{\Omega; \Psi \vdash \mathsf{snd}\ U : \mathcal{U}_2} \qquad \text{(U:Subtype)}\ \frac{\Omega; \Psi \vdash U : \mathcal{U}' \qquad \mathcal{U}' \sqsubseteq \mathcal{U}}{\Omega; \Psi \vdash U : \mathcal{U}}$$

Fig. 7. Vertex structure type system for $\lambda^{G\mu}$.

$$\text{(UT:ProdLeft)}\ \frac{}{\mathcal{U}_1^{a_1} \times \mathcal{U}_2^{a_2} \sqsubseteq \mathcal{U}_3^{\square} \times \mathcal{U}_2^{a_2}} \qquad \text{(UT:ProdRight)}\ \frac{}{\mathcal{U}_1^{a_1} \times \mathcal{U}_2^{a_2} \sqsubseteq \mathcal{U}_1^{a_1} \times \mathcal{U}_3^{\square}} \qquad \text{(UT:Prod)}\ \frac{\mathcal{U}_1 \sqsubseteq \mathcal{U}_1' \qquad \mathcal{U}_2 \sqsubseteq \mathcal{U}_2'}{\mathcal{U}_1^{a_1} \times \mathcal{U}_2^{a_2} \sqsubseteq \mathcal{U}_1'^{a_1} \times \mathcal{U}_2'^{a_2}}$$

$$\text{(UT:Corec1)}\ \frac{}{vt.\mathcal{U} \sqsubseteq \mathcal{U}[vt.\mathcal{U}/t]} \qquad \text{(UT:Corec2)}\ \frac{}{\mathcal{U}[vt.\mathcal{U}/t] \sqsubseteq vt.\mathcal{U}} \qquad \text{(UT:Reflexive)}\ \frac{}{\mathcal{U} \sqsubseteq \mathcal{U}} \qquad \text{(UT:Transitive)}\ \frac{\mathcal{U} \sqsubseteq \mathcal{U}'' \qquad \mathcal{U}'' \sqsubseteq \mathcal{U}'}{\mathcal{U} \sqsubseteq \mathcal{U}'}$$

Fig. 8. VS subtyping.

$$\text{(US:Prod)}\ \frac{}{\mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{\blacksquare} \rightsquigarrow \mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{\square} \boxplus \mathcal{U}_1^{\square} \times \mathcal{U}_2^{\blacksquare}} \qquad \text{(US:SplitBoth)}\ \frac{\mathcal{U}_1 \rightsquigarrow \mathcal{U}_1' \boxplus \mathcal{U}_1'' \qquad \mathcal{U}_2 \rightsquigarrow \mathcal{U}_2' \boxplus \mathcal{U}_2''}{\mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{\blacksquare} \rightsquigarrow \mathcal{U}_1'^{\blacksquare} \times \mathcal{U}_2'^{\blacksquare} \boxplus \mathcal{U}_1''^{\blacksquare} \times \mathcal{U}_2''^{\blacksquare}}$$

$$\text{(US:SplitLeft)}\ \frac{\mathcal{U}_1 \rightsquigarrow \mathcal{U}_1' \boxplus \mathcal{U}_1''}{\mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{a} \rightsquigarrow \mathcal{U}_1'^{\blacksquare} \times \mathcal{U}_2^{a} \boxplus \mathcal{U}_1''^{\blacksquare} \times \mathcal{U}_2^{\square}} \qquad \text{(US:SplitRight)}\ \frac{\mathcal{U}_2 \rightsquigarrow \mathcal{U}_2' \boxplus \mathcal{U}_2''}{\mathcal{U}_1^{a} \times \mathcal{U}_2^{\blacksquare} \rightsquigarrow \mathcal{U}_1^{a} \times \mathcal{U}_2'^{\blacksquare} \boxplus \mathcal{U}_1^{\square} \times \mathcal{U}_2''^{\blacksquare}}$$

$$\text{(US:Corecursive)}\ \frac{\mathcal{U}[vt.\mathcal{U}/t] \rightsquigarrow \mathcal{U}_1 \boxplus \mathcal{U}_2}{vt.\mathcal{U} \rightsquigarrow \mathcal{U}_1 \boxplus \mathcal{U}_2} \qquad \text{(US:Subtype)}\ \frac{\mathcal{U} \rightsquigarrow \mathcal{U}_1' \boxplus \mathcal{U}_2 \qquad \mathcal{U}_1' \sqsubseteq \mathcal{U}_1}{\mathcal{U} \rightsquigarrow \mathcal{U}_1 \boxplus \mathcal{U}_2} \qquad \text{(US:Commutative)}\ \frac{\mathcal{U} \rightsquigarrow \mathcal{U}_2 \boxplus \mathcal{U}_1}{\mathcal{U} \rightsquigarrow \mathcal{U}_1 \boxplus \mathcal{U}_2}$$

Fig. 9. Vertex structure type splitting.

$$\text{(OM:Empty)}\ \frac{}{\cdot \rightsquigarrow \cdot \boxplus \cdot} \qquad \text{(OM:Commutative)}\ \frac{\Omega \rightsquigarrow \Omega_2 \boxplus \Omega_1}{\Omega \rightsquigarrow \Omega_1 \boxplus \Omega_2} \qquad \text{(OM:Var)}\ \frac{\Omega \rightsquigarrow \Omega_1 \boxplus \Omega_2}{\Omega, u : \mathcal{U} \rightsquigarrow \Omega_1, u : \mathcal{U} \boxplus \Omega_2} \qquad \text{(OM:VarTypeSplit)}\ \frac{\Omega \rightsquigarrow \Omega_1 \boxplus \Omega_2 \qquad \mathcal{U} \rightsquigarrow \mathcal{U}_1 \boxplus \mathcal{U}_2}{\Omega, u : \mathcal{U} \rightsquigarrow \Omega_1, u : \mathcal{U}_1 \boxplus \Omega_2, u : \mathcal{U}_2}$$

Fig. 10. Ω context splitting.

$$\frac{\Omega \rightsquigarrow \Omega_1 \boxplus \Omega_2 \qquad \Delta; \Omega_1; \Psi \vdash G : * \qquad \Omega_2; \cdot \vdash U : \blacklozenge}{\Delta; \Omega; \Psi \vdash G \swarrow_U : *}\ (\text{DW:Spawn}) \qquad \frac{\cdot; \Psi \vdash U : \blacklozenge}{\Delta; \Omega; \Psi \vdash {}^{U}\!\searrow : *}\ (\text{DW:Touch})$$

$$\frac{\Delta; \Omega, u : \mathcal{U}; \Psi, u : \mathcal{U} \vdash G : *}{\Delta; \Omega; \Psi \vdash \mathsf{new}\ u{:}\mathcal{U}.G : *}\ (\text{DW:New}) \qquad \frac{\Delta, \gamma : \Pi[u_f{:}\mathcal{U}_f; u_t{:}\mathcal{U}_t].*; u_f : \mathcal{U}_f; \Psi, u_f : \mathcal{U}_f, u_t : \mathcal{U}_t \vdash G : *}{\Delta; \Omega; \Psi \vdash \mu\gamma.\Pi[u_f{:}\mathcal{U}_f; u_t{:}\mathcal{U}_t].G : \Pi[u_f{:}\mathcal{U}_f; u_t{:}\mathcal{U}_t].*}\ (\text{DW:RecPi})$$

$$\frac{\Delta; \Omega_1; \Psi \vdash G : \Pi[u_f{:}\mathcal{U}_f; u_t{:}\mathcal{U}_t].* \qquad \begin{array}{c}\Omega \rightsquigarrow \Omega_1 \boxplus \Omega_2 \\ \Omega_2; \cdot \vdash U_f : \mathcal{U}_f\end{array} \qquad \cdot; \Psi \vdash U_f : \mathcal{U}_f \qquad \cdot; \Psi \vdash U_t : \mathcal{U}_t}{\Delta; \Omega; \Psi \vdash G[U_f; U_t] : *}\ (\text{DW:App})$$

Fig. 11. Selected rules for graph type formation.

The VS typing rule U:Pair uses an auxiliary splitting relation $\bullet \rightsquigarrow \bullet \boxplus \bullet$ [O'Connor et al. 2021], which is described in Figure 10. This relation is responsible for enforcing the affine treatment of Ω contexts. The judgment $\Omega \rightsquigarrow \Omega_1 \boxplus \Omega_2$ states that Ω splits into the disjoint contexts Ω_1 and Ω_2. It is important that Ω_1 and Ω_2 be disjoint so that futures spawned under Ω_1 and under Ω_2 have distinct vertices. However, we allow a variable with a product VS type to appear in both contexts, as long as the availability of the products is in turn split between the two. This is allowed by OM:VarTypeSplit: $\Omega, u : \mathcal{U}$ may split to $\Omega_1, u : \mathcal{U}_1$ and $\Omega, u : \mathcal{U}_2$ if $\mathcal{U} \rightsquigarrow \mathcal{U}_1 \boxplus \mathcal{U}_2$ holds. Intuitively, $\mathcal{U}$, $\mathcal{U}_1$, and $\mathcal{U}_2$ are the same types but with different availabilities: if a component of a product VS type is available in $\mathcal{U}$, then that component is available in $\mathcal{U}_1$ or $\mathcal{U}_2$ or neither, but not both. For example, if $u{:}\mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{\blacksquare}$ appears in Ω, we may have $u{:}\mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{\square}$ in Ω_1 and $u{:}\mathcal{U}_1^{\square} \times \mathcal{U}_2^{\blacksquare}$ in Ω_2, but we cannot have $u{:}\mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{\blacksquare}$ appear in either Ω_1 and Ω_2.

The VS type splitting judgment $\mathcal{U} \rightsquigarrow \mathcal{U}_1 \boxplus \mathcal{U}_2$ is defined in Figure 9. The core mechanism of VS type splitting is US:Prod, which states a VS type $\mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{\blacksquare}$ may split into $\mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{\square}$ and $\mathcal{U}_1^{\square} \times \mathcal{U}_2^{\blacksquare}$. Most of the other rules are "search" rules allowing applications of US:Prod in nested VS types. The other significant VS type splitting rule is US:Subtype, which allows for "weakening" the VS types resulting from a split (by turning available sides of product VS types to unavailable).

3.2 Graph Types and Type Constructors

There are two kinding judgments to characterize well-formed types: one for graph types, and another one for type constructors. For graph types, the judgment $\Delta; \Omega; \Psi \vdash G : \kappa_G$ states that the graph type G has graph kind κ_G (Figure 11). The context Δ maps graph type variables to their kinds, and is used to check that recursive graph types are well-formed (see DW:RecPi). We show selected graph type formation rules for space reasons; others are similar and are given in the full version of the paper [Rinaldi et al. 2024].

For type constructors, the judgment $\Delta; \Psi; \Upsilon \vdash c :: \kappa$ states that the type constructor c is well-formed and has kind κ (Figure 12). The context Υ maps type variables to their kinds. Type constructors can be ordinary types, which are given the kind Ty (and for which we sometimes use the metavariable τ). Types may also be parameterized by vertex structures. This allows, for example, a type of lists of futures which is parameterized by the VS providing the vertices for the futures.[10] The

[10]We could use the same type parameter mechanism to allow types to be parameterized by other types, as in the ML type `'a list`, but this is orthogonal and we do not consider it in the formalism to streamline the presentation.

$$\frac{}{\Delta;\Psi;\Upsilon \vdash \mathsf{unit} :: \mathsf{Ty}}\;(\text{K:Unit}) \qquad \frac{}{\Delta;\Psi;\Upsilon,\alpha :: \kappa \vdash \alpha :: \kappa}\;(\text{K:Var})$$

$$\frac{\begin{array}{c}\Delta;\Psi,u_f : \mathcal{U}_f,u_t : \mathcal{U}_t;\Upsilon \vdash c_1 :: \mathsf{Ty}\\ \Delta;\Psi,u_f : \mathcal{U}_f,u_t : \mathcal{U}_t;\Upsilon \vdash c_2 :: \mathsf{Ty} \qquad \Delta;\cdot;\Psi \vdash G : \Pi[u_f{:}\mathcal{U}_f;u_t{:}\mathcal{U}_t].*\end{array}}{\Delta;\Psi;\Upsilon \vdash \Pi[u_f{:}\mathcal{U}_f;u_t{:}\mathcal{U}_t].c_1 \xrightarrow{G[u_f;u_t]} c_2 :: \mathsf{Ty}}\;(\text{K:Fun})$$

$$\frac{\Delta;\Psi;\Upsilon \vdash c_1 :: \mathsf{Ty} \qquad \Delta;\Psi;\Upsilon \vdash c_2 :: \mathsf{Ty}}{\Delta;\Psi;\Upsilon \vdash c_1 + c_2 :: \mathsf{Ty}}\;(\text{K:Sum}) \qquad \frac{\Delta;\Psi;\Upsilon \vdash c_1 :: \mathsf{Ty} \qquad \Delta;\Psi;\Upsilon \vdash c_2 :: \mathsf{Ty}}{\Delta;\Psi;\Upsilon \vdash c_1 \times c_2 :: \mathsf{Ty}}\;(\text{K:Prod})$$

$$\frac{\Delta;\Psi;\Upsilon \vdash c :: \mathsf{Ty} \qquad \cdot;\Psi \vdash U : \blacklozenge}{\Delta;\Psi;\Upsilon \vdash c\ \mathsf{future}[U] :: \mathsf{Ty}}\;(\text{K:Fut}) \qquad \frac{\Delta;\Psi,u : \mathcal{U};\Upsilon,\alpha :: \mathcal{U} \to \mathsf{Ty} \vdash c :: \mathsf{Ty} \qquad \cdot;\Psi \vdash U : \mathcal{U}}{\Delta;\Psi;\Upsilon \vdash \mu(\alpha;u{:}\mathcal{U}.c;U) :: \mathsf{Ty}}\;(\text{K:Rec})$$

$$\frac{\Delta;\Psi,u : \mathcal{U};\Upsilon \vdash c :: \mathsf{Ty}}{\Delta;\Psi;\Upsilon \vdash \lambda u{:}\mathcal{U}.c :: \mathcal{U} \to \mathsf{Ty}}\;(\text{K:Lambda}) \qquad \frac{\Delta;\Psi;\Upsilon \vdash c :: \mathcal{U} \to \mathsf{Ty} \qquad \cdot;\Psi \vdash U : \mathcal{U}}{\Delta;\Psi;\Upsilon \vdash c\ U :: \mathsf{Ty}}\;(\text{K:App})$$

Fig. 12. Type constructor kinding rules.

$$\frac{\Psi \vdash U \equiv (U_1,U_2) : \mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{a}}{\Psi \vdash \mathsf{fst}\ U \equiv U_1 : \mathcal{U}_1}\;(\text{UE:FstPair}) \qquad \frac{\Psi \vdash U \equiv (U_1,U_2) : \mathcal{U}_1^{a} \times \mathcal{U}_2^{\blacksquare}}{\Psi \vdash \mathsf{snd}\ U \equiv U_2 : \mathcal{U}_2}\;(\text{UE:SndPair})$$

Fig. 13. Selected rules for vertex structure equivalence.

kind $\mathcal{U} \to \mathsf{Ty}$ classifies type-level functions that take a VS of type $\mathcal{U}$ and return a type constructor of kind Ty. Rule K:Lambda describes how to assign the kind $\mathcal{U} \to \mathsf{Ty}$ to such functions.

As mentioned earlier, the motivation behind $\lambda^{G\mu}$ is to allow recursive types containing futures. We achieve this by parameterizing recursive types by VSs containing the vertices for these futures. The syntax for a parameterized recursive data type is $\mu(\alpha;u{:}\mathcal{U}.c;U)$, where $u{:}\mathcal{U}.c$ is a type level VS function (equivalent to $\lambda u : \mathcal{U}.c$), α is a recursive binding of $u : \mathcal{U}.c$, and U is the argument applied to $u{:}\mathcal{U}.c$ when the recursive type is unrolled. The formation of parameterized recursive types is performed by K:Rec, in which α has kind $\mathcal{U} \to \mathsf{Ty}$ within c since it represents a type-level function that passes a VS argument to the VS argument of the recursive instance (seen in more detail below). By applying a sub-VS of U to an instance of α in c, the type $\mu(\alpha;u{:}\mathcal{U}.c;U)$ is able to recur over U. (Note that non-parameterized recursive types do not require special syntax because we can parameterize them in a trivial way by using a dummy VS as the parameter.) We can now implement the type constructor for a list of integer futures as

$$\lambda u'{:}\mathsf{vstream}.\mu(\alpha;u{:}\mathsf{vstream}.(\mathsf{unit} + (\mathsf{int\ future}[\mathsf{fst}\ u] \times \alpha\ (\mathsf{snd}\ u)));u').$$

Because VSs can occur in types, type checking $\lambda^{G\mu}$ programs may require performing some type-level computation, notably when projecting vertices out of a VS. To address this, we introduce two

$$\text{(CE:FUT)}\quad \frac{\Delta;\Psi;\Upsilon \vdash c \equiv c' :: \mathsf{Ty} \qquad \Psi \vdash U \equiv U' : \blacklozenge}{\Delta;\Psi;\Upsilon \vdash c\ \mathsf{future}[U] \equiv c'\ \mathsf{future}[U'] :: \mathsf{Ty}} \qquad \text{(CE:BETAEQ)}\quad \frac{\Delta;\Psi, u : \mathcal{U};\Upsilon \vdash c :: \mathsf{Ty} \qquad \cdot;\Psi \vdash U : \mathcal{U}}{\Delta;\Psi;\Upsilon \vdash (\lambda u{:}\mathcal{U}.c)\ U \equiv c[U/u] :: \mathsf{Ty}}$$

Fig. 14. Selected rules for type constructor equivalence.

judgments: an equivalence judgment on VSs, $\Psi \vdash U \equiv U' : \mathcal{U}$ (see Figure 13), and an equivalence judgment on type constructors, $\Delta; \Psi; \Upsilon \vdash c_1 \equiv c_2 :: \kappa$ (see Figure 14). Regarding VSs, the most notable rules are UE:FSTPAIR and UE:SNDPAIR, which extract (available) components of a pair. For type constructors, equivalence has two purposes: performing a type-level VS function application, which is performed by CE:BETAEQ; and changing VSs within types to equivalent VSs according to the VS equivalence rules (CE:FUTURE is given as an example). Other rules are relatively straightforward and are deferred to the full version of the paper [Rinaldi et al. 2024] for space reasons.

3.3 Graph Type System

Figure 15 presents the type system for $\lambda^{G\mu}$, which assigns graph types (and types) to expressions. The judgment $\Delta; \Omega; \Psi; \Gamma \vdash e : \tau \mid G$ states that the expression e has type τ and graph type G. In addition to the graph type context Δ and VS contexts Ω and Ψ, this judgment uses the context Γ which, as usual, maps expression variables to their types.

We give a quick overview of the graph type system: $\bullet$ represents expressions that execute purely sequentially; $G_1 \oplus G_2$ represents expressions that execute an expression with graph type G_1 followed by an expression with graph type G_2; $G_1 \vee G_2$ represents expressions that execute an expression with graph type G_1 or an expression with graph type G_2; $\mu\gamma.G$ is a recursive graph type; $G \swarrow_U$ represents spawning a future at vertex U that executes an expression with graph type G in parallel; and ${}^{U}\searrow$ represents touching the future at vertex U. A parameterized graph type $\Pi[u_f{:}\mathcal{U}_f; u_t{:}\mathcal{U}_t].G$ accepts two VSs as arguments (u_f has VS type $\mathcal{U}_f$ and is added to Ω and Ψ while u_t has VS type $\mathcal{U}_t$ and is only added to Ψ). Such graph type functions are applied with the syntax $G[U; U]$. Finally, new $u{:}\mathcal{U}.G$ binds a new VS variable of VS type $\mathcal{U}$ and represents expressions that do the same.

Rule S:FUN types function expressions $\mathsf{fun}[u_f; u_t]\ f\ x = e$ where f is the name of the function, x is an expression parameter, u_f and u_t are two VS parameters, and e is the body of the function. Excluding bindings of new VS variables within e, the only VS variable that can be used for spawning futures in e is u_f, while future touches can use any VS variable within the context (including u_f and u_t), hence the function having two VS parameters. The type of functions is

$$\Pi[u_f{:}\mathcal{U}_f; u_t{:}\mathcal{U}_t].\tau_1 \xrightarrow{(\mu\gamma.\Pi[u_f:\mathcal{U}_f;u_t:\mathcal{U}_t].G)\,[u_f;u_t]} \tau_2$$

where τ_1 is the type of the parameter x, τ_2 is the type of the function body, G is the graph type of the function body, and $(\mu\gamma.\Pi[u_f{:}\mathcal{U}_f; u_t{:}\mathcal{U}_t].G)\,[u_f; u_t]$ is the graph type representing graphs produced by applying the function. The function type is parameterized by the VS parameters u_f and u_t (the same ones from the function expression), which have VS types $\mathcal{U}_f$ and $\mathcal{U}_t$ respectively. The graph type $(\mu\gamma.\Pi[u_f : \mathcal{U}_f; u_t : \mathcal{U}_t].G)\,[u_f; u_t]$ contains a recursive binding to a graph type function whose body is G (note that this function binds a new, separate u_f and u_t within G), and this recursive binding is applied to the u_f and u_t bound within the type. This allows G to pass different VS arguments to recursive instances of itself (this is useful, for example, when recursive instances access deeper levels of a vertex stream). In addition to adding the function expression's parameters to the context when typing the function body, we add f, the recursive binding of the function; and γ, the recursive binding of the graph type function whose body is G.

(S:Var)

$$\frac{}{\Delta;\Omega;\Psi;\Gamma,x:\tau\vdash x:\tau\mid\bullet}$$

(S:Unit)

$$\frac{}{\Delta;\Omega;\Psi;\Gamma\vdash\langle\rangle:\mathsf{unit}\mid\bullet}$$

(S:Fun)

$$\frac{\begin{array}{c}\Delta'=\Delta,\gamma:\Pi[u_f{:}\mathcal{U}_f;u_t{:}\mathcal{U}_t].* \qquad \Gamma'=\Gamma,f:\Pi[u_f{:}\mathcal{U}_f;u_t{:}\mathcal{U}_t].\tau_1\xrightarrow{\gamma[u_f;u_t]}\tau_2,x:\tau_1 \qquad \gamma\ \text{fresh}\\ \Psi'=\Psi,u_f:\mathcal{U}_f,u_t:\mathcal{U}_t \qquad \Delta';u_f:\mathcal{U}_f;\Psi';\Gamma'\vdash e:\tau_2\mid G \qquad \Delta;\Psi';\cdot\vdash\tau_1::\mathsf{Ty} \qquad \Delta;\Psi';\cdot\vdash\tau_2::\mathsf{Ty}\end{array}}{\Delta;\Omega;\Psi;\Gamma\vdash\mathsf{fun}[u_f;u_t]\ f\ x=e:\Pi[u_f{:}\mathcal{U}_f;u_t{:}\mathcal{U}_t].\tau_1\xrightarrow{(\mu\gamma.\Pi[u_f:\mathcal{U}_f;u_t:\mathcal{U}_t].G)\,[u_f;u_t]}\tau_2\mid\bullet}$$

(S:App)

$$\frac{\begin{array}{c}\Omega\rightsquigarrow\Omega_1\boxplus\Omega' \qquad \Omega'\rightsquigarrow\Omega_2\boxplus\Omega_3 \qquad \Delta;\Omega_1;\Psi;\Gamma\vdash e_1:\Pi[u_f{:}\mathcal{U}_f;u_t{:}\mathcal{U}_t].\tau_1\xrightarrow{G_3\,[u_f;u_t]}\tau_2\mid G_1\\ \Delta;\Omega_2;\Psi;\Gamma\vdash e_2:\tau_1[U_f/u_f][U_t/u_t]\mid G_2 \qquad \Omega_3;\cdot\vdash U_f:\mathcal{U}_f \qquad \cdot;\Psi\vdash U_f:\mathcal{U}_f \qquad \cdot;\Psi\vdash U_t:\mathcal{U}_t\end{array}}{\Delta;\Omega;\Psi;\Gamma\vdash e_1[U_f;U_t]\ e_2:\tau_2[U_f/u_f][U_t/u_t]\mid G_1\oplus G_2\oplus G_3[U_f;U_t]}$$

(S:Pair)

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

(S:Fst)

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

(S:Snd)

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

(S:InL)

$$\frac{\begin{array}{c}\Delta;\Omega;\Psi;\Gamma\vdash e:\tau_1\mid G\\ \Delta;\Psi;\cdot\vdash\tau_2::\mathsf{Ty}\end{array}}{\Delta;\Omega;\Psi;\Gamma\vdash\mathsf{inl}\ e:\tau_1+\tau_2\mid G}$$

(S:InR)

$$\frac{\begin{array}{c}\Delta;\Omega;\Psi;\Gamma\vdash e:\tau_2\mid G\\ \Delta;\Psi;\cdot\vdash\tau_1::\mathsf{Ty}\end{array}}{\Delta;\Omega;\Psi;\Gamma\vdash\mathsf{inr}\ e:\tau_1+\tau_2\mid G}$$

(S:Case)

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

(S:Roll)

$$\frac{\Delta;\Omega;\Psi;\Gamma\vdash e:\tau[U/u][\lambda u'{:}\mathcal{U}.\mu(\alpha;u{:}\mathcal{U}.\tau;u')/\alpha]\mid G \qquad \Delta;\Psi;\cdot\vdash\mu(\alpha;u{:}\mathcal{U}.\tau;U)::\mathsf{Ty}}{\Delta;\Omega;\Psi;\Gamma\vdash\mathsf{roll}\ e:\mu(\alpha;u{:}\mathcal{U}.\tau;U)\mid G}$$

(S:Unroll)

$$\frac{\Delta;\Omega;\Psi;\Gamma\vdash e:\mu(\alpha;u{:}\mathcal{U}.\tau;U)\mid G}{\Delta;\Omega;\Psi;\Gamma\vdash\mathsf{unroll}\ e:\tau[U/u][\lambda u'{:}\mathcal{U}.\mu(\alpha;u{:}\mathcal{U}.\tau;u')/\alpha]\mid G}$$

(S:Future)

$$\frac{\begin{array}{c}\Omega\rightsquigarrow\Omega_1\boxplus\Omega_2\\ \Delta;\Omega_1;\Psi;\Gamma\vdash e:\tau\mid G \qquad \Omega_2;\cdot\vdash U:\blacklozenge \qquad \cdot;\Psi\vdash U:\blacklozenge\end{array}}{\Delta;\Omega;\Psi;\Gamma\vdash\mathsf{future}[U]\ e:\tau\ \mathsf{future}[U]\mid G\swarrow_U}$$

(S:Touch)

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

(S:New)

$$\frac{\Delta;\Omega,u:\mathcal{U};\Psi,u:\mathcal{U};\Gamma\vdash e:\tau\mid G \qquad \Delta;\Psi;\cdot\vdash\tau::\mathsf{Ty}}{\Delta;\Omega;\Psi;\Gamma\vdash\mathsf{new}\ u{:}\mathcal{U}.e:\tau\mid\mathsf{new}\ u{:}\mathcal{U}.G}$$

(S:Type-Eq)

$$\frac{\Delta;\Psi;\cdot\vdash\tau_1\equiv\tau_2::\mathsf{Ty} \qquad \Delta;\Omega;\Psi;\Gamma\vdash e:\tau_1\mid G}{\Delta;\Omega;\Psi;\Gamma\vdash e:\tau_2\mid G}$$

Fig. 15. Graph type system for $\lambda^{G\mu}$.

$$\frac{}{u \Downarrow u}\,(\text{UV:Var}) \qquad \frac{}{\mathsf{gen}(\vec{u}) \Downarrow \mathsf{gen}(\vec{u})}\,(\text{UV:Path}) \qquad \frac{U_1 \Downarrow U_1' \quad U_2 \Downarrow U_2'}{(U_1, U_2) \Downarrow (U_1', U_2')}\,(\text{UV:Pair}) \qquad \frac{U \Downarrow (U_1, U_2)}{\mathsf{fst}\ U \Downarrow U_1}\,(\text{UV:FstPair}) \qquad \frac{U \Downarrow U' \quad U' \neq (U_1, U_2)}{\mathsf{fst}\ U \Downarrow \mathsf{fst}\ U'}\,(\text{UV:FstNotPair})$$

Fig. 16. Selected rules for vertex structure normalization.

Rule S:Type-Eq ensures that typing respects type constructor equivalence. Rules S:Roll and S:Unroll roll and unroll parameterized recursive data types: $\mu(\alpha; u{:}\mathcal{U}.\tau; U)$ unrolls to $\tau[U/u][\lambda u' : \mathcal{U}.\mu(\alpha; u : \mathcal{U}.\tau; u')/\alpha]$ by applying U to $u : \mathcal{U}.\tau$ (hence $\tau[U/u]$) and then substituting itself recursively. Instead of replacing the α with an instance of the recursive type, we replace it with a type-level VS function that passes its argument to the recursive type.

4 SOUNDNESS

The goal of this section is to prove the soundness of the graph type system for $\lambda^{G\mu}$; that is, that the computation graph of a program is described by its graph type. In order to prove this theorem, we must first formalize 1) the notion of a computation graph being "described by" a graph type and 2) the operational semantics by which a program evaluates to produce a computation graph. The first notion is one of *normalization* [Muller 2022], a process for constructing the set of computation graphs corresponding to a given graph type. The second is a *cost semantics*, which we present as a big-step semantics that evaluates an expression to a value and a computation graph.

The rest of this section is structured as follows. In Section 4.1, we discuss how we represent the creation of new vertices (which occurs during both normalization, as new bindings are normalized, and during evaluation, as they are evaluated). We then formalize normalization (Section 4.2), and finally present the cost semantics and prove soundness (Section 4.3).

4.1 Generation of Runtime Vertex Names

Recall that the constructs new $u{:}\mathcal{U}.G$ in graph types and new $u{:}\mathcal{U}.e$ in expressions bind "fresh" vertex structure (VS) variables to be used in the graph type and the expression, respectively. Because VSs can be infinite and of arbitrary type, some care must be taken in how to represent them at "runtime", i.e., in normalization and the cost semantics. The key insight is that VSs are (possibly infinite) trees with unique vertices at each leaf. It is thus possible to uniquely identify a vertex by the VS it comes from, and the path taken to reach it from the root of the VS. Paths in a vertex structure U are already represented in our syntax as sequences of projections, e.g., fst snd U, so most of the new conceptual work is in representing the roots of the vertex structures.

We use the syntax $\vec{u}$ and variants to represent a unique vertex name, called a *generator*, which serves as the root of a VS $\mathsf{gen}(\vec{u})$. Generators are included in the contexts Ω and Ψ like VS variables, but are meaningful runtime symbols representing unique VSs. We will use the notation Ω° and Ψ° to refer to contexts that contain only generators, and no variables. These are the only contexts that will exist at runtime for typing top-level terms and graph types, as such terms and graph types contain no free variables. We refer to these terms and graph types as *closed* even though they may contain free vertex names in the form of generators. The judgments for VS typing and Ω context splitting are extended with rules for generators that resemble the rules for variables.

Equipped with a way to represent the roots of new vertex structures, we turn our attention again to paths from the root to a vertex. Currently, the same path can be represented in multiple ways, for example $\Psi \vdash \mathsf{fst}\ (\mathsf{fst}\ U, \mathsf{snd}\ U) \equiv \mathsf{fst}\ U : \mathcal{U}$. It will be useful to have a normal form for paths, so we introduce a normalization operation on VSs with the judgment $U \Downarrow U$, defined in Figure 16

$$
\begin{array}{llll}
\textit{VSs} & U & ::= & \ldots \mid \mathit{VP} \\
\textit{Vertex Path} & \mathit{VP} & ::= & \mathsf{gen}(\vec{u}) \mid \mathsf{fst}\ \mathit{VP} \mid \mathsf{snd}\ \mathit{VP} \\
\textit{Expressions} & e & ::= & \ldots \mid v \\
\textit{Values} & v & ::= & \langle\rangle \mid \mathsf{fun}[u;u]\ f\ x = e \mid (v,v) \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \mathsf{roll}\ v \mid \mathsf{handle}[\mathit{VP}]\ v
\end{array}
$$

Fig. 17. Extended syntax for generators, vertex paths, and values

$$
\text{(S:Handle)}\quad \frac{\Delta; \cdot; \Psi; \Gamma \vdash v : \tau \mid \bullet \qquad \cdot; \Psi \vdash \mathit{VP} : \blacklozenge}{\Delta; \Omega; \Psi; \Gamma \vdash \mathsf{handle}[\mathit{VP}]\ v : \tau\ \mathsf{future}[\mathit{VP}] \mid \bullet}
$$

Fig. 18. Rule for typing handles.

$$
\text{(UR:Seq1)}\ \frac{G_1 \hookrightarrow G_1'}{G_1 \oplus G_2 \hookrightarrow G_1' \oplus G_2}
\qquad
\text{(UR:Seq2)}\ \frac{G_2 \hookrightarrow G_2'}{G_1 \oplus G_2 \hookrightarrow G_1 \oplus G_2'}
\qquad
\text{(UR:Rec)}\ \frac{}{\mu\gamma.G \hookrightarrow G[\mu\gamma.G/\gamma]}
$$

Fig. 19. Selected rules for graph type unrolling.

$$
\begin{array}{rcll}
\mathsf{NBNF}(\bullet) & \triangleq & \bullet & \\
\mathsf{NBNF}(G_1 \oplus G_2) & \triangleq & \mathsf{NBNF}(G_1) \oplus \mathsf{NBNF}(G_2) & \\
\mathsf{NBNF}(G_1 \vee G_2) & \triangleq & \mathsf{NBNF}(G_1) \vee \mathsf{NBNF}(G_2) & \\
\mathsf{NBNF}(\mu\gamma.G) & \triangleq & \mu\gamma.G & \\
\mathsf{NBNF}(G \swarrow_{U}) & \triangleq & \mathsf{NBNF}(G) \swarrow_{U'} & U \Downarrow U' \\
\mathsf{NBNF}(^{U}\searrow) & \triangleq & ^{U'}\searrow & U \Downarrow U' \\
\mathsf{NBNF}(\mathsf{new}\ u{:}\mathcal{U}.G) & \triangleq & \mathsf{NBNF}(G[\mathsf{gen}(\vec{u})/u]) & \vec{u}\ \text{fresh} \\
\mathsf{NBNF}(\Pi[u_f : \mathcal{U}_f; u_t : \mathcal{U}_t].G) & \triangleq & \Pi[u_f : \mathcal{U}_f; u_t : \mathcal{U}_t].G & \\
\mathsf{NBNF}(G[U_f; U_t]) & \triangleq & \mathsf{NBNF}(G'[U_f, U_t/u_f, u_t]) & \mathsf{NBNF}(G) = \Pi[u_f{:}\mathcal{U}_f; u_t{:}\mathcal{U}_t].G' \\
\mathsf{NBNF}(G[U_f; U_t]) & \triangleq & G[U_f; U_t] & \text{o.w.}
\end{array}
$$

Fig. 20. New-β normal form.

(rules symmetric to these are omitted). Intuitively, the operation β-reduces any projections of pairs (but leaves alone projections of vertex structures that are not syntactically pairs, e.g. fst gen($\vec{u}$)).

We use the term *vertex paths*, and the notation *VP*, to refer to closed, normal VSs. Figure 17 extends the syntax for VSs with generators and gives the syntax for vertex paths. We now have a way of producing references to unique vertices: generators give rise to unique, non-intersecting VSs, and unique vertex paths in a given VS refer to unique vertices. When evaluating a new binding, we will simply create a fresh generator. The remainder of the vertices in the VS are then created implicitly, and will be accessed by the program as it traverses the VS.

4.2 Normalization

Figures 19–21 present the machinery for normalization. Because a recursive graph type represents an infinite set of graphs (as it can be unrolled any number of times), we stage the construction of these sets so that every set constructed is finite. Constructing a set of graphs consists of three

$$\begin{array}{rcl}
Exp(\bullet) & \triangleq & \{\bullet\} \\
Exp(G_1 \oplus G_2) & \triangleq & \{g'_1 \oplus g'_2 \mid g'_1 \in Exp(G_1), g'_2 \in Exp(G_2)\} \\
Exp(G_1 \vee G_2) & \triangleq & Exp(G_1) \cup Exp(G_2) \\
Exp(\mu\gamma.G) & \triangleq & \emptyset \\
Exp(G[U_f; U_t]) & \triangleq & \emptyset \\
Exp(G \swarrow_U) & \triangleq & \{g' \swarrow_U \mid g' \in Exp(G)\} \\
Exp({}^{U}\searrow) & \triangleq & \{{}^{U}\searrow\}
\end{array}$$

Fig. 21. Graph type expansion.

operations, each of which performs some of the required tasks. First, recursive graph types are unrolled a desired number of times, yielding another graph type that is equivalent up to unrollings of recursive μ bindings. Next, the graph type is reduced to "New-β normal form" (NBNF), which "evaluates" any exposed "new" bindings by generating and substituting fresh vertex structures. This process also performs any applicable β reductions on exposed applications. At this point, we are left with a valid graph type, but one with no exposed "new" bindings or applications. Finally, the resulting graph type is *expanded* into the set of graphs; because there are no exposed "new" bindings, this process does not involve generating any new vertex names or structures. We will now discuss each of these steps in more detail.

Figure 19 gives a small-step semantics for unrolling recursive bindings in graph types. The bulk of the work is done by rule UR:Rec, which steps a binding $\mu\gamma.G$ to $G[\mu\gamma.G/\gamma]$. The remaining rules "search" the graph type for recursive bindings, so we defer most of these rules to the full version of the paper [Rinaldi et al. 2024]. Note that, unlike in a standard left-to-right (or right-to-left) operational semantics, the rules UR:Seq1 and UR:Seq2 allow any instance of recursion in the type to be unrolled at any time in a nondeterministic fashion. For example, both steps below are valid:

$$(\mu\gamma.\gamma \oplus \gamma) \oplus (\mu\gamma.\gamma \vee \gamma) \hookrightarrow ((\mu\gamma.\gamma \oplus \gamma) \oplus (\mu\gamma.\gamma \oplus \gamma)) \oplus (\mu\gamma.\gamma \vee \gamma)$$
$$(\mu\gamma.\gamma \oplus \gamma) \oplus (\mu\gamma.\gamma \vee \gamma) \hookrightarrow (\mu\gamma.\gamma \oplus \gamma) \oplus ((\mu\gamma.\gamma \vee \gamma) \oplus (\mu\gamma.\gamma \vee \gamma))$$

The rules for reducing to NBNF, given in Figure 20, eliminate "new" bindings by substituting fresh vertex structure generators, and perform any exposed applications. Evaluation proceeds recursively through sequential compositions and alternatives, but not under binders. As a result, closed sub-graph-types of NBNF graph types are themselves NBNF.

Finally, Figure 21 gives the rules for expanding a graph type into the set of graphs it represents. We represent a graph g formally as a 4-tuple (V, E, s, t) containing the sets of vertices V and edges E, as well as a designated "start" vertex s and "end" vertex t. We use shorthands for combining graphs sequentially and in parallel; these shorthands use many of the same operators as graph type composition, but should not be confused. Figure 22 gives formal definitions for these shorthands; for more description of them, the reader is referred to prior work [Muller 2022]. In brief, sequential composition $\oplus$ joins the end vertex of the first graph to the start vertex of the second graph. The "left composition" operator [Spoonhower 2009], written $g \swarrow_u$, adds a subgraph g corresponding to a future to the graph, with an edge representing the spawn. It also adds the vertex u as the sink of the future's graph. The "touch" operator ${}^{u}\searrow$ adds an edge from u.

Sequential compositions are expanded by expanding both subgraphs, and then sequentially composing the resulting graphs. Alternation simply takes the union of the two sets of graphs. Expansion does not perform any additional unrolling, so the expansion of a recursive graph type is the empty set of graphs. Expansion of the future $G \swarrow_U$ is performed by left-composing all of the resulting graphs with the vertex U, and touches ${}^{U}\searrow$ simply expand to the singleton graph

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

Fig. 22. Shorthands for combining graphs.

$$\text{(C:Value)}\ \frac{}{v \Downarrow v \mid \bullet} \qquad \text{(C:App)}\ \frac{e_1 \Downarrow \mathsf{fun}[u_f; u_t]\ f\ x = e \mid g_1 \qquad e_2 \Downarrow v' \mid g_2 \qquad e[\mathsf{fun}[u_f; u_t]\ f\ x = e/f][U_f/u_f][U_t/u_t][v'/x] \Downarrow v \mid g_3}{e_1[U_f; U_t]\ e_2 \Downarrow v \mid g_1 \oplus g_2 \oplus g_3} \qquad \text{(C:Pair)}\ \frac{e_1 \Downarrow v_1 \mid g_1 \qquad e_2 \Downarrow v_2 \mid g_2}{(e_1, e_2) \Downarrow (v_1, v_2) \mid g_1 \oplus g_2}$$

$$\text{(C:Fst)}\ \frac{e \Downarrow (v_1, v_2) \mid g}{\mathsf{fst}\ e \Downarrow v_1 \mid g} \qquad \text{(C:InL)}\ \frac{e \Downarrow v \mid g}{\mathsf{inl}\ e \Downarrow \mathsf{inl}\ v \mid g} \qquad \text{(C:CaseL)}\ \frac{e \Downarrow \mathsf{inl}\ v \mid g_1 \qquad e_1[v/x] \Downarrow v' \mid g_2}{\mathsf{case}\ e\ \{x.e_1; y.e_2\} \Downarrow v' \mid g_1 \oplus g_2} \qquad \text{(C:Roll)}\ \frac{e \Downarrow v \mid g}{\mathsf{roll}\ e \Downarrow \mathsf{roll}\ v \mid g}$$

$$\text{(C:Unroll)}\ \frac{e \Downarrow \mathsf{roll}\ v \mid g}{\mathsf{unroll}\ e \Downarrow v \mid g} \qquad \text{(C:Future)}\ \frac{e \Downarrow v \mid g \qquad U \Downarrow VP}{\mathsf{future}[U]\ e \Downarrow \mathsf{handle}[VP]\ v \mid g \swarrow_{VP}} \qquad \text{(C:Touch)}\ \frac{e \Downarrow \mathsf{handle}[VP]\ v \mid g}{\mathsf{touch}\ e \Downarrow v \mid g \oplus {}^{VP}\searrow} \qquad \text{(C:New)}\ \frac{\vec{u} \text{ fresh} \qquad e[\mathsf{gen}(\vec{u})/u] \Downarrow v \mid g}{\mathsf{new}\ u{:}\mathcal{U}.e \Downarrow v \mid g}$$

Fig. 23. Cost Semantics for $\lambda^{G\mu}$ (selected). Rules symmetric to these are omitted.

consisting of the touch. Note that because NBNF has already expanded all "new" bindings, there is no rule for expanding these, and expansion does not generate new vertex names. The latter is a key property in guaranteeing that expansion results in well-formed graphs.

These three operations combine to form a normalization process that is correct: any set of graphs that results from unrolling, normalizing, and expanding a well-formed graph type is well-formed. This result is formalized by Theorem 1, which is proven in the full version of the paper alongside several necessary technical lemmas.

Theorem 1. *If* $\cdot; \Omega; \Psi \vdash G : \kappa_G$ *and* $G \hookrightarrow^* G'$, *then* $\mathsf{NBNF}(G')$ *exists and if* $g \in \mathit{Exp}(\mathsf{NBNF}(G'))$, *then* g *is a well-formed graph.*

4.3 Cost Semantics and Soundness

We equip $\lambda^{G\mu}$ with a cost semantics, a big-step operational semantics that evaluates an expression and also produces the computation graph that represents the execution. The judgment is $e \Downarrow v \mid g$, meaning that expression e evaluates to value v, producing the cost graph g. The rules for this judgment are in Figure 23, and the syntax for values are in Figure 17. In C:Future, the body of the future is evaluated (in a real execution, the body of the future will be evaluated in parallel, but the big-step cost semantics deliberately abstracts away evaluation order) and the future evaluates to a handle, a new syntactic form which records the result of the future. In addition, we evaluate the vertex structure U used to spawn the future to a vertex path VP, which is recorded by the handle. The C:Touch rule extracts both the vertex path and future result from the handle. In C:New, a new generator $\vec{u}$ is created and used to generate a vertex structure which instantiates the variable u.

The soundness theorem for the graph type system of $\lambda^{G\mu}$ is that if a program e has a graph type G and evaluates to produce a graph g, then g is described by G (that is, g should be in the set of graphs obtained by normalizing G using the machinery in the previous subsection). This is stated formally as Theorem 2. The formal statement of the theorem also includes a context Ψ° containing generators created during execution which may be captured in the result value v.

Theorem 2. *If* $\cdot; \Omega; \Psi; \cdot \vdash e : \tau \mid G$ *and* $e \Downarrow v \mid g$, *then there exists a* Ψ° *such that* $\vec{u} \in \Psi^\circ$ *implies* $\vec{u}$ fresh *and* $\cdot; \cdot; \Psi, \Psi^\circ; \cdot \vdash v : \tau \mid \bullet$ *and there exists a* G' *such that* $G \hookrightarrow^* G'$ *and* $g \in \mathit{Exp}(\mathsf{NBNF}(G'))$.

The proof of the theorem, as well as statements and proofs of several necessary technical lemmas, appears in the full version of the paper [Rinaldi et al. 2024]. These lemmas include:

- A number of substitution results for expressions, types, vertex structures, etc.
- If $\Delta; \Omega; \Psi; \Gamma \vdash e : \tau \mid G$ then $\Delta; \Psi; \cdot \vdash \tau :: \mathsf{Ty}$ and $\Delta; \Omega; \Psi \vdash G : *$ (assuming Γ contains only well-kinded types).
- A "framing" property that allows us to repeatedly unroll a sub-graph type while keeping the rest of the graph type the same, for example, if $G_1 \hookrightarrow^* G_1'$ then $G_1 \oplus G_2 \hookrightarrow^* G_1' \oplus G_2$.

5 ELABORATION OF RECURSIVE TYPES WITH VERTEX STRUCTURES

Thus far, we have presented the annotated language $\lambda^{G\mu}$ containing recursive data types $\mu(\alpha; u : \mathcal{U}.c; \mathit{VP})$, annotated with a vertex path VP of type $\mathcal{U}$ that provides vertex names for data structures of the recursive type. We have motivated that VP should have a structure that in some sense "maps on" to the recursive structure of the list so that any futures in the structure have a corresponding vertex name. As examples, a list data type corresponds to an infinite stream of vertices, and a binary tree data type corresponds to an infinite binary tree of vertices. As discussed, the annotated language is provided merely as a core calculus for expressing the ideas of the graph type system; the annotations can be inferred from unannotated code by our implementation.

Other than the addition of vertex structures, the general structure of the algorithm for inferring these annotations is similar to that of GML [Muller 2022], and the details of the algorithm are largely outside the scope of this paper. However, one important and non-obvious fact for inferring annotations for $\lambda^{G\mu}$ is that it is indeed possible to annotate any recursive data structure with a corresponding vertex path. Showing this fact is the goal of this section. We do so by defining a set of rules for annotating unannotated types and values with vertex structure annotations. For simplicity, the system we present in this section is declarative and not algorithmic, so it still abstracts away many of the complexities of our inference algorithm, but we show that the rules are complete and thus that any recursive type may be so annotated.

We first define a syntax for unannotated types σ and unannotated values ϵ.

$$\begin{array}{rcl} \sigma & ::= & \alpha \mid \mathsf{unit} \mid \Pi[u_t : \mathcal{U}_t; u_f : \mathcal{U}_f].c_1 \to c_2 \mid \sigma \times \sigma \mid \sigma + \sigma \mid \sigma\ \mathsf{future} \mid \mu\alpha.\sigma \\ \epsilon & ::= & \langle\rangle \mid \mathsf{fun}[u; u]\ f\ x = e \mid (\epsilon, \epsilon) \mid \mathsf{inl}\ \epsilon \mid \mathsf{inr}\ \epsilon \mid \mathsf{roll}\ \epsilon \mid \mathsf{handle}\ \epsilon \end{array}$$

Unannotated types consist of the unit type, functions, products, and sums, as well as an unannotated future type and an unannotated recursive type. Note that the annotation of functions is orthogonal to the annotation of recursive data types; we assume that function types and function values have already been annotated and include annotated function types and annotated function values as unannotated types and unannotated values, respectively. The unannotated future type σ future is similar to the annotated future type c future$[U]$ but is not annotated with a VS. Similarly, the unannotated recursive type $\mu\alpha.\sigma$ binds a type variable but does not bind a VS variable and does not take a VS as an argument. Because unannotated types do not interact with vertex structures, there is no type-level lambda and all unannotated types have kind Ty (and so we do not distinguish between "unannotated types" and "unannotated type constructors"). Unannotated values differ from values v only in that future handles are not annotated with vertex paths.

Figure 24 defines the judgment $\Upsilon \vdash_U \sigma \rightsquigarrow c; \mathcal{U}$. This indicates that σ may be annotated to the type constructor c (which will, by construction, have kind Ty). It also returns a vertex structure type $\mathcal{U}$ that "corresponds" to the type σ. For recursive types $\mu\alpha.\sigma$, the VS type $\mathcal{U}$ is the type of the VS annotation for the recursive type (that is, $\mu\alpha.\sigma$ will be annotated to be $\mu(\alpha; u{:}\mathcal{U}.c; U)$ for some c and some U). As an example, if σ is the type of int future lists, then $\mathcal{U}$ will be (equivalent to) the type of vertex streams, $vt.\blacklozenge^{\blacksquare} \times t^{\blacksquare}$. The judgment takes a type variable context Υ mapping type variables to kinds (these will be annotated types and so their kinds will not be Ty). It is also parameterized by a vertex structure U to use for annotations. When annotating a closed unannotated type σ, this parameter will simply be instantiated with a fresh vertex path $\mathsf{gen}(\vec{u})$ to derive $\cdot \vdash_{\mathsf{gen}(\vec{u})} \sigma \rightsquigarrow \tau; \mathcal{U}$. The returned type τ would be annotated with projections of $\mathsf{gen}(\vec{u})$. The returned VS type $\mathcal{U}$ would be the type that $\vec{u}$ should be assigned in order for τ to be well-kinded.

Rule F:TyVar looks up the type variable α in the context. By construction, its kind will be of the form $\mathcal{U} \to \mathsf{Ty}$, indicating that to properly annotate the use of the variable α, it must be applied to a VS of type $\mathcal{U}$. We use the VS U for the annotation and return the type $\mathcal{U}$ as the required type of U. The unit and function types do not require additional annotations, and so are simply returned.[11] Rule F:Prod takes a VS U and annotates the first component σ_1 with the left projection of U and the second component σ_2 with the right projection. The type required for U is thus the product of the two returned types. Rule F:Sum, somewhat counterintuitively, also returns a product of the two VS types. This is because if a data structure can take one of two forms, the corresponding VS must offer either possibility.[12] Rule F:Fut takes U to be a product whose second component is a single vertex, which it uses to annotate the future; the first component is used to annotate the future's return type. Finally, rule F:Rec annotates a recursive type $\mu\alpha.\sigma$. It begins by adding α to the context with kind $vt.\mathcal{U} \to \mathsf{Ty}$ (this is the only truly non-algorithmic feature of these rules; we do not discuss how to construct $\mathcal{U}$). With this context, it annotates σ. The resulting VS type is rolled back into the corecursive type $vt.\mathcal{U}$, which is the type required for U.

Example. We can represent the type of a list of integer futures as an unannotated type σ:

$$\sigma = \mu\alpha.\mathsf{unit} + (\mathsf{int\ future} \times \alpha)$$

Using the rules of Figure 24, we can infer the following annotation for σ:

$$\cdot \vdash_U \sigma \rightsquigarrow \mu(\alpha; u{:}\mathcal{U}.\mathsf{unit} + (\mathsf{int\ future}[\mathsf{fst\ snd}\ u] \times \alpha\ (\mathsf{snd\ snd}\ u)); U); \mathcal{U}$$

where $\mathcal{U} = vt.\blacklozenge^{\blacksquare} \times (\blacklozenge^{\blacksquare} \times t^{\blacksquare})^{\blacksquare}$.

The VS corresponding to σ is a stream of vertices (note that because we treat VS types equi-corecursively, the VS type above is equivalent to $vt.\blacklozenge^{\blacksquare} \times t^{\blacksquare}$ but unrolled slightly). In the body of the recursive annotated type, which is $\mathsf{unit} + (\mathsf{int\ future}[\mathsf{fst\ snd}\ u] \times \alpha\ (\mathsf{snd\ snd}\ u))$, the first vertex of the stream is discarded (this is an effect of mapping the type unit to the VS type $\blacklozenge$ even though it does not need a vertex), the second vertex of the stream (the first vertex of the tail) is used for the future and the remainder (the tail of the tail) is passed to the recursive instance of the type.

The judgment described above declaratively shows a correspondence between unannotated types and the vertex structure types required to annotate them. Later in this section, we show that this relation is complete with respect to well-kinded unannotated types, and thus that any type has a corresponding VS type. We next wish to show that a VS of the returned VS type actually does suffice to provide all necessary vertices for a data structure of the given type. We do this using

[11]The returned VS type is $\blacklozenge$, which will result in the addition of unnecessary vertices to the final VS; it would be straightforward to add a multiplicative unit to VS types, which would be the most appropriate VS type to return here, but we have not done so this far to keep the VS type language as simple as possible.

[12]As an optimization, we could take a "maximum" over the two VS types. For example, in a 2-3 tree, where each node may have two or three children, the corresponding VS could always offer three branches and a 2-node would use the first two.

(F:TyVar)
$$\frac{}{\Upsilon, \alpha :: \mathcal{U} \to \mathsf{Ty} \vdash_U \alpha \rightsquigarrow \alpha\ U; \mathcal{U}}$$

(F:Unit)
$$\frac{}{\Upsilon \vdash_U \mathsf{unit} \rightsquigarrow \mathsf{unit}; \blacklozenge}$$

(F:Fun)
$$\frac{}{\Upsilon \vdash_U \Pi[u_f : \mathcal{U}_f; u_t : \mathcal{U}_t].c_1 \xrightarrow{G} c_2 \rightsquigarrow \Pi[u_f : \mathcal{U}_f : u_t \mathcal{U}_t; c_1 \xrightarrow{G} c_2].; \blacklozenge}$$

(F:Prod)
$$\frac{\Upsilon \vdash_{\mathsf{fst}\ U} \sigma_1 \rightsquigarrow c_1; \mathcal{U}_1 \qquad \Upsilon \vdash_{\mathsf{snd}\ U} \sigma_2 \rightsquigarrow c_2; \mathcal{U}_2}{\Upsilon \vdash_U \sigma_1 \times \sigma_2 \rightsquigarrow c_1 \times c_2; \mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{\blacksquare}}$$

(F:Sum)
$$\frac{\Upsilon \vdash_{\mathsf{fst}\ U} \sigma_1 \rightsquigarrow c_1; \mathcal{U}_1 \qquad \Upsilon \vdash_{\mathsf{snd}\ U} \sigma_2 \rightsquigarrow c_2; \mathcal{U}_2}{\Upsilon \vdash_U \sigma_1 + \sigma_2 \rightsquigarrow c_1 + c_2; \mathcal{U}_1^{\blacksquare} \times \mathcal{U}_2^{\blacksquare}}$$

(F:Fut)
$$\frac{\Upsilon \vdash_{\mathsf{fst}\ U} \sigma \rightsquigarrow c; \mathcal{U}}{\Upsilon \vdash_U \sigma\ \mathsf{future} \rightsquigarrow c\ \mathsf{future}[\mathsf{snd}\ U]; \mathcal{U}^{\blacksquare} \times \blacklozenge^{\blacksquare}}$$

(F:Rec)
$$\frac{\Upsilon, \alpha :: \nu t.\mathcal{U} \to \mathsf{Ty} \vdash_u \sigma \rightsquigarrow c; \mathcal{U}[\nu t.\mathcal{U}/t]}{\Upsilon \vdash_U \mu\alpha.\sigma \rightsquigarrow \mu(\alpha; u : \nu t.\mathcal{U}.c; U); \nu t.\mathcal{U}}$$

Fig. 24. Annotating types with vertex structures.

(FE:Unit)
$$\frac{}{\vdash_{VP} \langle\rangle \rightsquigarrow \langle\rangle}$$

(FE:Fun)
$$\frac{}{\vdash_{VP} \mathsf{fun}[u_f; u_t]\ f\ x = e \rightsquigarrow \mathsf{fun}[u_f; u_t]\ f\ x = e}$$

(FE:Pair)
$$\frac{\vdash_{\mathsf{fst}\ VP} \epsilon_1 \rightsquigarrow v_1 \qquad \vdash_{\mathsf{snd}\ VP} \epsilon_2 \rightsquigarrow v_2}{\vdash_{VP} (\epsilon_1, \epsilon_2) \rightsquigarrow (v_1, v_2)}$$

(FE:InL)
$$\frac{\vdash_{\mathsf{fst}\ VP} \epsilon \rightsquigarrow v}{\vdash_{VP} \mathsf{inl}\ \epsilon \rightsquigarrow \mathsf{inl}\ v}$$

(FE:InR)
$$\frac{\vdash_{\mathsf{snd}\ VP} \epsilon \rightsquigarrow v}{\vdash_{VP} \mathsf{inr}\ \epsilon \rightsquigarrow \mathsf{inr}\ v}$$

(FE:Roll)
$$\frac{\vdash_{VP} \epsilon \rightsquigarrow v}{\vdash_{VP} \mathsf{roll}\ \epsilon \rightsquigarrow \mathsf{roll}\ v}$$

(FE:Handle)
$$\frac{\vdash_{\mathsf{fst}\ VP} \epsilon \rightsquigarrow v}{\vdash_{VP} \mathsf{handle}\ \epsilon \rightsquigarrow \mathsf{handle}[\mathsf{snd}\ VP]\ v}$$

Fig. 25. Annotating values with vertex structures.

another judgment that annotates unannotated values. This judgment is defined in Figure 25 and takes the form $\vdash_{VP} \epsilon \rightsquigarrow v$, where VP is a vertex path to use for annotation (similar to the type annotation judgment above), ϵ is an unannotated value, and v is the annotated value. We restrict annotations of values to vertex paths since the value $\mathsf{handle}[VP]\ v$ may only use vertex paths VP as the handle. Otherwise, annotation of values proceeds in much the same way as annotation of types. Rule FE:Pair uses the two components of VP to annotate the components of the pair. Rules FE:InL and FE:InR use the first and second components, respectively, of VP to annotate left and right injections (recall that, for a sum type, VP is given a product type so that the two components of VP may be used for the two injections). Finally, just as F:Fut uses the first component of U to annotate the type of the future's payload and the second component as the vertex for the future, rule FE:Handle uses fst VP to annotate the payload and snd VP to annotate the handle itself.

Example. Consider the list of integer futures from above. We claimed that the correct VS type for this type is $\mathcal{U} = \nu t.\blacklozenge^{\blacksquare} \times t^{\blacksquare}$. The rules of Figure 25 provide a "recipe" for constructing a future list using a vertex path of VS type $\mathcal{U}$. As an example, consider the unannotated value

$$\mathsf{roll}\ \mathsf{inr}\ (\mathsf{handle}\ 1, \mathsf{roll}\ \mathsf{inr}\ (\mathsf{handle}\ 2, \mathsf{roll}\ \mathsf{inl}\ \langle\rangle))$$

(SV:Handle)

$$\frac{\Omega \rightsquigarrow \Omega_1 \boxplus \Omega_2 \qquad \Omega_1 \vdash_A v : \tau \qquad \Omega_2; \cdot \vdash VP : \blacklozenge}{\Omega \vdash_A \mathsf{handle}[VP]\ v : \tau\ \mathsf{future}[VP]}$$

Fig. 26. Affine typing rule for handle values.

$$\begin{aligned}
\mathsf{Unann}(\cdot) &\triangleq \cdot \\
\mathsf{Unann}(\Upsilon, \alpha :: \kappa) &\triangleq \mathsf{Unann}(\Upsilon), \alpha :: \mathsf{Ty} \\
\\
\mathsf{Ann}(\cdot) &\triangleq \cdot \\
\mathsf{Ann}(\Upsilon, \alpha :: \mathsf{Ty}) &\triangleq \mathsf{Ann}(\Upsilon), \alpha :: t \to \mathsf{Ty} \quad t \text{ fresh}
\end{aligned}$$

Fig. 27. Annotating and unannotating kinds in Υ.

which represents the list containing two future handles, one returning 1 and the other returning 2. Applying the rules, we get the expression

$$\mathsf{roll\ inr}\ (\mathsf{handle}[\mathsf{fst\ snd}\ VP]\ 1, \mathsf{roll\ inr}\ (\mathsf{handle}[\mathsf{fst\ snd\ snd\ snd}\ VP]\ 2, \mathsf{roll\ inl}\ \langle\rangle))$$

As described above, the futures take consecutive odd vertices from the stream *VP*.

The main result of this section has three components. First, any well-kinded unannotated type may be matched with an annotated type by the rules of Figure 24. Second, if a well-kinded unannotated type is annotated with a VS of the VS type returned by the annotation judgment, then the annotated type is also well-kinded. Third, if an unannotated type σ is annotated with a vertex path (that is, if $\cdot \vdash_{VP} \sigma \rightsquigarrow \tau; \mathcal{U}$), then any well-typed unannotated value of type σ may be annotated with *VP* by the rules of Figure 25, and the annotated value is well-typed when *VP* has type $\mathcal{U}$. Moreover, to show that *VP* has "enough" vertices to fully annotate the value with unique vertices, we show that the annotated value is well-typed under a new typing judgment that uses only an affine context for vertices. Usually, values would be typed with the unrestricted context Ψ, because a data structure is allowed to contain multiple handles to the same future, but in this case, we wish to show that we *can* restrict data structures to use new vertices for each handle. We write the new judgment $\Omega \vdash_A v : \tau$. The rules are similar to the standard typing rules, but use the affine context Ω for typing handles. This rule for typing handle values is given in Figure 26. Values always have the graph type $\bullet$, so we omit the graph type from the judgment. The remaining rules are straightforward and are deferred to the full version of the paper [Rinaldi et al. 2024].

Theorem 3 formalizes the main result of this section, that is, that 1) type annotation is complete with respect to well-kinded unannotated types, 2) type annotation annotates well-kinded unannotated types into well-kinded types, and 3) annotating values with vertex structures of the returned VS type results in well-typed values. In order to show this, we introduce a kinding judgment for unannotated types, $\Upsilon \vdash \sigma :: \mathsf{Ty}$, and a typing judgment for unannotated values, $\vdash \epsilon : \sigma$. The rules for these judgments are similar to those for annotated types and expressions, so we defer them to the full version of the paper [Rinaldi et al. 2024]. Since the kinds of type variables bound by unannotated and annotated recursive types are different (Ty and $\mathcal{U} \to \mathsf{Ty}$ respectively), we need some way to change the kinds that these type variables are bound to. We address this with the functions Unann and Ann. $\mathsf{Unann}(\Upsilon)$ takes a context Υ suitable for annotating types and kinding annotated types (where type variables can, and will always, have kind $\mathcal{U} \to \mathsf{Ty}$), and return an *unannotated context*, one suitable for kinding unannotated types (where every type variable has kind Ty). $\mathsf{Ann}(\Upsilon)$ performs this process in reverse, where the VS type expected by every type variable in Υ is a fresh VS type variable unique to that type variable (each which can be substituted with the desired VS type).

The proof of Theorem 3, as well as statements and proofs of several necessary technical lemmas, appears in the full version of the paper [Rinaldi et al. 2024].

Theorem 3.

(1) *For an unannotated context* Υ*, if* $\Upsilon \vdash \sigma :: \mathsf{Ty}$*, then for any* U*, there exist* τ *and* $\mathcal{U}$ *such that* $\mathsf{Ann}(\Upsilon) \vdash_U \sigma \rightsquigarrow \tau; \mathcal{U}$*.*
(2) *If* $\Upsilon \vdash_U \sigma \rightsquigarrow \tau; \mathcal{U}$ *and* $\mathsf{Unann}(\Upsilon) \vdash \sigma :: \mathsf{Ty}$ *and* $\cdot; \Psi \vdash U : \mathcal{U}$*, then* $\cdot; \Psi; \Upsilon \vdash \tau :: \mathsf{Ty}$*.*
(3) *If* $\cdot \vdash_{VP} \sigma \rightsquigarrow \tau; \mathcal{U}$ *and* $\vdash \epsilon : \sigma$ *and* $\cdot \vdash \sigma :: \mathsf{Ty}$*, then there exists* v *such that* $\vdash_{VP} \epsilon \rightsquigarrow v$ *and for any* Ω *such that* $\Omega; \cdot \vdash VP : \mathcal{U}$*. we have* $\Omega \vdash_A v : \tau$*.*

6 IMPLEMENTATION AND EXAMPLES

We have implemented a prototype graph inference algorithm for $\lambda^{G\mu}$ on top of GML [Muller 2022], an existing graph type inference algorithm. The goal of the implementation, which we call GML$^\mu$, is to infer vertex structure annotations and graph types from ordinary, unannotated OCaml programs. GML extends OCaml syntax with the keywords `future` for spawning expressions into a future, `touch` for joining a future handle's value to the current thread, and a type `'a future`. Additionally, GML$^\mu$ supports OCaml's user-definable recursive datatypes, which were not previously supported by GML (there are some limitations, which we discuss at the end of this section). For example, we can define the `'a pipe` type from Sections 1 and 2 using standard OCaml syntax:

```
type 'a pipe = Pipe of 'a * 'a pipe future ;;
```

Our extension of GML successfully infers the corresponding vertex structure annotations, for the type itself and for all of its uses in the code in Figure 5.

In addition, we implemented (by extending facilities existing in GML) a visualizer that uses several heuristics to generate a visualization of a representative graph corresponding to each inferred graph type.[13] This allows developers to see at a glance how their program will parallelize. We have used GML$^\mu$ to infer graph types for all example programs in this paper.

The details of the implementation are out of the scope of the paper. However, the main challenge in extending GML with support for algebraic data types is generating the VS type corresponding to an ADT. Our algorithm for this closely follows the presentation of Section 5.[14] When processing a type declaration, GML generates the associated VS type, and also generates a constructor and deconstructor function for each constructor. Constructor applications are desugared to ordinary applications of the constructor function and the deconstructor function is used during pattern matching. Another major challenge is implementing unification on vertex structures. At the moment, our implementation uses a set of heuristics that are not guaranteed to be complete (i.e., unification may fail for VSs that could be unified, resulting in a spurious type error) but work well in practice on the large examples tested.

In addition to extending the subset of OCaml supported by GML, we have also substantially re-architected the code. In GML$^\mu$, the graph type checker is completely separate from the type checker. This simplifies the implementation and has a number of other benefits. First, all futures in a program are known by the time graph checking begins. This allows the implementation to infer graph types in several instances where type annotations would previously have been required (one such instance is noted in prior work [Muller 2022] as a limitation of GML, which is not a limitation of GML$^\mu$). Additionally, this architecture would simplify the process of integrating graph checking as an extension of the OCaml compiler, as an additional pass on type-checked ASTs.

[13] Once the graph type is unrolled to generate the representative graph, we output a file that can be turned into a visualization using GraphViz [Gansner and North 2000].
[14] As discussed in that section, the only non-algorithmic detail of the presentation was constructing $\mathcal{U}$ in F:Rec; in the implementation, we add $t \to \mathsf{Ty}$ to the context instead of $\nu t.\mathcal{U} \to \mathsf{Ty}$. This means the context contains non-well-formed VS types, which makes the theory more unwieldy but yields a convenient implementation.

```
type 'a flist =
| FNil
| FCons of 'a * ('a flist future);;

let rec produce n =
  if n < 0 then FNil
  else FCons (n, future (produce (n - 1)));;

let rec consume sumxs =
  let (sum, xs) = sumxs in
  match xs with
  | FNil -> sum
  | FCons (x, xs) -> consume (x + sum, touch xs);;

consume (0, (produce n));;
```

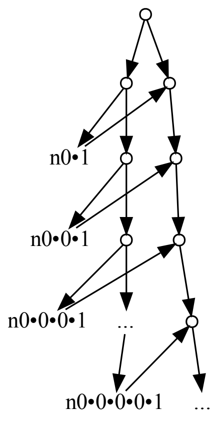

Fig. 28. Blelloch-Reid-Miller produce-consume example

6.1 Examples

To show the utility of GML$^\mu$, we discuss several example programs for which it can infer and visualize graph types.

Produce-consume. The producer-consumer example of Blelloch and Reid-Miller [1997], shown in Figure 28, is similar to the pi_pipeline function of Figure 5, but allows the pipelined list to be finite (ending with FNil). As in the pipeline example, the FCons constructor allows the tail of the flist to continue being computed in a future. We compose produce, which (for the sake of a simple example) outputs a list of the numbers 1–n, with the consume function which calculates the sum of the list. In the graph of the composed functions (right side of the figure), the touches of consume happen in parallel with the production of the list.

Tree Sum. In Figure 29, we present operations on a pipelined tree data structure [Blelloch and Reid-Miller 1997]. As with flist, the two subtrees of an ftree are futures, so they may be computed asynchronously while the value at the node is used. The function bst generates a tree of numbers 0 to 10, then tree_sum calculates the sum of elements in the generated tree. While the particular application of summing a binary tree is fairly simple, one can imagine using the same structure for more complicated use-cases. Because of the design of the data structure, bst immediately returns a future and then tree_sum can perform its calculation as later recursive steps of bst are still executing.

Tree Reverse. The function in Figure 30 reverses a pipelined tree of the type defined in Figure 29. Here, the interesting feature of the output was not the visualization of the function's graph type, which shows a similar structure to tree_sum and bst, but the function type, which is

$$\text{reverse} : \Pi[u_f : \text{vtree}; u_t : \text{vtree}].\text{ftree}\ u_t \xrightarrow{G} \text{ftree}\ u_f$$

where vtree $= vt.t \times \blacklozenge \times t \times \blacklozenge$. We omit the graph type G for clarity. The function takes two VS parameters and a tree indexed by u_t and returns a tree indexed by u_f. At first glance, this may seem imprecise because one might expect the VSs parameterizing the input and output tree to be related (after all, the output tree is the reverse of the input). However, this is not correct: reverse

```
type ftree =
| Empty
| Node of int * ftree future * ftree future;;

let rec bst lohi =
  let (lo, hi) = lohi in
  if lo >= hi then Empty
  else
    let mid = (lo + hi) / 2 in
    Node (mid, future (bst (lo, mid)),
          future (bst (mid, hi)));;

let rec tree_sum tree =
  match tree with
  | Empty -> 0
  | Node (x, l, r) ->
    let left_sum_fut = future (tree_sum (touch l)) in
    let right_sum = tree_sum (touch r) in
    let left_sum = touch left_sum_fut in
    x + left_sum + right_sum;;

tree_sum (bst (0, 10));;
```

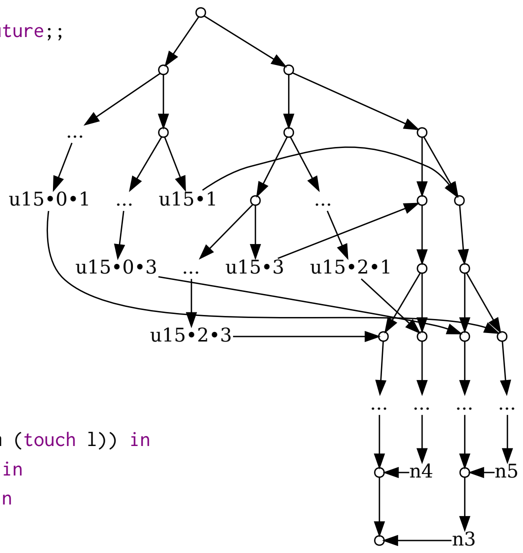

Fig. 29. Sum over elements in a pipelined tree.

```
let rec reverse tree =
  match tree with
  | Empty -> Empty
  | Node (x, l, r) -> Node (x, future (reverse (touch r)), future (reverse (touch l)))
```

Fig. 30. Reverse a pipelined tree.

touches (in a pipelined way) all of the futures of the input tree and constructs a new tree with the reversed *values* but *new futures* from the VS u_f. Here, not just the graph type but the return type parameterized by its VS can correct a misunderstanding about the parallel behavior of a program.

6.2 Limitations

Though our inference algorithm checks most useful programs, there are some limitations. First, polymorphic types cannot be instantiated with types that include futures. For example, a list of futures would have to be explicitly defined as a new type `'a futlist` rather than by instantiating built-in lists to form the type `'a future list`. This represents a design trade-off; graph types are not currently expressive enough to represent, say, a `reverse` function on lists of futures. If the standard list type could be instantiated with a future, the polymorphic `reverse` function would need to be assigned a general enough type to cover all instantiations of `'a`, which wouldn't be possible. A limitation we inherit from GML is that functional arguments of higher order functions cannot spawn futures. This is possible in $\lambda^{G\mu}$, but cannot be inferred without annotations.

7 RELATED WORK

Graph Types and Related Analyses. The use of graphs to represent the parallel programs dates back to at least the late 1960s [Karp and Miller 1966; Rodriguez Bezos 1969]. Our notation is most directly inspired by the work of Blelloch and Greiner [1995, 1996] and Spoonhower [2009], who extended these graphs with notations for futures. In this work, graphs were produced dynamically from programs using a *cost semantics*, which abstractly evaluates the program to form the graph (or a family of graphs if execution is nondeterministically). The first work we are aware of on statically approximating such graphs for fine-grained parallel programs was the prior work of one of us [Muller 2022], which developed a calculus λ^G and corresponding graph type system for inferring graph types of parallel programs with futures. Our work builds on λ^G, including the use of an affine type system to ensure that vertex names are unique and therefore do not appear twice in a graph, which would result in an invalid graph. However, the main thrust of this paper is overcoming the significant limitation in λ^G that affine treatment of vertex names prevents building collections of futures.

Dependency graphs are frequently used to represent control dependencies in coarse-grained parallel programs and these have been the target of several static analyses (e.g., [Chen et al. 2002; Cheng 1993; Kasahara et al. 1995]) but such tools do not contend with the substantial dynamicity inherent in fine-grained parallel programs, especially those with futures. Dependency graphs are also used to represent other dependencies in a program, including data dependencies; analyzing the structure of such dependencies is a form of program slicing (e.g., [Korel 1987; Weiser 1984]).

As observed in prior work [Muller 2022], graph type systems draw on ideas from *region type systems* [Tofte and Talpin 1997], where assigning a vertex to a future corresponds to allocating an object within a *region* of memory, in order to aid in memory management and/or ensure safety (e.g. [Fluet et al. 2006]), including in the presence of concurrency and complex, dynamic data structures [Milano et al. 2022]. It is not possible to list all of the related work on regions and related systems, so we refer the interested reader to the chapter by Henglein et al. [2005]. Two major differences with region systems are that vertex assignments must be unique (whereas typically many objects are allocated within a single region) and that, to generate useful graphs, we wish for vertex assignments to be visible at a global scope (see the example from the Introduction of why locally allocated vertices are not suitable for graph types of data structures).

Heterogeneous and Indexed Data Structures. Indexed types [Xi and Pfenning 1999; Zenger 1997], a limited form of dependent types in which a type is *indexed* by a value from a specified domain, have long been used to add expressiveness to types—a classic example is a type of vectors indexed with a natural number giving the vector's length. We index recursive data types with a vertex structure to assign unique vertices to futures in recursive data structures. Vertex structures have a non-trivial semantics of their own but are not first-class objects at the expression level, so computation on VSs may be seen as an instance of *type-level computation.* A similar indexing idea and type-level computation have been previously combined to achieve heterogeneity in HList [Kiselyov et al. 2004], a Haskell library that expresses heterogeneous lists by indexing the list type constructor with a type-level list. Their work does not appear to generalize beyond lists or to infinite indices.

Richer forms of type-level computation have been explored, and could be used to further generalize the theory of vertex structures. Yorgey et al. [2012] extend Haskell's kind system with features for expressing a variety of type-level data structures. As another example, we have considered extending vertex structures with sums (so a vertex structure could, e.g., represent a finite list) and using type-level matching [Blanvillain et al. 2022] to constrain the lengths of lists by the length of the vertex structure parameter. While this would expand the expressiveness of vertex structures, extending VSs beyond tree-like corecursive structures causes a problem for inference and so it seems likely that such an extension would require programmer annotations in some cases.

We note that this paper enters a rich design space of combining data and codata (e.g. [Thibodeau et al. 2016]). We have shown in Section 5 that any data type can be "overapproximated by" a codata type in the sense that there is a straightforward, local mapping from nodes in the data type's AST to those of the codata type (the vertex structure in our case). Whether this has a deeper meaning in the theory of data and codata is left to future work.

Affine Type Systems. We use an affine type system to handle vertex structures and ensure that vertex names in output graphs are unique. Affine type systems have been used in a number of languages to ensure safe usage of resources (broadly construed), notably including Cyclone [Jim et al. 2002] and Rust [Rus [n.d.]]. Our notation for splitting and availability is inspired by Cogent [O'Connor et al. 2021], which uses these ideas for an affine treatment of record types.

Encoding in Rust. As a memory safety focused language, Rust's type system would likely benefit from the features GML. Though Rust is able to encode other type systems such as session types [Jespersen et al. 2015; Lagaillardie et al. 2020], we do not believe GML could be usefully or at least conveniently encoded in Rust as is. The main limiting factor we foresee is that we believe each vertex name would need its own lifetime. This is a problem because Rust requires all lifetimes to be declared statically, and each piece of code can only refer to finitely many lifetimes. However, in GML, vertex names are generated dynamically, therefore a function might manipulate infinitely many vertices.

8 CONCLUSION

We have presented a type system for annotating parallel programs with futures with *graph types*, which compactly represent the parallel structure of the program. Unlike prior work, we support complex data structures containing futures. As evidenced by our prototype implementation, it is possible to infer these graph types automatically for examples that use this feature for efficient pipelined algorithms. Our implementation is also able to generate visualizations of the resulting graph types, which can aid in understanding the structure of parallel code and finding bugs.

In the future, we hope to expand on the types of program analysis and bug-finding that can be done with these graph types. For example, we could build on the prototype analyses of Muller [2022] to study deadlock and asymptotic complexity in the complex, pipelined graphs that arise from programs with data structures of futures. We also plan to scale the implementation up to support a larger subset of OCaml, with the goal of integrating the analysis into the OCaml compiler. Finally, because the graph type system itself does not depend on a particular source language, we plan to explore implementing the graph type system in front-ends for a variety of languages, so that more programmers can benefit from graph types as an analysis tool and reasoning aid.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful feedback. This paper was based on work partially supported by the National Science Foundation under awards CCF-2107289, CCF-2007784, and CNS-2314323.

DATA AVAILABILITY STATEMENT

The artifact associated with this paper, consisting of the implementation of GML$^{\mu}$ and the examples described in Section 6, is available at https://zenodo.org/record/8424018.

REFERENCES

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

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

Olivier Blanvillain, Jonathan Immanuel Brachthäuser, Maxime Kjaer, and Martin Odersky. 2022. Type-level programming with match types. *Proceedings of the ACM on Programming Languages* 6 (1 2022). Issue POPL. https://doi.org/10.1145/3498698 We could use something like this to give stronger guarantees with VSs, e.g., have some finite VSs.

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

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

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. https://doi.org/10.1007/3-540-48046-3_17

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

Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear Regions Are All You Need. In *Proceedings of the 15th European Conference on Programming Languages and Systems* (Vienna, Austria) *(ESOP'06)*. Springer-Verlag, Berlin, Heidelberg, 7–21. https://doi.org/10.1007/11693024_2

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. https://doi.org/10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N

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.

Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session Types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming* (Vancouver, BC, Canada) *(WGP 2015)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/2808098.2808100

Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02)*. USENIX Association, USA, 275–288.

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

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly typed heterogeneous collections. *Proceedings of the ACM SIGPLAN 2004 Haskell Workshop, Haskell'04* (2004), 96–107. https://doi.org/10.1145/1017472.1017488

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

Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. 2020. Implementing Multiparty Session Types in Rust. In *Coordination Models and Languages: 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15–19, 2020, Proceedings* (Valletta, Malta). Springer-Verlag, Berlin, Heidelberg, 127–136. https://doi.org/10.1007/978-3-030-50029-0_8

Mae Milano, Joshua Turcotti, and Andrew C. Myers. 2022. A Flexible Type System for Fearless Concurrency. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 458–473. https://doi.org/10.1145/3519939.3523443

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

Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. Cogent: uniqueness types and certifying compilation. *J. Funct. Program.* 31 (2021), e25. https://doi.org/10.1017/S095679682100023X

Francis Rinaldi, june wunder, Arthur Azevedo de Amorim, and Stefan K. Muller. 2024. Pipelines and Beyond: Graph Types for ADTs with Futures. TODO XXX FIXME: ArXiV citation.

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

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

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

David Thibodeau, Andrew Cave, and Brigitte Pientka. 2016. Indexed Codata Types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 351–363. https://doi.org/10.1145/2951913.2951929

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

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

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(TLDI '12)*. Association for Computing Machinery, New York, NY, USA, 53–66. https://doi.org/10.1145/2103786.2103795

Christoph Zenger. 1997. Indexed types. *Theoretical Computer Science* 187, 1 (1997), 147–165. https://doi.org/10.1016/S0304-3975(97)00062-5

Received 2023-07-11; accepted 2023-11-07