TypeDis: A Type System for Disentanglement

ALEXANDRE MOINE, New York University, USA
STEPHANIE BALZER, Carnegie Mellon University, USA
ALEX XU, Carnegie Mellon University, USA
SAM WESTRICK, New York University, USA

Disentanglement is a runtime property of parallel programs intuitively guaranteeing that parallel tasks remain oblivious to each other's allocations. As demonstrated in the MaPLe compiler and run-time system, disentanglement can be exploited for fast automatic memory management, especially task-local garbage collection with no synchronization between parallel tasks. However, as a low-level property, disentanglement can be difficult to reason about for programmers. The only means of statically verifying disentanglement so far has been DisLog, an Iris-fueled variant of separation logic, mechanized in the Rocq proof assistant. DisLog is a fully-featured program logic, allowing for proof of functional correctness as well as verification of disentanglement. Yet its employment requires significant expertise and per-program proof effort.

This paper explores the route of automatic verification via a type system, ensuring that any well-typed program is disentangled and lifting the burden of carrying out manual proofs from the programmer. It contributes TypeDis, a type system inspired by region types, where each type is annotated by a timestamp, identifying the task that allocated it. TypeDis supports iso-recursive types, as well as polymorphism over both types and timestamps. Crucially, timestamps are allowed to change during type-checking, at join points as well as via a form of subtyping, dubbed *subtiming*. The paper illustrates TypeDis and its features on a range of examples. The soundness of TypeDis and the examples are mechanized in the Rocq proof assistant, using an improved version of DisLog, dubbed DisLog2.

Additional Key Words and Phrases: disentanglement, parallelism, type system, separation logic

1 Introduction

A recent line of work has identified a key memory property of parallel programs called *disentanglement* [Acar et al. 2015; Arora et al. 2024, 2021, 2023; Guatto et al. 2018; Moine et al. 2024; Raghunathan et al. 2016; Westrick et al. 2022, 2020]. Roughly speaking, disentanglement is the property that concurrent tasks remain oblivous to each other's memory allocations. As demonstrated by the MaPLe compiler [Acar et al. 2020], this property makes it possible to perform task-local memory management (allocations and garbage collections) independently, in parallel, without any synchronization between concurrent tasks. MaPLe in particular features a provably efficient memory management system for a dialect of Parallel ML—a parallel functional programming language—and offers competitive performance in practice relative to low-level parallel code written in languages such as C/C++ [Arora et al. 2023].

The key idea behind disentanglement is to coschedule memory and computation, taking advantage of structured fork-join parallelism. Memory is organized into a dynamic tree of heaps mirroring the parent/child relationship between tasks: as tasks fork and join, the tree grows and contracts, respectively. Each task thus has its own task-local heap, in which it allocates memory objects and may perform garbage collection independently, in parallel. On this tree of heaps, disentanglement can be defined as a "no cross-pointers" invariant. Specifically, disentanglement allows for *up-pointers* from descendant heaps to ancestors, as well as *down-pointers* from ancestors to descendants, but disallows *cross-pointers* pointers between concurrent tasks (siblings, cousins, etc.). The existence of cross-pointers is called *entanglement*.

Authors' Contact Information: Alexandre Moine, alexandre.moine@nyu.edu, New York University, New York, USA; Stephanie Balzer, balzers@cs.cmu.edu, Carnegie Mellon University, Pittsburgh, USA; Alex Xu, alexxu@andrew.cmu.edu, Carnegie Mellon University, Pittsburgh, USA; Sam Westrick, shw8119@nyu.edu, New York University, New York, USA.

Entanglement arises from a particular communication pattern, where one task allocates a local heap object and then another task (executing concurrently, relative to the first task) acquires a pointer to the object. At this point, the task-local memories of the two concurrent tasks have become intertwined and neither task can perform garbage collection without synchronizing with the other. These additional synchronizations lead to significant performance degradation in the automatic memory management system [Arora et al. 2023]. In this sense, entanglement is a performance hazard.

One way to rule out entanglement is to disallow side effects entirely. Indeed, the original study of disentanglement emerged out of an interest in improving the performance of parallel functional programming techniques, which naturally have a high rate of allocation and whose scalability and efficiency is largely determined by the performance of automatic memory management. In this setting, disentanglement is guaranteed by construction due to a lack of side effects. But the full power of disentanglement lies in its expressivity beyond purely functional programming—in particular, disentanglement allows for judicious utilization of side effects such as in-place updates and irregular and/or data-dependent access patterns in shared memory. Such side effects are crucial for efficiency in state-of-the-art implementations of parallel algorithms [Abdi et al. 2024; Anderson et al. 2022; Shun et al. 2012], and all of these implementations have been found to be naturally disentangled [Westrick et al. 2022].

In these more general settings, where there numerous opportunities for efficient utilization of side effects, it is easy to accidentally *entangle* concurrent tasks. To ensure good performance—especially the efficiency and scalability of automatic memory management—a question immediately arises of how to verify disentanglement.

Verification of disentanglement. Two approaches to check for and/or verify disentanglement have been proposed thus far. First, as currently implemented in the MaPLe compiler, the programmer can rely on a runtime *entanglement detector* [Westrick et al. 2022]. This approach is similar in principle to dynamic race detection [Flanagan and Freund 2009]. In the case of entanglement, dynamic detection has been shown to have low overhead, making it suitable for automatic run-time management of entanglement [Arora et al. 2023]. However, run-time detection cannot guarantee disentanglement due to the inherent non-determinism of entanglement, which typically arises due to race conditions and may or may not occur in individual executions. The second approach, as developed by Moine et al. [2024], is full-blown static verification of disentanglement using a separation logic called DisLog, proven sound in Rocq[1] on top of the Iris framework [Jung et al. 2018b]. This approach can be used to statically verify disentanglement, no matter how complex the program is—for example, even for non-deterministic programs that utilize intricate lock-free data structures in shared memory. However, static verification with DisLog is difficult, requiring significant expertise and effort even to verify small examples. Additionally, both of these prior approaches are inherently low-level, requiring the programmer to reason about disentanglement at the level of individual memory allocations and accesses, which is not only cumbersome but also fundamentally at odds with the goal of expressive high-level parallel programming.

In context of this prior work, an interesting question is whether it is possible to guarantee disentanglement statically through a type system. This would have the advantage of being mostly automatic, requiring (ideally) only a modest amount of type annotation. Most importantly, a type system would raise the level of abstraction at which the programmer can reason about disentanglement, clarifying how the property interacts with high-level abstractions such as parametric polymorphism, higher-order functions, algebraic datatypes, and other desirable features.

[1]The Rocq prover is the new name of the Coq proof assistant, see https://rocq-prover.org/.

A type system for disentanglement. In this paper, we present ***TypeDis***, the first static type system for disentanglement. We consider a core ML-like language with a number of expressive features, as well as in-place atomic operations on shared memory and structured fork-join parallelism. The language features a single parallel construct, written $\mathsf{par}(f, g)$, which calls $f()$ and $g()$ in parallel, waits for both to complete, and returns their results as a pair. Here, we think of the execution of the two function calls as two child tasks, which themselves might execute par(...) recursively, creating a dynamic tree (parent-child) relationship between tasks.

TypeDis identifies tasks with timestamp variables δ, and annotates every value computed during execution with the timestamp of the task that allocated that value. This is tracked explicitly in the type of the value. For example, $s : \mathsf{string}@\delta$ indicates that the value s is a string that was allocated by a task δ. The type system implicitly maintains a partial order over timestamps, written $\delta' \preccurlyeq \delta$, intuitively corresponding to the tree relationship between tasks. Crucially, TypeDis guarantees an invariant that we call the **up-pointer invariant**: for every task running at timestamp δ, every value accessed by this task must have a timestamp $\delta' \preccurlyeq \delta$, i.e., the value must have been allocated "before" the current timestamp. In other words, the key insight in this paper is to *restrict all memory references to point backwards in time*, which is checked statically. This restriction is a deep invariant over values: every data structure will only contain values allocated at the same timestamp or a preceding timestamp. As a result, all loads in the language are guaranteed to be safe for disentanglement.

Although the up-pointer invariant places some restrictions on programs written within TypeDis, we have nevertheless found it to be surprisingly expressive. The up-pointer invariant is well-suited for immutable data (which naturally adheres to the invariant), as well as parallel batch processing of pre-allocated data. Pre-allocation especially is a common pattern in many performance-oriented parallel programs (to avoid the performance overheads of dynamic allocation along the "hot" path), and pre-allocation generally leads to more up-pointers. The up-pointer invariant also allows for structure sharing, even in the presence of mutable state.

Maintaining the up-pointer invariant. To maintain the up-pointer invariant in the presence of mutable state, TypeDis places a restriction on writes (in-place updates), requiring that the timestamp of the written value precedes the timestamp of its container. This restriction is implemented in the type system with a form of subtyping, dubbed **subtiming**, which affects only the timestamps of values within their types. The idea is to allow for any value to be (conservatively) restamped with a *newer* timestamp. Subtiming makes it possible to express the restriction on writes as a simple unification over the type of the contents of a mutable reference or array.

Restamping with an *older* timestamp would be unsound in TypeDis, as it would allow for a child's (heap-allocated) value to be written into a parent's container, potentially making that value accessible to a concurrent sibling. This is prevented throughout the type system, except in one place: at the join point of par. At this point, the two sub-tasks have completed and their parent inherits the values they allocated. To allow the parent task to access these values, TypeDis restamps the result of par with the timestamp of the parent. We dub this operation **backtiming**.

TypeDis features first-class function types ($\alpha \to^{\delta} \beta$), annotated by a timestamp variable δ, indicating which task the function may be called by. Timestamp variables can be universally quantified, effectively allowing for **timestamp polymorphism**. For example, pure functions that have no side-effects are type-able as $(\forall\delta.\, \alpha \to^{\delta} \beta)$, indicating that the function may be safely called by any task. TypeDis also allows for **constrained timestamp polymorphism**. For example, a function of type $(\forall\delta' \preccurlyeq \delta.\, \mathsf{string}@\delta' \to^{\delta} ())$ only accepts as argument strings timestamped at some δ' that precede the timestamp δ of the calling task. Typically, such constraints arise from the use of closures, especially those that close over mutable state.

Soundness. The soundness of TypeDis is verified in Rocq on top of the Iris separation logic framework. More precisely, we use the approach of *semantic typing* [Constable et al. 1986; Martin-Löf 1982; Timany et al. 2024], and define a logical relation targeting a variation of DisLog [Moine et al. 2024], from which we reuse the technical parts. As illustrated by RustBelt [Jung et al. 2018a], semantic typing facilitates manual verification of programs that are correct (e.g. disentangled), but ill-typed, by carrying out a logical relation inhabitation proof using the program logic—overcoming incompleteness inherent to any type system. For example, in the case of TypeDis, this allows the user to verify part of the code that use pointers pointing forward in time. We do not yet make use of this feature in this paper, but it would be interesting to explore in more detail in future work.

Contributions. Our contributions include:

- TypeDis, the first static type system for disentanglement. It is a type system with the notion of a timestamp to track which object is accessible by which task. TypeDis includes (iso-)recursive types as well as polymorphism over types and over timestamps.
- Two mechanisms to update a timestamp annotation: via subtiming, a form of subtyping, and specifically at join points via the new operation of backtiming.
- A soundness proof of TypeDis mechanized in the Rocq prover using the Iris framework. We use semantic typing [Timany et al. 2024] and DisLog$_2$, an improved version of DisLog relying on a new alternative definition of disentanglement.
- A range of case studies, including building and iterating over an immutable tree in parallel, as well the challenging example of deduplication via concurrent hashing.

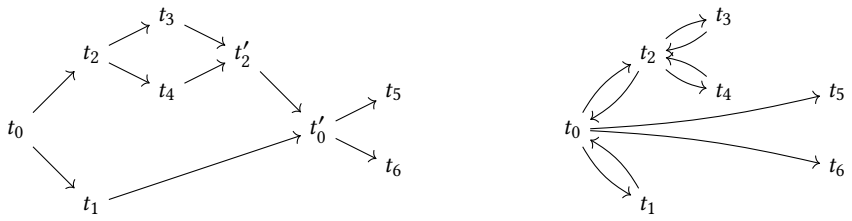
2 Key Ideas

In this section, we cover the key ideas of our work. We start by recalling the definition of disentanglement (§2.1). We then present the main idea of TypeDis: adding task identifiers, specifically *timestamp variables*, within types (§2.2). We then illustrate with two examples two core principles of TypeDis allowing for updating timestamps within types: backtiming (§2.3) and subtiming (§2.4).

2.1 Background and Preliminaries

Nested fork-join parallelism and task trees. We consider programs written in terms of a single parallel primitive: $\mathsf{par}(f_1, f_2)$, which creates two new child tasks to execute $f_1()$ and $f_2()$ in parallel, waits for both of child tasks to complete, and then returns the results of the two calls as an immutable pair. Creating the two child tasks is called a ***fork***, and waiting for the two children to complete is called a ***join***. The behavior of the par primitive guarantees that every fork has a corresponding join. Any task may (recursively) fork and join, and in this sense the parallelism of the program is *nested*, giving rise to a dynamic tree during execution called the ***task tree***. The nodes of the task tree correspond to (parent) tasks that are waiting for their children to join, and the leaves of the task tree correspond to tasks which may actively take a step. Whenever two sibling tasks join, the children are removed from the tree and the parent resumes as a leaf task. The task tree therefore dynamically grows and shrinks as tasks fork and join. In this paper, we will use the letter t to denote tasks (leaves of the task tree), and will equivalently refer to these as ***timestamps***.

Computation graphs. The evolution of the task tree over time can be recorded as a ***computation graph***, where vertices correspond to tasks and edges correspond to scheduling dependencies. The computation graph records not just the current tree of tasks, but also the history of tasks that have joined. When a task t forks into two children t_1 and t_2, two edges (t, t_1) and (t, t_2) are added to the graph; later when t_1 and t_2 join, two edges (t_1, t) and (t_2, t) are added to the graph. We say that



(a) Standard computation graph (b) Equivalent "cyclic" approach used in this paper

Fig. 1. Comparison of two computation graphs equivalent for disentanglement

t *precedes* t' in graph G, and write $G \vdash t \preccurlyeq t'$, when there exists a sequence of edges from t to t'. Note that $\preccurlyeq$ is reflexive. Two tasks are ***concurrent*** when neither precedes the other.

Standard versus cyclic computation graphs. It is worth mentioning that our presentation of computation graphs differs slightly from the standard presentation used in prior work [Acar et al. 2016; Moine et al. 2024; Westrick et al. 2020]. The standard approach is to use a fresh task identifier at each join point, effectively renaming the resumed parent task. In our approach, which we call the ***cyclic approach***, we instead use the same task identifier after the join point. Figure 1 illustrates the difference between the two approaches. It presents two computation graphs representing the same computation: Figure 1a shows the standard approach, and Figure 1b shows the (new) cyclic approach. The distinction occurs when two tasks join. In Figure 1a tasks t_3 and t_4 join and form a new task t_2' whereas in Figure 1b the two tasks join by going back to task t_2. This distinction occurs again when t_2' (resp. t_2) join to form t_0' (resp. t_0).

The cyclic approach considerably reduces the need to manipulate timestamps, not only in our proofs (for example the soundness proof of backtiming), but also in the design of the type system itself as well as in the underlying program logic (§5.3). As the name suggests, the cyclic approach introduces cycles in the graph, which may seem dangerous. However, for the purpose of verifying disentanglement, we prove the two approaches equivalent.[2]

Roots. At any moment, every task has a set of task-local ***roots*** which are the memory locations directly mentioned within a subexpression of that task. For example, the expression '$\mathsf{let}\, x = (\ell_1, \ell_2)\, \mathsf{in}\, \mathsf{fst}(x)$' has roots $\{\ell_1, \ell_2\}$, where (formally) ℓ_1 and ℓ_2 are locations within a the memory store. Note that the roots of a task change over time: for example, the above expression eventually steps to ℓ_1 at which point it only has one root, $\{\ell_1\}$. The set of roots can grow due to allocations and loads from memory.

Disentanglement. Disentanglement restricts the set of possible task-local roots. **A program state is disentangled if each root of a task has been allocated by some preceding task**. More precisely, a program state with a computation graph G is disentangled if, for a root ℓ of a task t, ℓ has been allocated by a task t' such that $G \vdash t' \preccurlyeq t$, that is, such that t' precedes t in G. Following the computation graph definition, preceding tasks include t itself, parent tasks, but also children tasks that have terminated. The formal definition of disentanglement appears in Section 3.3.

TypeDis, the type system we present, verifies that a program is disentangled, that is, every reachable program state is disentangled.

[2]Intuitively, the two approaches are equivalent because we never need to check reachability between two tasks that have both completed. We have formally proven this equivalence with a simulation theorem: every reduction in a semantics with the standard approach implies the existence of a reduction reaching the same expression in the semantics with the cyclic approach, and vice-versa [Anon. 2025].

2.2 TypeDis 101: Timestamps in Types

In order to keep track of which task allocated which location, TypeDis incorporates timestamps in types. More precisely, every heap-allocated ("boxed") type is annotated by a *timestamp variable*, written δ, which can be understood as the timestamp of the task that allocated the underlying location. For example, a reference allocated by task δ on a (unboxed) integer has type $\mathsf{ref}(\mathsf{int})@\delta$.

Timestamp polymorphism. Functions in TypeDis are annotated by a timestamp variable, restricting which task they may run on. Such a variable can be universally quantified, allowing for functions to be run by different tasks. For example, consider the function `fun x -> newref(x)` which allocates a new mutable reference containing an integer `x`. This function can be given the type $\forall\delta.\ \mathsf{int} \to^{\delta} \mathsf{ref}(\mathsf{int})@\delta$. The superscript δ on the arrow indicates that the function must run on a task at timestamp δ, and the result type $\mathsf{ref}(\mathsf{int})@\delta$ indicates that the resulting reference will be allocated at the same timestamp δ. By universally quantifying δ, the function is permitted to run on any task, with the type system tracking that the resulting reference will be allocated at the same timestamp as the caller.

Type polymorphism. TypeDis allows for universal quantification over type variables α. Considering again the function `fun x -> newref(x)`, we can give this function the more general type $\forall\alpha.\forall\delta.\ \alpha \to^{\delta} \mathsf{ref}(\alpha)@\delta$, indicating that it is polymorphic over the type α of the contents of the mutable reference. Corresponding get and set primitives for mutable references can then be typed as shown in Figure 2, all of which are polymorphic over the type variable α. Note that, in the type of the set primitive, we use a notational convention: functions taking multiple arguments only specify a timestamp variable on the last arrow.

The up-pointer invariant. In Figure 2, the type of get is given as $\forall\alpha.\,\forall\delta\,\delta'.\,\mathsf{ref}(\alpha)@\delta' \to^{\delta} \alpha$. Note that this type is parameterized over both a caller time δ as well as a (potentially different) timestamp δ' associated with the input reference. Intuitively, this type specifies that get is safe to call at any moment, by any task, with any reference given as argument. The design of TypeDis in general guarantees that all loads from memory, both mutable and immutable, are always safe. Specifically, this is guaranteed by enforcing an invariant that we call the ***up-pointer invariant***: all data structures in the language may only contain values allocated at the same timestamp or a preceding timestamp. For example, given two non-equal timestamps δ_1 and δ_2 where $\delta_1 \prec \delta_2$, the type $\mathsf{ref}(\mathsf{ref}(\mathsf{int})@\delta_1)@\delta_2$ is valid, but $\mathsf{ref}(\mathsf{ref}(\mathsf{int})@\delta_2)@\delta_1$ is not.

Closures. In TypeDis, functions are first-class values and may be passed as argument to other functions, or stored in data structures, etc. Function values are implemented as heap-allocated closures [Appel 1992; Landin 1964], and must be given a timestamp indicating when they were allocated. For example, consider the definition of function f in Figure 3, which closes over a mutable reference r and an immutable string w, both allocated at timestamps δ_0 which (in this example) is the timestamp of the current task. We can give f the type $(\forall\delta.\ () \to^{\delta} ())@\delta_0$, indicating that f itself was allocated at timestamp δ_0. Additionally, the type of f specifies that it may be freely called at any timestamp; this is safe for disentanglement because f preserves the up-pointer invariant, regardless of *when* it will be called. Contrast this with the definition of function g, which (when called) allocates a new string and writes this string into the reference r. If g were called at some timestamp δ_1 where $\delta_0 \prec \delta_1$, then this would violate the up-pointer invariant for r. The function g does however admit the type $(\mathsf{int} \to^{\delta_0} ())@\delta_0$, indicating that g may be safely called only by tasks at time δ_0 (the same timestamp as the reference r).

$$\begin{aligned} \text{newref} &: \forall\alpha.\,\forall\delta.\ \alpha \to^{\delta} \text{ref}(\alpha)@\delta \\ \text{get} &: \forall\alpha.\,\forall\delta\,\delta'.\ \text{ref}(\alpha)@\delta' \to^{\delta} \alpha \\ \text{set} &: \forall\alpha.\,\forall\delta\,\delta'.\ \text{ref}(\alpha)@\delta' \to \alpha \to^{\delta} () \end{aligned}$$

Fig. 2. Example: typing reference primitives

```
let r = newref "hello"      r : ref(string@δ0)@δ0
let w = "world"             w : string@δ0
let f () = set r w          f : (∀δ. () →δ ())@δ0
let g i =                   g : (int →δ0 ())@δ0
  set r (Int.to_string i)
```

Fig. 3. Example: typing closures

```
type tree@δ = ( int + (tree@δ × tree@δ)@δ )@δ
let leaf x = inj1 x                          leaf : (∀δ. int →δ tree@δ)@δ0
let node x y = inj2 (x,y)                    node : (∀δ. tree@δ → tree@δ →δ tree@δ)@δ0
let rec build n x =                          build : (∀δ. int → int →δ tree@δ)@δ0
  if n <= 0 then leaf x else
  let n' = n - 1 in
  let (l,r) = par (fun () -> build n' x) (fun () -> build n' (x + pow2 n')) in
  node l r
```

Fig. 4. Example: building a tree in parallel

2.3 Backtiming the Result of a par

As explained earlier (§2.1), we consider in this paper the parallel primitive par(...), which executes two closures in parallel and returns their result as an immutable pair. The par primitive can be used to build data structures in parallel. Consider the code presented in Figure 4. The recursive type tree@δ = (int + (tree@δ × tree@δ)@δ)@δ describes a binary tree with integer leaves. It consists of an immutable sum of either an integer (a leaf) or a product of two subtrees (a node). All the parts of a tree are specified in the type to have been allocated at the same timestamp δ. A leaf is built with the first injection, and a node with the second injection. The function `build n x` builds in parallel a binary tree of depth n, with leaves labeled from x to $x + 2^n - 1$ in left-to-right order.

TypeDis type-checks `build` with the type $\forall\delta.$ int $\to$ int $\to^{\delta}$ tree@δ. The reader may be surprised: we announced that the type `tree@`δ has all of its parts allocated at the same timestamp δ, but we are showing a function that builds a tree in parallel, hence with some parts allocated by different tasks at different timestamps. What's the trick?

The key observation is that *we can pretend that the objects allocated by a completed sub-task were instead allocated by its parent.* Indeed, disentanglement prevents sharing of data allocated in parallel, but as soon as the parallel phase has ended, there is no restriction anymore!

In TypeDis, the par primitive implements **backtiming**, meaning that it replaces the timestamp of the child task by the timestamp of the parent task in the return type of the closures executed in parallel. Indeed, the par primitive admits the following, specialized for `build`, type:

$$\forall\delta\,\delta_l\,\delta_r.\ (\forall\delta'.\,() \to^{\delta'} \text{tree@}\delta')@\delta_l \to (\forall\delta'.\,() \to^{\delta'} \text{tree@}\delta')@\delta_r \to^{\delta} (\text{tree@}\delta \times \text{tree@}\delta)@\delta$$

This type for par does exactly what we need: it returns the result of the two closures in a pair as-if they were called at time δ. Backtiming is a powerful feature: it reduces parallelism to almost an implementation detail. Indeed, the type of `build` does not reveal its internal use of parallelism.

2.4 Making Something New out of Something Old with Subtiming

A common practice (especially in functional programming) is data structural *sharing*, where components of an old structure are reused inside part of a new structure. In the context of TypeDis, data structural sharing is interesting in that it mixes data of potentially different timestamps within the same structure. Here we consider one such example and a describe a key feature of TypeDis which enables such "mixing" of timestamps.

```
let rec selectmap p f t =                    selectmap : (∀δ δ_p δ_f δ_t. (∀δ'. int →^δ' bool)@δ_p
  match t with                                          → (∀δ'. int →^δ' int)@δ_f
  | inj1 x -> if p x then leaf (f x) else t             → tree@δ_t →^δ tree@δ)@δ_0
  | inj2 (l,r) ->
    let (nl,nr) = par (fun () -> selectmap p f l) (fun () -> selectmap p f r) in
    if nl == l && nr == r then t else node nl nr
```

Fig. 5. Example: the selectmap function

Figure 5 presents the `selectmap p f t` function, which selectively applies the function `f` to the leaves of the tree `t`, following a predicate `p` on integers. The `selectmap` function traverses the tree in parallel and crucially preserves sharing as much as possible. Specifically, when none of the leaves of the tree satisfy the predicate, the function returns the original input tree as-is, instead of building another identical tree. To type this function within TypeDis, it may not be immediately clear what the timestamp of the resulting tree should be: `selectmap` might directly return the argument passed as argument (potentially coming from an older task), or it might return a new tree. TypeDis typechecks `selectmap` with the type

$$\forall \delta\, \delta_p\, \delta_f\, \delta_t.\ (\forall \delta'.\ \text{int} \to^{\delta'} \text{bool})@\delta_p \to (\forall \delta'.\ \text{int} \to^{\delta'} \text{int})@\delta_f \to \text{tree}@\delta_t \to^{\delta} \text{tree}@\delta$$

This type universally quantifies over δ (the timestamp at which `selectmap` will run), δ_p and δ_f (the timestamps of the two closure arguments), and δ_t (the timestamp of the tree argument). Crucially, the result is of type tree@δ, as-if the whole result tree was allocated by δ. What's the trick?

TypeDis supports **subtiming**, that is, a way of "advancing" timestamps within a type, following the precedence. The rules of subtiming are as follows. For a mutable type (e.g. an array or a reference), subtiming is *shallow*: the outermost timestamp can be updated, but not the inner timestamps; this is due to well-known variance issues [Pierce 2002, §15]. For an immutable type (e.g. products and sums), subtiming is *deep*: any timestamp within the type can be advanced, as long as the up-pointer invariant is preserved.

In the case of `selectmap`, we need to use deep subtiming on the recursive immutable type tree@δ_t in order to update it to tree@δ. How can we be sure that δ_t, the timestamp of the tree, precedes δ, the timestamp at which we call `selectmap`? We unveil a key invariant of TypeDis: every timestamp in the typing environment precedes the "current" timestamp, that is, the timestamp of the task executing the function. In our case the current timestamp is precisely δ. We hence deduce that δ_t precedes δ, allowing us to use subtiming to "restamp" the value t : tree@δ_t as t : tree@δ.

To allow the user to express additional knowledge about the dependencies between timestamps, TypeDis annotates universal timestamp quantification with a set of *constraints*, which are supposed to hold while typing the function body, and are verified at call sites. For example, the following function `let par' f g = ignore (par f g)` that executes two closures `f` and `g` from unit to unit in parallel and ignore the result can be given the type:

$$\forall \delta\, \delta_1\, \delta_2.\ (\forall \delta'\ \delta \preccurlyeq \delta'.\ () \to^{\delta'} ())@\delta_1 \to (\forall \delta'\ \delta \preccurlyeq \delta'.\ () \to^{\delta'} ())@\delta_2 \to^{\delta} ()$$

This type says that, if `par'` gets called at timestamp δ with arguments f and g, then f and g can assume that they will be called at timestamp δ' such that $\delta \preccurlyeq \delta'$. These constraints are discussed in Section 4, and the fully general type of par is presented in Section 4.6.

3 Syntax and Semantics

The formal language we study, dubbed DisLang$_2$, can be understood as an extension of DisLang, the language studied by Moine et al. [2024]. DisLang$_2$ adds support for immutable pairs and sums,

Values	$v, w ::= ()$	$\mid b \in \{\mathsf{true}, \mathsf{false}\}$	$\mid i \in \mathbb{Z}$	$\mid \ell \in \mathcal{L}$	$\mid \mathsf{vfold}\, v$
Blocks	$r ::= \vec{w}$	$\mid (v, v)$	$\mid \mathsf{inj}_{\iota \in \{0,1\}}\, v$	$\mid \mu f.\, \lambda \vec{x}.\, e$	
Primitives	$\bowtie ::= +$	$\mid -$	$\mid \times$	$\mid \div$	$\mid \mathsf{mod} \mid == \mid < \mid \leq \mid > \mid \geq \mid \vee \mid \wedge$
Expressions	$e ::= v$	$\mid x \in \mathcal{V}$	$\mid \mathsf{let}\, x = e\, \mathsf{in}\, e$	$\mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e$	$\mid e \bowtie e$
	$\mid \mu f.\, \Lambda \vec{\delta}\, \Delta.\, \lambda \vec{x}!\delta.\, e$	$\mid e\, \langle \vec{\delta} \rangle\, \vec{e}$	$\mid \Lambda \alpha :: \kappa.\, e$	$\mid e\, \langle \rho \rangle$	*closures and universal types*
	$\mid (e, e)$	$\mid \mathsf{proj}_{\iota \in \{1,2\}}\, e$			*pairs*
	$\mid \mathsf{inj}_{i \in \{1,2\}}\, e$	$\mid \mathsf{match}\, e\, \mathsf{with}\, \mathsf{inj}_1\, x \Rightarrow e \mid \mathsf{inj}_2\, x \Rightarrow e\, \mathsf{end}$			*sums*
	$\mid \mathsf{alloc}\, e\, e$	$\mid e[e]$	$\mid e[e] \leftarrow e$	$\mid \mathsf{length}\, e$	*arrays*
	$\mid \mathsf{fold}\, e$	$\mid \mathsf{unfold}\, e$			*iso-recursive types*
	$\mid \mathsf{par}(e, e)$	$\mid e \parallel e$	$\mid \mathsf{CAS}\, e\, e\, e$		*parallelism and concurrency*
Contexts	$K ::= \mathsf{let}\, x = \square\, \mathsf{in}\, e$	$\mid \mathsf{if}\, \square\, \mathsf{then}\, e\, \mathsf{else}\, e$	$\mid \mathsf{alloc}\, \square\, e$	$\mid \mathsf{alloc}\, v\, \square$	$\mid \mathsf{length}\, \square$
	$\mid \square[e]$	$\mid v[\square]$	$\mid \square[e] \leftarrow e$	$\mid v[\square] \leftarrow e$	$\mid v[v] \leftarrow \square$
	$\mid \square \bowtie e$	$\mid v \bowtie \square$	$\mid \square\, \vec{e}$	$\mid v\, (\vec{v} +\!\!+ \square +\!\!+ \vec{e})$	$\mid \mathsf{fold}\, \square$
	$\mid \mathsf{unfold}\, \square$	$\mid (\square, e)$	$\mid (v, \square)$	$\mid \mathsf{proj}_\iota\, \square$	$\mid \mathsf{inj}_i\, \square$
	$\mid \mathsf{match}\, \square\, \mathsf{with}\, \mathsf{inj}_1\, x \Rightarrow e \mid \mathsf{inj}_2\, x \Rightarrow e\, \mathsf{end}$			$\mid \mathsf{par}(\square, e)$	$\mid \mathsf{par}(v, \square)$
	$\mid \mathsf{CAS}\, \square\, e\, e$	$\mid \mathsf{CAS}\, v\, \square\, e$	$\mid \mathsf{CAS}\, v\, v\, \square$	$\mid \mathsf{CAS}\, v\, v\, v\, \square$	

Fig. 6. Syntax of DisLang$_2$. Constructs in red are source-level and constructs in blue are runtime-level.

iso-recursive types, and directly offers the par primitive for fork-join parallelism. We present the syntax of DisLang$_2$ (§3.1), its semantics (§3.2), and the formal definition of disentanglement (§3.3).

3.1 Syntax

The syntax of DisLang$_2$ appears in Figure 6. The constructs in red occur only in the source program and are erased at runtime; conversely, the constructs in blue are forbidden in the source program and occur only at runtime.

A value $v \in \mathcal{V}$ can be the unit value $()$, a boolean $b \in \{\mathsf{true}, \mathsf{false}\}$, an idealized integer $i \in \mathbb{Z}$, a *memory location* $\ell \in \mathcal{L}$, where $\mathcal{L}$ is an infinite set of locations, or a folded value $\mathsf{vfold}\, v$, witnessing our use of iso-recursive types [Pierce 2002, §20].

A *block* describes the contents of a heap cell, amounting to either an array of values, written $\vec{w}$, an immutable pair (v, v), the first injection $\mathsf{inj}_1\, v$ or the second injection $\mathsf{inj}_2\, v$ of an immutable sum, or a *λ-abstraction* $\mu f.\, \lambda \vec{x}.\, e$. Lambdas can close over free variables, compilers of functional languages usually implement them as *closures* [Appel 1992; Landin 1964]. A closure is a heap-allocated object carrying a code pointer as well as an environment, recording the values of the free variable. Thus, acquiring a closure can create entanglement. Moreover, because functions and tuples are heap allocated, currying and uncurrying—that is, converting a function taking multiple arguments to a function taking a tuple of arguments and vice-versa—does not come for free. Hence, we chose to present a version of the language were every function takes possibly multiple arguments.

Expressions e range over the usual constructs, but some of them (related to closures and universal quantification) carry additionally timestamp variables δ, logical graphs Δ (formally defined as a set of pairs of timestamps), kinds κ and types ρ to guide type checking. These annotations disappear at runtime (§4.1).

Closure allocation is written $\mu f.\, \Lambda \vec{\delta}\, \Delta.\, \lambda \vec{x}!\delta.\, e$. This notation binds a recursive name f, a list of timestamps $\vec{\delta}$, a logical graph Δ, argument names $\vec{x}$ and execution timestamp δ in the expression e. A function call is written $e\, \langle \vec{\delta} \rangle\, \vec{e}$, instantiating the (type-level) timestamps arguments of e with $\vec{\delta}$ and function arguments with $\vec{e}$. Universal quantification is written $\Lambda \alpha :: \kappa.\, e$, quantifying over type variable α with kind κ in e. Type application is written $e\, \langle \rho \rangle$.

$$\textsc{HeadAlloc}\ \frac{0 \le n \qquad \ell \notin \mathrm{dom}(\sigma) \qquad \ell \notin \mathrm{dom}(\alpha)}{G, t \vdash \sigma \setminus \alpha \setminus \mathsf{alloc}\ n\ v \longrightarrow [\ell := v^n]\sigma \setminus [\ell := t]\alpha \setminus \ell}$$

$$\textsc{HeadLoad}\ \frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{G, t \vdash \sigma \setminus \alpha \setminus \ell[i] \longrightarrow \sigma \setminus \alpha \setminus v}$$

$$\textsc{HeadStore}\ \frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}|}{G, t \vdash \sigma \setminus \alpha \setminus \ell[i] \leftarrow v \longrightarrow [\ell := [i := v]\vec{w}]\sigma \setminus \alpha \setminus ()}$$

$$\textsc{HeadLetVal}\ G, t \vdash \sigma \setminus \alpha \setminus \mathsf{let}\ x = v\ \mathsf{in}\ e \longrightarrow \sigma \setminus \alpha \setminus [v/x]e$$

$$\textsc{HeadIfTrue}\ G, t \vdash \sigma \setminus \alpha \setminus \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow \sigma \setminus \alpha \setminus e_1$$

$$\textsc{HeadIfFalse}\ G, t \vdash \sigma \setminus \alpha \setminus \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow \sigma \setminus \alpha \setminus e_2$$

$$\textsc{HeadClosure}\ \frac{\ell \notin \mathrm{dom}(\sigma) \qquad \ell \notin \mathrm{dom}(\alpha)}{G, t \vdash \sigma \setminus \alpha \setminus \mu f.\, \lambda \vec{x}.\, e \longrightarrow [\ell := \mu f.\, \lambda \vec{x}.\, e]\sigma \setminus [\ell := t]\alpha \setminus \ell}$$

$$\textsc{HeadCall}\ \frac{\sigma(\ell) = \mu f.\, \lambda \vec{x}.\, e \qquad |\vec{x}| = |\vec{w}|}{G, t \vdash \sigma \setminus \alpha \setminus \ell\, \vec{w} \longrightarrow \sigma \setminus \alpha \setminus [\ell/f][\vec{w}/\vec{x}]e}$$

$$\textsc{HeadCallPrim}\ \frac{v_1 \bowtie v_2 \xrightarrow{\text{pure}} v}{G, t \vdash \sigma \setminus \alpha \setminus v_1 \bowtie v_2 \longrightarrow \sigma \setminus \alpha \setminus v}$$

$$\textsc{HeadPair}\ \frac{\ell \notin \mathrm{dom}(\sigma) \qquad \ell \notin \mathrm{dom}(\alpha)}{G, t \vdash \sigma \setminus \alpha \setminus (v_1, v_2) \longrightarrow [\ell := (v_1, v_2)]\sigma \setminus [\ell := t]\alpha \setminus \ell}$$

$$\textsc{HeadProj}\ \frac{\sigma(\ell) = (v_1, v_2)}{G, t \vdash \sigma \setminus \alpha \setminus \mathsf{proj}_\iota\, \ell \longrightarrow \sigma \setminus \alpha \setminus v_\iota}$$

$$\textsc{HeadInj}\ \frac{\ell \notin \mathrm{dom}(\sigma) \qquad \ell \notin \mathrm{dom}(\alpha)}{G, t \vdash \sigma \setminus \alpha \setminus \mathsf{inj}_i\, v \longrightarrow [\ell := \mathsf{inj}_i\, v]\sigma \setminus [\ell := t]\alpha \setminus \ell}$$

$$\textsc{HeadCase}\ \frac{\sigma(\ell) = \mathsf{inj}_\iota\, v}{G, t \vdash \sigma \setminus \alpha \setminus (\mathsf{match}\ \ell\ \mathsf{with}\ \mathsf{inj}_1\, x_1 \Rightarrow e_1 \mid \mathsf{inj}_2\, x_2 \Rightarrow e_2\ \mathsf{end}) \longrightarrow \sigma \setminus \alpha \setminus [v/x_\iota]e_\iota}$$

$$\textsc{HeadFold}\ G, t \vdash \sigma \setminus \alpha \setminus \mathsf{fold}\, v \longrightarrow \sigma \setminus \alpha \setminus \mathsf{vfold}\, v$$

$$\textsc{HeadUnfold}\ G, t \vdash \sigma \setminus \alpha \setminus \mathsf{unfold}\ (\mathsf{vfold}\, v) \longrightarrow \sigma \setminus \alpha \setminus v$$

Fig. 7. Head reduction (selected rules)

In DisLang$_2$, fork-join parallelism is available via the parallel primitive $\mathrm{par}(e_1, e_2)$, which reduces e_1 and e_2 to closures, calls them in parallel, and returns their result as an immutable pair. This parallel computation is represented by the *active* parallel pair $e_1 \parallel e_2$, appearing only at runtime. DisLang$_2$ supports a compare-and-swap instruction CAS $e\ e\ e\ e$, which targets an array, and is parameterized by 4 arguments: the location of the array, the index in the array, the old value and the new value. A (sequential) evaluation context K describes a term with a hole, written $\square$. The syntax of evaluation contexts dictates a left-to-right *call-by-value* evaluation strategy. Note that evaluation contexts K in this presentation are sequential. Specifically, we intentionally excluded active parallel pairs $(- \parallel -)$ from the grammar of K. The evaluation strategy for active parallel pairs allows for interleaving of small steps, which is handled separately by a "scheduler reduction" relation in the operational semantics (§3.2).

3.2 Operational Semantics

Head reduction relation. A *head configuration* $\sigma \setminus \alpha \setminus e$ is composed of a store σ, an allocation map α, and an expression e. The store σ represents the heap and consists of a finite map of locations to blocks. The allocation map α is a finite map of locations to timestamps, recording the timestamps at which locations were allocated. Figure 7 presents parts of the definition of the head reduction relation between two head configurations $G, t \vdash \sigma \setminus \alpha \setminus e \longrightarrow \sigma' \setminus \alpha' \setminus e'$ occurring at the (local) task of timestamp t in the (global) computation graph G. A head configuration consists of the

$$\textsc{SchedHead}\quad \frac{G, t \vdash \sigma \backslash \alpha \backslash e \longrightarrow \sigma' \backslash \alpha' \backslash e'}{\sigma / \alpha / G / t / e \xrightarrow{\text{sched}} \sigma' / \alpha' / G / t / e'}$$

$$\textsc{SchedFork}\quad \frac{t_1, t_2 \notin \text{vertices}(G) \qquad G' = G \cup \{(t_0, t_1), (t_0, t_2)\} \qquad e' = v_1\,[()] \parallel v_2\,[()]}{\sigma / \alpha / G / t_0 / \text{par}(v_1, v_2) \xrightarrow{\text{sched}} \sigma / \alpha / G' / t_1 \otimes_t t_2 / e'}$$

$$\textsc{SchedJoin}\quad \frac{\ell \notin \text{dom}(\sigma) \qquad \ell \notin \text{dom}(\alpha) \qquad G' = G \cup \{(t_1, t), (t_2, t)\}}{\sigma / \alpha / G / t_1 \otimes_t t_2 / v_1 \parallel v_2 \xrightarrow{\text{sched}} [\ell := (v_1, v_2)]\sigma / [\ell := t]\alpha / G' / t / \ell}$$

$$\textsc{StepSched}\quad \frac{\sigma / \alpha / G / T / e \xrightarrow{\text{sched}} \sigma' / \alpha' / G' / T' / e'}{(\sigma, \alpha, G) / T / e \xrightarrow{\text{step}} (\sigma', \alpha', G') / T' / e'}$$

$$\textsc{StepBind}\quad \frac{S / T / e \xrightarrow{\text{step}} S' / T' / e'}{S / T / K[e] \xrightarrow{\text{step}} S' / T' / K[e']}$$

$$\textsc{StepParL}\quad \frac{S / T_1 / e_1 \xrightarrow{\text{step}} S' / T_1' / e_1'}{S / T_1 \otimes_t T_2 / e_1 \parallel e_2 \xrightarrow{\text{step}} S' / T_1' \otimes_t T_2 / e_1' \parallel e_2}$$

$$\textsc{StepParR}\quad \frac{S / T_2 / e_2 \xrightarrow{\text{step}} S' / T_2' / e_2'}{S / T_1 \otimes_t T_2 / e_1 \parallel e_2 \xrightarrow{\text{step}} S' / T_1 \otimes_t T_2' / e_1 \parallel e_2'}$$

Fig. 8. Reduction under a context and parallelism

expression e being evaluated, the store σ, and an allocation map α. Figure 7 omits rules for the length array primitive as well as the atomic compare-and-swap on arrays.

We write $\sigma(\ell)$ to denote the block stored at the location ℓ in the store σ. We write $[\ell := r]\sigma$ for the insertion of the block r at location ℓ in σ. Note that only arrays can be updated; closures, pairs and sums are immutable. We write $\vec{w}(i)$ to refer to the index i of an array $\vec{w}$. We write $[i := v]\vec{w}$ for an update to an array, and we similarly write $[\ell := t]\alpha$ for an insertion in the allocation map. We write v^n for an array of length n, where each element of the array is initialized with the value v.

HeadAlloc allocates an array, extending the store and the allocation map. HeadLoad acquires the value v from an index of an array. HeadStore, HeadLetVal, HeadIfTrue and HeadIfFalse are standard. HeadClosure allocates a closure and HeadCall calls a closure. HeadCallPrim calls a primitive, whose result is computed at the meta-level by the $\xrightarrow{\text{pure}}$ relation. HeadPair and HeadProj allocate and project immutable pairs, respectively. HeadInj and HeadCase allocate and case over immutable sums, respectively. HeadFold and HeadUnfold handle iso-recursive types in a standard way.

Scheduler reduction relation. In order to keep track of the timestamp of each task and whether the task is activated or suspended, we follow Westrick et al. [2020] and enrich the semantics with an auxiliary structure called a *task tree*, written T, of the following formal grammar: $T \triangleq t \in \mathcal{T} \mid T \otimes_t T$. A leaf t indicates an active task denoted by its timestamp. A node $T_1 \otimes_t T_2$ represents a suspended task t that has forked two parallel computations, recursively described by the task trees T_1 and T_2.

Figure 8 presents the scheduling reduction relation $\sigma / \alpha / G / T / e \xrightarrow{\text{sched}} \sigma' / \alpha' / G' / T' / e'$ as either a head step, a fork, or a join. In this reduction relation, σ is a store, α an allocation map, G a computation graph, T a task tree, and e an expression. The SchedHead reduction describes a head reduction. The SchedFork reduction describes a fork: the task tree consists of a leaf t_0 and the expression of $\text{par}(v_1, v_2)$, where both v_1 and v_2 are closures to be executed in parallel. The reduction generates two fresh timestamps t_1 and t_2, adds the corresponding edges to the computation graph, and updates the task tree to comprise the node with two leaves $t_1 \otimes_t t_2$. The reduction then updates the expression to the active parallel pair $v_1\,[()] \parallel v_2\,[()]$, reflecting the parallel call of the two closures v_1 and v_2, each one called with a single argument, the unit value $()$. The SchedJoin

$$\textsc{DELeaf}\ \frac{\forall \ell.\, \ell \in \mathit{roots}(e) \implies G \vdash \alpha(\ell) \preccurlyeq t}{\text{Disentangled } (\_, \alpha, G) \,/\, t \,/\, e} \qquad \textsc{DEPar}\ \frac{\text{Disentangled } S \,/\, T_1 \,/\, e_1 \qquad \text{Disentangled } S \,/\, T_2 \,/\, e_2}{\text{Disentangled } S \,/\, T_1 \otimes_t T_2 \,/\, e_1 \parallel e_2}$$

$$\textsc{DEBind}\ \frac{S = (\_, \alpha, G) \qquad \text{Disentangled } S \,/\, T_1 \otimes_t T_2 \,/\, e \qquad \forall \ell.\, \ell \in \mathit{roots}(K) \implies \forall t'.\, t' \in \text{leaves}(T_1) \cup \text{leaves}(T_2) \implies G \vdash \alpha(\ell) \preccurlyeq t'}{\text{Disentangled } S \,/\, T_1 \otimes_t T_2 \,/\, K[e]}$$

Fig. 9. Definition of Disentanglement

reduction describes a join, and differs from prior semantics for disentanglement [Moine et al. 2024; Westrick et al. 2022] because it reuses a timestamp (§ 2.1). The task tree is at a node t with two leaves $t_1 \otimes_t t_2$, and both leaves reached a value. The reduction adds edges (t_1, t) and (t_2, t) to the computation graph, and allocates a memory cell to store the result of the (active) parallel pair. It then updates the task tree to the leaf t.

Parallelism and reduction under a context. The lower part of Figure 8 presents the main reduction relation $S \,/\, T \,/\, e \xrightarrow{\text{step}} S' \,/\, T' \,/\, e'$, which describe a scheduling reduction inside the whole parallel program [Moine et al. 2024]. A *configuration* $S \,/\, T \,/\, e$ consists of the program state S, the task tree T, and an expression e. This expression e can consist of multiple tasks, governed by the nesting of active parallel pairs $(e_1 \parallel e_2)$. The corresponding timestamps of these tasks are given by the accompanying task tree T. A state S consists of the tuple (σ, α, G), denoting a store σ, an allocation map α, and a computation graph G. The StepSched reduction describes a scheduling step. The other reductions describe *where* the scheduling reduction takes place in the whole parallel program. The StepBind reduction describes a reduction under an evaluation context. The StepParL and StepParR reductions are non-deterministic: if a node of the task tree is encountered facing an active parallel pair, the left side or the right side can reduce.

3.3 Definition of Disentanglement

The property Disentangled $S \,/\, T \,/\, e$ asserts that, given a program state S and a task tree T, the expression e is disentangled—that is, the roots of each task in e were allocated by preceding tasks. Figure 9 gives the inductive definition of Disentangled $S \,/\, T \,/\, e$. If the program state has an allocation map α and a computation graph G, and if the task tree is a leaf t, DELeaf requires that for every location ℓ in $\mathit{roots}(e)$, that is, the set of locations syntactically occurring in e, the location ℓ has been allocated by a task $\alpha(t)$ preceding t in G. If the task tree is a node $T_1 \otimes_t T_2$, there are two cases. In the first case, if the expression is an active parallel pair, DEPar requires that the two sub-expressions are disentangled. Otherwise, the expression must be of the form $K[e]$, and then DEBind requires that e itself is disentangled and that for every location ℓ occurring in the evaluation context K, the location ℓ has been allocated before every leaf t' of T_1 and T_2. This definition is similar to the key invariant rootsde by Westrick et al. [2022] and Moine et al. [2024].

4 Type System

In this section, we describe TypeDis in depth. First, we present the formal syntax of types (§ 4.1) as well as the typing judgment (§4.2). We then comment on typing rules for mutable heap blocks (§4.3), which enforce disentanglement. Next, we present the rules for creating and calling closures (§ 4.4), which are crucial for understanding our approach for typing the par primitive (§ 4.5). We then focus on advanced features of TypeDis: general recursive types and type polymorphism (§ 4.6). We conclude by presenting subtiming (§ 4.7).

$$
\begin{array}{rrcl}
\text{Timestamp variables} & \delta & & \\
\text{Type variables} & \alpha & & \\
\text{Logical graphs} & \Delta & \triangleq & \Delta,\ \delta \preccurlyeq \delta \mid \emptyset \\
\text{Kinds} & \kappa & \triangleq & \star \mid \boxtimes \Rightarrow \kappa \\
\text{Unboxed types} & \tau & \triangleq & () \mid \mathsf{bool} \mid \mathsf{int} \\
\text{Boxed types} & \sigma & \triangleq & \mathsf{array}(\rho) \mid (\rho \times \rho) \mid (\rho + \rho) \mid \forall \vec{\delta}\, \Delta.\ \vec{\rho} \to^{\delta} \rho \\
\text{Types} & \rho & \triangleq & \tau \mid \lambda \delta.\ \rho \mid \rho\, \delta \mid \forall \alpha :: \kappa.\ \rho \mid \mu \alpha.\ \sigma @ \delta \mid \alpha \mid \sigma @ \delta \\
\text{Environments} & \Gamma & \triangleq & x : \rho, \Gamma \mid \alpha :: \kappa, \Gamma \mid \emptyset
\end{array}
$$

Fig. 10. Syntax of types

4.1 Syntax of Types

Figure 10 presents the syntax of types.

To reason statically about the runtime notions of timestamps t and computation graphs G (§3.2), we introduce their corresponding static notions: *timestamp variables* δ and *logical graphs* Δ, respectively. A logical graph Δ is a set of pairs $\delta_1 \preccurlyeq \delta_2$, asserting that the timestamp δ_1 *precedes* the timestamp δ_2, that is, everything allocated by the task at δ_1 is safe to acquire for the task at δ_2. A logical graph can be understood as a static approximation of (a part of) the computation graph.

A powerful feature of our type system is its support for *timestamp polymorphism*, facilitated through *higher-order types*. This higher-order feature is instrumental in typing the par primitive (§4.5), and thus supporting the cyclic approach detailed in §2.1. Because our system is high-order, we introduce *kinds*, written κ, which capture the number of timestamps a type expects as arguments. The ground kind, written $\star$, indicates that the type does not take a timestamp argument. The successor kind, written $\boxtimes \Rightarrow \kappa$, indicates that the type expects $\kappa + 1$ timestamp arguments.

A base type τ describes an *unboxed* value, that is, a value that is not allocated on the heap. Base types include the unit type, Booleans, and integers.

The syntax of a types ρ is mutually inductive with the syntax of allocated types σ. A type ρ is either a base type τ, a type taking a timestamp argument $\lambda \delta.\ \rho$, an application of a type to a timestamp $\rho\, \delta$, a universal quantification of a type variable with some kind $\forall \alpha :: \kappa.\ \rho$, a recursive type $\mu \alpha.\ \sigma @ \delta$, a type variable α, or an allocated type annotated by a timestamp $\sigma @ \delta$. When the timestamp δ does not matter, we write $\sigma @ \_$. An allocated type σ is either an array $\mathsf{array}(\rho)$, an immutable pair $(\rho \times \rho)$, an immutable sum $(\rho + \rho)$, or a function $\forall \vec{\delta}\, \Delta.\ \vec{\rho} \to^{\delta} \rho$.

Types support α-equivalence for both type and timestamp variables, as well as β-reduction.

4.2 The Typing Judgment

A typing environment Γ is a map from free program variables to types, and from free type variables to kinds. The general form of the typing judgment of TypeDis is:

$$\Delta \mid \Gamma \vdash e : \rho \triangleright \delta$$

where Δ is a logical graph, Γ a typing environment, e the expression being type-checked at type ρ and at current timestamp δ.

Selected rules of the type system appear in Figure 11. In this figure, we follow Barendregt's convention [Barendregt 1984], meaning that bound variables are always assumed fresh, that is, distinct from any other variable in scope.

Various rules are standard: the rule T-Var type-checks variables and the rules T-Unit, T-Int and T-Bool type-check base types. The structural rules T-Let and T-If are also standard, and type-check let bindings and if statements, respectively. In the remainder, we discuss the rules that deserve special attention with regard to disentanglement.

T-Var
$$\frac{\Gamma(x) = \rho}{\Delta \mid \Gamma \vdash x : \rho \triangleright \delta}$$

T-Unit
$$\Delta \mid \Gamma \vdash () : \mathsf{unit} \triangleright \delta$$

T-Int
$$\Delta \mid \Gamma \vdash i : \mathsf{int} \triangleright \delta$$

T-Bool
$$\Delta \mid \Gamma \vdash b : \mathsf{bool} \triangleright \delta$$

T-Let
$$\frac{\Delta \mid \Gamma \vdash e_1 : \rho' \triangleright \delta \qquad \Delta \mid x : \rho', \Gamma \vdash e_2 : \rho \triangleright \delta}{\Delta \mid \Gamma \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \rho \triangleright \delta}$$

T-If
$$\frac{\Delta \mid \Gamma \vdash e_1 : \mathsf{bool} \triangleright \delta \qquad \Delta \mid \Gamma \vdash e_2 : \rho \triangleright \delta \qquad \Delta \mid \Gamma \vdash e_3 : \rho \triangleright \delta}{\Delta \mid \Gamma \vdash \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 : \rho \triangleright \delta}$$

T-Pair
$$\frac{\Delta \mid \Gamma \vdash e_1 : \rho_1 \triangleright \delta \qquad \Delta \mid \Gamma \vdash e_2 : \rho_2 \triangleright \delta}{\Delta \mid \Gamma \vdash (e_1, e_2) : (\rho_1 \times \rho_2)@\delta \triangleright \delta}$$

T-Proj
$$\frac{\Delta \mid \Gamma \vdash e : (\rho_1 \times \rho_2)@\_ \triangleright \delta}{\Delta \mid \Gamma \vdash \mathsf{proj}_\iota\, e : \rho_i \triangleright \delta}$$

T-Inj
$$\frac{\Delta \mid \Gamma \vdash e : \rho_i \triangleright \delta}{\Delta \mid \Gamma \vdash \mathsf{inj}_\iota\, e : (\rho_1 + \rho_2)@\delta \triangleright \delta}$$

T-Case
$$\frac{\Delta \mid \Gamma \vdash e : (\rho_1 + \rho_2)@\_ \triangleright \delta \qquad \Delta \mid x_1 : \rho_1, \Gamma \vdash e_1 : \rho \triangleright \delta \qquad \Delta \mid x_2 : \rho_2, \Gamma \vdash e_2 : \rho \triangleright \delta}{\Delta \mid \Gamma \vdash \mathsf{match}\, e \,\mathsf{with}\, \mathsf{inj}_1\, x_1 \implies e_1 \mid \mathsf{inj}_2\, x_2 \implies e_2 \,\mathsf{end} : \rho \triangleright \delta}$$

T-Array
$$\frac{\Delta \mid \Gamma \vdash e_1 : \mathsf{int} \triangleright \delta \qquad \Delta \mid \Gamma \vdash e_2 : \rho \triangleright \delta}{\Delta \mid \Gamma \vdash \mathsf{alloc}\, e_1\, e_2 : \mathsf{array}(\rho)@\delta \triangleright \delta}$$

T-Store
$$\frac{\Delta \mid \Gamma \vdash e_1 : \mathsf{array}(\rho)@\_ \triangleright \delta \qquad \Delta \mid \Gamma \vdash e_2 : \mathsf{int} \triangleright \delta \qquad \Delta \mid \Gamma \vdash e_3 : \rho \triangleright \delta}{\Delta \mid \Gamma \vdash e_1[e_2] \leftarrow e_3 : () \triangleright \delta}$$

T-Load
$$\frac{\Delta \mid \Gamma \vdash e_1 : \mathsf{array}(\rho)@\_ \triangleright \delta \qquad \Delta \mid \Gamma \vdash e_2 : \mathsf{int} \triangleright \delta}{\Delta \mid \Gamma \vdash e_1[e_2] : \rho \triangleright \delta}$$

T-Abs
$$\frac{\Delta, \Delta_1, \delta \preccurlyeq \delta_f \mid f : (\forall \vec{\delta}_1\, \Delta_1.\ \vec{\rho}_1 \to^{\delta_f} \rho_2)@\delta, (\vec{x} : \vec{\rho}_1), \Gamma \vdash e : \rho_2 \triangleright \delta_f}{\Delta \mid \Gamma \vdash \mu f.\, \Lambda \vec{\delta}_1\, \Delta_1.\, \lambda \vec{x}!\delta_f.\, e : (\forall \vec{\delta}_1\, \Delta_1.\ \vec{\rho}_1 \to^{\delta_f} \rho_2)@\delta \triangleright \delta}$$

T-App
$$\frac{\begin{array}{c}\delta = [\vec{\delta}'_1/\vec{\delta}_1]\delta_f \qquad \vec{\rho}'_1 = [\vec{\delta}'_1/\vec{\delta}_1]\vec{\rho}_1 \qquad \rho'_2 = [\vec{\delta}'_1/\vec{\delta}_1]\rho_2 \qquad \Delta'_1 = [\vec{\delta}'_1/\vec{\delta}_1]\Delta_1 \\ \Delta \mid \Gamma \vdash e : (\forall \vec{\delta}_1\, \Delta_1.\ \vec{\rho}_1 \to^{\delta_f} \rho_2)@\_ \triangleright \delta \qquad \Delta \vdash \Delta'_1 \qquad \Delta \mid \Gamma \vdash \vec{e'} : \vec{\rho}'_1 \triangleright \delta\end{array}}{\Delta \mid \Gamma \vdash e\, \langle \vec{\delta}'_1 \rangle\, \vec{e'} : \rho'_2 \triangleright \delta}$$

T-Par
$$\frac{\begin{array}{c}\Gamma \vdash \varphi_1 :: \mathbb{X} \Rightarrow \star \qquad \Gamma \vdash \varphi_2 :: \mathbb{X} \Rightarrow \star \\ \Delta \mid \Gamma \vdash e_1 : (\forall \delta'\, \delta \preccurlyeq \delta'.\ () \to^{\delta'} \varphi_1\, \delta')@\_ \triangleright \delta \qquad \Delta \mid \Gamma \vdash e_2 : (\forall \delta'\, \delta \preccurlyeq \delta'.\ () \to^{\delta'} \varphi_2\, \delta')@\_ \triangleright \delta\end{array}}{\Delta \mid \Gamma \vdash \mathsf{par}(e_1, e_2) : (\varphi_1\, \delta \times \varphi_2\, \delta)@\delta \triangleright \delta}$$

T-Fold
$$\frac{\Gamma \vdash \mu\alpha.\, \sigma@\delta :: \star \qquad \Delta \mid \Gamma \vdash e : ([\mu\alpha.\, \sigma@\delta/\alpha]\sigma)@\delta \triangleright \delta}{\Delta \mid \Gamma \vdash \mathsf{fold}\, e : \mu\alpha.\, \sigma@\delta \triangleright \delta}$$

T-Unfold
$$\frac{\Gamma \vdash \mu\alpha.\, \sigma@\delta :: \star \qquad \Delta \mid \Gamma \vdash e : \mu\alpha.\, \sigma@\delta \triangleright \delta}{\Delta \mid \Gamma \vdash \mathsf{unfold}\, e : ([\mu\alpha.\, \sigma@\delta/\alpha]\sigma)@\delta \triangleright \delta}$$

T-TAbs
$$\frac{\Delta \mid \alpha :: \kappa, \Gamma \vdash e : \rho \triangleright \delta \qquad \mathsf{veryPure}\, e}{\Delta \mid \Gamma \vdash \Lambda\alpha :: \kappa.\, e : \forall \alpha :: \kappa.\ \rho \triangleright \delta}$$

T-TApp
$$\frac{\Gamma \vdash \rho' :: \kappa \qquad \Delta \mid \Gamma \vdash e : \forall \alpha :: \kappa.\ \rho \triangleright \delta}{\Delta \mid \Gamma \vdash e\, \langle \rho' \rangle : [\rho'/\alpha]\rho \triangleright \delta}$$

T-GetRoot
$$\frac{\begin{array}{c}\Gamma(x) = \mathsf{array}(\sigma')@\delta' \ \vee\ \Gamma(x) = \mu\alpha.\, \sigma'@\delta' \\ \{(\delta', \delta)\} \cup \Delta \mid \Gamma \vdash e : \rho \triangleright \delta\end{array}}{\Delta \mid \Gamma \vdash e : \rho \triangleright \delta}$$

T-Subtiming
$$\frac{\Delta \mid \Gamma \vdash e : \rho \triangleright \delta \qquad \Delta \vdash \rho \subseteq_\delta \rho'}{\Delta \mid \Gamma \vdash e : \rho' \triangleright \delta}$$

Fig. 11. The type system (selected rules)

4.3 Typing Rules for Heap Blocks

Heap blocks must be handled with care to guarantee disentanglement: every time the program acquires a location—that is, the address of a heap block—we must ensure that this location has been allocated by a preceding task. Otherwise, this newly created root would break the disentanglement invariant (§3.3).

Load operations are so common in a programming language that we chose to enforce the following invariant on the typing judgment $\Delta \mid \Gamma \vdash e : \rho \triangleright \delta$: every location that can be acquired from Γ was allocated before the current timestamp δ (§2.4). Hence, load operations (from immutable blocks and from mutable blocks) do not have any timestamp check.

Operations on immutable blocks are type-checked by T-Pair and T-Proj, for pairs, and by T-Inj and T-Case, for sums. In particular, T-Pair and T-Inj reflect that pair allocation and injection allocation *allocate heap blocks*, hence, the resulting type is annotated by $@\delta$, marking the allocating timestamp inside the type.

Operations on mutable blocks are type-checked by T-Array, T-Store and T-Load.

4.4 Abstractions and Timestamp Polymorphism

A function can be seen as a *delayed computation*. In our case, this notion of "delay" plays an interesting role: a function can run on a task distinct from the one that allocated it. Hence, functions in TypeDis have three non-standard features related to timestamps, roughly describing the status of the computation graph when the function will run. First, a function takes timestamp parameters, which are universally quantified. Second, a function takes constraint over these timestamps, as a logical graph. Third, a function is annotated by a timestamp representing the task it will run on.

Let us focus on the abstraction rule T-Abs. This rule type-checks a function definition of the form $\mu f.\, \Lambda\vec{\delta}_1\, \Delta_1.\, \lambda\vec{x}!\delta_f.\, e$, which binds, in the function body e, the recursive name f, the timestamp parameters $\vec{\delta}_1$, the logical graph Δ_1, the function arguments $\vec{x}$, and the running timestamp δ_f. The current timestamp is δ and the type associated to the function is $(\forall\vec{\delta}_1\, \Delta_1.\, \vec{\rho}_1 \to^{\delta_f} \rho_2)@\delta$. This type asserts that, if *(i)* there is some instantiation of $\vec{\delta}_1$ satisfying Δ_1, *(ii)* there are some arguments of type $\vec{\rho}_1$, and *(iii)* the timestamp of the calling task is δ_f, then the function will produce a result of type ρ_2. This type also reminds us that a function is a heap-allocated object, and is hence annotated by the task that allocated it, here δ. The premise of the T-Abs changes the current timestamp to be δ_f, the timestamp of the invoking task, and requires the body e to be of type ρ_2. T-Abs is in fact the sole rule of the system "changing" the current timestamp while type checking. The logical graph is augmented with Δ_1 plus the knowledge that δ precedes δ_f, conveying the fact that a function can only be called at a subsequent timestamp. The environment Γ is extended with the parameters $(\vec{x} : \vec{\rho}_1)$ as well as the recursive name f. Note that timestamp parameters $\vec{\delta}_1$ and logical graph Δ_1, are before the arguments $\vec{x}$. This means that the body e will be able to recursively call f with different timestamp arguments (potentially including a different δ_f), for example after it forked. We already saw such an example for the `build` and `selectmap` functions in Sections 2.3 and 2.4.

Let us now focus on T-App, type-checking a function application. The conclusion type-checks the expression $e\ \langle\vec{\delta}'_1\rangle\ \vec{e}'$ to be of type ρ'_2 at the current timestamp δ. The premise of T-App requires e to be a function of type $\forall\vec{\delta}_1\, \Delta_1.\, \vec{\rho}_1 \to^{\delta_f} \rho_2$, allocated by some irrelevant task. The premise then substitutes in all the relevant parts the user-supplied timestamps $\vec{\delta}'_1$ in place of $\vec{\delta}_1$. Hence, the result type ρ'_2 is in fact equal to $[\vec{\delta}'_1/\vec{\delta}_1]\rho_2$. In particular, the premise $\delta = [\vec{\delta}'_1/\vec{\delta}_1]\delta_f$ requires that the running timestamp δ_f to be equal to δ, the current timestamp. The premise also requires the logical graph Δ'_1 to be a subgraph of the logical graph Δ, written $\Delta \vdash \Delta'_1$ meaning that every pair of vertices reachable in Δ'_1 must be also reachable in Δ. This property is formally defined in Appendix A.3. Finally, the premise requires the arguments $\vec{e}$ to be of the correct type $\vec{\rho}'_1$.

4.5 The Par Rule

The typing rule for the par primitive is at the core of TypeDis.

T-Par type-checks $\mathsf{par}(e_1, e_2)$ at current timestamp δ. Recall (§3.2) that the results of e_1 and e_2 must be closures; these closures are then called in parallel and their results are returned as an immutable pair. To preserve disentanglement, the two closures must not communicate allocations they make with each other. Hence, the premise of T-Par requires the two expressions e_1 and e_2 to be of type $\forall \delta'\ \delta \preccurlyeq \delta'.\ () \to^{\delta'} \ldots$, signaling that they must be closures that are expected to run on a task δ', universally quantified, and subsequent to δ. Because of this universal quantification over the running timestamp δ' and because the rules allocating blocks (T-Array, T-Pair, T-Proj and T-Abs) always tag the value they allocate with the running timestamp, the tasks will not be able to communicate allocations they make.

After these two closure calls terminate, and their underlying tasks join, the parent task gain access to everything the two children allocated. In fact, from the point of view of disentanglement, we can even pretend that the parent task itself allocated these locations! T-Par does more than pretending and **backtimes** the return types of the two closures, by substituting the running timestamp of the children δ' by the running timestamp of the parent δ. Indeed, the return types of the closures, $\varphi_1\ \delta'$ for e_1 and $\varphi_2\ \delta'$, for e_2, signal that these two closures will return some type, parametrized by the running timestamp δ'. This formulation allows the rule to type-check the original $\mathsf{par}(e_1, e_2)$ as $(\varphi_1\ \delta \times \varphi_2\ \delta)@\delta$, that is, a pair of the two types returned by the closures, *but where the running timestamp of the child δ' was replaced by the running timestamp of the parent δ.*

4.6 Recursive Types and Type Polymorphism

Recursive types. TypeDis supports iso-recursive types [Pierce 2002, §20.2]. In TypeDis, a recursive type takes the form $\mu\alpha.\ \sigma@\delta$, binding the recursive name α in the allocated type σ which must have been allocated at δ. This syntax ensures that types are somehow well-formed, and forbids meaningless types $\mu\alpha.\ \alpha$ as well as useless types $\mu\alpha.\ \mu\beta.\ \rho$. T-Fold and T-Unfold allow for going from $\mu\alpha.\ \sigma@\delta$ to $([\mu\alpha.\ \sigma@\delta/\alpha]\sigma)@\delta$ and vice-versa. Note that this approach requires that the recursive occurrence of α are all allocated at the same timestamp; all the nodes of the recursive data structures must have been allocated at the same timestamp. This may seem restrictive, but subtiming will relax this requirement (§4.7).

Let us give an example. The type of lists allocated at timestamp δ containing integers is:

$$\mu\alpha.\ (() + (\mathsf{int} \times \alpha)@\delta)@\delta$$

This type describes that a list of integers is either the unit value (describing the nil case), or the pair of an integer and a list of integers (describing the cons case).

Type polymorphism. TypeDis supports type polymorphism, through type abstraction T-TAbs and type application T-TApp. The latter has a standard form but the former has an unusual premise $\mathsf{veryPure}\ e$.

Indeed, it is well-known that mutable state and polymorphism is unsound if unrestricted. The modern solution is called the *value restriction* [Wright 1995]. This simple technique consists of allowing type abstraction only in front of *values*, that is in a standard lambda-calculus, functions. However, DisLang_2 has an unusual aspect: functions are *not* values, they are allocated on the heap (§3.1). Hence, the value restriction is not applicable as-is, as it is crucial to allow universal type quantification in front of functions. We present here a variant of the value restriction, that allows type quantification in front of any *pure expression that does not call a function, projects a pair, cases over a sum, or fork new tasks.* This includes function allocation, pair allocation, sums injection, as well as other control-flow constructs. This syntactic check is ensured by the predicate $\mathsf{veryPure}\ e$

$$\textsc{S-Refl}\ \ \Delta \vdash \rho \subseteq_\delta \rho \qquad \textsc{S-ReflAt}\ \ \Delta \vdash \sigma \subseteq_\delta \sigma$$

$$\textsc{S-TAbs}\ \ \frac{\Delta \vdash \rho_1 \subseteq_\delta \rho_2}{\Delta \vdash \forall \alpha :: \kappa.\ \rho_1 \subseteq_\delta \forall \alpha :: \kappa.\ \rho_2} \qquad \textsc{S-Pair}\ \ \frac{\Delta \vdash \rho l_1 \subseteq_\delta \rho l_2 \qquad \Delta \vdash \rho r_1 \subseteq_\delta \rho r_2}{\Delta \vdash (\rho l_1 \times \rho r_1) \subseteq_\delta (\rho l_2 \times \rho r_2)}$$

$$\textsc{S-Sum}\ \ \frac{\Delta \vdash \rho l_1 \subseteq_\delta \rho l_2 \qquad \Delta \vdash \rho r_1 \subseteq_\delta \rho r_2}{\Delta \vdash (\rho l_1 + \rho r_1) \subseteq_\delta (\rho l_2 + \rho r_2)} \qquad \textsc{S-At}\ \ \frac{\Delta \vdash \delta_1 \preccurlyeq \delta_2 \qquad (\delta_1 \neq \delta_2 \implies \Delta \vdash \delta_2 \preccurlyeq \delta) \qquad \Delta \vdash \sigma_1 \subseteq_{\delta_2} \sigma_2}{\Delta \vdash \sigma_1@\delta_1 \subseteq_\delta \sigma_2@\delta_2}$$

$$\textsc{S-Rec}\ \ \frac{\Delta \vdash \delta_1 \preccurlyeq \delta_2 \qquad (\delta_1 \neq \delta_2 \implies \Delta \vdash \delta_2 \preccurlyeq \delta) \qquad \Delta \vdash \sigma_1 \subseteq_{\delta_2} \sigma_2 \qquad \Delta \mid \alpha \mapsto \delta_2 \vdash_{\delta_2} \sigma_2}{\Delta \vdash \mu\alpha.\ \sigma_1@\delta_1 \subseteq_\delta \mu\alpha.\ \sigma_2@\delta_2}$$

$$\textsc{S-Abs}\ \ \frac{\Delta' = \Delta \cup \Delta_2 \qquad \Delta' \vdash \Delta_1 \qquad \Delta' \vdash \rho\vec{s}_2 \subseteq_{\delta_f} \rho\vec{s}_1 \qquad \Delta' \vdash \rho_1 \subseteq_{\delta_f} \rho_2}{\Delta \vdash \forall \vec{\delta s}\, \Delta_1.\ \rho\vec{s}_1 \to^{\delta_f} \rho_1 \subseteq_\delta \forall \vec{\delta s}\, \Delta_2.\ \rho\vec{s}_2 \to^{\delta_f} \rho_2}$$

Fig. 12. The subtiming judgement

that appears as a premise of the type abstraction rule T-TAbs. The predicate veryPure e is defined in Appendix A.2.

TypeDis supports higher-kind type polymorphism. For example, reminding of the typing rule T-Par, one could present par as a higher-order function of the following type

$$\mathsf{par}\ :\ \forall(\varphi_1 :: \mathbb{X} \Rightarrow \star)\,(\varphi_2 :: \mathbb{X} \Rightarrow \star).$$
$$\forall\delta\,\delta_1\,\delta_2.\,(\forall\delta'\,\delta \preccurlyeq \delta'.\,() \to^{\delta'} \varphi_1\,\delta')@\delta_1 \to (\forall\delta'\,\delta \preccurlyeq \delta'.\,() \to^{\delta'} \varphi_2\,\delta')@\delta_2 \to^{\delta} (\varphi_1\,\delta \times \varphi_2\,\delta)@\delta$$

Note that taking $\varphi_1 = \varphi_2 = \lambda\delta.\,\mathsf{tree}@\delta$ and doing beta reduction matches the type presented in Section 2.3.

4.7 Subtiming

As presented so far, backtiming—that is, substituting the timestamp of a child task by the one of its parent task at the join point—is the only way of changing a timestamp inside a type (§4.5). We propose here another mechanism that we dub *subtiming*. As the name suggests, subtiming is a form of subtyping [Pierce 2002, §15] for timestamps.

At a high-level, subtiming allows for "advancing" a timestamp within a type, as long as this update makes sense. This notion of "advancing" relates to the notion of *precedence*, describing the reachability between two timestamps. We write $\Delta \vdash \delta_1 \preccurlyeq \delta_2$ to describe that δ_1 can reach δ_2 in Δ (Appendix A.3). Equipped with this reachability predicate, we make a first attempt at capturing the idea of subtiming as follows:

$$\textsc{Special-Case-of-Subtiming}\ \ \frac{\Delta \mid \Gamma \vdash e : \sigma@\delta_1 \triangleright \delta \qquad \Delta \vdash \delta_1 \preccurlyeq \delta_2 \qquad \Delta \vdash \delta_2 \preccurlyeq \delta}{\Delta \mid \Gamma \vdash e : \sigma@\delta_2 \triangleright \delta}$$

Special-Case-of-Subtiming asserts that an expression of type $\sigma@\delta_1$ can be viewed as an expression of type $\sigma@\delta_2$ as long as δ_1 precedes δ_2, and that δ_2 is not ahead of time, that is δ_2 precedes the current timestamp δ. Indeed, TypeDis enforces that, if $\Delta \mid \Gamma \vdash e : \sigma@\delta' \triangleright \delta$ holds, then δ' precedes δ.

While Special-Case-of-Subtiming is admissible in TypeDis, it is not general enough, as it only considers the timestamp at the root of a type. This motivates rule T-Subtiming in Figure 11, which relies on the subtiming judgment $\Delta \vdash \rho \subseteq_\delta \rho'$, given in Figure 12, and acts as a subsumption rule. Intuitively, the judgment $\Delta \vdash \rho \subseteq_\delta \rho'$ captures the fact the timestamps in ρ precede the timestamps in ρ' under logical graph Δ, knowing that the every timestamp occurring in ρ' must

precede δ. The definition of the judgment now allows changing the timestamps inside immutable types. Because of variance issues (see [Pierce 2002, §15.5]), however, subtiming for mutable types is only shallow: a timestamp can be changed only at the root of an array type. Subtiming in TypeDis is thus *semi-shallow*.

The subtiming judgment $\Delta \vdash \rho \subseteq_\delta \rho'$ assumes that types are in β-normal form. S-Refl and S-ReflAt assert that the subtiming judgment is reflexive for both types and allocated types. S-TAbs asserts that subtiming goes below type quantifiers. S-Pair and S-Sum reflect that subtiming for immutable types is deep.

S-At illustrates the case presented in Special-Case-of-Subtiming. This rule asserts that, with logical graph Δ and maximum allowed timestamp δ, the allocated type $\sigma_1@\delta_1$ is a subtype of $\sigma_2@\delta_2$ if three conditions are met. First, δ_1 must precede δ_2. Second, if subtiming is applied here, that is, if $\delta_1 \neq \delta_2$, then δ_2 must precede δ, the maximum timestamp allowed. Third, σ_1 must recursively be a subtype of σ_2, with maximum timestamp allowed δ_2. Indeed, recall the TypeDis allows only for up-pointers: every timestamp in σ_2 must precede δ_2.

S-Rec allows subtiming for recursive types $\mu\alpha.\, \sigma_1@\delta_1$ and $\mu\alpha.\, \sigma_2@\delta_2$. The three first premises (in the left-to-right, top-to-bottom order) are the same as for S-At. A fourth premise $\Delta \mid \alpha \mapsto \delta_2 \vdash_{\delta_2} \sigma_2$ requires explanations. This predicate, dubbed the "valid variable" judgment and whose formal definition appears in Appendix A.4, ensures two properties. First, that α does not appear in an array type (because subtiming is not allowed at this position) or in an arrow or another recursive type (for simplicity). Second, that if α appears under a timestamp δ, then δ_2 must precede δ.

S-Abs allows subtiming for function types $\forall \vec{\delta s}\, \Delta_1.\ \vec{\rho s}_1 \to^{\delta_f} \rho_1$ and $\forall \vec{\delta s}\, \Delta_2.\ \vec{\rho s}_2 \to^{\delta_f} \rho_2$. The quantified timestamps $\vec{\delta s}$ and the calling timestamp δ_f must be the same. The extended logical graph Δ', equal to $\Delta \cup \Delta_2$, must subsume Δ_1. Moreover, the arguments $\vec{\rho s}_2$ must subtime $\vec{\rho s}_1$ (note the polarity inversion). The return type ρ_1 must subtime ρ_2.

Before using subtiming, information about precedence may be needed. TypeDis guarantees a strong invariant: every timestamp occurring in the typing environment comes before the current timestamp. Such an invariant is illustrated by T-GetRoot, which allows adding to the logical graph Δ an edge (δ', δ), where δ' is a timestamp in the environment and δ the current timestamp.

5 Soundness

In this section, we state the soundness of TypeDis and give an intuition for its proof, which takes the form of a logical relation in Iris and is mechanized in Rocq [Anon. 2025]. We first enunciate the soundness theorem (§ 5.1). We then recall the concepts of Iris we need (§ 5.2) and present DisLog$_2$ (§ 5.3), the particular instantiation of Iris we use. We then devote our attention to the formal proof, by presenting the high-level ideas of the logical relation (§ 5.4) we developed, its fundamental theorem (§ 5.5). We conclude by assembling all the building blocks we presented and sketch the soundness proof of TypeDis (§ 5.6).

5.1 Soundness Statement of TypeDis

Our soundness statement adapts Milner [1978]' slogan "well-typed programs cannot go wrong" by proving that the reduction of a well-typed program reaches only configurations that are *safe* and *disentangled*.

We already formally defined the concept of disentanglement (§3.3). What about safety? Intuitively, a configuration is safe if *all tasks can take a step* or, conversely, *no task is ever stuck*. However, this property is too strong for our type system due to reasons unrelated to disentanglement. Being purposefully designed for disentanglement, our type system is not capable of verifying arbitrary functional correctness conditions. In particular, while the semantics of DisLang$_2$ ensures that

$$\textsc{OOB-Alloc}\ \frac{i < 0}{\mathsf{OOB}\ \sigma\ (\mathsf{alloc}\ i\ v)} \qquad \textsc{OOB-Load}\ \frac{\sigma(\ell) = \vec{v} \quad i < 0 \vee i \geq |\vec{v}|}{\mathsf{OOB}\ \sigma\ (\ell[i])} \qquad \textsc{OOB-Store}\ \frac{\sigma(\ell) = \vec{v} \quad i < 0 \vee i \geq |\vec{v}|}{\mathsf{OOB}\ \sigma\ (\ell[i] \leftarrow w)}$$

$$\textsc{OOB-CAS}\ \frac{\sigma(\ell) = \vec{v} \quad i < 0 \vee i \geq |\vec{v}|}{\mathsf{OOB}\ \sigma\ (\mathsf{CAS}\ \ell\ i\ w_1\ w_2)}$$

$$\textsc{Red-Sched}\ \frac{S\,/\,T\,/\,e \xrightarrow{\text{sched}} S'\,/\,T'\,/\,e'}{\mathsf{AllRedOrOOB}\ S\,/\,T\,/\,e} \qquad \textsc{Red-OOB}\ \frac{\mathsf{OOB}\ \sigma\ e}{\mathsf{AllRedOrOOB}\ (\sigma, \alpha, G)\,/\,t\,/\,e} \qquad \textsc{Red-Ctx}\ \frac{\mathsf{AllRedOrOOB}\ S\,/\,T\,/\,e}{\mathsf{AllRedOrOOB}\ S\,/\,T\,/\,K[e]}$$

$$\textsc{Red-Par}\ \frac{\begin{array}{c}(e_1 \notin \mathcal{V} \ \vee\ e_2 \notin \mathcal{V}) \\ (e_1 \notin \mathcal{V} \implies \mathsf{AllRedOrOOB}\ S\,/\,T_1\,/\,e_1) \\ (e_2 \notin \mathcal{V} \implies \mathsf{AllRedOrOOB}\ S\,/\,T_2\,/\,e_2)\end{array}}{\mathsf{AllRedOrOOB}\ S\,/\,T_1 \otimes_t T_2\,/\,e_1 \parallel e_2}$$

$$\textsc{Safe-Final}\ \mathsf{Safe}\ S\,/\,t\,/\,v \qquad \textsc{Safe-NonFinal}\ \frac{\mathsf{AllRedOrOOB}\ S\,/\,T\,/\,e}{\mathsf{Safe}\ S\,/\,T\,/\,e}$$

Fig. 13. The OOB, AllRedOrOOB and Safe predicates

accesses to arrays by load and store operations are within bounds and thus cannot cause a task to get stuck, our type system does not enforce that. This restriction comes at the advantage of freeing programmers from carrying out the correctness themselves, and having it instead been carried out by the type checker. Intuitively, we say that a configuration is safe if either it is final or each task can either take a step, or is facing a load or a store operation outside of bounds. We formalize these properties in Figure 13. The property $\mathsf{OOB}\ \sigma\ e$ asserts that the expression e faces an out-of-bounds operation: that is, an allocation, a load, a store, or a CAS outside the bounds. The property $\mathsf{AllRedOrOOB}\ S\,/\,T\,/\,e$ asserts that, within the configuration of the program state S, the task tree T and the expression e, every task of the task tree can either take a step or face an out-of-bounds operation. Red-Sched asserts that the configuration can take a scheduling step (that is, either a head step, a fork or a join). Red-OOB asserts that the configuration is at a leaf and faces an out-of-bounds operation. Red-Ctx asserts that an expression under an evaluation is reducible if this very expression is reducible. Red-Par asserts that an active parallel pair $e_1 \parallel e_2$ is reducible if at least one of $\{e_1, e_2\}$ is not a value, and that whichever of $\{e_1, e_2\}$ is not a value is reducible. (If both expressions are values, a join is possible). The property $\mathsf{Safe}\ S\,/\,T\,/\,e$ asserts that the configuration $S\,/\,T\,/\,e$ is either final (Safe-Final), that is, the task tree is at a leaf and the expression is a value, or that every task every task of the task tree can either take a step or face an out-of-bounds operation (Safe-NonFinal).

An expression e is ***always safe and disentangled*** if $(\emptyset, \emptyset, \{(t_0, t_0)\})\,/\,t\,/\,e \xrightarrow{\text{step}}{}^{*} S'\,/\,T'\,/\,e'$ implies that $\mathsf{Safe}\ S'\,/\,T'\,/\,e'$ and $\mathsf{Disentangled}\ S'\,/\,T'\,/\,e'$ hold, for some t_0 an initial timestamp.

Theorem 5.1 (Soundness of TypeDis). *If $\emptyset \mid \emptyset \vdash e : \rho \triangleright \delta$ then e is always safe and disentangled.*

Proof. We prove this theorem using a logical relation [Timany et al. 2024], which makes use of DisLog$_2$, a variation of DisLog [Moine et al. 2024]. We present the proof sketch in Section 5.6. □

5.2 Iris Primer

We set up our proofs in Iris [Jung et al. 2018b], and recall here the base notations. Iris' assertions are of type *iProp*. We write Φ for an assertion, $\ulcorner P \urcorner$ for a pure assertion, $\Phi_1 * \Phi_2$ for a separating conjunction, and $\Phi_1 \mathrel{-\!\!*} \Phi_2$ for a separating implication. We write a postcondition—that is, a predicate over values—using Ψ.

One of the most important feature of Iris consists of *invariants*. An invariant over an assertion Φ, written $\boxed{\Phi}$ intuitively denotes that Φ holds true in-between every computation step. (Formally,

$$\textsc{D-Load}\quad \frac{\ulcorner 0 \le i < |\vec{w}| \ \wedge\ \vec{w}(i) = v \urcorner \qquad \ell \mapsto_p \vec{w} \qquad v \clockwise t}{\mathsf{wp}\,\langle t,\ \ell[i]\rangle\,\{\lambda\, v'.\ \ulcorner v' = v\urcorner \ *\ \ell \mapsto_p \vec{w}\}}$$

$$\textsc{D-LoadOOB}\quad \frac{\ulcorner i < 0 \vee i \ge |\vec{w}|\urcorner \qquad \ell \mapsto_p \vec{w}}{\mathsf{wp}\,\langle t,\ \ell[i]\rangle\,\{\lambda\, \_.\ \bot\}}$$

$$\textsc{D-Par}\quad \frac{\begin{array}{c}\forall t_1\, t_2.\ \ t \preccurlyeq t_1 \ *\ t \preccurlyeq t_2 \Rrightarrow \exists \Psi_1\, \Psi_2.\ \ \mathsf{wp}\,\langle t_1,\ell_1\,[()]\rangle\,\{\Psi_1\} \ *\ \mathsf{wp}\,\langle t_2,\ell_2\,[()]\rangle\,\{\Psi_2\} \ * \\ \left(\forall v_1\, v_2\, \ell.\ \ \Psi_1\, v_1 \ *\ \Psi_2\, v_2 \ *\ t_1 \preccurlyeq t \ *\ t_2 \preccurlyeq t \ *\ \ell \mapsto (v_1, v_2) \mathrel{-\!\!*} \Psi\, \ell\right)\end{array}}{\mathsf{wp}\,\langle t,\ \mathsf{par}(\ell_1,\ell_2)\rangle\,\{\Psi\}}$$

$$\textsc{D-ClockMono}\quad \frac{v \clockwise t_1 \qquad t_1 \preccurlyeq t_2}{v \clockwise t_2}$$

Fig. 14. Selected rules of DisLog$_2$

invariants are annotated with so-called masks [Jung et al. 2018b, §2.2], we omit them for brevity.) Invariants, as well as other logical resources in Iris, are implemented using *ghost state*. We write $\Phi_1 \Rrightarrow \Phi_2$ a *ghost update*—that is, an update of the ghost state between Φ_1 and Φ_2.

Iris features a variety of modalities. In this work we use two of them extensively. First, the *persistence* modality, written $\square\,\Phi$ asserts that the assertion Φ is persistent, meaning in particular that $\square\,\Phi$ is duplicable. Second, the *later* modality, written $\triangleright\,\Phi$ asserts that Φ holds "one step of computation later".

We write $\ell \mapsto \vec{v}$ to denote that ℓ points-to an array with content $\vec{v}$. We write $\ell \mapsto_{\square} r$, with a *discarded fraction* [Vindum and Birkedal 2021], to denote that ℓ points-to an immutable block r (that is, either a closure, an immutable pair, or an immutable sum). This latter assertion is persistent.

5.3 Taking Advantage of the Cyclic Approach with DisLog$_2$

Moine et al. [2024] contributed DisLog, the first program logic for verifying disentanglement. DisLog depends on the very definition of disentanglement, and uses the standard approach presented in Section 2.1: when two tasks join, they form a new task with a fresh timestamp. This choice impacts the logic: the *weakest precondition* (WP) modality of DisLog takes the form $\mathsf{wp}\,\langle t,\ e\rangle\,\{\lambda\, t'\, v.\ \Phi\}$ and asserts that the expression e running on *current timestamp* t is disentangled, and if the evaluation of e terminates, it does so on the *end timestamp* t', with final value v and satisfying the assertion Φ. In particular, t and t' may not be the same, for example if e contains a call to par.

To accommodate the cyclic approach for disentanglement (§2.1), we had to develop a new version of DisLog, yielding the logic DisLog$_2$. DisLog$_2$ allows reusing the timestamp of the forking task for the child tasks upon join. As a result, the current timestamp and end timestamp of an expression always coincide, allowing us to simplify the WP of DisLog by simply removing the end timestamp parameter of the postcondition. Formally, the WP of DisLog$_2$ then takes the form

$$\mathsf{wp}\,\langle t,\ e\rangle\,\{\lambda\, v.\ \Phi\}$$

and asserts that the expression e running on timestamp t is disentangled, and if the evaluation of e terminates, it does so with final value v and satisfying the assertion Φ. Compared to DisLog, DisLog$_2$ tolerates out-of-bounds accesses.

Otherwise, DisLog$_2$ adapts all the ideas of DisLog. In particular, the logic features two persistent assertions related to timestamps. First, the clock assertion $\ell \clockwise t$ asserts that the location ℓ was allocated by a task that precedes t. The overloaded assertion $v \clockwise t$, for an arbitrary value v, is defined to $\ell \clockwise t$ if v is a location ℓ, and is sent to the trivial assertion $\ulcorner True \urcorner$ otherwise. Second, the precedence assertion $t_1 \preccurlyeq t_2$ asserts that task t_1 precedes task t_2 in the underlying computation graph. The precedence assertion forms a pre-order: it is reflexive and transitive. Crucially, the clock assertion *is monotonic with respect to the precedence pre-order* [Moine et al. 2024].

In the remaining of the paper, we write $t_1 \approx t_2$ to denote that t_1 and t_2 are *equivalent*, that is, both $t_1 \preccurlyeq t_2$ and $t_2 \preccurlyeq t_1$ hold.

Selected rules of DisLog$_2$. Figure 14 presents four key rules of DisLog$_2$. The premise of these rules are implicitly separated by a separating conjunction $*$.

The D-Load rule, targeting a load operation on the array ℓ at offset i on task t ensures disentanglement. Indeed, the rule requires that ℓ points-to the array $\vec{w}$ and that the offset i in $\vec{w}$ corresponds to the value v. It also requires the assertion $v \circledcirc t$, witnessing that if v is a location, then this location must have allocated by a preceding task. The D-LoadOOB rule is unusual in a program logic, and reflects that we purposefully allow for OOB accesses in verified programs, because our type system does. Because an OOB access results in a crash, the postcondition of the WP is $\ulcorner False \urcorner$, that is, allows the user to conclude anything. The D-Par rule is at the heart of DisLog$_2$ and allows verifying a parallel call to two closures ℓ_1 and ℓ_2 at timestamp t. The premise universally quantifies over t_1 and t_2, the two timestamps of the forked tasks, that are both preceded by t. Then, the user must provide two postconditions, Ψ_1 and Ψ_2 for the two tasks, and verify that the closure call $\ell_1\ [()]$ (resp. $\ell_2\ [()]$) is safe at timestamp t_1 (resp. t_2) with postcondition Ψ_1 (resp. Ψ_2). The second line of the premise requires the user to prove that, after the two tasks terminated and joined, the initial postcondition $\Psi\ \ell$ must hold, for some location ℓ pointing to the pair (v_1, v_2) where v_1 is the final result of t_1 and v_2 of t_2. The D-ClockMono rule formalizes the monotonicity of the clock assertion.

The soundness theorem of DisLog$_2$. The soundness theorem of DisLog$_2$ asserts that if e can be verified using the program logic, then e is always safe and disentangled.

Theorem 5.2 (Adequacy of DisLog$_2$). *If $wp\,\langle t,\, e\rangle\,\{\Psi\}$ holds, then e is always safe and disentangled.*

Proof. Similar to the adequacy proof of DisLog; see our mechanization [Anon. 2025]. □

5.4 A Logical Relation

The very heart of the soundness proof of TypeDis is a *logical relation*, set up in Iris using DisLog$_2$. *Logical relations* [Girard 1972; Pitts and Stark 1998; Plotkin 1973; Statman 1985; Tait 1967] are a technique that allows one to prescribe properties of valid programs in terms of their *behavior*, as opposed to solely their static properties. We adopt the *semantic approach* [Constable et al. 1986; Martin-Löf 1982; Timany et al. 2024], which allows terms that are not necessarily (syntactically) well-typed to be an inhabitant of the logical relation and has been successfully deployed in the RustBelt project [Jung et al. 2018a], for example.

Our logical relation is presented in Appendix A.6. We comment here on the high-level ideas.

The goal is to define the *interpretation* of a type ρ, intuitively returning a predicate over values, describing the values on relation with ρ, that is, the values inhabiting this type. Because our types have higher-kinds (that is, are functions over timestamps) some predicates involved in our logical relation are also predicates over timestamps. In particular, the interpretation of a type ρ with kind κ waits for κ timestamps, and then produces a predicate over values. This means that the interpretation of ρ at kind $\star$ is a regular predicate over values.

The presented relations involve two sorts of mappings, relating variables occuring in types to concrete values. First, a *timestamp mapping*, written h, which is a finite map from timestamp variables δ to concrete timestamps t. Second, a *type mapping*, written m, which is a finite map from type variables to tuples of a kind κ and a tuple of two functions depending on κ. The first function waits for κ timestamps and produces a predicate over values; it represents the semantic interpretation of the type by which the variable will be instantiated. The second function waits for κ timestamps and produces a timestamp; its result corresponds to the root timestamp of the type by which the variable will be instantiated.

$$
\begin{aligned}
[\![\, \Delta \mid \Gamma \vdash e : \rho \triangleright \delta \,]\!] \triangleq\ & \square \forall h\, m\, u.\ \ulcorner \mathrm{dom}\, \Gamma = \mathrm{dom}\, u \urcorner \mathrel{-\!\!*} \\
& \ulcorner \forall \alpha\, \kappa\, \Psi\, r.\ m(\alpha) = (\kappa, (\Psi, r)) \implies \mathrm{proper}_\kappa\, \Psi \ \wedge\ \mathrm{regular}_\kappa\, r \urcorner \mathrel{-\!\!*} \\
& \mathbin{*}_{(t_1,t_2)\in\Delta}\ h(t_1) \preccurlyeq h(t_2) \mathrel{-\!\!*} \\
& \mathbin{*}_{(x,\rho)\in\Gamma,\,(x,v)\in u}\ (\mathrm{root}\, \rho \preccurlyeq^h_m \delta \ *\ [\![\rho]\!]^h_m \star v) \mathrel{-\!\!*} \\
& \forall t.\ t \approx h(\delta) \mathrel{-\!\!*} \mathrm{wp}\, \langle t, [u/]e \rangle\, \{\lambda v.\ \mathrm{root}\, \rho \preccurlyeq^h_m \delta \ *\ [\![\rho]\!]^h_m \star v\}
\end{aligned}
$$

Fig. 15. The interpretation of typing judgments

The interpretation of a type guarantees that this type contains only *up-pointers* (§2.2), that is, the interpretation of $\sigma@\delta$ ensures that if δ' appears in σ, then δ' precedes δ. Such an invariant would be tedious to enforce as-is. Hence, our approach makes use of transitivity: the interpretation of $\sigma@\delta$ ensures that, for each outermost ρ encountered in σ, *the root timestamp of* ρ—conceptually, the outermost timestamp in ρ—precedes δ. Because this invariant is enforced at each stage of the type interpretation, and because precedence is transitive, we guarantee that there are only up-pointers. Appendix A.5 presents a function computing the root timestamp of a type, and defines the assertion $\mathrm{root}\, \rho \preccurlyeq^h_m \delta$, asserting that the root timestamp of ρ comes before δ with the mappings h and m.

The main relation is the *type relation* $[\![\rho]\!]^h_m\, \kappa$. It produces a predicate waiting for κ timestamps, a value v, and captures that v is of type ρ, within the timestamp mapping h and type mapping m.

Apart from timestamps, the seasoned reader of logical relations in Iris will not be surprised by our approach, as it follows the standard recipe [Timany et al. 2024]: a recursive type is interpreted using a guarded fixpoint, universal type quantification is interpreted as a universal quantification in the logic, an array is interpreted using an invariant, and an arrow using the WP. Moreover, every predicate is designed such that it is persistent.

5.5 Interpretation of Typing Judgments

We now focus on the interpretation of typing judgments of TypeDis. The overall form is standard: the interpretation of a judgment reflects that the interpretation of the typing environment implies the interpretation of the expression, itself being a WP whose postcondition asserts that the returned value is in relation with the interpretation of the type.

Figure 15 presents the definition of the interpretation of our typing judgment. The judgment under consideration has logical graph Δ, type environment Γ, expression e with type ρ at timestamp δ. The interpretation starts by quantifying over three mappings: the timestamp mapping h, the type mapping m as well as the *variable mapping* u, a map from code variable to values.

The variable mapping must have the same domain as the environment Γ. The type mapping m is restricted such that type variables are given only a proper interpretation (via the $\mathrm{proper}_\kappa\, \Psi$ property) and a regular root function (via the $\mathrm{regular}_\kappa\, r$ property). The property $\mathrm{proper}_\kappa\, \Psi$ captures that any timestamp parameter of Ψ can be replaced by an equivalent one. The property $\mathrm{regular}_\kappa\, r$ captures that the function r either ignores all its arguments or returns one of them. These two properties are needed in order to prove the correctness of T-Par.

Then, the interpretation requires that Δ is a valid logical graph, that is, each edge between two timestamp variables in Δ corresponds to an edge between their mapping. The interpretation also requires that, for every variable x that has type ρ in Γ and is associated to value v in u, then the root timestamp of v precedes the interpretation of δ and v is in relation with the interpretation of ρ. Next, the definition quantifies over a timestamp t, equivalent to the interpretation of δ, and asserts the WP at timestamp t, of the expression e in which variables are substituted by values following the variable mapping u. The postcondition asserts that the root timestamp of v precedes the interpretation of δ and that returned value v is in relation with the interpretation of the type ρ.

Once the interpretation of typing judgment is defined, we can state the *fundamental theorem*, relating the syntactical typing system (§4) and its semantic interpretation.

Theorem 5.3 (Fundamental). *If* $\Delta \mid \Gamma \vdash e : \rho \triangleright \delta$ *holds, then* $[\![\Delta \mid \Gamma \vdash e : \rho \triangleright \delta]\!]$ *holds too.*

Proof. By induction over the typing derivation; see our mechanization [Anon. 2025]. □

5.6 Putting Pieces Together: The Soundness Proof of TypeDis

We can finally unveil the proof of the soundness Theorem 5.1 of TypeDis, which we formally establish in Rocq. Let us suppose that $\emptyset \mid \emptyset \vdash e : \rho \triangleright \delta$ holds. Making use of the fundamental Theorem 5.3, we deduce that $[\![\emptyset \mid \emptyset \vdash e : \rho \triangleright \delta]\!]$ holds too. Unfolding the definition (Figure 15), instantiating the timestamp mapping h with the singleton map $[\delta := t_0]$—for t_0 some initial timestamp—and the type mapping m, and the variable mapping u with empty maps, and simplifying trivial premises concerning these mappings, we are left with the statement $\forall t.\ t \approx t_0 \twoheadrightarrow \mathsf{wp}\,\langle t,\, e\rangle\,\{\lambda\, v.\ \mathsf{root}\,\rho \preccurlyeq^h_m \delta * [\![\rho]\!]^{[\delta:=t_0]}_{\emptyset} \star v\}$. Instantiating t with t_0, we deduce that $\mathsf{wp}\,\langle t_0,\, e\rangle\,\{\lambda\, v.\ \mathsf{root}\,\rho \preccurlyeq^h_m \delta * [\![\rho]\!]^{[\delta:=t_0]}_{\emptyset} \star v\}$ holds. We finally use the adequacy Theorem 5.2 of DisLog_2 and deduce that e is always safe and disentangled.

6 Case Studies

We evaluate the usefulness of TypeDis by type checking several case studies in Rocq using the rules presented in Section 4.

We verify the examples presented in the "Key Ideas" Section 2. These examples illustrate: simple mechanics of the type system (§2.2), backtiming (§2.3) and subtiming (§2.4). In particular, the two last examples, `build` and `selectmap`, illustrate the use of TypeDis with high-order functions and a recursive immutable type (a binary tree with integer leaves).

Our largest case study consists of the typing of a parallel deduplication algorithm via concurrent hashing. This example is a case study of DisLog [Moine et al. 2024, §6.3]. Deduplication consists in removing duplicates in an array—something that can be done efficiently in a parallel and disentangled setting [Westrick 2022]. The algorithm rests on a folklore [VerifyThis 2022] concurrent, lock-free, fixed-capacity hash set using *open addressing* and *linear probing* to handle collision [Knuth 1998]. The main deduplication function allocates a new hash set, insert in parallel every element in the hash set using a parallel for loop, and finally returns the elements of the set. We implement the parallel for loop as a direct translation of code from standard library of MPL [Acar et al. 2020].

Parallel deduplication is an interesting case study: it involves the use of high order functions (because of the parallel for loop, for which we give a general type close to the one of the par primitive) as well as concurrent reads and writes inside an array using atomic operations. These reads and writes are disentangled because they concern data that was allocated before the parallel phase, and more precisely because they preserve the up-pointer invariant.

Intuitively, the deduplication function takes three arguments: a hashing function, a dummy element in order to populate the result array, and the array to deduplicate. We type-check the deduplication function with the type

$$\forall \alpha :: \star.\, \forall \delta\, \delta_1\, \delta_2.\, (\forall \delta'\, \delta \preccurlyeq \delta'.\, \alpha \to^{\delta'} \mathsf{int})@\delta_1 \to \alpha \to \mathsf{array}(\alpha)@\delta_2 \to^{\delta} \mathsf{array}(\alpha)@\delta$$

This type quantify over the type α of the elements of the array to deduplicate, and then quantify over δ, the calling timestamp, and δ_1 and δ_2, the (irrelevant) allocation timestamps of the first and third argument, respectively. The first argument is a closure of a hashing function on α, that will be called at subsequent tasks δ'. The second argument is a dummy element of type α. The third argument is the array to deduplicate.

7 Related Work

Disentanglement. The specific property we consider in this paper is based on the definition by Westrick et al. [2020] which was then formalized by Moine et al. [2024] in their development of DisLog, a program logic for disentanglement. Most of the existing work on disentanglement considers structured fork-join parallel code, as we do in this paper. More recently, Arora et al. [2024] showed that disentanglement is also applicable in a more general setting involving state and parallel futures. The authors specifically prove that disentanglement ensures deadlock-freedom in this setting. A question that we plan to investigate in future work is whether or not TypeDis could be extended to support futures.

Region-based Systems. TypeDis associates timestamp variables with values in their types. Immediately, we note similarities with region-based type and effect systems [Grossman et al. 2002; Tofte et al. 2004; Tofte and Talpin 1997] which have also recently received attention in supporting parallelism [Elsman and Henriksen 2023]. The timestamps in our setting are somewhat analogous to regions, with parent-child relationships between timestamps and the up-pointer invariant of TypeDis bearing resemblance to the stack discipline of region-based memory management systems. However, there are a number of key differences. In region-based systems, allocations may occur within any region, and all values within a region are all deallocated at the same moment; one challenge in such systems is statically predicting or conservatively bounding the lifetime of every value. In contrast, in TypeDis, allocations only ever occur at the "current" timestamp, and timestamps tell you nothing about deallocation—every value in our approach is dynamically garbage collected. Each timestamp in TypeDis is associated with a task within a nested fork-join task structure, and values with the same timestamp are all allocated by the same task (or one of its subtasks).

Possible Worlds Type Systems. Our type system falls into what can broadly be categorized as a *possible worlds* type system. These type systems augment the typing judgement with world modalities (in our case timestamp variables δ) that occur as syntactic objects in propositions (a.k.a. types), and typing is then carried out relative to an accessibility relation (in our case the logical graph Δ). While our work is the first to contribute a possible worlds type system for disentanglement, world modalities have been successfully used for other purposes. In the context of fork-join parallelism, Muller et al. [2017] employed world modalities to track priorities of tasks and guarantee absence of priority inversions, ensuring responsiveness and interactivity. While Muller et al. [2017] also require their priorities to be partially ordered, as we require timestamps to be partially ordered, their priorities are fixed, whereas ours are not. In the context of message-passing concurrency, world modalities have been employed to verify deadlock-freedom [Balzer et al. 2019], domain accessibility [Caires et al. 2019], and information flow control [Derakhshan et al. 2021, 2024]. This line of work not only differs in underlying computation model, considering a process calculus, but also adopts linear typing to control data races and non-determinism. While disentanglement does not forbid races, adopting some form of linear typing may be an interesting avenue for future work, to admit even more disentangled programs as well-typed, e.g. those with down-pointers.

Information Flow Control Type Systems. Information flow type systems [Sabelfeld and Myers 2003; Smith and Volpano 1998; Volpano et al. 1996] can also be viewed as representatives of possible worlds type systems, where modalities capture confidentiality (or integrity) and pc labels, and the accessibility relation is a lattice. Typically, modalities can change by typing. For example, when type checking the branches of an if statement the pc label is increased to the join of its current value and the confidentiality label of the branching condition. A similar phenomenon happens in TypeDis upon type checking a fork, where the sibling threads are type checked at a later timestamp. Besides these similarities in techniques employed, the fundamental invariants preserved by type checking

are different. In our setting it is the "no cross-pointers invariant", whereas it is noninterference for IFC type systems. As a result, the metatheory employed also differs: whereas we use a unary logical relation, noninterference demands a binary logical relation. Such a binary logical relation for termination-insensitive noninterference in the context of a sequential, higher-order language with higher-order store, for example, has been recently contributed by Gregersen et al. [2021]. The authors develop an IFC type system, in the spirit of Flow Caml [Pottier and Simonet 2003; Simonet 2003], with label polymorphism, akin to our timestamp polymorphism. Like our work, the authors use the semantic typing approach supported by the Iris separation logic framework. Similarly, the authors support subtyping on labels, allowing a label to be raised in accordance with the lattice, akin to our subtiming, in accordance with the precedence relation.

Type Systems for Parallelism and Concurrency. There has been significant work on developing static techniques, especially type systems, to guarantee correctness and safety properties (such as race-freedom, deadlock-freedom, determinism, etc.) for parallel and concurrent programs. For example, the idea of ownership [Clarke et al. 1998; Dietl and Müller 2005; Müller 2002; Noble et al. 1998] has been exploited to rule out races and deadlocks among threads [Boyapati et al. 2002, 2003; Boyapati and Rinard 2001]. Ownership is also enforced by linear type systems [Wadler 1990], which rule out races by construction and have been successfully employed in message-passing concurrency [Caires et al. 2019; Wadler 2012]. The approach has then been popularized by Rust [Klabnik and Nichols 2023], in particular, focusing on statically restricting aliasing and mutability [Jung et al. 2018a], which in Rust takes the form of ownership and borrowing as well as reference-counted mutexes for maximal flexibility. Recently flexible mode-based systems have been explored, as present in the work on DRFCaml [Georges et al. 2025], which exploits modes (extending Lorenzen et al. [2024]) to distinguish values that can and cannot be safely shared between threads. Other systems leverage region-based techniques to restrict concurrent threads, ensuring safe disjoint access to the heap with minimal annotations [Milano et al. 2022], or leveraging explicit annotations to limit the set of permissible effects on shared parts of the heap [Bocchino Jr. et al. 2009].

Much of these related works focus on the hazards of concurrency, especially data races, race conditions, non-determinism, and similar issues. Disentanglement (and by extension, TypeDis) focuses on an equally important but different issue, namely, the *performance* of parallel programs. TypeDis in particular is designed to allow for unrestricted sharing of immutable data (for example, as demonstrated in the data structure sharing example of Section 2.4) mixed with disentangled sharing of mutable data (for example, in Section 6). This support for data sharing is motivated by the implementation of efficient parallel algorithms, many of which rely upon access to shared memory with irregular and/or data-dependent access patterns, which are difficult to statically analyze for safety. For example, Abdi et al. [2024] find that many standard implementations of parallel algorithms are rejected by the Rust type system, yet these same implementations have previously been shown to be disentangled [Westrick et al. 2022]. We consider one such implementation as a case study in Section 6 and confirm that it is typeable under TypeDis.

8 Conclusion and Future Work

Disentanglement is an important property of parallel programs, which can in particular serve for improving performance. This paper introduces TypeDis, a static type system that proves disentanglement. TypeDis annotates types with timestamps, recording for each object the task that allocated it. Moreover, TypeDis supports iso-recursive types, as well as type and timestamp polymorphism. TypeDis allows restamping the timestamps in types using a particular form of subtyping we dub *subtiming*.

This paper focuses on *type checking*, that is, given a program annotated with types, checking if these types are valid. We are currently working on a prototype type checker, written in OCaml. An immediate direction for future work is *type inference*, that is, generating a valid type for a program. For future work, we plan to use the framework of Odersky et al. [1999], which adapts Hindley-Milner to a system with constrained universal quantification. We believe subtiming and backtiming will be inferrable. One challenging case will be mixing polymorphic recursion with par, which might require annotations in order to remain decidable (this is a known problem in region-based type systems [Tofte and Birkedal 1998]).

References

Javad Abdi, Gilead Posluns, Guozheng Zhang, Boxuan Wang, and Mark C. Jeffrey. 2024. When Is Parallelism Fearless and Zero-Cost with Rust?. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2024, Nantes, France, June 17-21, 2024*, Kunal Agrawal and Erez Petrank (Eds.). ACM, 27–40. doi:10.1145/3626183.3659966

Umut A. Acar, Jatin Arora, Matthew Fluet, Ram Raghunathan, Sam Westrick, and Rohan Yadav. 2020. MPL: A high-performance compiler for Parallel ML. https://github.com/MPLLang/mpl

Umut A. Acar, Guy E. Blelloch, Matthew Fluet, Stefan K. Muller, and Ram Raghunathan. 2015. Coupling Memory and Computation for Locality Management. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs, Vol. 32)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1–14. doi:10.4230/LIPICS.SNAPL.2015.1

Umut A. Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. 2016. Dag-calculus: a calculus for parallel computation. In *International Conference on Functional Programming (ICFP)*. 18–32. https://doi.org/10.1145/2951913.2951946

Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. 2022. The problem-based benchmark suite (PBBS), V2. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 445–447. doi:10.1145/3503221.3508422

Anon. 2025. Supplementary Material. Submitted together with this paper.

Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press. http://www.cambridge.org/9780521033114

Jatin Arora, Stefan K. Muller, and Umut A. Acar. 2024. Disentanglement with Futures, State, and Interaction. *Proc. ACM Program. Lang.* 8, POPL (2024), 1569–1599. doi:10.1145/3632895

Jatin Arora, Sam Westrick, and Umut A. Acar. 2021. Provably space-efficient parallel functional programming. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–33. doi:10.1145/3434299

Jatin Arora, Sam Westrick, and Umut A. Acar. 2023. Efficient Parallel Functional Programming with Effects. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1558–1583. doi:10.1145/3591284

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *28th European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 11423)*. Springer, 611–639. doi:10.1007/978-3-030-17184-1_22

Henk P. Barendregt. 1984. *The Lambda Calculus, Its Syntax and Semantics*. Elsevier. http://www.elsevier.com/wps/find/bookdescription.cws_home/501727/description

Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 97–116. doi:10.1145/1640089.1640097

Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. 2002. Ownership types for safe programming: preventing data races and deadlocks. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 211–230. doi:10.1145/582419.582440

Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. 2003. Ownership types for object encapsulation. In *30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 213–223. doi:10.1145/604131.604156

Chandrasekhar Boyapati and Martin C. Rinard. 2001. A Parameterized Type System for Race-Free Java Programs. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 56–69. doi:10.1145/504282.504287

Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2019. Domain-Aware Session Types. In *30th International Conference on Concurrency Theory (CONCUR) (LIPIcs, Vol. 140)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik,

39:1–39:17. doi:10.4230/LIPICS.CONCUR.2019.39

David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 48–64. doi:10.1145/286936.286947

Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, Robert Harper, Douglas J. Howe, Todd B. Knoblock, Nax Paul Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. 1986. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall. http://dl.acm.org/citation.cfm?id=10510

Farzaneh Derakhshan, Stephanie Balzer, and Limin Jia. 2021. Session Logical Relations for Noninterference. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 1–14. doi:10.1109/LICS52264.2021.9470654

Farzaneh Derakhshan, Stephanie Balzer, and Yue Yao. 2024. Regrading Policies for Flexible Information Flow Control in Session-Typed Concurrency. In *38th European Conference on Object-Oriented Programming (ECOOP) (LIPIcs, Vol. 313)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:29. doi:10.4230/LIPICS.ECOOP.2024.11

Werner Dietl and Peter Müller. 2005. Universes: Lightweight Ownership for JML. *Journal of Object Technology* 4, 8 (2005), 5–32. doi:10.5381/JOT.2005.4.8.A1

Martin Elsman and Troels Henriksen. 2023. Parallelism in a Region Inference Context. *Proc. ACM Program. Lang.* 7, PLDI (2023), 884–906. doi:10.1145/3591256

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 121–133. doi:10.1145/1542476.1542490

Aïna Linn Georges, Benjamin Peters, Laila Elbeheiry, Leo White, Stephen Dolan, Richard A. Eisenberg, Chris Casinghino, François Pottier, and Derek Dreyer. 2025. Data Race Freedom à la Mode. *Proc. ACM Program. Lang.* 9, POPL (2025), 656–686. doi:10.1145/3704859

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'État. Université Paris 7. https://girard.perso.math.cnrs.fr/These.pdf

Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. 2021. Mechanized logical relations for termination-insensitive noninterference. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. doi:10.1145/3434291

Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 282–293. doi:10.1145/512529.512563

Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut A. Acar, and Matthew Fluet. 2018. Hierarchical memory management for mutable state. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, Andreas Krall and Thomas R. Gross (Eds.). ACM, 81–93. doi:10.1145/3178487.3178494

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 66:1–66:34. https://people.mpi-sws.org/~dreyer/papers/rustbelt/paper.pdf

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf

Steve Klabnik and Carol Nichols. 2023. *The Rust programming language*. No Starch Press.

Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA.

Peter J. Landin. 1964. The Mechanical Evaluation of Expressions. *Computer Journal* 6, 4 (Jan. 1964), 308–320.

Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. *Proc. ACM Program. Lang.* 8, ICFP (2024), 485–514. doi:10.1145/3674642

Per Martin-Löf. 1982. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science VI*. Studies in Logic and the Foundations of Mathematics, Vol. 104. Elsevier, 153–175. doi:10.1016/S0049-237X(09)70189-2

Mae Milano, Joshua Turcotti, and Andrew C. Myers. 2022. A flexible type system for fearless concurrency. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 458–473. doi:10.1145/3519939.3523443

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (Dec. 1978), 348–375. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.67.5276

Alexandre Moine, Sam Westrick, and Stephanie Balzer. 2024. DisLog: A Separation Logic for Disentanglement. *Proc. ACM Program. Lang.* 8, POPL, Article 11 (Jan. 2024), 30 pages. doi:10.1145/3632853

Peter Müller. 2002. *Modular Specification and Verification of Object-Oriented Programs*. Lecture Notes in Computer Science, Vol. 2262. Springer. doi:10.1007/3-540-45651-1

Stefan K. Muller, Umut A. Acar, and Robert Harper. 2017. Responsive parallel computation: bridging competitive and cooperative threading. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 677–692. doi:10.1145/3062341.3062370

James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *12th European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science, Vol. 1445)*. Springer, 158–185. doi:10.1007/BFB0054091

Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *Theory and Practice of Object Systems* 5, 1 (1999), 35–55. https://doi.org/10.1002/(SICI)1096-9942(199901/03)5:1%3C35::AID-TAPO4%3E3.0.CO;2-4

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

Andrew M. Pitts and Ian Stark. 1998. Operational Reasoning for Functions with Local State. *Higher Order Operational Techniques in Semantics (HOOTS)* (1998), 227–273.

Gordon D. Plotkin. 1973. *Lambda-definability and logical relations*. Technical Report. University of Edinburgh.

François Pottier and Vincent Simonet. 2003. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25, 1 (2003), 117–158. doi:10.1145/596980.596983

Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy E. Blelloch. 2016. Hierarchical memory management for parallel programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 392–406. doi:10.1145/2951913.2951935

Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE J. Sel. Areas Commun.* 21, 1 (2003), 5–19.

Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: the problem based benchmark suite. In *24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, Pittsburgh, PA, USA, June 25-27, 2012*, Guy E. Blelloch and Maurice Herlihy (Eds.). ACM, 68–70. doi:10.1145/2312005.2312018

Vincent Simonet. 2003. Flow Caml in a Nutshell. In *1st APPSEM-II Workshop*, Graham Hutton (Ed.).

Geoffrey Smith and Dennis M. Volpano. 1998. Secure Information Flow in a Multi-Threaded Imperative Language. In *POPL*. ACM, 355–364.

Richard Statman. 1985. Logical Relations and the Typed λ-calculus. *Information and Control* 65, 2/3 (1985), 85–97. doi:10.1016/S0019-9958(85)80001-2

William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *The Journal of Symbolic Logic* 32, 2 (1967), 198–212. http://www.jstor.org/stable/2271658

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, 6, Article 40 (Nov. 2024), 75 pages. doi:10.1145/3676954

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

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher-Order and Symbolic Computation* 17, 3 (Sept. 2004), 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and Computation* 132, 2 (1997), 109–176. http://www.irisa.fr/prive/talpin/papers/ic97.pdf

VerifyThis. 2022. Challenge 3 - The World's Simplest Lock-Free Hash Set. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Verify%20This/Challenges2022/verifyThis2022-challenge3.pdf

Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue. In *Certified Programs and Proofs (CPP)*. 76–90. https://cs.au.dk/~birke/papers/2021-ms-queue-final.pdf

Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2/3 (1996), 167–188.

Philip Wadler. 1990. Linear Types Can Change the World!. In *IFIP Working Group 2.2, 2.3 on Programming Concepts and Methods*. North-Holland, 561.

Philip Wadler. 2012. Propositions as Sessions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 273–286. doi:10.1145/2364527.2364568

Sam Westrick. 2022. *Efficient and Scalable Parallel Functional Programming through Disentanglement*. Ph. D. Dissertation. Department of Computer Science, Carnegie Mellon University.

Sam Westrick, Jatin Arora, and Umut A. Acar. 2022. Entanglement Detection with Near-Zero Cost. *Proc. ACM Program. Lang.* 6, ICFP, Article 115 (aug 2022), 32 pages. doi:10.1145/3547646

Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2020. Disentanglement in Nested-Parallel Programs. *Proc. ACM Program. Lang.* 4, POPL, Article 47 (jan 2020), 32 pages. doi:10.1145/3371115

Andrew K. Wright. 1995. Simple Imperative Polymorphism. *Lisp and Symbolic Computation* 8, 4 (Dec. 1995), 343–356. http://www.cs.rice.edu/CS/PLT/Publications/Scheme/lasc95-w.ps.gz

A Appendix

A.1 The Kinding Judgment

$$\textsc{K-Var}\ \frac{x :: \kappa \in \Gamma}{\Gamma \vdash x :: \kappa} \qquad \textsc{K-Unboxed}\ \Gamma \vdash \tau :: \star \qquad \textsc{K-At}\ \frac{\Gamma \vdash \sigma :: \star}{\Gamma \vdash \sigma@\delta :: \star} \qquad \textsc{K-Lam}\ \frac{\Gamma \vdash \rho :: \kappa}{\Gamma \vdash \lambda\delta.\, \rho :: \mathbb{X} \Rightarrow \kappa} \qquad \textsc{K-App}\ \frac{\Gamma \vdash \rho :: \mathbb{X} \Rightarrow \kappa}{\Gamma \vdash \rho\, \delta :: \kappa}$$

$$\textsc{K-TAbs}\ \frac{\alpha :: \kappa, \Gamma \vdash \rho :: \star}{\Gamma \vdash \forall \alpha :: \kappa.\ \rho :: \star} \qquad \textsc{K-Rec}\ \frac{\alpha :: \star, \Gamma \vdash \sigma :: \star}{\Gamma \vdash \mu\alpha.\, \sigma@\delta :: \star} \qquad \textsc{K-Array}\ \frac{\Gamma \vdash \rho :: \star}{\Gamma \vdash \mathsf{array}(\rho) :: \star} \qquad \textsc{K-Pair}\ \frac{\Gamma \vdash \rho_1 :: \star \qquad \Gamma \vdash \rho_2 :: \star}{\Gamma \vdash (\rho_1 \times \rho_2) :: \star}$$

$$\textsc{K-Sum}\ \frac{\Gamma \vdash \rho_1 :: \star \qquad \Gamma \vdash \rho_2 :: \star}{\Gamma \vdash (\rho_1 + \rho_2) :: \star} \qquad \textsc{K-Arrow}\ \frac{(\forall \rho.\ \rho \in \vec{\rho_1} \implies \Gamma \vdash \rho :: \star) \qquad \Gamma \vdash \rho_2 :: \star}{\Gamma \vdash \forall \vec{\delta_1}\, \Delta.\ \vec{\rho_1} \to^{\delta_2} \rho_2 :: \star}$$

Fig. 16. Kinding judgment

Figure 16 presents the kinding judgment $\Gamma \vdash \rho :: \kappa$, asserting that type ρ has kind κ considering the environment Γ.

A.2 The veryPure Predicate

$$\textsc{VP-Val}\ \mathsf{veryPure}\ v \qquad \textsc{VP-Abs}\ \mathsf{veryPure}\ (\mu f.\, \Lambda \vec{\delta_1}\, \Delta_1.\, \lambda \vec{x}!\delta_f.\, e) \qquad \textsc{VP-Var}\ \mathsf{veryPure}\ x \qquad \textsc{VP-Prim}\ \frac{\mathsf{veryPure}\ e_1 \qquad \mathsf{veryPure}\ e_2}{e_1 \bowtie e_2}$$

$$\textsc{VP-Let}\ \frac{\mathsf{veryPure}\ e_1 \qquad \mathsf{veryPure}\ e_2}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2} \qquad \textsc{VP-Pair}\ \frac{\mathsf{veryPure}\ e_1 \qquad \mathsf{veryPure}\ e_2}{(e_1, e_2)} \qquad \textsc{VP-Fold}\ \frac{\mathsf{veryPure}\ e}{\mathsf{veryPure}\ \mathsf{fold}\ e} \qquad \textsc{VP-Fold}\ \frac{\mathsf{veryPure}\ e}{\mathsf{veryPure}\ \mathsf{unfold}\ e}$$

$$\textsc{VP-If}\ \frac{\mathsf{veryPure}\ e_1 \qquad \mathsf{veryPure}\ e_2 \qquad \mathsf{veryPure}\ e_3}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3}$$

Fig. 17. The veryPure predicate

Figure 17 presents the veryPure predicate over an expression. This predicate ensures that the expression does not contain any array allocation, load, store, par, projection, case, or function call.

A.3 Reachability Predicates

$$\textsc{R-Refl}\ \Delta \vdash \delta \preccurlyeq \delta \qquad \textsc{R-Cons}\ \frac{(\delta_1, \delta_2) \in \Delta \qquad \Delta \vdash \delta_2 \preccurlyeq \delta_3}{\Delta \vdash \delta_1 \preccurlyeq \delta_3} \qquad \textsc{R-Logical}\ \frac{\forall \delta_1\, \delta_2.\ (\delta_1, \delta_2) \in \Delta' \implies \Delta \vdash \delta_1 \preccurlyeq \delta_2}{\Delta \vdash \Delta'}$$

Fig. 18. The reachability predicates

Figure 18 presents the reachability predicates that appear in T-App and in Figure 12.

R-Refl asserts that a timestamp can always reach itself. R-Cons asserts that if there is an edge between δ_1 and δ_2 and if δ_2 can reach δ_3, then δ_1 can reach δ_3.

R-Logical asserts that a logical graph Δ subsumes a logical graph Δ' if every edge between δ_1 and δ_2 in Δ' can be simulated in Δ.

A.4 The "Valid Variable" Judgment

$$\textsc{VA-Var}\ \frac{\alpha = \alpha' \implies \Delta \vdash \delta_1 \preccurlyeq \delta_2}{\Delta \mid \alpha \mapsto \delta_1 \vdash_{\delta_2} \alpha'} \qquad \textsc{VA-Base}\ \Delta \mid \alpha \mapsto \delta_1 \vdash_{\delta_2} \tau \qquad \textsc{VA-At}\ \frac{\Delta \mid \alpha \mapsto \delta_1 \vdash_{\delta} \sigma}{\Delta \mid \alpha \mapsto \delta_1 \vdash_{\delta_2} \sigma@\delta} \qquad \textsc{VA-TAbs}\ \frac{\alpha \neq \alpha' \implies \Delta \mid \alpha \mapsto \delta_1 \vdash_{\delta_2} \rho}{\Delta \mid \alpha \mapsto \delta_1 \vdash_{\delta_2} \forall \alpha' :: \kappa.\ \rho}$$

$$\textsc{VA-Pair}\ \frac{\Delta \mid \alpha \mapsto \delta_1 \vdash_{\delta_2} \rho_1 \qquad \Delta \mid \alpha \mapsto \delta_1 \vdash_{\delta_2} \rho_2}{\Delta \mid \alpha \mapsto \delta_1 \vdash_{\delta_2} (\rho_1 \times \rho_2)} \qquad \textsc{VA-Sum}\ \frac{\Delta \mid \alpha \mapsto \delta_1 \vdash_{\delta_2} \rho_1 \qquad \Delta \mid \alpha \mapsto \delta_1 \vdash_{\delta_2} \rho_2}{\Delta \mid \alpha \mapsto \delta_1 \vdash_{\delta_2} (\rho_1 + \rho_2)}$$

$$\textsc{VA-TRec}\ \frac{\alpha \notin \mathsf{fv}(\rho) \setminus \{\alpha'\}}{\Delta \mid \alpha \mapsto \delta_1 \vdash_{\delta_2} \mu\alpha'.\, \sigma@\delta} \qquad \textsc{VA-Array}\ \frac{\alpha \notin \mathsf{fv}(\rho)}{\Delta \mid \alpha \mapsto \delta_1 \vdash_{\delta_2} \mathsf{array}(\rho)} \qquad \textsc{VA-Abs}\ \frac{\alpha \notin \mathsf{fv}(\vec{\rho'}) \cup \mathsf{fv}(\rho'')}{\Delta \mid \alpha \mapsto \delta_1 \vdash_{\delta_2} \forall \vec{\delta'}\ \Delta'.\ \vec{\rho'} \to^{\delta_f} \rho''}$$

Fig. 19. The "valid variable" judgment

Figure 19 presents the "valid variable" judgment that is used for subtiming recursive types (S-Rec).

A.5 The Root Functions and Assertions

$$\begin{aligned}
\mathsf{answer} &\triangleq \mathsf{Timestamp}\ \delta \mid \mathsf{Unboxed} \mid \mathsf{Nonsense} \\
\mathsf{rootf}\ h\ m\ \kappa\ \rho &: \mathsf{fkind}\ \kappa\ \mathsf{answer} \\
\mathsf{rootf}\ h\ m\ \kappa\ \alpha &\triangleq \mathsf{if}\ m(\alpha) = (\kappa, (_, r))\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{Nonsense}_\kappa \\
\mathsf{rootf}\ h\ m\ \star\ \tau &\triangleq \mathsf{Unboxed} \\
\mathsf{rootf}\ h\ m\ \star\ (\sigma@\delta) &\triangleq \mathsf{Timestamp}\ \delta \\
\mathsf{rootf}\ h\ m\ (\mathbb{X} \implies \kappa)\ (\lambda\delta.\, \rho) &\triangleq \lambda t.\, \mathsf{rootf}\ ([\delta := t]h)\ m\ \kappa\ \rho \\
\mathsf{rootf}\ h\ m\ \kappa\ (\rho\ \delta) &\triangleq (\mathsf{rootf}\ h\ m\ (\mathbb{X} \implies \kappa)\ \rho)\ (h(\delta)) \\
\mathsf{rootf}\ h\ m\ \star\ (\forall\alpha :: \kappa.\ \rho) &\triangleq \mathsf{rootf}\ h\ ([\alpha := \mathsf{Nonsense}_\kappa]m)\ \star\ \rho \\
\mathsf{rootf}\ h\ m\ \star\ (\mu\alpha.\, \sigma@\delta) &\triangleq \mathsf{Timestamp}\ \delta \\
\\
\mathsf{root}\ \rho\ \preccurlyeq^h_m\ \delta &\triangleq \mathsf{match}\ (\mathsf{rootf}\ h\ m\ \star\ \rho)\ \mathsf{with} \\
&\quad \mid \mathsf{Unboxed} \quad \Rightarrow \ulcorner True \urcorner \\
&\quad \mid \mathsf{Nonsense} \quad \Rightarrow \ulcorner False \urcorner \\
&\quad \mid \mathsf{Timestamp}\ \delta' \Rightarrow h(\delta') \preccurlyeq h(\delta)
\end{aligned}$$

Fig. 20. Root-related functions and assertions

Figure 20 presents the rootf function, expecting a timestamp mapping h, a type mapping m, a kind κ and a type ρ, and produces a function expecting κ timestamps and returning an "answer", representing the root timestamp of ρ. An answer is either a timestamp, Unboxed to indicate an unboxed type, of Nonsense if the type has no sensible root timestamp (for example, $\forall\alpha :: \kappa.\ \alpha$). In this definition, we write $\mathsf{Nonsense}_\kappa$ the function expecting κ arguments and returning Nonsense.

The assertion $\mathsf{root}\ \rho\ \preccurlyeq^h_m\ \delta$, also presented in Figure 20 matches root timestamp of ρ. If it is unboxed, the assertion is true, if it is nonsensical, the assertion is false, and if it is a regular timestamp δ', then $h(\delta')$ must precede $h(\delta)$.

A.6 Definition of our Logical Relations

Figure 21 presents the logical relations we define. Formally, we define the (meta-type-level) function $\mathsf{fkind}\ \kappa\ A$ producing a function waiting for κ timestamps and returning something of type A. This function is defined by induction over the kind κ.

$$
\begin{aligned}
\mathsf{fkind}\ \star\ a &\triangleq A \\
\mathsf{fkind}\ (\mathbb{X} \Rightarrow \kappa)\ A &\triangleq \mathcal{T} \to \mathsf{fkind}\ \kappa\ A
\end{aligned}
$$

$$
\begin{aligned}
[\![\rho]\!]^h_m\ \kappa &: \mathsf{fkind}\ \kappa\ (\mathcal{V} \to \mathit{iProp}) \\
[\![\alpha]\!]^h_m\ \kappa &\triangleq \forall (\Psi : \mathsf{fkind}\ \kappa\ (\mathcal{V} \to \mathit{iProp}))\ (r : \mathsf{fkind}\ \kappa\ \mathcal{T}). \\
&\quad \mathsf{if}\ m(\alpha) = (\kappa, (\Psi, r))\ \mathsf{then}\ \Psi\ *_\kappa\ \circledcirc_\kappa\ r\ \mathsf{else}\ \bot_\kappa \\
[\![\tau]\!]^h_m\ \star &\triangleq \lambda v.\ \ulcorner \tau = () \land v = () \urcorner\ \lor\ \ulcorner \tau = \mathsf{bool} \land v \in \{\mathsf{true}, \mathsf{false}\} \urcorner\ \lor\ \ulcorner \tau = \mathsf{int} \land v \in \mathbb{Z} \urcorner \\
[\![\sigma@\delta]\!]^h_m\ \star &\triangleq \lambda v.\ v \circledcirc h(\delta)\ *\ [\![\sigma]\!]^h_m\ \delta\ v \\
[\![\mu\alpha.\, \sigma@\delta]\!]^h_m\ \star &\triangleq \mu(\Psi : \mathcal{V} \to \mathit{iProp}). \\
&\quad \lambda v.\ \exists w.\ \ulcorner v = \mathsf{vfold}\ w \urcorner\ *\ w \circledcirc h(\delta)\ *\ \triangleright\ [\![\sigma]\!]^h_{[\alpha := (\star, (\Psi, h(\delta)))]m}\ \delta\ v \\
[\![\forall \alpha :: \kappa.\ \rho]\!]^h_m\ \star &\triangleq \lambda v.\ \Box \forall (\Psi : \mathsf{fkind}\ \kappa\ (\mathcal{V} \to \mathit{iProp}))\ (r : \mathsf{fkind}\ \kappa\ \mathcal{T}). \\
&\quad \ulcorner \mathsf{proper}_\kappa\ \Psi \land \mathsf{regular}_\kappa\ r \urcorner \mathrel{-\!\!*} [\![\rho]\!]^h_{[\alpha := (\kappa, (\Psi, r))]m}\ \star\ v \\
[\![\lambda\delta.\ \rho]\!]^h_m\ (\mathbb{X} \Rightarrow \kappa) &\triangleq \lambda t.\ [\![\rho]\!]^{[\delta := t]h}_m\ \kappa \\
[\![\rho\ \delta]\!]^h_m\ \kappa &\triangleq [\![\rho]\!]^h_m\ (\mathbb{X} \Rightarrow \kappa)\ h(\delta)
\end{aligned}
$$

$$
(\!|\rho|\!)^h_m \triangleq \lambda \delta\, v.\ \mathsf{root}\ \rho \preccurlyeq^h_m \delta\ *\ [\![\rho]\!]^h_m\ \star\ v
$$

$$
\begin{aligned}
[\![\sigma]\!]^h_m &: \mathcal{T} \to \mathcal{V} \to \mathit{iProp} \\
[\![\mathsf{array}(\rho)]\!]^h_m &\triangleq \lambda \delta\, v.\ \exists \ell.\ \ulcorner v = \ell \urcorner\ * \boxed{\exists \vec{w}.\ \ell \mapsto \vec{w}\ *\ \textstyle\ast_{v' \in \vec{w}}\ (\!|\rho|\!)^h_m\ \delta\ v'} \\
[\![(\rho_1 \times \rho_2)]\!]^h_m &\triangleq \lambda \delta\, v.\ \exists \ell\, v_1\, v_2.\ \ulcorner v = \ell \urcorner\ *\ \ell \mapsto_\square (v_1, v_2)\ *\ (\!|\rho_1|\!)^h_m\ \delta\ v_1\ *\ (\!|\rho_2|\!)^h_m\ \delta\ v_2 \\
[\![(\rho_1 + \rho_2)]\!]^h_m &\triangleq \lambda \delta\, v.\ \exists \ell\, v'.\ \ulcorner v = \ell \urcorner\ * \\
&\quad (\ell \mapsto_\square \mathsf{inj}_1\ v'\ *\ (\!|\rho_1|\!)^h_m\ \delta\ v')\ \lor\ (\ell \mapsto_\square \mathsf{inj}_2\ v'\ *\ (\!|\rho_2|\!)^h_m\ \delta\ v') \\
[\![\forall \vec{\delta_1}\ \Delta_1.\ \vec{\rho_1} \to^{\delta_f} \rho_2]\!]^h_m &\triangleq \lambda \delta\, v.\ \Box \forall \vec{t}\ \vec{w}.\ \ulcorner |\vec{\delta_1}| = |\vec{t}|\ \land\ |\vec{\rho_1}| = |\vec{w}| \urcorner \mathrel{-\!\!*} \\
&\quad \mathsf{let}\ h' = [\vec{\delta_1} := \vec{t}]h\ \mathsf{in} \\
&\quad h(\delta) \preccurlyeq h'(\delta_f) \mathrel{-\!\!*} [\![\Delta_1]\!]^{h'} \mathrel{-\!\!*} \textstyle\ast_{\rho \in \vec{\rho_1}, v' \in \vec{w}}\ (\!|\rho|\!)^{h'}_m\ v') \mathrel{-\!\!*} \\
&\quad \forall t'.\ t' \approx h'(\delta_f) \mathrel{-\!\!*} \mathsf{wp}\ \langle t',\ v\ \vec{w} \rangle\ \{\lambda\, v'.\ (\!|\rho_2|\!)^{h'}_m\ v'\}
\end{aligned}
$$

Fig. 21. The interpretation of types

More precisely, Figure 21 presents the type relation and the allocated type relation. The *type relation* $[\![\rho]\!]^h_m\ \kappa$ produces a function waiting for κ timestamps, a value v, and captures that v is of type ρ, within the timestamp mapping h and type mapping m. The *allocated type relation* $[\![\sigma]\!]^h_m$ produces a predicate over a timestamp δ and a value v, capturing that v is of type σ allocated at δ, within the timestamp mapping h and type mapping m. The type relation and the allocated type relation are defined by mutual induction over their type argument. The omitted cases are all sent to $\bot_\kappa$, the always false predicate ignoring its κ arguments.

Interpretation of types. Let us first present the relation $[\![\rho]\!]^h_m\ \kappa$.

If ρ is a variable α, then α must have kind κ in the type mapping m, linked with predicate Ψ and timestamp function r. The relation returns the predicate $\Psi\ *_\kappa\ \circledcirc_\kappa\ r$. The operator $*_\kappa$ lifts the separating conjunction to predicated in $\mathsf{fkind}\ \kappa\ (\mathcal{V} \to \mathit{iProp})$ by distributing κ timestamp arguments and a value to Ψ and $\circledcirc_\kappa\ r$. The predicate $\circledcirc_\kappa\ r$ is of type $\mathsf{fkind}\ \kappa\ (\mathcal{V} \to \mathit{iProp})$; it feeds κ timestamps to r, and asserts that the value argument was allocated before the result of r. For example, in the particular case of $\kappa = \mathbb{X} \Rightarrow \star$, we have that $\Psi\ *_\kappa\ \circledcirc_\kappa\ r\ =\ \lambda \delta\, v.\ \Psi\ \delta\ v\ *\ v \circledcirc (r\ \delta)$.

If ρ is a base type τ, the kind must be the base kind $\star$, and the relation binds a value which must correspond to the particular base type under consideration.

If ρ is an allocated type $\sigma@\rho$, the kind must be $\star$, and the relation binds a value v which must have been allocated before $h(\delta)$, the timestamp associated to δ in h. The relation also ensures that v recursively satisfies the interpretation of σ.

If ρ is a recursive type $\mu\alpha.\, \sigma@\delta$, the kind must be $\star$, and the relation is expressed as a *guarded fixed-point*. Intuitively, the predicate Ψ, from a value to *iProp*, captures the interpretation of the recursive type itself. The interpretation binds a value v and asserts that it is of the form vfold w. The interpretation is then similar to an allocated type: w must have been allocated before $h(\delta)$ and be in relation with the interpretation of σ, with a type environment updated to bind α to Ψ as well as the root timestamp $h(\delta)$.

If ρ is a type abstraction $\forall\alpha :: \kappa.\ \rho$, the kind must be $\star$, and the relation binds a value v. The relation universally quantifies over the predicate Ψ and the timestamp function r, which will be instantiated during the T-TApp rule. Both Ψ and r are constrained. The property $\text{proper}_\kappa\ \Psi$ captures that any timestamp parameter of Ψ can be replaced by an equivalent one. The property $\text{regular}_\kappa\ r$ captures that the function r either ignores all its arguments or returns one of them. These two properties are needed in order to prove that T-Par is sound. The relation then calls itself recursively on ρ, augmenting the type mapping by associating α to its kind κ and the pair of Ψ and r.

If ρ is a timestamp abstraction $\lambda\delta.\, \rho$, the kind must be of the form $\mathbb{x} \Rightarrow \kappa$, and the relation expands to a function waiting for a timestamp δ and adding it to the timestamp mapping h.

If ρ is a timestamp application $\rho\ \delta$ at some kind κ, then the relation applies the timestamp $h(\delta)$ to the interpretation of ρ at kind $\mathbb{x} \Rightarrow \kappa$.

Interpretation of allocated types. The *enriched type interpretation* $(\!|\rho|\!)^h_m$, defined next in Figure 21, is a predicate over a timestamp δ and a value v. It asserts that the root timestamp of ρ comes before δ and that v is in relation with the interpretation of ρ. This wrapper in used for the interpretation of allocated types, which we present next. The interpretation of allocated types is written $[\![\sigma]\!]^h_m$ and is a predicate over a timestamp variable δ and a value v.

If σ is an array $\text{array}(\rho)$, then v must be a location ℓ, such that ℓ points-to an array $\vec{w}$ and that for each value v' in $\vec{w}$ is in relation with the enriched interpretation of ρ. The points-to assertion and the relation on the values of the array appears inside an invariant, ensuring their persistence.

If σ is a pair $(\rho_1 \times \rho_2)$, then v must be a location ℓ pointing to a pair of values (v_1, v_2) such that v_1 (resp. v_2) is in relation with the enriched interpretation of ρ_1 (resp. ρ_2). The sum case is similar.

If σ is an arrow $\forall\vec{\delta}_1\ \Delta_1.\ \vec{\rho}_1 \to^{\delta_f} \rho_2$, then the interpretation quantifies over the list of timestamp arguments $\vec{t}$ and the list of arguments of the function $\vec{w}$, which must both have the correct length. The relation then defines h', the new timestamp environment, being h where $\vec{\delta}_1$ are instantiated with $\vec{t}$. The relation next requires that the allocation timestamp $h(\delta)$ precedes the timestamp of the caller $h'(\delta_f)$, and that every value in $\vec{w}$ is of the correct type. Last, the relation requires that for any timestamp equivalent to $h'(\delta_f)$, the WP of the function call holds, and that the returned value is in relation with the enriched interpretation of the return type ρ_2.