Linear Promises: Towards Safer Concurrent Programming

Ohad Rau
Georgia Institute of Technology, Atlanta, GA, USA

Caleb Voss
Georgia Institute of Technology, Atlanta, GA, USA

Vivek Sarkar
Georgia Institute of Technology, Atlanta, GA, USA

Abstract

In this paper, we introduce a new type system based on linear typing, and show how it can be incorporated in a concurrent programming language to track ownership of promises. By tracking write operations on each promise, the language is able to guarantee exactly one write operation is ever performed on any given promise. This language thus precludes a number of common bugs found in promise-based programs, such as failing to write to a promise and writing to the same promise multiple times. We also present an implementation of the language, complete with an efficient type checking algorithm and high-level programming constructs. This language serves as a safer platform for writing high-level concurrent code.

2012 ACM Subject Classification Software and its engineering → Concurrent programming languages; Theory of computation → Operational semantics; Theory of computation → Type theory

Keywords and phrases promises, type systems, linear typing, operational semantics, concurrency

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.13

Supplementary Material *Software (Source Code)*: `https://github.com/OhadRau/LinearPromises` archived at `swh:1:dir:311764ac58400c3720161e108bb3611fcab4c2d9`
Software (ECOOP 2021 Artifact Evaluation approved artifact): `https://doi.org/10.4230/DARTS.7.2.15`

1 Introduction

In recent decades, the prevalence of concurrent programming[1] has increased as programmers strive to take advantage of increasingly parallel machines. Unstructured concurrency has proven to be highly error-prone, and programmers have subsequently undertaken efforts to formalize concurrent programming using structured techniques. One such technique is the *promise* – a container used to refer to a value that is produced asynchronously [14]. Promises typically include both a "read" (or "await") operation – which blocks until a value is available and returns that value – as well as a "write" (or "fulfill") operation – which provides the value for the promise. Promises are frequently used to communicate between threads or tasks, where one thread awaits a value and another performs some asynchronous computation before writing the value to the promise. This creates a higher-level abstraction as compared to more traditional concurrency primitives like the lock/mutex (which allows programs to ensure mutually-exclusive access to shared resources) and a more general abstraction than the future (which represents a placeholder for the result of a function being evaluated by a predetermined task or thread). In recent years, promises have been incorporated into mainstream languages such as C++, JavaScript, and Java as popular concurrency primitives.

[1] In this paper, we use "concurrent" interchangeably with "parallel", though we recognize that a distinction is made between the two terms in other contexts.

Despite the value of structured concurrency in preventing bugs, these techniques – including the promise – are not without their own share of bugs. Madsen et al. [15] identify several common promise-related concurrency bugs, including the *omitted write* – a promise that is never fulfilled – and the *double write* – a promise that is fulfilled multiple times. In their case study, these two bugs are the direct cause of 6 of the 21 concurrency bugs analyzed.

In addition to the omitted write and double write bugs, there are other kinds of concurrency errors that can occur with the use of promises. A *deadlock* is composed of a cyclical dependency among a set of threads [11]. Further, an *unowned write* occurs when one thread assumes ownership over a promise only to have another thread perform the actual write operation. This is distinct from the double write, as it can make it difficult for the programmer to reason about the source of a promise's fulfillment and often results in a race to *fulfill* the promise between two threads. This comes in contrast to a double write to an owned promise, which would only exhibit a race when reading[2].

Each of these bugs has the potential to create critical issues in a program, ranging from making the program inoperable to creating major security flaws. For example, both omitted writes and deadlocks can cause a program to hang indefinitely, while unowned or double writes may contribute to data races and, by extension, inconsistent, unpredictable, or even undefined behavior. In fact, these bugs have been cited [9, 10] as a disadvantage of promises relative to futures, which by definition cannot result in unowned, double, or omitted writes[3].

Over the years, programmers have devised many tools that can be used to quickly detect bugs in programs. In general, these fall into two major categories: dynamic analyses, which attempt to identify bugs during a program's execution, and static analyses, which attempt to identify bugs before execution [8]. Each method has its own advantages, with dynamic analyses generally producing fewer false positives as well as more detailed results (due to the availability of more information about the program) and static analyses revealing possible bugs before the code is run [7]. Beyond that, static analyses present the advantage of proving the *absence* of behaviors (i.e. that a program can *never* exhibit a certain bug). This allows for bugs to be found that would only appear intermittently or rarely in the running program; in contrast, such bugs can be very challenging to find using dynamic analyses.

One particularly useful class of static analysis takes the form of type checking. *Substructural* type systems are a class of type systems in which restrictions are placed on the number of times a variable may be used. One such system is known as *linear* typing, in which linear variables must be used *exactly once* [23]. As an example, with a linear variable x both the program $x + x$ and 0 cannot be typed, as neither uses the variable x exactly once. In recent years, substructural type systems have risen in popularity through their use in languages such as Rust [24], Haskell [2], ATS [26], and F* [21].

The contributions of this paper are as follows:

1. We present the design and implementation of a new concurrent programming language aimed at preventing certain classes of promise-related bugs.
2. We introduce a linear type system that tracks ownership of promises.
3. We provide a formal semantics for the behavior of concurrent, promise-based programs.
4. We prove that it is *impossible* to create double writes and unowned writes in this language, as well as omitted writes in terminating programs.

[2] The write permission of an owned promise is only owned by one thread at a time, so no two writers can race to fulfill the same *owned* promise. Both situations can result in a race when *reading*, because the value read depends on the non-deterministic order of read and write operations.

[3] Recall that a future can only be fulfilled as the result of a function. Given a function f known to return exactly once, a future yielding f's result will be fulfilled exactly once (and only by f).

5. We provide a decidable type checking algorithm, and experimentally evaluate its speed.
6. We demonstrate the abilities of this language to find bugs in real-world programs.

2 Language Formalisms

Our work revolves around a custom programming language intended to create safer type-level primitives for interacting with promises. However, we believe that this approach can be incorporated into other strongly-typed, concurrent languages with promises. Voss and Sarkar [22] introduce the concept of promise ownership as a key mechanism for dynamically identifying omitted write bugs and deadlocks. Specifically, ownership of a promise p by a task T is defined as the responsibility of T to either fulfill p or transfer ownership to another task. Based on their approach, we leverage a linear type system to statically track promise ownership. By using the type system, we can not only *statically* prevent omitted writes in terminating programs, but also unowned and double writes. As in their work, we can also leverage the language's semantics to *dynamically* detect deadlocks.

This notion of promise ownership is established by splitting promises into two components, a "read-end" and a "write-end," similar to the split between "futures" and "promises" in C++, respectively [12]. In this system, the write-end of a promise is linearly typed so as to ensure it is fulfilled exactly once. Compound data types, such as products and sums, are linear if they contain one or more linear components. All other variables remain unrestricted. By applying linear typing in this way, we enable liberal use of the read-end, while restricting the write-end to only a single use by its owner. We formally define the programming language in three parts (syntax, type system, and semantics), taking careful consideration to create as simple a language as possible.

2.1 Syntax

The language consists of multi-procedure programs, which are able to spawn asynchronous tasks. Values are integers, promise read and write handles, sums, products, and named functions. Control flow is expressed by matching on sums and products, while looping is expressed via recursion. Functions take a single parameter and return a single value. Types take the form of the base Int type for integers, the promise read handle type $\mathsf{Promise}(\tau)$, the promise write handle type $\mathsf{Promise}^*(\tau)$, binary sum and product types[4], and function types.

Figure 1 defines the syntax, and features several interesting constructs:

- $\mathsf{async}\ e$ – Schedules the expression e to be performed asynchronously.
- $\mathsf{promise}\ \tau$ – Constructs a promise of a τ, yielding a pair of the write and read handles.
- $?x$ – Blocks on and retrieves the value contained by a promise x (e.g. "await").
- $x \leftarrow e$ – Writes the value of e to the owned promise x (resolving/fulfilling the promise).

2.2 Type System

Before specifying the typing rules, we must first introduce the notion of a linear type [23]. Variables of linear types have the property that they must be used exactly once along any path of execution. For example, if a linear variable x is in scope, then the expression 1 cannot be typed, since it does not use the linear variable x. Likewise, the expression $x + x$

[4] Unbounded data was left out of the language definition for simplicity. The language can be trivially extended with recursive data types (Section 4.1) to allow for dynamically-sized data types. Such a modification does not invalidate any of the language's properties.

$$
\begin{aligned}
\tau \in \mathsf{Type} &::= \mathsf{Int} \mid \mathsf{Promise}(\tau) \mid \mathsf{Promise}^*(\tau) \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \\
e \in \mathsf{Expr} &::= i \mid x \mid \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2 \mid f(e) \mid \mathsf{async}\ e \mid (e_1, e_2) \\
&\mid \mathsf{Inl}_{\tau_L,\tau_R}\ e \mid \mathsf{Inr}_{\tau_L,\tau_R}\ e \\
&\mid \mathsf{match}\ e_0\ \{\ (x_1, x_2) \Rightarrow e_1\ \} \\
&\mid \mathsf{match}\ e_0\ \{\ \mathsf{Inl}\ x_1 \Rightarrow e_1\ ,\ \mathsf{Inr}\ x_2 \Rightarrow e_2\ \} \\
&\mid \mathsf{promise}\ \tau \mid ?x \mid x \leftarrow \mathsf{e} \\
F \in \mathsf{Function} &::= \mathsf{fun}\ f(x : \tau_1) : \tau_2\ \{\ e\ \} \\
P \in \mathsf{Program} &::= F \mid F\ P
\end{aligned}
$$

Figure 1 Syntax rules for the language.

$$
\frac{}{\textsc{IsLinear}(\mathsf{Promise}^*(\tau))}
$$

$$
\frac{\textsc{IsLinear}(\tau_1) \vee \textsc{IsLinear}(\tau_2)}{\textsc{IsLinear}(\tau_1 \times \tau_2)} \qquad \frac{\textsc{IsLinear}(\tau_1) \vee \textsc{IsLinear}(\tau_2)}{\textsc{IsLinear}(\tau_1 + \tau_2)}
$$

Figure 2 The IsLinear judgment, which is used to determine whether a type τ is linear.

cannot be typed, since it uses the linear variable x twice. The linear types in our language are types of the form $\mathsf{Promise}^*(\tau)$ (i.e. write handles) and any compound type containing at least one $\mathsf{Promise}^*$. Note that functions in this type system need not be linear, as the lack of closures and global variables ensures that a call to a function cannot use a linear variable without receiving it as a parameter or creating it. A generalization to allow closures is possible, and would require functions to become linear when capturing linear variables[5]. Types are determined to be linear by the judgment IsLinear, defined in Figure 2.

We define the typing environment Γ as a sequence of statements of the form $x : \tau$, such that each statement denotes the type τ of the variable x in the environment. Note that the same environment contains both linearly and non-linearly typed variables.

$$
\Gamma \in \mathsf{Environment} ::= \emptyset \mid \Gamma, x : \tau
$$

Following the notation of Slepak [20], Figure 3 defines the "environment splitting" relation $\boxplus$, such that given an environment Γ, one can state that $\Gamma = \Gamma_1 \boxplus \Gamma_2$ if the environment Γ can be split into Γ_1 and Γ_2. This splitting relation asserts that the *linear* variables in the environments Γ_1 and Γ_2 are *disjoint* and that each linear variable in Γ belongs to *exactly one* of Γ_1 and Γ_2. However, *non-linear* variables *can* be duplicated between both Γ_1 and Γ_2. For example, given a linear variable v and a standard variable w, we cannot split the environment in such a way as to share the variable v between two scopes (i.e. if $\Gamma = \Gamma_1 \boxplus \Gamma_2$, then $v \in \Gamma_1 \not\veebar v \in \Gamma_2$). However, the same restriction does not apply to w, which can be present in one or both environments.

[5] A linear closure would be restricted to a single call. This is overly conservative in some cases, but has been successfully employed in other languages with substructural type systems (e.g. FnOnce in Rust).

$$\frac{}{\emptyset = \emptyset \boxplus \emptyset} \qquad \frac{\Gamma = \Gamma_1 \boxplus \Gamma_2}{\Gamma, x : \tau = (\Gamma_1, x : \tau) \boxplus \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \boxplus \Gamma_2}{\Gamma, x : \tau = \Gamma_1 \boxplus (\Gamma_2, x : \tau)}$$

$$\frac{\Gamma = \Gamma_1 \boxplus \Gamma_2 \qquad \neg \textsc{IsLinear}(\tau)}{\Gamma, x : \tau = (\Gamma_1, x : \tau) \boxplus (\Gamma_2, x : \tau)}$$

Figure 3 The environment splitting relation $\boxplus$.

$$\text{(T-Weaken)}\ \frac{\Gamma_1 \vdash e : \tau_1 \qquad \nexists x : \tau_2 \in \Gamma_2 . \textsc{IsLinear}(\tau_2)}{\Gamma_1 \boxplus \Gamma_2 \vdash e : \tau_1}$$

$$\text{(T-Int)}\ \frac{}{\vdash i : \mathsf{Int}} \qquad \text{(T-Var)}\ \frac{}{x : \tau \vdash x : \tau} \qquad \text{(T-Let)}\ \frac{\Gamma_1 \vdash e_v : \tau_v \qquad \Gamma_2, x_v : \tau_v \vdash e_b : \tau_b}{\Gamma_1 \boxplus \Gamma_2 \vdash \mathsf{let}\ x_v := e_v\ \mathsf{in}\ e_b : \tau_b}$$

$$\text{(T-App)}\ \frac{f : \tau \to \tau' \vdash f : \tau \to \tau' \qquad \Gamma \vdash e : \tau}{\Gamma \boxplus f : \tau \to \tau' \vdash f(e) : \tau'} \qquad \text{(T-Async)}\ \frac{\Gamma \vdash e : \mathsf{Int}}{\Gamma \vdash \mathsf{async}\ e : \mathsf{Int}}$$

$$\text{(T-Product)}\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1 \boxplus \Gamma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \text{(T-Left)}\ \frac{\Gamma \vdash e_L : \tau_L}{\Gamma \vdash \mathsf{Inl}_{\tau_L, \tau_R}\ e_L : \tau_L + \tau_R}$$

$$\text{(T-Right)}\ \frac{\Gamma \vdash e_R : \tau_R}{\Gamma \vdash \mathsf{Inr}_{\tau_L, \tau_R}\ e_R : \tau_L + \tau_R}$$

$$\text{(T-MatchProduct)}\ \frac{\Gamma_1 \vdash e_v : \tau_1 \times \tau_2 \qquad \Gamma_2, x_1 : \tau_1, x_2 : \tau_2 \vdash e_b : \tau_b}{\Gamma_1 \boxplus \Gamma_2 \vdash \mathsf{match}\ e_v\ \{\ (x_1, x_2) \Rightarrow e_b\ \} : \tau_b}$$

$$\text{(T-MatchSum)}\ \frac{\Gamma_1 \vdash e_v : \tau_L + \tau_R \qquad \Gamma_2, x_L : \tau_L \vdash e_L : \tau_b \qquad \Gamma_2, x_R : \tau_R \vdash e_R : \tau_b}{\Gamma_1 \boxplus \Gamma_2 \vdash \mathsf{match}\ e_v\ \{\ \mathsf{Inl}\ x_L \Rightarrow e_L\ ,\ \mathsf{Inr}\ x_R \Rightarrow e_R\ \} : \tau_b}$$

$$\text{(T-Promise)}\ \frac{\neg \textsc{IsLinear}(\tau)}{\vdash \mathsf{promise}\ \tau : \mathsf{Promise}^*(\tau) \times \mathsf{Promise}(\tau)} \qquad \text{(T-Read)}\ \frac{\Gamma \vdash x : \mathsf{Promise}(\tau)}{\Gamma \vdash ?x : \tau}$$

$$\text{(T-Write)}\ \frac{\Gamma_1 \vdash x_p : \mathsf{Promise}^*(\tau) \qquad \Gamma_2 \vdash e_v : \tau}{\Gamma_1 \boxplus \Gamma_2 \vdash x_p \leftarrow e_v : \mathsf{Int}}$$

$$\text{(T-Func)}\ \frac{x : \tau \vdash e : \tau'}{\vdash \mathsf{fun}\ f(x : \tau) : \tau' \{e\} : \tau \to \tau'}$$

Figure 4 The typing rules for the language[a].

[a] The results of (T-Async) and (T-Write) are meaningless, but evaluate to placeholder integers due to the absence of a void/unit type.

Listing 1 Program violating double write restriction.

```
1 fun doubleWrite(p: Promise*(Int)): Int {
2   let _ := p ← 0 in
3   p ← 1
4 }
```

Figure 4 defines the typing rules. While most of the rules are fairly standard, it is important to observe the mechanics used for linear typing. Each rule only allows the most restricted possible environment[6] to be present (e.g. (T-Var) operates on an environment that contains only the referenced variable). This prevents the program from dropping a linear variable without using it.

The (T-Weaken) rule is employed to relax the restriction of only applying a rule to the smallest environment. Specifically, (T-Weaken) allows for unrestricted variables to be dropped arbitrarily. For example, take the program $\mathsf{let}\ x := 5\ \mathsf{in}\ 0$. Since the (T-Int) rule only operates on an empty environment, we cannot directly apply it to type 0 as an Int with x in the context. However, (T-Weaken) allows us to drop the variable x when type-checking 0, because x is an unrestricted variable. In practice, this means that unrestricted variables behave exactly as they would in languages without linear typing. Note that (T-Weaken) *cannot* be applied to linear variables, since dropping a linear variable would allow it to escape from being used (breaking the guarantee that each linear variable is used exactly once).

Environment splitting accomplishes the other half of linear typing. Whenever a linear variable appears in the environment for an expression containing multiple sub-expressions, we must ensure that it is only available in one sub-expression. For example, consider a product (e_1, e_2) and a linear variable x. If x appeared in both e_1 and e_2, it would clearly allow for x to be used twice. To circumvent this, typing rules with multiple sub-expressions *split* the environment into several sub-environments, such that each linear variable appears in only one sub-environment. In this case, x could only be in scope for one of e_1 and e_2.

Several rules relate directly to promises and are critical to implementing the desired safety guarantees. Specifically, note the fact that the promise constructor in (T-Promise) returns both a linear/owned handle for the promise and a standard handle for the promise; this creates the split between the write-end and read-end of the promise, respectively. In the case of (T-Read), observe that it cannot accept a write-end to a promise, as reading from the write-end would "use" the promise and result in never writing to it. Likewise, observe that in (T-Write) the promise must be a linear write-end, so as to enforce only writing to a promise a single time and only in an owned context.

Put together, these typing rules enforce the critical property that promises must be written to exactly once. Since only one un-copyable and un-droppable handle exists for writing to a given promise, it is impossible to write to a promise twice or to forget to write to a promise. Further, the establishment of ownership by the linear type system enforces the property that a promise can only ever be written to by its owner.

As an illustration of the typing rules and their effects, let us consider several small programs that fail to type check and, more importantly, demonstrate bugs that can occur via incorrect use of promises.

[6] This is similar to the "Small Footprint" rule used in separation logics, in that the typing rules contain the smallest possible environment but can be generalized by (T-Weaken).

Listing 2 Program violating omitted write restriction.

```
1 fun omittedWrite(p: Promise*(Int)): Int {
2   match Inl_{Int,Int} 0 {
3     Inl i ⇒ 0,
4     Inr i ⇒ p ← 1
5   }
6 }
```

Listing 3 Program violating unowned write restriction.

```
1 fun unownedWrite(p: Promise(Int)): Int {
2   p ← 0
3 }
```

Listing 1 illustrates a double write bug. When type checking this function, we must split the environment on line 2 (via (T-Let)) such that p belongs to the right-hand side of the let expression and is no longer owned in line 3. Thus, it becomes impossible to type check line 3 in this context, since (T-Write) demands that p be owned when writing. In other words, there is *no* way to type this function. If allowed to execute, this program would attempt to assign a second value to the promise p. This may lead to undefined behavior; if, for example, two threads await the promise p, they may read different values depending on when the second write occurs. While other languages handle this scenario as a runtime error – for example, this same error is handled as an exception future_errc::promise_already_satisfied in C++ and by simply ignoring subsequent writes to the promise in JavaScript – our language can statically prevent this behavior from *ever* occurring.

Listing 2 demonstrates an omitted write bug, in which a promise is not fulfilled in some paths of execution. As an analogy, compare this to how the Java compiler rejects functions that are missing a return statement in one branch of a conditional. When type checking the match expression on line 2 with (T-MatchSum), we split the environment between the value being matched and the two branches, requiring that both branches type check under the same environment. In this case, we assign an empty environment to the value being matched and an environment consisting of p to the two branches. While the branch on line 4 type checks via (T-Write), the variable p is unused on line 3. Since p is a linear variable, it is impossible to eliminate p via (T-Weaken), and thus the program cannot type check. When executing this program, the match expression would reach line 3 and fail to assign a value to p before returning.

Finally, Listing 3 demonstrates writing to a promise that is not owned. This is in contrast to Listing 1, where a write is attempted while *no* promise is in scope. When attempting to type check line 2, we find it impossible to apply (T-Write) since the type of p is not of the form Promise*(Int). During execution, this program would allow various bugs to occur due to the inability to track this write operation elsewhere in the program. For example, it could perform a second write to an already-fulfilled promise, creating a double write bug.

2.3 Operational Semantics

To define the operational semantics of a program, we must first introduce some extra syntax to represent the runtime state of a program. First, we extend the expression syntax with the construct $\&i_\tau$, which represents a reference to the read-end of a promise of τ with the

$$\text{(T-PromiseRef)}\ \frac{}{\vdash \&i_\tau : \mathsf{Promise}(\tau)} \qquad \text{(T-PromiseRef}^*\text{)}\ \frac{}{\vdash \&^* i_\tau : \mathsf{Promise}^*(\tau)}$$

Figure 5 Typing rules for promise references.

$$\begin{aligned} m \in \mathsf{Thread\ ID} \quad &::= i \\ T \in \mathsf{Threads} \quad &::= \langle m, e \rangle \mid T \parallel T \end{aligned}$$

Figure 6 Syntax for thread trees.

unique ID i. We also add the construct $\&^* i_\tau$, which represents a reference to the write-end of a promise of τ with the unique ID i. We augment the type system with two new rules, defined in Figure 5, to give the promise-reference literals a type.

Figure 6 defines the currently-executing state of a program as a binary tree of threads, known as a *fork-tree*. In such a fork-tree, each node contains a thread ID and an expression. Each fork operation splits the node to a left thread, representing the currently executing code, and a right thread, representing the new task. These terms are composed together to create a tree of all concurrently executing tasks. We encode the entire state of the program as the quadruple (i, P, O, t): the next-available ID for a promise or thread, a mapping of promise IDs to an optional value, a mapping of promises to their owner thread, and the thread tree for the program. Semantically, P stores the values (or lack thereof) associated with a given promise, and also indicates whether a given promise has been fulfilled. This is encoded by the type $\tau + \mathsf{Int}$, where τ corresponds to a value and Int corresponds to a placeholder for an unfulfilled promise (the value of which is ignored). The initial state of a program is $(1, \emptyset, \emptyset, \langle 0, \mathit{main}(0) \rangle)$, where $\mathit{main} : \mathsf{Int} \to \mathsf{Int}$.

Figure 7 defines the IsValue predicate, used to identify expressions that are fully evaluated. Finally, we utilize contextual semantics to simplify the notation for the operational semantics. Specifically, Figure 8 defines two contexts K and J to represent an expression with a hole and a fork-tree with a hole, respectively. While the normal transition rules (e.g. (S-Write)) implement the transition rules for base-case expressions, the K contextual semantics allow us to recursively apply these transformations to sub-expressions and the J contextual semantics allow us to apply these transformations to any thread in the tree.

Figure 9 defines the operational semantics. Of particular interest here are the rules for modeling non-determinism. The binary fork-tree structure of the program allows us to apply the basic stepping rules (such as function application, let evaluation, etc.) to any thread at

$$\frac{}{\text{IsValue}(i)} \qquad \frac{}{\text{IsValue}(\&i_\tau)} \qquad \frac{}{\text{IsValue}(\&^* i_\tau)} \qquad \frac{\text{IsValue}(v_1) \wedge \text{IsValue}(v_1)}{\text{IsValue}((v_1, v_2))}$$

$$\frac{\text{IsValue}(v)}{\text{IsValue}(\mathsf{Inl}_{\tau_L, \tau_R}\ v)} \qquad \frac{\text{IsValue}(v)}{\text{IsValue}(\mathsf{Inr}_{\tau_L, \tau_R}\ v)}$$

Figure 7 The IsValue predicate, which determines whether an expression is a terminal value.

$$
\begin{aligned}
K \in \textsf{Expr Context} \quad ::= \; & \cdot \mid \textsf{let } x := K \textsf{ in } e \mid f(K) \mid \textsf{Inl } K \mid \textsf{Inr } K \\
\mid \; & (K, e) \mid (e, K) \mid \textsf{match } K \{ \textsf{ Inl } x_1 \Rightarrow e_1, \textsf{Inr } x_2 \Rightarrow e_2 \} \\
\mid \; & \textsf{match } K \{ (x_1, x_2) \Rightarrow e \} \mid ?K \mid K \leftarrow e \mid e \leftarrow K \\
J \in \textsf{Thread Context} \quad ::= \; & \cdot \mid J \parallel T \mid T \parallel J
\end{aligned}
$$

Figure 8 K (expression-level) and J (thread-level) contextual semantics.

$$
\text{(SJ-Schedule)} \quad \frac{(i, P, O, \langle m, e\rangle) \longrightarrow (i', P', O', T')}{(i, P, O, J[\langle m, e\rangle]) \longrightarrow (i', P', O', J[T'])}
$$

$$
\text{(SK-RunExpr)} \quad \frac{(i, P, O, \langle m, e\rangle) \longrightarrow (i', P', O', \langle m, e'\rangle)}{(i, P, O, \langle m, K[e]\rangle) \longrightarrow (i', P', O', \langle m, K[e']\rangle)}
$$

$$
\text{(SK-RunFork)} \quad \frac{(i, P, O, \langle m, e\rangle) \longrightarrow (i', P', O', \langle m, e'\rangle \parallel T)}{(i, P, O, \langle m, K[e]\rangle) \longrightarrow (i', P', O', \langle m, K[e']\rangle \parallel T)}
$$

$$
\text{(S-Fork)} \quad \frac{\Gamma \vdash e : \tau \qquad O_L = \{p \mapsto m \mid p \mapsto m \in O, p \notin \Gamma\} \qquad O_R = \{p \mapsto i \mid p \mapsto m \in O, p \in \Gamma\}}{(i, P, O, \langle m, \textsf{async } e\rangle) \longrightarrow (i+1, P, O_L \cup O_R, \langle m, 0\rangle \parallel \langle i, e\rangle)}
$$

$$
\text{(S-Let)} \quad \frac{\textsc{IsValue}(v)}{(i, P, O, \langle m, \textsf{let } x := v \textsf{ in } e\rangle) \longrightarrow (i, P, O, \langle m, e[v/x]\rangle)}
$$

$$
\text{(S-App)} \quad \frac{\textsc{IsValue}(v)}{(i, P, O, \langle m, f(v)\rangle) \longrightarrow (i, P, O, \langle m, \textsc{Body}(f)[v/\textsc{Arg}(f)]\rangle)}
$$

$$
\text{(S-MatchProduct)} \quad \frac{\textsc{IsValue}(v_1) \qquad \textsc{IsValue}(v_2)}{\begin{array}{c}(i, P, O, \langle m, \textsf{match } (v_1, v_2)\{(x_1, x_2) \Rightarrow e\}\rangle) \longrightarrow \\ (i, P, O, \langle m, e[v_1/x_1, v_2/x_2]\rangle)\end{array}}
$$

$$
\text{(S-MatchSumL)} \quad \frac{\textsc{IsValue}(v)}{\begin{array}{c}(i, P, O, \langle m, \textsf{match Inl } v\{\textsf{Inl } x_L \Rightarrow e_L, \textsf{Inr } x_R \Rightarrow e_R\}\rangle) \longrightarrow \\ (i, P, O, \langle m, e_L[v/x_L]\rangle)\end{array}}
$$

$$
\text{(S-MatchSumR)} \quad \frac{\textsc{IsValue}(v)}{\begin{array}{c}(i, P, O, \langle m, \textsf{match Inr } v\{\textsf{Inl } x_L \Rightarrow e_L, \textsf{Inr } x_R \Rightarrow e_R\}\rangle) \longrightarrow \\ (i, P, O, \langle m, e_R[v/x_R]\rangle)\end{array}}
$$

$$
\text{(S-Promise)} \quad \frac{}{(i, P, O, \langle m, \textsf{promise } \tau\rangle) \longrightarrow (i+1, [i \mapsto \textsf{Inr } 0; P], [i \mapsto m; O], \langle m, (\&^* i_\tau, \&i_\tau)\rangle)}
$$

$$
\text{(S-Read)} \quad \frac{e_p = \&(i_p)_\tau \qquad P(i_p) = \textsf{Inl } v}{(i, P, O, \langle m, ?e_p\rangle) \longrightarrow (i, P, O, \langle m, v\rangle)}
$$

$$
\text{(S-Write)} \quad \frac{e_p = \&^*(i_p)_\tau \qquad \textsc{IsValue}(v)}{(i, P, O, \langle m, e_p \leftarrow v\rangle) \longrightarrow (i, [i_p \mapsto \textsf{Inl } v; P], O, \langle m, 0\rangle)}
$$

Figure 9 Operational semantics for the language.

any time, while also enabling forking to occur at any program point. In other words, we rely on the ability to *substitute* a single thread (e.g. $\langle \mathsf{let}\ x := \mathsf{async}\ e_1\ \mathsf{in}\ e_2 \rangle$) with a sub-tree (e.g. $\langle \mathsf{let}\ x := 0\ \mathsf{in}\ e_2 \rangle \parallel \langle e_1 \rangle$) to allow for forking within the scope of an arbitrary expression.

Much of the heavy-lifting with regards to concurrency occurs via the contextual semantics, which are implemented via three transition rules: (SJ-Schedule), (SK-RunExpr), and (SK-RunFork). (SJ-Schedule) forms the basis of the threading model, allowing us to treat any individual task in the fork-tree as a hole (where we can apply other transition rules). In practice, this implements scheduling for the program by picking a thread and performing transformations on it. The two K-level rules, (SK-RunExpr) and (SK-RunFork) are extremely similar, in that they focus on a hole in the expression tree and apply further transformations to it. In practice, these holes take the form of larger expressions that require reduction before evaluation; for example, consider how (S-App) requires the function's argument to be reduced to a value before calling the function. The two rules differ in that (SK-RunFork) specifically allows for a fork operation to occur within the sub-expression, whereas (SK-RunExpr) runs the sub-expression as a singular thread. As an example, imagine that the argument to a function involves an async operation. When evaluating the argument, the hole is no longer a single expression, but rather a fork-tree of two expressions. When substituting this subtree into the program, care must be taken to ensure that the old thread's expression is substituted in while the new thread stays distinct.

The rule (S-Fork) is also of particular interest, as it is used to spawn new threads. When (S-Fork) creates a new thread, it must assign a new thread ID to it (picking a unique value from the state of the program i). Note that a promise write handle's owner is the thread that will write to that promise. That is, since two thread IDs now exist, each promise in scope will be mapped to one of these IDs in the ownership map O. For example, if the newly spawned thread will use an owned promise p then we must transfer the ownership of p to the new thread. This runtime ownership tracking is used only for proofs, and can be removed from the semantics of the language for an actual implementation without any effect. At the same time, it provides a tool that could be leveraged to perform deadlock detection at runtime [22], which may be useful for an implementation.

Finally, let us briefly consider the mechanics of the (S-Promise), (S-Read) and (S-Write) rules. These rules all rely on the map P, which tracks each promise's state. (S-Promise) adds an entry for i (the ID of the new promise), mapping it to $\mathsf{Inr}\ 0$ – a placeholder indicating that p hasn't been fulfilled. In (S-Write), P is extended with a mapping of i_p (the ID of the promise p) to $\mathsf{Inl}\ v$ (a value v). The use of the Inl constructor indicates that p has been fulfilled, and the value v corresponds to p's new value. (S-Read) checks the state of i_p, and only advances once the value stored in $P(i_p)$ is fulfilled (indicated by the Inl constructor). Once p is fulfilled, (S-Read) evaluates to the stored value. In effect, this forces (S-Read) to block until a value for p becomes available.

3 Theoretical Guarantees

The goal of our language is designing a system that is free of a number of common concurrency bugs. Specifically, we argue that our language cannot exhibit an omitted write bug (creating a promise but never fulfilling it) in terminating programs, and cannot ever exhibit an unowned write bug (writing to a promise that is not owned by the current thread) or a double write bug (fulfilling the same promise twice).

▶ **Theorem 1** (Linearity). *Given a linear variable $x : \tau_x$, if $\Gamma, x : \tau_x \vdash e : \tau_e$ then it is always the case that $\Gamma \vdash e : \tau_e$ cannot be well-typed. Likewise, if $\Gamma \vdash e : \tau_e$ then $\Gamma, x : \tau_x \vdash e : \tau_e$ cannot be well-typed. That is, a linear variable cannot be added/dropped in Γ for the same e.*

Proof. We argue that whenever a linear variable $x : \tau_x$ exists in the environment $\Gamma, x : \tau_x$ for a well-typed program e, then x cannot be dropped while retaining a well-typed program. Further, whenever $x : \tau_x$ *does not exist* in Γ for a well-typed e, x cannot be added to Γ while retaining a well-typed program. Each can be shown by trivial induction on the typing rules. ◀

3.1 Soundness

We define soundness to mean that if a program type-checks it can only get "stuck" due to promise dependency cycles. That is, either a program is done executing, is able to progress via the operational semantics, or it contains a set of threads T such that every thread in T is blocked on a promise owned by another thread in T. This is critical to proving the various other properties of our language.

▶ **Lemma 2** (Substitution Preservation). *Given a well-typed program, such that* $\Gamma \vdash e : \tau$*, if* $\Gamma \vdash x_0 : \tau_0$ *and* $\Gamma \vdash e_0 : \tau_0$ *then* $\Gamma \vdash e[e_0/x_0] : \tau$*.*

Proof. Observe that the typing relation is defined such that if an expression is well-typed, any sub-expression must also be well-typed. Since x_0 and e_0 share the same type, substitution cannot effect the overall typing judgment. This can be trivially shown by induction on e. ◀

▶ **Lemma 3** (Promise Preservation). *Given a promise* $p : \mathit{Promise}(\tau)$*,* $\vdash P(p) : \tau + \mathit{Int}$*.*

Proof. First, observe that an owned promise $p : \mathsf{Promise}^*(\tau)$ can only be fulfilled via the (T-Write) rule on an expression $p \leftarrow v$, where the value v is an inhabitant of the type τ. By definition, the value *written* to the promise will always match the correct type τ.

Next, observe that the write and read ends of a promise always share their type. The only way to create a promise literal of the form $\&i_\tau$ or $\&^*i_\tau$ is via the (S-Promise) rule. By definition this rule assigns the same *unique* ID to both ends, so each newly created read-end and write-end will correspond to a matching type τ.

Finally, notice that the type of both a read-end and a write-end of a promise will never change throughout the program. Via (T-PromiseRef) and (T-PromiseRef*), a promise literal's type is directly encoded in the syntax of the promise literal. By case analysis, it is trivially true that there is no way to transform the type encoded in the promise literal. There is no possible syntax that can allow for the wrong type of value to be written to a promise and there is also no possible syntax to attempt reading a promise as the wrong type.

Thus, in any step $e \longrightarrow e'$ where e involves a write operation, both the value placed into the promise and the promise itself always retain their types. Therefore it can be seen that no possible sequence of steps can result in a promise p containing a value of the wrong type. ◀

▶ **Lemma 4** (Local Preservation). *Given a program* $\langle m, e\rangle$ *such that* $\vdash e : \tau$*, then if* $\langle m, e\rangle \longrightarrow \langle m, e'\rangle$ *or* $\langle m, e\rangle \longrightarrow \langle m, e'\rangle \parallel t$ *it must be true that* $\vdash e' : \tau$*.*

Proof. By simple induction over the transition rules and appeal to Lemmas 2 and 3, it can be seen that any step $e \longrightarrow e'$ yields e' with the same type τ (and when forking, a second thread with any type). Therefore, in all cases the original thread retains its type τ. ◀

▶ **Theorem 5** (Global Preservation). *For any well-typed program, any operational semantics step taken will leave us with a well-typed program where each thread either retains its old type or has been newly created.*

Proof. By induction on the step that is taken, observe that either:

1. The program consists of a single thread and does not fork, in which case it is trivially true that the program's type remains globally preserved via Lemma 4.
2. A new thread is created and the parent thread is modified via (S-Fork). By Lemma 4 we know that the parent thread is preserved. Since the newly created thread was not previously present in the tree, and the sub-expression it is created from was well-typed, then it must be true that the new thread is well-typed.
3. A thread is modified in a program consisting of many threads, which can only be done by using the thread-level contextual semantics to navigate to that thread's subtree. In this case we can apply the inductive hypothesis to the thread's subtree in order to show that all threads in the subtree are preserved once they are modified. The threads that were not modified are preserved by definition, therefore it must be true that the entire tree is preserved after substituting in the modified subtree. ◀

▶ **Lemma 6** (No Ownership on Termination). *For any thread $\langle m, e\rangle$ with well-typed e, e cannot be reduced to a terminal value while still owning a promise.*

Proof. Observe that, by definition, all terminal values can only be well-typed in the empty environment. We cannot drop an owned promise (or any linear variable that might contain a promise) via (T-Weaken). No task can terminate as a linear value, as the async expression requires a type of Int for the spawned task and the *main* function evaluates to Int (i.e. all threads evaluate to Int). Thus, before termination a thread must *always* eliminate all owned promises. ◀

▶ **Definition 7** (ContainsCycle(O, T)). *Given a promise-ownership map O and a tree of tasks T, let G be a graph containing a vertex for each task in T. Then, for every task t_1 that is currently reading a promise p, where p's owner $O(p) = t_2$, create a directed edge from t_1 to t_2. The pair (O, T) are said to contain a cycle iff G contains a cycle.*

▶ **Theorem 8** (Progress). *Given a well-typed program (i, P, O, T), either $(i, P, O, T) \longrightarrow (i', P', O', T') \vee \forall t \in T.\textsc{IsValue}(t) \vee \textsc{ContainsCycle}(O, T)$.*

Proof. We structure our progress theorem as the statement that given a well-typed program either the program can make progress, the program has terminated (i.e. every thread is a value literal), or the program contains a promise dependency cycle. In any case where a step can be made or all threads have terminated, there is nothing to show. Thus, if we assume that progress is impossible but the program has not terminated, we argue that it must be due to the existence of such a promise dependency cycle.

Let us assume that all non-value threads in T are not steppable and that at least one thread $t = \langle m, e\rangle$ is not reduced to a value. For a non-value thread to be un-steppable, it must be performing a read operation applied to a promise reference: $K[?(\&i_\tau)]$. This can be seen by induction over e: if e is not a terminal value, the program can always unconditionally step forward or reduce further using K-level contextual semantics *unless* it is in the form of a read operation $K[?(\&i_\tau)]$ where the promise $\&i_\tau$ is unfulfilled (in which case e is *blocked* on the promise i). Let the owner thread $o = O(i)$, the owner of the promise i. We consider a recursive walk through the tree of threads, in which there are three cases:

1. o refers to the current thread. In this case, we trivially observe a cycle from o to o.
2. o refers to a thread that is a basic value and cannot be further evaluated. This case is trivially shown to be impossible, since o is unable to own the promise i via Lemma 6.
3. o refers to another blocked thread, and we recursively apply the same logic to o.

Given that there is a finite set of threads, by the pigeonhole principle we observe that the process of traversing dependencies must eventually terminate with a cycle (note that we have already rejected the base case of the thread being able to step forward). By this logic, it can be seen that the only case where a program is not terminated and cannot step forward is when a cycle exists in (O, T).

Therefore we have shown that for any well-typed program, the condition holds that the program is finished evaluating, can step forward, or contains a dependency cycle. ◀

▶ **Theorem 9** (Soundness). *Given a well-typed program* (i, P, O, T)*, if* $(i, P, O, T) \longrightarrow^* (i', P', O', T')$ *then either the program* T' *can continue, all threads have reached a terminal value, or it contains a cycle.*

Proof. By induction over the set of steps $(i, P, O, T) \longrightarrow^* (i', P', O', T')$:

1. If no steps were taken, we argue that since the program is well-typed then by Theorem 8 it must be true that either T' can step forward, it has terminated, or it contains a cycle.
2. If at least one step $(i, P, O, T) \longrightarrow (i', P', O', T')$ has been taken, then we apply Theorem 5 to show that the program (i', P', O', T') remains well-typed. Now, by the inductive hypothesis we argue that the rest of the sequence of steps must either be able to continue, have already terminated, or contain a cycle.

Therefore, for any sequence of steps it must be the case that the final result can continue, has terminated, or contains a cycle. ◀

3.2 Additional Properties

Building on Theorem 9, we can easily show that each of the language's guarantees still holds.

▶ **Lemma 10** (Single-Write of Promises). *For any well-typed program, all promises in that program are written to* at most *once. For any well-typed program that* terminates successfully*, all promises in that program are written to* exactly *once.*

Proof. This can be easily observed, as there are only two ways that an owned promise can be used: transferring ownership via a function call/function return/alias or writing to it. Since the single-use property of linear variables is unconditionally true in the program by Theorem 1, then it must be the case that, for any promise p:

1. Given a write operation, a promise p is used and can no longer be reused.
2. Given a transfer operation, a promise p is used and an alias p' is created, which must then be used by the program.
3. Given any other operation, the promise p remains available and must be used.

In order for the above to reach termination, it must eventually reach the base case of writing to a promise. Therefore, for any program that has successfully terminated, all promises have been written to exactly once. Until the point of termination, there is no operation that allows for the duplication of a promise, and thus we can create an upper-bound of at most one write to any promise for programs that have not yet terminated. ◀

▶ **Corollary 11** (No Omitted Writes). *If a program is well-typed and terminates successfully, all created promises will be fulfilled during execution.*

▶ **Corollary 12** (No Double Writes). *If a program is well-typed, no promise will be fulfilled multiple times.*

▶ **Theorem 13** (No Unowned Writes). *If a program is well-typed, any promise that is written to will be owned by the currently executing thread (i.e. the writer).*

Proof. Promises can only become owned by the currently executing thread when created in it or when transferred to it. Transfer can only occur via async, as the only other inter-thread communication consists of promises (which cannot contain linear values). By definition (S-Fork) transfers ownership to the thread that will use the promise. Since the only promise we can write to is an owned promise, and other values cannot change types to become an owned promise by Theorem 5, then it is impossible to write to anything except a promise owned by the current thread. ◀

We have shown that our language does not permit double or unowned writes, and that it does not permit omitted writes in terminating programs. Note that a diverging program may indefinitely postpone promise writes; for example, consider a program that runs an infinite loop before issuing a write to a promise. While this is a limit to the language's guarantees, observe that futures have this same limitation: a future created from a function that does not terminate will never be fulfilled. In effect, the type system gives promises similar safety characteristics to futures: both promises and futures will always be fulfilled at most once and only by their owner, and will always be fulfilled exactly once if the program reaches termination. This side-steps the safety gap between promises and futures noted by [9, 10].

▶ **Corollary 14** (Only Cyclical Deadlocks). *If a program is well-typed, the program can only ever get stuck*[7] *due to a cyclical deadlock.*

Proof. This directly follows from Theorem 9. ◀

With the result of Corollary 14, it's useful to note the advantages our language presents for runtime deadlock detection. Voss and Sarkar [22] present an algorithm for dynamically detecting deadlocks in promise-based programs, which consists of two conditions: that all promises are fulfilled and that there are no cyclical dependencies among tasks caused by promises awaiting a result. Since all promises are known to be eventually fulfilled by Corollary 11, simply identifying the cycles at runtime must be sufficient to catch all possible deadlocks encountered during a program's execution. Notably, the thread/promise-ownership table used for deadlock detection in their algorithm is already tracked at runtime per the operational semantics of the language. In effect, this allows us to trivially implement dynamic deadlock detection in our language without altering the syntax, type system, or semantics.

4 Implementation

In order to evaluate our language, we implemented a compiler that performed type checking and translated our language to a subset of Java[8]. Specifically, the compiler built upon the formalisms of the language and carefully extended it to allow for more user-friendly programming. Thanks to the type system's guarantees, no additional runtime overhead is added to the generated programs.

[7] In the sense that no operational semantics steps can be taken and the program is not done executing.
[8] The implementation is included as part of the supplementary materials.

4.1 Language Extensions

In designing the compiler, a number of additional features were added to the language that are not represented in Section 2. First, we extended the language to allow named and *recursive* types, as the current language definition only supports anonymous, non-recursive sum and product types. Through a simple syntax for defining new types (known as "records" and "unions"), the language is able to apply similar typing rules to the specified product and sum types, respectively. The addition of named types follows the existing type system rules, so that $\mathsf{IsLinear}(\tau)$ is true for all user-defined τ that contain linear members. In other words, we can still reason about a named type being linear, since naming a type does not hide its linearity. This generalization enables users of the language to create and interact with various recursive types, such as linked-lists and binary trees.

As another quality of life improvement, the language implementation was generalized to allow N-ary functions, whereas the type system described in Section 2 only allows for unary functions (functions with a single parameter). This, combined with support for additional types such as `Unit`, `Bool` and `String`, eases translation for many real-world programs.

Both `for` and `while` loops were also added to the language. However, due to the nature of loops – which may execute any number of times – it is impossible to guarantee that variables in a loop will be used exactly once. For example, consider the program `while condition() { p <- 0 }`. If p is a value of type $\mathsf{Promise}^*(\mathsf{Int})$, we can observe two possible bugs in the above program. Firstly, the function `condition()` may initially return `false`, in which case the write would *never* occur. Likewise, it is possible that `condition()` returns `true` multiple times in a row, in which case the write would occur *more than once*. Both of these possibilities would clearly violate the language's guarantees, and thus must be disallowed. To do so, the compiler prevents capturing a free linear variable in a loop. In this case, unless p is defined within the loop, it is impossible to reference it from within the loop. Note that while such a restriction is overly conservative (i.e. a loop *could* be designed to only write once), similar restrictions exist in other substructural type systems. In practice, such a restriction has not prevented the adoption of other substructurally-typed languages (e.g. Rust, which has partially addressed this problem through the use of iterators).

The compiler also provides a function unsafeWrite, which enables writing to a promise's read-end. At runtime, this operation is equivalent to writing to the write-end, but allows for multiple writes to the same promise during type checking. This enables an escape hatch, which can be useful when directly translating a foreign program or dealing with conditions that are difficult to reason about (e.g. consider a loop that only fulfills a promise on the first iteration). While this construct introduces an unowned write bug (and potentially double writes), it *does not* negate the language's soundness and still prevents the omitted write bug. This must be the case because of the existence of the write-end, which ensures that at least one write unconditionally occurs. If unsafeWrite were implemented as a no-op, the program would still be sound by definition (it is equivalent to the program without unsafeWrite). Now observe that there is no way for an additional write to cause the program to get stuck, because threads can *only* block while awaiting an *unfulfilled* promise. Therefore, it must be the case that all well-typed programs remain sound with unsafeWrite enabled.

4.2 The Type Checking Algorithm

The type checking algorithm is largely built upon the notion of environment splitting. However, to perform splitting efficiently a $\textsc{split}(\Gamma, e_1, e_2)$ function had to be devised such that $\textsc{split}(\Gamma, e_1, e_2) = (\Gamma_1, \Gamma_2) \implies \Gamma = \Gamma_1 \boxplus \Gamma_2 \wedge \Gamma_1 \vdash e_1 : \tau_1 \wedge \Gamma_2 \vdash e_2 : \tau_2$. In

other words, the SPLIT function would inspect two expressions and assign variables from the environment to each expression, creating their corresponding type checking environments. This is used to to mimic the behavior of the $\Gamma_1 \boxplus \Gamma_2$ relation in the type system.

In Voss and Sarkar [22], user-supplied annotations indicate the transfer of promise ownership between tasks. These annotations take the form of a list of promises to "move" at every async expression, and the newly-spawned task becomes the owner of the provided promises. This mechanism is integral to both the dynamic detection of omitted writes and the dynamic deadlock detection algorithm that Voss and Sarkar introduce. By encoding ownership information in the type system and *inferring* the "moves" via our environment splitting algorithm, our type checker provides a way to lower the burden of explicit annotations for such a system. The key, in this case, is the ability of the split operation to statically determine which sub-expression (the async task or the code that follows spawning the new task) should become the owner of any given promise and, through the linearity constraints of promises, force this ownership model to be adhered to.

Algorithm 1 Environment Splitting between Expressions.

1: **function** SPLIT(Γ, e_1, e_2)
2: $\quad \text{free}_1 \leftarrow \mathsf{free}(e_1)$
3: $\quad \text{free}_2 \leftarrow \mathsf{free}(e_2)$
4: $\quad \Gamma_1 \leftarrow \{v : \tau \mid v : \tau \in \Gamma, v \in \text{free}_1\}$
5: $\quad \Gamma_2 \leftarrow \{v : \tau \mid v : \tau \in \Gamma, v \in \text{free}_2\}$
6: $\quad \text{allUsed} \leftarrow \forall v : \tau \in \Gamma \;.\; v \in \Gamma_1 \vee v \in \Gamma_2 \vee \neg \mathsf{IsLinear}(\tau)$
7: $\quad \text{noneReused} \leftarrow \nexists v : \tau \in \Gamma \;.\; \mathsf{IsLinear}(\tau) \wedge v \in \Gamma_1 \wedge v \in \Gamma_2$
8: $\quad$ **if** allUsed $\wedge$ noneReused **then return** (Γ_1, Γ_2)
9: $\quad$ **else type error**

The SPLIT function, defined in Algorithm 1, operates on an environment Γ and two expressions, e_1 and e_2. It begins by finding the free variables in each expression with respect to an empty environment, revealing which variables will be referenced in each expression (lines 2-3). Then, it creates an environment for each expression by taking all variables from Γ that are in that expression's free set (lines 4-5). We then define allUsed as the statement that every variable in Γ either appears in one of the two environments or is non-linear; that is, *are 0 linear variables dropped?* Next, we define noneReused to be the statement that there are no variables in Γ that are linear and used in both environments; that is, *are 0 linear variables copied?* If both conditions hold true, we return the two environments (line 8); otherwise, we throw a type error, as the environment cannot be validly split (line 9).

Since functions are generalized to be N-ary, special care must be taken to ensure that the environment is split correctly for each argument. Algorithm 2 thus defines a special form of the SPLIT function called SPLIT_SEQUENCE(Γ, es). This function utilizes the same approach as the SPLIT function, but operates recursively on a single expression at a time. In doing so, it is able to confirm that at every step the all-used and none-reused conditions still hold.

SPLIT_SEQUENCE functions as a generalization of SPLIT, taking an environment Γ and a list of expressions es. This algorithm recursively walks through es and splits the environment at each expression. The base case of an empty list returns an empty list of environments (line 2). Otherwise, the function continues to its recursive step on line 4. First, we define expr to be the head of the list and rest to be the remainder of the list (lines 4-5). We then find the free variables for expr (line 6) and define free_rest to be the union of the free variables for each expression in rest (line 7). As with SPLIT, we construct a Γ for each set of free variables

Algorithm 2 Environment Splitting between a Sequence of Expressions.

```
1:  function SPLIT_SEQUENCE(Γ, es)
2:      if es = [] then return []
3:      else
4:          expr ← head(es)
5:          rest ← tail(es)
6:          free_expr ← free(expr)
7:          free_rest ← ⋃_{e∈rest} free(e)
8:          Γ1 ← {v : τ | v : τ ∈ Γ, v ∈ free_expr}
9:          Γ2 ← {v : τ | v : τ ∈ Γ, v ∈ free_rest}
10:         allUsed ← ∀v : τ ∈ Γ . v ∈ Γ1 ∨ v ∈ Γ2 ∨ ¬IsLinear(τ)
11:         noneReused ← ∄v : τ ∈ Γ . IsLinear(τ) ∧ v ∈ Γ1 ∧ v ∈ Γ2
12:         if allUsed ∧ noneReused then return append(Γ1, SPLIT_SEQUENCE(Γ2, rest))
13:         else type error
```

(lines 8-9) and then check whether all linear variables are used and none of them are reused (lines 10-12). If so, the current expression can be split, so we return Γ_1 appended to the result of SPLIT_SEQUENCE on rest with Γ_2 (line 12). Otherwise, we throw a type error to signal that the environment cannot be split (line 13).

At a high-level, the type checker performs pattern matching against the various syntactic constructs. Each case is handled as per the typing judgments, utilizing the SPLIT and SPLIT_SEQUENCE functions to assign the environments for each sub-expression. For the base cases of the type checker, special care must be taken to ensure that weakening is applied correctly; in this case, splitting the environment between the actual expression and an empty/dummy expression ensures that only used variables (including linear variables) remain. In the case of branches, both branches must be split from the condition/matched value (and not from each other) so that it is ensured that the same linear variables occur in each branch.

In Algorithm 3, the TYPECHECK function takes an environment Γ and an expression e to type check in Γ. The function defines a variable complete as the result of splitting Γ between e and a dummy expression (line 2). This is used to determine whether all linear variables in Γ are used by e: if e cannot be split, it must be due to dropping or reusing a linear variable in Γ. Next, the function performs pattern matching against e (line 3) to determine the syntactic construct that appears.

Lines 4-7 handle references to variables (called x, in this case). Because this is a base case, care must be taken to ensure that no linear variables are dropped; this is done by checking that complete references a valid environment (line 5). If so, the type of x is fetched from the environment Γ (line 6); otherwise, the function throws a type error (line 7).

Lines 8-12 handle the let syntax. In particular, when a let expression is encountered, the environment must be split so that the value and body are checked in different environments. Γ is split into Γ_{rhs} and Γ_{body}, which are used for the RHS and the body, respectively (line 9). We recurse to determine the type of the RHS (line 10). We then add $id : \tau_{rhs}$ to the body's environment so that the bound variable id is available in the body's scope (line 11). Finally, we return the body's type by recursively calling TYPECHECK (line 12).

Lines 13-22 handle the if syntax. When an if statement is encountered, the environment splitting is a little more difficult. For example, consider the case where one branch uses a linear variable and the other does not; clearly, this would violate the language's guarantees. Thus, we must ensure that both branches use all linear variables that are not assigned to the

Algorithm 3 Type Checker Implementation for Selected Constructs.

```
1:  function TYPECHECK(Γ, e)
2:      complete ← SPLIT(Γ, e, ())
3:      match e with
4:          case x ⇒
5:              if complete is not a type error then
6:                  return Γ(x)
7:              else type error
8:          case let id := rhs in body ⇒
9:              Γ_rhs, Γ_body ← SPLIT(Γ, rhs, body)
10:             τ_rhs ← TYPECHECK(Γ_rhs, rhs)
11:             Γ'_body ← Γ_body, id : τ_rhs
12:             return TYPECHECK(Γ'_body, body)
13:         case if cond then thenBranch else elseBranch ⇒
14:             Γ_cond,0, Γ_then ← SPLIT(Γ, cond, thenBranch)
15:             Γ_cond,1, Γ_else ← SPLIT(Γ, cond, elseBranch)
16:             if Γ_cond,0 = Γ_cond,1 ∧ TYPECHECK(Γ_cond,0, cond) = 'Bool' then
17:                 τ_then ← TYPECHECK(Γ_then, thenBranch)
18:                 τ_else ← TYPECHECK(Γ_then, elseBranch)  ▷ Must both typecheck in Γ_then
19:                 if τ_then = τ_else then
20:                     return τ_then
21:                 else type error
22:             else type error
23:         case f(args...) ⇒
24:             match Γ(f) with
25:                 case (τ_expected ...) → τ' ⇒
26:                     Γ_args ← SPLIT_SEQUENCE(Γ, args)
27:                     τ_args ← {TYPECHECK(Γ_args,i, args_i) | i ∈ 0..LENGTH(args)}
28:                     if τ_args = τ_expected then
29:                         return τ'
30:                     else type error
31:                 otherwise ⇒ type error
32:             end match
33:         case async e ⇒
34:             if TYPECHECK(Γ, e) = 'Unit' then
35:                 return 'Unit'
36:             else type error
37:         ...   ▷ Other cases have been omitted for brevity, but follow a similar approach
38:     end match
```

condition. We begin by splitting the environment between *cond* and *thenBranch*, and *cond* and *elseBranch*, respectively (lines 14-15). This leaves us with two valid environments for *cond* and guarantees that the two branches are split in such a way that they cannot drop any linear variables. We then check whether the two *cond* environments are identical and that *cond* evaluates to a boolean (line 16). If not, we throw a type error (line 22). Next, we check that the two branches evaluate to the same type (line 19); if so, we return that type (line 20) and if not, we throw a type error (line 21).

Lines 23-32 handle function calls. Note that in contrast to the typing rules presented earlier, functions here are N-ary. As a result, *args* is a list of N arguments to the function f. We begin by looking up the type of f in Γ to confirm it is indeed a function, calling its argument types $\tau_{expected}$ and return type τ' (lines 24-25). Whenever f cannot be found in Γ or it is not a function type, we throw a type error (line 31). To type check the arguments, we need each to have its own environment. To do so, we call the SPLIT_SEQUENCE function on *args* to split Γ for each argument (line 26). Then, we map over each argument and recursively call TYPECHECK on it with the corresponding environment, storing the results in τ_{args} (line 27). When τ_{args} matches $\tau_{expected}$, we return the function's return type τ' (lines 28-29); otherwise, we throw a type error (30).

Lines 33-36 handle the async syntax. Because async only contains a single sub-expression, knowing that the sub-expression is well-typed is enough to ensure that the entire async expression is also well-typed; thus we do not need to check the complete property. Given an expression e to run asynchronously, we simply TYPECHECK it in Γ and verify that it evaluates to the unit type (line 34). If so, we can return the unit type for the entire async expression (line 35). Otherwise, we signal a type error (line 36).

The various other cases are left out for brevity, as they use the same techniques demonstrated in the above cases.

Since the TYPECHECK function visits each syntax node exactly once, and at each syntax node performs a SPLIT operation, the time complexity is a linear function of the cost of type checking and the cost of splitting. The complexity of splitting is proportional to the size of the environment (i.e. the number of variables in scope) and the time complexity of the free function. With memoization, it is possible to create an amortized $O(1)$ implementation of the free function since the same nodes are repeatedly visited (the worst-case – an unvisited node – being linear with respect to the size of the subtree). Thus, we believe that with an efficient implementation of free the amortized worst-case time complexity of TYPECHECK is $O(|\Gamma| \times |e|)$, where Γ is the environment and e is the syntax tree.

5 Evaluation

We evaluate our compiler implementation on two separate metrics. First, we measured the speed at which a number of test programs of various sizes could be type checked. This was important to evaluate *how practical* opting into such a type system would be. Second, we conduct a case study on the language's ability to catch bugs. This is done by translating a number of JavaScript programs containing promise-related bugs. In performing this case study, we are able to evaluate *how useful* opting into such a type system would be.

5.1 Type Checker Performance

While a theoretical worst-case time complexity was calculated for the type checker, we were also interested in its the real-world performance. To estimate the scalability of the type checking algorithm in realistic conditions, we created several synthetic test programs that

Table 1 Compile time evaluation.

Program	Lines of Code	Type Checking μs per run[1)]	Full Compile μs per run[1)]	% time in type checker[2)]
Infer	5	6.5	9.1	71.4%
Strings	6	26.3	28.8	91.3%
State	12	25.3	28.6	88.5%
Square	26	58.9	68.6	85.9%
Useful	29	59.5	64.8	91.8%
Basic	34	57.1	67.2	85.0%
Cppreference	38	76.9	109.6	70.2%
IO	45	189.5	211.6	89.6%
Long	231	1107.0	1359.8	81.4%

1) Averaged over 1000 runs.
2) The compiler performs a straight-shot translation to source-level Java, and thus requires very little time to generate code. Type-checking therefore comprises the bulk of the work.

demonstrated various language constructs (available in the supplementary materials). These programs ranged in length from 5 lines to 231 lines, and ranged in complexity from simple tests to programs that implemented buffered, asynchronous reading and parsing of files. To measure the performance, we created a benchmarking mode in the compiler. This mode allowed us to measure runtime for 1000 runs of both the type checker and the entire compiler pipeline. Care was taken to time *only* the operation in question, so that for example parsing time would not contribute to the type checking benchmark and the overhead of file I/O syscalls would never contribute to the full pipeline benchmark. Thus, all necessary input was pre-computed to allow for a simple loop of the runs to be timed together.

Table 1 shows the average runtime for type checking and compiling each program, as well as the proportion of the total compile time spent type checking. The benchmarks were run in Windows 10 on an Intel i5-7300U CPU clocked to 2.71GHz with 8GB of RAM.

In general, we find that the results in Table 1 are empirically consistent with an approximately linear average time complexity with respect to the length of the program. However, variance does exist due to the actual factors (number of syntax nodes and variables in scope) not necessarily scaling proportionally to the overall length of programs. For example, consider the worst-case for our type checker: a program composed of one or more extremely long functions, which introduce hundreds of variables into scope. In such a program, we might expect approximately quadratic time with respect to the length of the program due to the cost of splitting the environment at each syntax node. Overall, we believe that our type checker is sufficiently performant for most real-world use cases, as our benchmarks show that the performance typically exceeds the worst-case time complexity.

5.2 Case Study

Madsen et al. [15] perform a case study of common promise-based bugs in JavaScript using programs from StackOverflow. To evaluate our compiler, we attempted to port these programs to our language; while not all features were possible, we tried to capture the general behavior of each program. For example, whenever callbacks were registered as event listeners, we replaced these with asynchronous infinite loops that would conditionally call a function representing the callback. A number of programs could not be translated to our language, as they relied on various JavaScript features that could not be emulated. Several programs used

Table 2 Case study of JavaScript promise bugs posted to StackOverflow.

StackOverflow ID	Type of Bug	Detected?
41268953	Omitted Write	✓
41488363	Omitted Write	✓
42304958	Double Write	✓
42551854	Double Write	✓
42672914	Omitted Write	✓
42777771	Double Write	✓
29210234	Fork in Promise Chain	
41699046	Missing Return in `.then`	✓
41764399	`.then` replaces error	
42163367	Fork in Promise Chain	
42408234	Missing Return leads to Omitted Write	✓
42577647	Failure to Return Promise	✓
42719050	Missing Return leads to Omitted Write	✓
42788603	Missing Return leads to Omitted Write	✓
42828856	Missing Return leads to Omitted Write	✓

exceptions, which our language does not support. Several other programs passed incorrect arguments to the `.then()` method, making their semantics nonsensical (e.g. using a promise value when a function was expected). The source code of the successfully translated programs is included in the `stackoverflow/` directory of the supplementary materials.

Madsen et al. [15] identify 6 out of 21 programs as either omitted writes or double writes, though 4 other programs also exhibit omitted writes indirectly. The additional omitted writes were all caused by a missing return value in the `.then()` method; the callback for this method is expected to return a new value for the promise, which was missing in several programs. Table 2 shows the complete set of tested questions (sourced from the original case study [15]). Since we have proven that these bugs cannot exist in our language (Corollaries 11 and 12), we expected that these programs would not type check.

Listing 4 Translation of StackOverflow question 42777771 with corresponding error.

```
1 func doIt(): Promise(String) begin
2   let numKeys = 1 in
3   promise p, resolve: String in
4   resolve <- "resolve called!";
5   for key = 0 to numKeys begin
6     resolve <- "inside resolve called"
7   end;
8   p
9 end
```

```
Cannot reuse linear variables [resolve: Promise*(String)] in both
  resolve <- "resolve called!"
and
  for key = 0 to numKeys begin
    resolve <- "inside resolve called"
  end;
  p
```

All 10 programs exhibiting omitted or double writes failed to compile due to a linearity error under our type checker (e.g. Listing 4). Two other programs failed through basic type errors, due to type mismatches between the branches of if statements.

6 Related Work

6.1 Program Analysis

In the past few decades, numerous approaches have been devised for program analysis [8]. Empirical studies have demonstrated the widespread use of program analysis tools in industrial software engineering [6]. Program analysis techniques can be divided into two classes: dynamic analysis, which occurs during program execution, and static analysis, which occurs without executing the program. While dynamic analysis techniques are generally able to make more precise conclusions due to additional information available only at runtime, their results cannot be generalized to program paths that are not executed in testing. In contrast, static analysis often provides more conservative (and sound) results, usually with fewer or no false negatives [7]. Christakis and Bird [6] observe that "60% [of survey respondents] reported that they would accept a slower analyzer if it captured more issues (fewer false negatives)" and "50.7% felt that finding more real issues was worth the cost of dealing with false positives", showing that programmers may often find value in the more conservative nature of static analysis. In the same study, approximately one third of respondents stated that they would like program analyzers to detect concurrency bugs.

A large body of research has been conducted into dynamic analysis for concurrency bugs [22, 18, 1, 17]. While these works offer promising results in precisely finding concurrency-related bugs, dynamic analysis is not suited for all use-cases. Specifically, the inherent lack of soundness guarantees for dynamic analysis could lead to too many false negatives in rarely executed code paths. Likewise, in performance-critical environments, the runtime or memory overhead of performing dynamic analysis may not be acceptable.

Voss and Sarkar [22] describe a promise ownership policy in which ownership of a promise by a task denotes the task's responsibility to fulfill the promise or transfer its ownership. This policy is utilized to dynamically detect two classes of bugs: omitted writes and deadlocks. Our static promise ownership model draws on this design, encoding the same ownership relation as a compile-time property. In doing so, we are able to translate the dynamic detection of omitted writes employed by Voss and Sarkar into a compile-time guarantee – precluding false negatives. Further, although the algorithm described by Voss and Sarkar requires syntactic annotations indicating the transfer of promise ownership between tasks, our type system statically infers the same information. As such, our type checking algorithm makes it possible to remove these annotations, thus lowering the user cost of enabling online deadlock detection for programs written in our language.

Two major camps currently exist for static analysis. On the one hand, many static analysis tools are built-in to compilers, often in the form of type systems. These tools generally offer a streamlined approach to analyzing programs as a result of their deep integration into the language. On the other hand, third-party static analysis tools often exist as standalone programs or frameworks meant to augment the guarantees of the language itself. These are often able to provide a valuable addition to large/existing code bases, in that these analyses can easily be retrofit into existing environments. At the same time, because of their nature as language extensions, they may only be able to analyze a subset of programs (leading to situations where errors can leak in through dependencies).

In terms of static analysis for concurrent programs, much of the research has focused on the use of advanced type systems. Boyapati et al. [3] present an extension to Java that encodes a partial ordering of locks using ownership types. This work provides promising results, including preventing all data-races and deadlocks. This work predates the widespread adoption of promises as a concurrency abstraction, and thus does not offer a similarly high-level abstraction for writing concurrent code. The need for annotations also complicates adoption for such a system. Westbrook et al. [25] introduce an extension to Java utilizing fractional permissions to statically prevent data races. This provides analysis using only minor modifications over normal Java programs and uses gradual typing to facilitate an easier translation process. This differs from our work in that it does not allow for the compile-time detection of the promise-based bugs our language prevents. Carbone and Montesi [4] present a deadlock-free type system based on multiparty session types. This also provides promising results, as the language is able to statically prevent a large class of bugs. As with our own work, this language is not based on existing mainstream programming languages. In contrast to our language, however, programming begins with a global specification of the program's behavior. This can then allow for projection of the global protocol to various programming languages to implement the program's business logic, but due to the development methodology requires a new approach to designing concurrent programs.

Ábrahám et al. [27] introduce a programming language that uses affine types for promises. This is somewhat similar to our own approach of using linear types, but does not require the programmer to fulfill all promises along each execution path (because affine types allow for weakening, i.e. an affine variable can be left unused). This prevents the language from statically identifying omitted writes, and by extension means that programs without any deadlocks can still get stuck. The language also disallows promises contained in heap-allocated data, a restriction which we have loosened in our type system. A final difference is that we introduce a deterministic and efficient type checking algorithm for our type system, which could aid in the implementation of such a type system.

Niehren et al. [16] present a linear type system and semantics for a lambda calculus with promises, $\lambda(fut)$. We follow a similar model in describing a type system that makes promise handles linear. Both systems guarantee similar properties about the safety of promises, such as all promises being fulfilled exactly once. However, $\lambda(fut)$'s linear type system is not intended to be used for programming. It is a proof system that one may apply to an existing program to verify correctness properties. This fundamentally differs from our language, in which the use of linear types in the program itself enables the compiler to immediately verify these correctness properties and enables users to build on language-defined abstractions using their safety guarantees. Beyond that, by diverging from the λ-calculus basis of $\lambda(fut)$, we provide a language more akin to mainstream programming languages. Consequently, we believe that we have designed a linear type system that is more intuitive for users. In practice, this means that users are better able to translate conventional code without having to think hard about how to represent it. Additionally, our introduction of a decidable and efficient type checking algorithm creates a simpler path for adoption.

As compared to related works using type systems to prevent concurrency bugs [3, 25, 4, 27, 16], we believe that our language strikes a desirable balance between powerful high-level abstractions, an intuitive programming model, and strong compile-time guarantees. Due to the basis of promises as the core concurrency primitive, we believe that translation of existing promise-based programs to our language would be relatively straight-forward. Based on the familiar programming model (as compared to other general purpose languages) and the lack of complex typing rules/annotations, we also believe that our type system could be easier to

pick up than similar languages. This is especially true with the introduction of substructural typing into mainstream programming languages via Rust, which has shown that similar type systems can be easily employed for general-purpose programming tasks [24].

6.2 Semantics for Concurrent Programs

Various works have explored formalizing the operational semantics of concurrent and/or promise-based programs [15, 13, 16, 25]. Though other works have similar goals in defining their semantics, we find that none mapped perfectly to the goals of this language.

The semantics of $\lambda(fut)$ share many similarities with our own semantics [16]. For example, our semantics models tasks and parallelism in a similar way to $\lambda(fut)$ and builds on similar concurrency constructs, such as promises and forking. In contrast, we believe that our semantics better fit the characteristics of promises in most contemporary programming languages. Specifically, $\lambda(fut)$'s semantics perform promise reads implicitly and only by need: if the value of a promise is never used, then the program will not await the promise before continuing. Such a system could lead to confusing semantics for users unfamiliar with non-strict evaluation and, more importantly, cannot be retrofit onto languages with strict evaluation schemes such as C++, Java, or JavaScript. Thus, we believe that our semantics serve a more appropriate basis for implementation in popular programming languages.

Lee and Palsberg [13] present operational semantics for Featherweight X10, which models threads as a tree of forked tasks. This design fits very well into our work and we drew upon these semantics as inspiration. In contrast to the similar operational semantics of Featherweight X10, however, our rules allow for the free use of async in any expression (via our use of contextual semantics to substitute threads in the tree). This is advantageous in that it generalizes the language so that spawning asynchronous tasks may occur from any expression without restricting us to a linear instruction-based program. As an example, one could not spawn an asynchronous task as part of the condition for an if expression or the right-hand side of a let expression in Featherweight X10.

Westbrook et al. [25] present another operational semantics for concurrent programs. The operational semantics introduces special tracking of heap values and their associated permissions. This design was inspirational in terms of tracking promises and their owners in our operational semantics. However, these semantics build heavily on notions such as permissions, which were not included in our language. Additionally, both the semantics of Featherweight X10 [13] and Westbrook et al. [25] are based on async-finish parallelism, which did not map well to our language due to the lack of a finish construct.

Madsen et al. [15] focus more heavily on promise-based concurrency, defining a language λ_p that formalizes the semantics of JavaScript's promises. Our semantics share many similarities with those of λ_p, such as drawing on a similar model for tracking promise states and describing many of the same classes of promise-based bugs using our semantics. Due to the single-threaded nature of JavaScript, however, JavaScript-style promises rely on *reactions* to events (such as fulfillment or rejection of a promise) rather than distinct, parallel threads of execution. Since our language is inherently parallel and does not feature promise reactions, we could not reuse the same semantics to describe all possible programs in our language.

7 Conclusion & Future Work

Promises are a powerful structured concurrency primitive, which are increasingly being used in modern programming languages. While they represent a huge step forward from

unstructured concurrency, promises can still contribute to their own fair share of bugs. To address this:

1. We have introduced a complete language featuring a novel type system and operational semantics for promise-based programming.
2. Utilizing a linear type system in which promises must be fulfilled exactly once, our language precludes several classes of promise-based bugs – omitted writes in terminating programs, double writes, and unowned writes – with no runtime overhead.
3. Further, by integrating the results directly into the language rather than an external proof system, the language offers the advantages of first-class support for this style of programming. For example, as in other advanced type systems (e.g. Rust's substructural type system [24]), all libraries included in a program satisfy the language's guarantees using no extra instrumentation or annotations. This is in contrast to gradual typing systems – such as Flow [5] and TypeScript [19] – that can only guarantee safety properties for the subset of code that features the proper annotations.
4. With an efficient implementation of the type checker (linearly proportional to the size of the program times the number of variables in scope), this would enable quick verification of large programs. In doing so, a large swathe of promise-related bugs normally only found at runtime will *always* be caught before the program is ever run.

In future work, this language could be extended with several features – such as closures, generics, and arrays – in order to incorporate its design into existing languages like Java or C++. Further, developing a gradual type system based on our approach may serve to streamline integration into gradually typed languages such as TypeScript. Due to the reliance on promises as a core abstraction in JavaScript and TypeScript, this type system could have far reaching effects in the JavaScript/TypeScript ecosystem. Likewise, Haskell could serve as an interesting target for implementation with its recent addition of a linear typing extension.

The role that a type system with owned promises can serve in statically detecting deadlocks remains an open question. Through the availability of more information on promise ownership and thread relationships at compile-time, it may be possible to identify deadlocks early via minor extensions to the type system. Finally, while data race freedom has not been proven for the language, we believe that data races are likely impossible given our design.

We believe that our language provides a strong foundation for designing safer promise abstractions in both novel and existing programming languages. Due to a decidable and efficient type checking algorithm, we believe that the required analysis can be performed at sufficient speeds for real-world programs. Further, we believe that accessing these new guarantees is not overly burdensome to users, as no extra annotations are required to enforce promise ownership. With a similar programming model to promises in C++, we believe that our language provides the proper abstractions to write real-world code. Finally, we believe that, with minimal additions to our type system, it is feasible to extend various *existing* languages in order to adopt the same guarantees.

References

1 Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding broken promises in asynchronous javascript programs. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276532.

2 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158093.

3 Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, page 211–230, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/582419.582440.

4 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: Multiparty asynchronous global programming. *SIGPLAN Not.*, 48(1):263–274, January 2013. doi:10.1145/2480359.2429101.

5 Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for JavaScript. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi:10.1145/3133872.

6 Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 332–343, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2970276.2970347.

7 Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

8 R. E. Fairley. Tutorial: Static analysis and dynamic testing of computer software. *Computer*, 11(4):14–23, 1978. doi:10.1109/C-M.1978.218132.

9 Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, and Huu-Phuc Vo. Forward to a promising future. In Giovanna Di Marzo Serugendo and Michele Loreti, editors, *Coordination Models and Languages*, pages 162–180, Cham, 2018. Springer International Publishing.

10 Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen, and Tobias Wrigstad. Godot: All the Benefits of Implicit and Explicit Futures. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:28, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2019.2.

11 Richard C. Holt. Some deadlock properties of computer systems. *ACM Comput. Surv.*, 4(3):179–196, 1972. doi:10.1145/356603.356607.

12 ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*, pages 1391–1407. International Organization for Standardization, Geneva, Switzerland, fifth edition, 2017. URL: https://www.iso.org/standard/68564.html.

13 Jonathan K. Lee and Jens Palsberg. Featherweight X10: A core calculus for async-finish parallelism. *SIGPLAN Not.*, 45(5):25–36, 2010. doi:10.1145/1837853.1693459.

14 B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.*, 23(7):260–267, 1988. doi:10.1145/960116.54016.

15 Magnus Madsen, Ondřej Lhoták, and Frank Tip. A model for reasoning about JavaScript promises. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017. doi:10.1145/3133910.

16 J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, 2006. doi:10.1016/j.tcs.2006.08.016.

17 Jihyun Park, Byoungju Choi, and Seungyeun Jang. Dynamic analysis method for concurrency bugs in multi-process/multi-thread environments. *International Journal of Parallel Programming*, 48, December 2020. doi:10.1007/s10766-020-00661-3.

18 Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 531–542, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2254064.2254127.

19 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-*

SIGACT Symposium on Principles of Programming Languages, POPL '15, page 167–180, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2676726.2676971.

20 Justin Slepak. Notes on substructural types. URL: http://www.ccs.neu.edu/~jrslepak/substruct-notes.pdf.

21 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bharagavan, and Jean Yang. Secure distributed programming with value-dependent types. Technical Report MSR-TR-2011-37, Microsoft Research, March 2011. This is an extended version of the conference paper (ICFP '11) with the same title. A final version of this full technical report is forthcoming. URL: https://www.microsoft.com/en-us/research/publication/secure-distributed-programming-with-value-dependent-types/.

22 Caleb Voss and Vivek Sarkar. An ownership policy and deadlock detector for promises, 2021. arXiv:2101.01312.

23 P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.

24 Aaron Weiss, Olek Gierczak, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The essence of Rust, 2020. arXiv:1903.00982v3.

25 Edwin Westbrook, Jisheng Zhao, Zoran Budimlić, and Vivek Sarkar. Practical permissions for race-free parallelism. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 614–639, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

26 Dengping Zhu. *To Memory Safety through Proofs*. PhD thesis, Boston University, USA, 2006.

27 Erika Ábrahám, Immo Grabe, Andreas Grüner, and Martin Steffen. Behavioral interface description of an object-oriented language with futures and promises. *The Journal of Logic and Algebraic Programming*, 78(7):491–518, 2009. The 19th Nordic Workshop on Programming Theory (NWPT 2007). doi:10.1016/j.jlap.2009.01.001.