# Witnessing Side-Effects *

Tachio Terauchi
EECS Department
University of California, Berkeley
tachio@cs.berkeley.edu

Alex Aiken
Computer Science Department
Stanford University
aiken@cs.stanford.edu

## Abstract

We present a new approach to the old problem of adding side effects to purely functional languages. Our idea is to extend the language with "witnesses," which is based on an arguably more pragmatic motivation than past approaches. We give a semantic condition for correctness and prove it is sufficient. We also give a static checking algorithm that makes use of a network flow property equivalent to the semantic condition.

***Categories and Subject Descriptors*** D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features

***General Terms*** Design, Languages, Theory

***Keywords*** Functional languages, side-effects

## 1. Introduction

Adding side-effects to a purely functional language is a well-known problem with a number of solutions [7, 9, 11, 15, 6, 1] with monads [10, 11, 7, 9] being arguably the most popular. In this paper, we propose a new approach to this old problem by attacking it from a different angle. Instead of starting from a language theoretic point of view, we start by introducing a simple programming feature called *witnesses* so that programs can explicitly order side-effects. This feature is motivated by a pragmatic observation and is straightforward. The catch is that, because it is so simple, it actually does not guarantee that a program is *correct* (i.e., that it can be viewed as a purely functional program). Instead, we argue that the feature makes it easy for programmers to write correct programs. We then formally state a natural semantic condition that is sufficient to guarantee correctness and give a static checking algorithm. The result is a new framework for guaranteeing correctness of side-effects in purely functional programs.

Besides arguably being more intuitive to programmers, our approach is more expressive than previous approaches. In particular, our approach does not force side-effects to occur in a sequential order. For example, a program is allowed to read from a reference cell in two unordered contexts as well as write to two different cells in two unordered contexts.

Besides providing new insights into the old problem of fitting side effects into functional languages for conventional von Neumann architectures, our work is motivated by the emergence of commercial parallel computer architectures (e.g., chip-multiprocessors or "multi-core" chips) that encourage parallel programming. It is well-known that the "explicit dependence" property of functional languages makes parallelization easier for both programmers and compilers.[1] However, frequent use of side-effects, namely manual destructive memory/resource updates, are believed to be important for programming high-performance parallel applications in practice. Hence a functional way to add side-effects without imposing parallelism-destroying sequentiality may be of practical interest for exploiting parallelism within these new architectures.

### 1.1 Contributions and Overview

This paper makes the following contributions:

- A simple language feature called *witnesses* that can be used to order side-effects. (Section 2)
- A semantic condition called *witness race freedom* for correct usage of witnesses and a proof of its sufficiency. (Section 3)
- An automatic algorithm for checking the afore-mentioned semantic condition that makes use of a network flow property. (Section 4)

The semantic condition is intuitive in the sense that it is directly motivated by the implications of writing race-free programs. The automatic algorithm is derived as a type inference algorithm for a substructural type system. The type system and its inference problem are somewhat subtle and interesting in their own right. Section 5 discusses related work. Section 6 concludes.

## 2. Preliminaries

We need a precise definition of what it means for side-effects to be "correct" within a functional language. A helpful idea is to show that a program's semantics is independent of a class of "functional" program transformation rules. However, there is no consensus on the right set of transformations. For example, the transformation ($\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e'$) $\equiv$ ($e'[x/e]$) for $x \notin \mathit{free}(e)$ is not always true in systems based on linear-types. (Here $\mathit{free}(e)$ is the set of free variables of $e$.)

To define correctness, we fix a set of program transformations expressive enough to model different functional reduction strate-

* This research was supported in part by NASA Grant No. NNA04CI57A; NSF Grant Nos. CCR-0234689, CCR-0085949, and CCR-0326577. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*ICFP'05* September 26–28, 2005, Tallinn, Estonia.
Copyright © 2005 ACM 1-59593-064-7/05/0009...$5.00.

[1] But not easy, since there are other challenging issues such as selecting the right granularity of parallelism, but these issues apply equally to other languages and many solutions such as data-parallel operators and thread annotations already exist.

$$
\begin{array}{rcl}
e & := & x \mid i \mid \lambda x.e \mid e\ e' \mid \texttt{let}\ x = e\ \texttt{in}\ e' \mid e \otimes e' \mid \pi_i(e) \\
& \mid & \texttt{write}\ e_1\ e_2\ e_3 \mid \texttt{read}\ e\ e' \mid \texttt{ref}\ e \mid \texttt{join}\ e\ e' \mid \bullet
\end{array}
$$

Figure 1: The syntax of the language $\lambda_{wit}$.

gies, including call-by-value, call-by-need (i.e., lazy-evaluation), and parallel evaluation.[2] So, for example, a program that is invariant under this set of transformations evaluates to the same result regardless of whether the evaluation order is call-by-value or call-by-need. In parallel evaluation, invariance implies that a program is deterministic under any evaluation schedule.

For ease of exposition, we include the afore-mentioned program transformations directly in the semantics as non-deterministic reduction rules. The correctness criteria then reduces to showing that a program is confluent with respect to this semantics. Also, for the purpose of exposition, we restrict side-effects to imperative operations on first class references.

Figure 1 gives the syntax of $\lambda_{wit}$, a simple functional language with side-effects and witnesses. Note $\lambda_{wit}$ has the usual features of a functional language: variables $x$, integers $i$, function abstractions $\lambda x.e$, function applications $e\ e'$, variable bindings $\texttt{let}\ x = e\ \texttt{in}\ e'$, pairs $e \otimes e'$, and projections $\pi_i(e)$ where $i = 1$ or $i = 2$. Bindings $\texttt{let}\ x = e\ \texttt{in}\ e'$ can be recursive, i.e., $x \in \mathit{free}(e)$. Three expression kinds work with references: reference writes $\texttt{write}\ e_1\ e_2\ e_3$, reference reads $\texttt{read}\ e\ e'$, and reference creations $\texttt{ref}\ e$. A read $\texttt{read}\ e\ e'$ has a *witness* parameter $e'$ along with a reference parameter $e$ such that it does not read the reference $e$ until it sees the witness $e'$. (Section 3 defines the formal meaning of "seeing the witness.") Similarly, a write $\texttt{write}\ e_1\ e_2\ e_3$ writes the expression $e_2$ to the reference $e_1$ after it sees the witness $e_3$. After completion of the read, $\texttt{read}\ e\ e'$ returns a pair of the read value and a witness. Similarly, $\texttt{write}\ e_1\ e_2\ e_3$ returns a witness after the write. In general, any side-effect primitive returns a witness of performing the corresponding side-effect; in the case of $\lambda_{wit}$, the side-effect primitives are just $\texttt{write}\ e_1\ e_2\ e_3$ and $\texttt{read}\ e\ e'$.

Before describing the formal semantics of $\lambda_{wit}$, we describe novel properties of $\lambda_{wit}$ informally by examples.

Programs in $\lambda_{wit}$ can use witnesses to order side-effects. For example, the following program returns 2 regardless of the evaluation order because the read requires a witness of the write:

$$\texttt{let}\ x = (\texttt{ref}\ 1)\ \texttt{in}\ \texttt{let}\ w = (\texttt{write}\ x\ 2\ \bullet)\ \texttt{in}\ \texttt{read}\ x\ w$$

(The symbol $\bullet$ is used for dummy witnesses.) On the other hand, $\lambda_{wit}$ does not guarantee correctness. For example, the following $\lambda_{wit}$ program has no ordering between the read and the write and hence may return 1 or 2 depending on the evaluation order:

$$\texttt{let}\ x = (\texttt{ref}\ 1)\ \texttt{in}\ \texttt{let}\ w = (\texttt{write}\ x\ 2\ \bullet)\ \texttt{in}\ \texttt{read}\ x\ \bullet$$

The expression kind $\texttt{join}\ e\ e'$ joins two witnesses by waiting until it sees the witness $e$ and the witness $e'$ and returning a witness. For example, the following program returns the pair $1 \otimes 1$ regardless of the evaluation order because the write waits until it sees witnesses of both reads:

$$
\begin{array}{l}
\texttt{let}\ x = (\texttt{ref}\ 1)\ \texttt{in} \\
\quad \texttt{let}\ y = (\texttt{read}\ x\ \bullet)\ \texttt{in}\ \texttt{let}\ z = (\texttt{read}\ x\ \bullet)\ \texttt{in} \\
\quad\quad \texttt{let}\ w = (\texttt{write}\ x\ 2\ (\texttt{join}\ \pi_2(y)\ \pi_2(z)))\ \texttt{in} \\
\quad\quad\quad \pi_1(y) \otimes \pi_1(z)
\end{array}
$$

Note that the two reads may be evaluated in any order. In general, witnesses are first class values and hence they can be passed to and returned from a function, captured in function closures, and even written to and read from a reference. Witnesses are a simple feature that can be used to order side-effects in a straightforward manner.

[2] The results in Section 3 are general enough for most other functional transformations too, but the static checking algorithm requires a more stringent definition.

$$
\begin{array}{rcl}
E & := & D \cup \{a \mapsto E\} \mid [\,] \mid E\ e \mid e\ E \mid E \otimes e \mid e \otimes E \\
& \mid & \pi_i(E) \mid \texttt{write}\ E\ e\ e' \mid \texttt{write}\ e\ E\ e' \mid \texttt{write}\ e\ e'\ E \\
& \mid & \texttt{read}\ e\ E \mid \texttt{read}\ E\ e \mid \texttt{ref}\ E \mid \texttt{join}\ E\ e \mid \texttt{join}\ e\ E
\end{array}
$$

$$
\begin{array}{ll}
\textbf{App} & (S, E[(\lambda x.e)\ e']) \Rightarrow (S, E[e[a/x]] \uplus \{a \mapsto e'\}) \\
\textbf{Let} & (S, E[\texttt{let}\ x = e\ \texttt{in}\ e']) \\
& \quad \Rightarrow (S, E[e'[a/x]] \uplus \{a \mapsto e[a/x]\}) \\
\textbf{Pair} & (S, E[\pi_i(e_1 \otimes e_2)]) \Rightarrow (S, E[e_i]) \\
\textbf{Write} & (S, E[\texttt{write}\ \ell\ e\ \bullet]) \Rightarrow (S[\ell \leftarrow a], E[\bullet] \uplus \{a \mapsto e\}) \\
\textbf{Read} & (S, E[\texttt{read}\ \ell\ \bullet]) \Rightarrow (S, E[S(\ell) \otimes \bullet]) \\
\textbf{Ref} & (S, E[\texttt{ref}\ e]) \Rightarrow (S \uplus \{\ell \mapsto a\}, E[\ell] \uplus \{a \mapsto e\}) \\
\textbf{Join} & (S, E[\texttt{join}\ \bullet\ \bullet]) \Rightarrow (S, E[\bullet]) \\
\textbf{Arrive} & (S, E[a] \uplus \{a \mapsto e\}) \Rightarrow (S, E[e] \uplus \{a \mapsto e\}) \\
& \quad \text{where } e \in V \\
\textbf{GC} & (S, D \uplus D') \Rightarrow (S, D) \\
& \quad \text{where } \diamond \notin \mathit{dom}(D') \wedge \mathit{dom}(D') \cap \mathit{free}(D) = \emptyset
\end{array}
$$

Figure 2: The semantics of $\lambda_{wit}$.

In the rest of this section, we describe the semantics of $\lambda_{wit}$ so that we can formally define when a $\lambda_{wit}$ program is correct, i.e., when it is confluent. Figure 2 shows the semantics of $\lambda_{wit}$, which is defined via reduction rules of the form $(S, D) \Rightarrow (S', D')$ where $S, S'$ are *reference stores* and $D, D'$ are *expression stores*. A reference store is a function from a set of *reference locations* $\ell$ to *ports* $a$, and an expression store is a function from a set of ports to expressions. Here, expressions include any expression from the source syntax extended with reference locations and ports. Given a program $e$, evaluation of $e$ starts from the initial state $(\emptyset, \{\diamond \mapsto e\})$ where the symbol $\diamond$ denotes the special *root port*. Ports are used for evaluation sharing.[3] The reduction rules are parametrized by the evaluation contexts $E$. For an expression $e$, $\mathit{free}(e)$ is the set of free variables, ports, and reference locations of $e$. For an expression store $D$, $\mathit{free}(D) = \mathit{dom}(D) \cup \bigcup_{e \in \mathit{ran}(D)} \mathit{free}(e)$.

We briefly describe the reduction rules from top-to-bottom. The rule **App** corresponds to a function application. For functions $F$ and $F'$, $F \uplus F'$ denotes $F \cup F'$ if $\mathit{dom}(F) \cap \mathit{dom}(F') = \emptyset$ and is undefined otherwise. Thus **App** creates a fresh port $a$ and stores $e'$ at $a$. The rule **Let** is similar. The rule **Pair** projects the $i$th element of the pair. The rule **Write** creates a fresh port $a$, stores the expression $e'$ at the port $a$, and stores the port $a$ at the reference location $\ell$. We use $S[\ell \leftarrow a]$ as a shorthand for $\{\ell' \mapsto S(\ell') \mid \ell' \in \mathit{dom}(S) \wedge \ell' \neq \ell\} \cup \{\ell \mapsto a\}$. Note that we use the dummy witness symbol $\bullet$ as the run-time representation of any witness because only its presence is important to the semantics, i.e., operationally, a witness is like a dataflow token in dataflow machines. The rule **Read** reads from the reference location $\ell$ and, as noted above, returns the value paired with a witness. The rule **Ref** creates a fresh location $\ell$ and a fresh port $a$, initializes $a$ to the expression $e$ and $\ell$ to the port $a$. The rule **Join** takes two witnesses and returns one witness.

The rule **Arrive** might look somewhat unfamiliar. Here $V$ is the set of "safe to duplicate" expressions. Partly for the sake of the static-checking algorithm to be presented later, we fix $V$ to *values* generated by the following grammar:

$$v := x \mid i \mid a \mid \bullet \mid \ell \mid v \otimes v' \mid \lambda x.e$$

[3] In the literature, top-level let-bound variables often double as variables and ports.

**Arrive** says that if $e$ is safe to duplicate, then we can replace $a$ by $e$; we say a safe to duplicate expression has *arrived* at port $a$. In essence, while standard operational semantics for functional languages [12, 8] implicitly combine **Arrive** with other rules, we separate **Arrive** for increased freedom in the evaluation order. Lastly, the rule **GC** garbage-collects unreachable (from the root port $\diamond$) portions of the expression store.

Here is an example of a $\lambda_{wit}$ evaluation:

$$
\begin{array}{ll}
(\emptyset, \{\diamond \mapsto (\lambda x.\texttt{read}\, x\, \bullet)\ \texttt{ref}\ 1\}) & \\
\quad \Rightarrow (\{\ell \mapsto a\}, \{\diamond \mapsto (\lambda x.\texttt{read}\, x\, \bullet)\ \ell, a \mapsto 1\}) & \textbf{Ref} \\
\quad \Rightarrow (\{\ell \mapsto a\}, \{\diamond \mapsto \texttt{read}\, a'\, \bullet, a \mapsto 1, a' \mapsto \ell\}) & \textbf{App} \\
\quad \Rightarrow (\{\ell \mapsto a\}, \{\diamond \mapsto \texttt{read}\, \ell\, \bullet, a \mapsto 1, a' \mapsto \ell\}) & \textbf{Arrive} \\
\quad \Rightarrow (\{\ell \mapsto a\}, \{\diamond \mapsto a \otimes \bullet, a \mapsto 1, a' \mapsto \ell\}) & \textbf{Read} \\
\quad \Rightarrow (\{\ell \mapsto a\}, \{\diamond \mapsto 1 \otimes \bullet, a \mapsto 1, a' \mapsto \ell\}) & \textbf{Arrive} \\
\quad \Rightarrow (\{\ell \mapsto a\}, \{\diamond \mapsto 1 \otimes \bullet\}) & \textbf{GC}
\end{array}
$$

The semantics is non-deterministic and therefore also allows other reduction sequences for the same program. For example, we may take an **App** step immediately instead of first creating a new reference by a **Ref** step:

$$
\begin{array}{ll}
(\emptyset, \{\diamond \mapsto (\lambda x.\texttt{read}\, x\, \bullet)\ \texttt{ref}\ 1\}) & \\
\quad \Rightarrow (\emptyset, \{\diamond \mapsto \texttt{read}\, a\, \bullet, a \mapsto \texttt{ref}\ 1\}) & \textbf{App}
\end{array}
$$

Before defining confluence, we point out several important properties of this semantics. Firstly, note that the evaluation contexts $E$ do not extend to subexpressions of a $\lambda$ abstraction $\lambda x.e$, i.e., we do not reduce under $\lambda$. The evaluation contexts also do not extend to subexpressions of an expression $\texttt{let}\ x = e\ \texttt{in}\ e'$, but $e$ and $e'$ may become available for evaluation via applications of the **Let** rule. As with call-by-value evaluation or call-by-need evaluation, evaluation of an expression is shared. For example, in the program $(\lambda x.x \otimes x)\ e$, the expression $e$ is evaluated at most once.

The semantics of $\lambda_{wit}$ has strictly more freedom in evaluation order than both call-by-value and call-by-need. In particular, call-by-need evaluation can be obtained by using the same reduction rules but restricting the evaluation contexts to the following

$$
\begin{array}{rcl}
E & := & D \cup \{\diamond \mapsto E\} \mid [\,] \mid E\ e \mid \pi_i(E) \mid \texttt{write}\, E\, e\, e' \\
 & \mid & \texttt{write}\, \ell\, e\, E \mid \texttt{read}\, E\, e \mid \texttt{read}\, \ell\, E \\
 & \mid & \texttt{join}\, E\, e \mid \texttt{join}\, \bullet\, E
\end{array}
$$

Call-by-value evaluation can be obtained by adding the following contexts to the evaluation contexts of the call-by-need evaluation

$$
\begin{array}{rcl}
E & := & \ldots \mid (\lambda x.e)\ E \mid E \otimes e \mid v \otimes E \mid \texttt{write}\, \ell\, E\, \bullet \\
 & \mid & \texttt{ref}\ E \mid \texttt{let}\ x = E\ \texttt{in}\ e
\end{array}
$$

in addition to restricting the rule **App** to the case $e' \in V$, the rule **Let** to the case $e \in V$ , the rule **Pair** to the case $e_1, e_2 \in V$, the rule **Write** to the case $e' \in V$, and the rule **Ref** to the case $e \in V$.[4] Note that both lazy writes and strict writes are possible in $\lambda_{wit}$.

It is important to understand that we are only concerned with side-effects via references, and hence we are not concerned about issues like the number of ports that are created during an evaluation.

Having defined the semantics, we can formally define when a $\lambda_{wit}$ program is confluent. To this end, we define *observational equivalence* as the smallest reflexive and transitive relation $D \approx D'$ on expression stores satisfying:

- $D \approx D[a/a']$ where $a \notin \mathit{free}(D)$
- $D \approx D[\ell/\ell']$ where $\ell \notin \mathit{free}(D)$

That is, expression stores are observationally equivalent if they are equivalent up to consistent renaming of free ports and reference locations. Let $\Rightarrow^*$ be a sequence of zero or more $\Rightarrow$.

[4] Strictly speaking, the context $\texttt{let}\ x = E\ \texttt{in}\ e$ is not in the semantics of $\lambda_{wit}$. But $\lambda_{wit}$ can simulate the behavior via a **Let** step and then reducing $e[a/x]$ which is now in an evaluation context.

DEFINITION 1 (Confluence). *A program state $(S, D)$ is confluent if for any two states $(S_1, D_1)$ and $(S_2, D_2)$ such that $(S, D) \Rightarrow^* (S_1, D_1)$ and $(S, D) \Rightarrow^* (S_2, D_2)$, there exist two states $(S_1', D_1')$ and $(S_2', D_2')$ such that $(S_1, D_1) \Rightarrow^* (S_1', D_1')$, $(S_2, D_2) \Rightarrow^* (S_2', D_2')$, and $D_1' \approx D_2'$. A program $e$ is confluent if its initial state $(\emptyset, \{\diamond \mapsto e\})$ is confluent.*

Note that the definition does not require any relation between reference location stores $S_1'$ and $S_2'$. So, for example, a program that writes but never reads would be confluent. As shown before, $\lambda_{wit}$ contains programs that are not confluent. Indeed, the difference between call-by-need and call-by-value is enough to demonstrate non-confluence:

$$(\lambda x.\texttt{read}\, x\, \bullet)\ (\texttt{let}\ x = (\texttt{ref}\ 1)\ \texttt{in}\ \texttt{let}\ y = (\texttt{write}\, x\, 2\, \bullet)\ \texttt{in}\ x)$$

The above program evaluates to the pair $1 \otimes \bullet$ under call-by-need and to the pair $2 \otimes \bullet$ under call-by-value. No further reductions can make the two states observationally equivalent. (Here we implicitly read back the top-level expression from the root port instead of showing the actual expression stores for brevity.)

We have shown earlier that witnesses can aid in writing correct programs by directly ordering side-effects. Witnesses are first class values and hence can be treated like other expressions. For example, the program below captures a witness in a function which itself returns a witness to ensure that reads and writes happen in a correct order:

$$
\begin{array}{l}
\texttt{let}\ x = \texttt{ref}\ 1\ \texttt{in} \\
\quad \texttt{let}\ w = \texttt{write}\, x\, 2\, \bullet\ \texttt{in} \\
\quad\quad \texttt{let}\ f = \lambda y.\texttt{read}\, x\, w\ \texttt{in} \\
\quad\quad\quad \texttt{let}\ z = (f\ 0) \otimes (f\ 0)\ \texttt{in} \\
\quad\quad\quad\quad \texttt{let}\ w = \texttt{write}\, x\, 3\ \texttt{join}\, \pi_2(\pi_1(z))\ \pi_2(\pi_2(z))\ \texttt{in}\ z
\end{array}
$$

Note that a witness of the first write is captured in the function $f$. Hence both reads from the two calls to $f$ see a witness of the write. A witness of each read is returned by $f$, and the last write waits until it sees witnesses from both reads. Therefore, the result of the program is $(2 \otimes \bullet) \otimes (2 \otimes \bullet)$ regardless of the evaluation order. Note that the two calls to $f$, and thus the reads in the calls, can occur in either order.

## 3. Witness Race Freedom

As discussed in Section 2, witnesses aid in writing correct programs in the presence of side-effects but do not enforce correctness. In this section, we give a sufficient condition for guaranteeing confluence.

The guideline for writing a correct program should be intuitively clear at this point: we ensure that reads and writes happen in some race-free order by partially ordering them via witnesses. We now make this intuition more precise. First, we formally define what we mean by the phrase " side-effect $A$ sees a witness of side effect $B$" that we have used informally up to this point.

Intuitively, a *trace graph* is a program trace with all information other than reads, writes, and witnesses elided. There are three kinds of nodes: read nodes *read*($\ell$), write nodes *write*($\ell$), and the join node *join*. Read and write nodes are parametrized by a reference location $\ell$. There is a directed edge $(A, B)$ from node $A$ to node $B$ if $B$ directly sees a witness of $A$. A trace graph $(\mathbb{V}, \mathbb{E})$ is

constructed as the program evaluates a modified semantics:

**Write** $(S, E[\texttt{write}\, \ell\, e\, A]) \Rightarrow (S[\ell \leftarrow a], E[B] \uplus \{a \mapsto e\})$
$\mathbb{V} := \mathbb{V} \cup \{B\}$ where $B$ is a new *write*($\ell$) node
$\mathbb{E} := \mathbb{E} \cup \{(A, B)\}$

**Read** $(S, E[\texttt{read}\, \ell\, A]) \Rightarrow (S, E[S(\ell) \otimes B])$
$\mathbb{V} := \mathbb{V} \cup \{B\}$ where $B$ is a new *read*($\ell$) node
$\mathbb{E} := \mathbb{E} \cup \{(A, B)\}$

**Join** $(S, E[\texttt{join}\, A\, B]) \Rightarrow (S, E[C])$
$\mathbb{V} := \mathbb{V} \cup \{C\}$ where $C$ is a new *join* node
$\mathbb{E} := \mathbb{E} \cup \{(A, C), (B, C)\}$

Note that we now use nodes as witnesses instead of $\bullet$. The line below each reduction rule shows the graph update action corresponding to that rule. The other rules remain unmodified and hence have no graph update actions. An evaluation starts with $\mathbb{V} = \mathbb{E} = \emptyset$ and performs the corresponding graph update when taking a **Write** step, a **Read** step, or a **Join** step. Notice that a trace graph and the annotated semantics are only needed to state the semantic condition for correctness and are not needed in the actual execution of a $\lambda_{wit}$ program.

We can now define what it means for a node $A$ to *see a witness* of a node $B$, a notion we have used informally until now.

DEFINITION 2. *Given a trace graph, we say that a node $A$ sees a witness of a node $B$ if there is a path from $B$ to $A$ in the trace graph. We write $B \rightsquigarrow A$.*

The following is a trivial observation:

THEOREM 1. *If $B \rightsquigarrow A$ in a trace graph, then the side effect corresponding to $B$ must have happened before the side effect corresponding to $A$ in the evaluation that generated the trace graph.*

Clearly, any trace graph is acyclic.

Having defined trace graphs and the $\rightsquigarrow$ relation, we are now ready to state the semantic condition for correctness. Before showing the condition formally, we informally motivate it by making analogies to the conventional programming guideline for writing correct concurrent programs: prevent race conditions.

We first note that a program could produce different trace graphs depending on the choice of reductions, even when those trace graphs are from terminating evaluations. Furthermore, it is not necessarily the case that such a program is non-confluent. Therefore, instead of trying to argue about confluence by comparing different trace graphs, we shall define a condition that can be checked by observing each individual trace graph in isolation.

Let us write $A$ : *nodetype* as a shorthand for a node $A$ of type *nodetype*. If we have $A$ : *read*($\ell$) and $B$ : *write*($\ell$), then we want either $A \rightsquigarrow B$ or $B \rightsquigarrow A$ to ensure that $A$ always happens before $B$ or $B$ always happens before $A$ because otherwise we may get a read-write race condition due to non-determinism. Also, for any $A$ : *read*($\ell$), if there are two nodes $B_1, B_2$ : *write*($\ell$) such that neither $B_1 \rightsquigarrow B_2$ nor $B_2 \rightsquigarrow B_1$ (so we do not know which write occurs first) and $A$ could happen after both $B_1$ and $B_2$, then we want $C$ : *write*($\ell$) such that $C \rightsquigarrow A$, $B_1 \rightsquigarrow C$ and $B_2 \rightsquigarrow C$, because otherwise the read at $A$ might depend on whether the evaluation chose to do $B_1$ first or $B_2$ first, i.e., we have another kind of race-condition. Perhaps somewhat surprisingly, satisfying these two conditions turns out to be sufficient to ensure confluence.

We now formalizes this discussion. For $B$ : *write*($\ell$), we use the shorthand $B \overset{!}{\rightsquigarrow} A$ if for any $C$ : *write*($\ell$) such that $C \rightsquigarrow A$ and $C \neq B$, we have $C \rightsquigarrow B$. Now, for any $A$ : *read*($\ell$), there exists at most one $B$ : *write*($\ell$) such that $B \overset{!}{\rightsquigarrow} A$. Note that the second condition above is equivalent to requiring that for any $A$ : *read*($\ell$), either there is no $B$ : *write*($\ell$) such that $B \rightsquigarrow A$ or there is a $B$ : *write*($\ell$) such that $B \overset{!}{\rightsquigarrow} A$

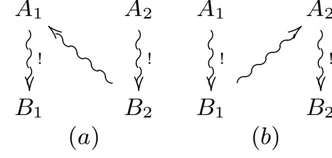


Figure 3: Possible orderings between pairs of reads and writes in a witness race free trace graph.

DEFINITION 3 (Witness Race Freedom). *We say that a trace graph $(\mathbb{V}, \mathbb{E})$ is witness race free if for every location $\ell$,*

- *for every $A$ : read($\ell$) $\in \mathbb{V}$ and $B$ : write($\ell$) $\in \mathbb{V}$, either $A \rightsquigarrow B$ or $B \rightsquigarrow A$, and*
- *for every $A$ : read($\ell$) $\in \mathbb{V}$, either there is no $B$ : write($\ell$) $\in \mathbb{V}$ such that $B \rightsquigarrow A$ or there is a $B$ : write($\ell$) $\in \mathbb{V}$ such that $B \overset{!}{\rightsquigarrow} A$.*

*We say that a program $e$ is witness race free if every trace graph of $e$ is witness race free.*

THEOREM 2. *If $e$ is witness race free then $e$ is confluent.*

The proof appears in our companion technical report [13].

While witness race freedom is a sufficient condition, it is not necessary. For example, if for each reference location writes happen to never change the location's value, then the program is trivially confluent regardless of the order of reads and writes. Another example is a program using implicit order of evaluation: e.g., in $\lambda_{wit}$, expressions are not reduced under $\lambda$ so a function body is evaluated only after a call. Hence a program that stores a function in a reference location, reads the reference location to call the function, and then writes in the same reference location from the body of the called function is confluent because the write always happens after the read despite the write not seeing the witness of the read.

Nevertheless, witness race freedom is "almost complete" in a sense that if the only way to order two side-effects is to make one see a witness of the other, and if we cannot assume anything about what expressions are written and how the contents are used, then it is the weakest condition guaranteeing correctness. In particular, if the trace graphs are the only information available about a program, then witness race freedom becomes a necessary condition.

The result in this section can be extended to most other functional program transformations, because witness race freedom is an entirely semantic condition. However, the static checking algorithm described in Section 4 is not as forgiving, which is why we have restricted the set of program transformations to that of the semantics of $\lambda_{wit}$. For example, the algorithm is unsound for general call-by-name reduction.

## 4. Types for Statically Checking Witness Race Freedom

While the concept of witnesses is straightforward, it may nevertheless be desirable to have an automated way of checking whether an arbitrary $\lambda_{wit}$ program is witness race free. Witness race freedom may be checked directly by checking every program trace, which is computationally infeasible. Instead, we exploit a special property of witness-race-free trace graphs to design a sound algorithm that can efficiently verify a large subset of witness-race-free $\lambda_{wit}$ programs.

The key observation is that any witness-race-free trace graph contains for each reference location $\ell$ a subgraph that we shall call

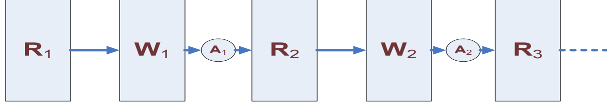


Figure 4: A read-write pipeline with bottlenecks for a reference location $\ell$.

a *read-write pipeline with bottlenecks*. We shall design an algorithm that detects these subgraphs instead of directly checking the witness race freedom condition. Consider a witness-race-free trace graph. Suppose there are $A_1, A_2$ : *write*$(\ell)$ and $B_1, B_2$ : *read*$(\ell)$ such that $A_1 \neq A_2$, $A_1 \overset{!}{\rightsquigarrow} B_1$ and $A_2 \overset{!}{\rightsquigarrow} B_2$. Due to witness race freedom, it must be the case that $B_2 \rightsquigarrow A_1$ or $A_1 \rightsquigarrow B_2$. If the former is the case, we have the relation as depicted in Figure 3 (a). Suppose that the latter is the case. Then, since $A_2 \overset{!}{\rightsquigarrow} B_2$, it must be the case that $A_1 \rightsquigarrow A_2$. Consider $A_2$ and $B_1$. Due to witness race freedom again, it must be the case that either $A_2 \rightsquigarrow B_1$ or $B_1 \rightsquigarrow A_2$. But if $A_2 \rightsquigarrow B_1$, then since $A_1 \overset{!}{\rightsquigarrow} B_1$, it must be the case that $A_2 \rightsquigarrow A_1$. But this is impossible since $A_2 \rightsquigarrow A_1 \rightsquigarrow A_2$ forms a cycle. So it must be the case that $B_1 \rightsquigarrow A_2$, and we have the relation as depicted in Figure 3 (b).

Further reasoning along this line of thought reveals that for a witness-race-free trace graph, for any reference location $\ell$, the nodes in the set $X = \{A : \mathit{write}(\ell) \mid \exists B : \mathit{read}(\ell). A \overset{!}{\rightsquigarrow} B\}$ are totally ordered (with $\rightsquigarrow$ as the ordering relation), and that these nodes partition all *read*$(\ell)$ nodes and *write*$(\ell)$ nodes in a way depicted in Figure 4 where $X = \{A_1, \ldots, A_n\}$. In the figure, each $\mathbb{R}_i$ and $\mathbb{W}_i$ is a collection of nodes. No $\mathbb{R}_i$ contains a *write*$(\ell)$ node and no $\mathbb{W}_i$ contains a *read*$(\ell)$ nodes. Each $A_i$ is one *write*$(\ell)$ node. An arrow from $X$ to $Y$ means that there is a path from each *write*$(\ell)$ node or *read*$(\ell)$ node in $X$ to each *write*$(\ell)$ node or *read*$(\ell)$ node in $Y$, except if a $\mathbb{W}_i$ contains no such node, then there is a path from each *read*$(\ell) \in \mathbb{R}_i$ to $A_i$. Each $\mathbb{R}_i$ for $i \neq 1$ must contain at least one *read*$(\ell)$. Arrows just imply the presence of paths, and hence there can be more paths than the ones implied by the arrows, e.g., paths to/from nodes that are not in the diagram, paths to and from nodes in the same collection, and even paths relating the collections in the diagram such as one that goes directly from $\mathbb{R}_i$ to $A_i$, bypassing $\mathbb{W}_i$.

The graph in Figure 4 can be described formally as a subgraph of the trace graph satisfying certain properties.

DEFINITION 4. *Given a trace graph $(\mathbb{V}, \mathbb{E})$ and a reference location $\ell$, we call its subgraph $G_\ell$ a read-write pipeline with bottlenecks if $G_\ell$ consists of collections of nodes $\mathbb{R}_1, \mathbb{R}_2, \ldots, \mathbb{R}_n$ and $\mathbb{W}_1, \mathbb{W}_2, \ldots, \mathbb{W}_n$ with the following properties:*

- $\{A \mid A : \mathit{read}(\ell) \in \mathbb{V}\} \subseteq \bigcup_{i=1}^{n} \mathbb{R}_i$,
- $\{A \mid A : \mathit{write}(\ell) \in \mathbb{V}\} \subseteq \bigcup_{i=1}^{n} \mathbb{W}_i$,
- $\mathbb{R}_1, \ldots, \mathbb{R}_n, \mathbb{W}_1, \ldots, \mathbb{W}_n$, *restricted to write$(\ell)$ nodes and read$(\ell)$ nodes are pairwise disjoint,*
- *for each $A : \mathit{read}(\ell) \in \mathbb{R}_i$ and $B : \mathit{write}(\ell) \in \mathbb{W}_i$, $A \rightsquigarrow B$,*
- *for each $\mathbb{R}_i$ such that $i \neq 1$, there exists at least one $A : \mathit{read}(\ell) \in \mathbb{R}_i$, and*
- *there exists $A : \mathit{write}(\ell) \in \mathbb{W}_i$ for all $i \neq n$ such that for all $B : \mathit{read}(\ell) \in \mathbb{R}_{i+1}$ and all $C : \mathit{write}(\ell) \in \mathbb{W}_i$, $A \overset{!}{\rightsquigarrow} B$ and $C \rightsquigarrow A$.*

Note that each collection $\mathbb{R}_i$ and $\mathbb{W}_i$ corresponds to the collection of nodes marked by the same name in Figure 4 but with each node $A_i$ included in the collection $\mathbb{W}_i$. The "bottlenecks" are the $A_i$'s. Note that a trace graph $(\mathbb{V}, \mathbb{E})$ actually contains a read-write pipeline with bottlenecks per each reference location $\ell$ as a subgraph $G_\ell$ (but the subgraphs may not be disjoint because the paths may involve other locations and share join nodes).

The following theorem formalizes our earlier informal discussion.

THEOREM 3. *A trace graph $(\mathbb{V}, \mathbb{E})$ is witness race free if and only if it has a read-write pipeline with bottlenecks for every $\ell$.*

The proof appears in our companion technical report [13]. The following is an immediate consequence:

COROLLARY 1. *A $\lambda_{wit}$ program $e$ is witness race free if and only if every trace graph of $e$ has a read-write pipeline with bottlenecks for every reference location $\ell$.*

### 4.1 Regions

Corollary 1 reduces the problem of deciding whether a program $e$ is witness race free to the problem of deciding if every trace graph of $e$ has a read-write pipeline with bottlenecks for every reference location $\ell$. Therefore it suffices to design an algorithm for solving the latter problem. But before we do so, we make a slight change to $\lambda_{wit}$ to make the problem more tractable. In $\lambda_{wit}$, there is a read-write pipeline with bottlenecks for each reference location $\ell$, but distinguishing dynamically allocated reference locations individually is difficult for a compile-time algorithm. Therefore, we add *regions* to the language so that programs can explicitly group reference locations that are to be tracked together.

Figure 5 shows $\lambda_{wit}^{reg}$, $\lambda_{wit}$ extended with regions. The syntax contains two new expression kinds: $\texttt{letreg}\ x\ e$ which creates a

$$
\begin{array}{lcl}
e & := & x \mid i \mid \lambda x.e \mid e\ e' \mid \texttt{let}\ x = e\ \texttt{in}\ e' \mid e \otimes e' \mid \pi_i(e) \\
& \mid & \texttt{write}\ e_1\ e_2\ e_3 \mid \texttt{read}\ e\ e' \mid \texttt{ref}\ e\ e' \mid \texttt{join}\ e\ e' \mid \bullet \\
& \mid & \texttt{letreg}\ x\ e
\end{array}
$$

Figure 5: The syntax of $\lambda_{wit}^{reg}$.

$$
\begin{array}{lcl}
E & := & D \cup \{a \mapsto E\} \mid [\,] \mid E\ e \mid e\ E \mid E \otimes e \mid e \otimes E \\
& \mid & \pi_i(E) \mid \texttt{write}\ E\ e\ e' \mid \texttt{write}\ e\ E\ e' \mid \texttt{write}\ e\ e'\ E \\
& \mid & \texttt{read}\ e\ E \mid \texttt{read}\ E\ e \mid \texttt{ref}\ E\ e \mid \texttt{ref}\ e\ E \\
& \mid & \texttt{join}\ E\ e \mid \texttt{join}\ e\ E
\end{array}
$$

$$
\begin{array}{ll}
\textbf{App} & (R, S, E[(\lambda x.e)\ e']) \\
& \quad \Rightarrow (R, S, E[e[a/x]] \uplus \{a \mapsto e'\}) \\
\textbf{Let} & (R, S, E[\texttt{let}\ x = e\ \texttt{in}\ e']) \\
& \quad \Rightarrow (R, S, E[e'[a/x]] \uplus \{a \mapsto e[a/x]\}) \\
\textbf{Pair} & (R, S, E[\pi_i(e_1 \otimes e_2)]) \Rightarrow (R, S, E[e_i]) \\
\textbf{Write} & (R, S, E[\texttt{write}\ \ell\ e\ \bullet] \\
& \quad \Rightarrow (R, S[\ell \leftarrow a], E[\bullet] \uplus \{a \mapsto e\}) \\
\textbf{Read} & (R, S, E[\texttt{read}\ \ell\ \bullet]) \Rightarrow (R, S, E[S(\ell) \otimes \bullet]) \\
\textbf{Ref} & (R, S, E[\texttt{ref}\ e\ r]) \\
& \quad \Rightarrow (R, S \uplus \{\ell \mapsto a\}, E[\ell] \uplus \{a \mapsto e\}) \\
\textbf{Join} & (R, S, E[\texttt{join}\ \bullet\ \bullet]) \Rightarrow (R, S, E[\bullet]) \\
\textbf{LetReg} & (R, S, E[\texttt{letreg}\ x\ e]) \\
& \quad \Rightarrow (R \uplus \{r\}, S, E[e[a/x]] \uplus \{a \mapsto r\}) \\
\textbf{Arrive} & (R, S, E[a] \uplus \{a \mapsto e\}) \Rightarrow (R, S, E[e] \uplus \{a \mapsto e\}) \\
& \quad \text{where } e \in V \\
\textbf{GC} & (R, S, D \uplus D') \Rightarrow (R, S, D) \\
& \quad \text{where } \diamond \notin \mathit{dom}(D') \wedge \mathit{dom}(D') \cap \mathit{free}(D) = \emptyset
\end{array}
$$

Figure 6: The semantics of $\lambda_{wit}^{reg}$.

new region and `ref` $e\,e'$ which places the newly created reference in region $e'$; `ref` $e\,e'$ replaces `ref` $e$. Figure 6 gives the semantics of $\lambda_{wit}^{reg}$ which differs from $\lambda_{wit}$ in two small ways. First, a state now contains a set of regions $R$. We use symbols $r, r', r_i$, etc. to denote regions. Regions are safe to duplicated, i.e., $r \in V$. The $R$'s are used only for ensuring that the newly created region $r$ at a **LetReg** step is fresh. (We overload the symbol $\uplus$ such that $R \uplus R' = R \cup R'$ if $R \cap R' = \emptyset$ and, is undefined otherwise.) Note that evaluation contexts $E$ do not extend to the subexpressions of `letreg` $x\ e$. The second difference is that a **Ref** step now takes a region $r$ along with the initializer $e$ to indicate that the newly created reference location $\ell$ belongs to the region $r$. Note that the semantics does not actually associate the reference location $\ell$ and the region $r$, and therefore grouping of reference locations via regions is entirely conceptual.[5]

Regions force programs to abide by witness race freedom at the granularity of regions instead of at the granularity of individual reference locations. That is, instead of *read*($\ell$) nodes and *write*($\ell$) nodes, we use *read*($r$) nodes and *write*($r$) nodes. Formally, a trace graph for $\lambda_{wit}^{reg}$ is constructed by the following graph construction semantics:

**Write** $(R, K, S, E[\texttt{write}\ \ell\ e\ A])$
$\Rightarrow (R, K, S[\ell \leftarrow a], E[B] \uplus \{a \mapsto e\})$
$\mathbb{V} := \mathbb{V} \cup \{B\}$ where $B$ is a new *write*($K(\ell)$) node
$\mathbb{E} := \mathbb{E} \cup \{(A, B)\}$

**Read** $(R, K, S, E[\texttt{read}\ \ell\ A]) \Rightarrow (R, K, S, E[S(\ell) \otimes B])$
$\mathbb{V} := \mathbb{V} \cup \{B\}$ where $B$ is a new *read*($K(\ell)$) node
$\mathbb{E} := \mathbb{E} \cup \{(A, B)\}$

**Join** $(R, K, S, E[\texttt{join}\ A\ B]) \Rightarrow (R, K, S, E[C])$
$\mathbb{V} := \mathbb{V} \cup \{C\}$ where $C$ is a new *join* node
$\mathbb{E} := \mathbb{E} \cup \{(A, C), (B, C)\}$

**Ref** $(R, K, S, E[\texttt{ref}\ e\ r])$
$\Rightarrow (R, K \uplus \{\ell \mapsto r\}, S \uplus \{\ell \mapsto a\}, E[\ell] \uplus \{a \mapsto e\})$

Note that these reduction rules use an additional function $K$ which is a mapping from reference locations to regions. The mapping $K$ starts empty at the beginning of evaluation. Other reductions rules are unmodified except that the function $K$ is passed from left to right in the obvious way.

Since there is less information available in a $\lambda_{wit}^{reg}$ trace graph than in a $\lambda_{wit}$ trace graph, the witness race freedom condition is more conservative for $\lambda_{wit}^{reg}$. That is, we still need the condition that for any $A$ : *write*($r$) and $B$ : *read*($r$), either $A \rightsquigarrow B$ or $B \rightsquigarrow A$. But we need to tighten the second condition so that for any $A$ : *read*($r$) if there are $B_1, B_2$ : *write*($r$) such that $B_1 \rightsquigarrow A$ and $B_2 \rightsquigarrow A$, then either $B_1 \rightsquigarrow B_2$ or $B_2 \rightsquigarrow B_1$. This condition is strictly more conservative than for $\lambda_{wit}$, which only requires some $C$ : *write*($r$) such that $C \overset{!}{\rightsquigarrow} A$ in such a situation. The reason for this conservativeness is that we do not know from a trace graph of $\lambda_{wit}^{reg}$ whether $B_1$ and $B_2$ both write to the same reference location.

Formally, witness race freedom for $\lambda_{wit}^{reg}$ can be defined as follows.

DEFINITION 5 (Witness Race Freedom for $\lambda_{wit}^{reg}$). *We say that a $\lambda_{wit}^{reg}$ trace graph $(\mathbb{V}, \mathbb{E})$ is witness race free if for every region $r$,*

- *for every $A$ : read($r$) $\in \mathbb{V}$ and $B$ : read($r$) $\in \mathbb{V}$, either $A \rightsquigarrow B$ or $B \rightsquigarrow A$, and*
- *for every $A$ : read($r$) $\in \mathbb{V}$ and $B_1, B_2$ : write($r$) $\in \mathbb{V}$ such that $B_1 \rightsquigarrow A$ and $B_2 \rightsquigarrow A$, we have $B_1 \rightsquigarrow B_2$ or $B_2 \rightsquigarrow B_1$.*

THEOREM 4. *If a $\lambda_{wit}^{reg}$ program $e$ is witness race free, then $e$ is confluent.*

Proof (Sketch): For any evaluation of $e$, carry out the same reduction sequence with the trace graph building action of $\lambda_{wit}$, i.e., the trace graph $G$ generated is at the granularity of reference locations. Then it is easy to see that if $G$ satisfies the above two conditions, $G$ also satisfies the two conditions of Theorem 2. □

It is easy to see that Definition 5 is the weakest possible restriction to the original witness race freedom under the region abstraction because for any $\lambda_{wit}^{reg}$ trace graph that is not witness race free, one can easily find a non-confluent program that produces the graph.

In a witness-race-free trace graph for $\lambda_{wit}^{reg}$, the read-write pipeline with bottlenecks for a region $r$ consisting of the collections $(\mathbb{R}_1, \ldots, \mathbb{R}_n, \mathbb{W}_1, \ldots, \mathbb{W}_n)$ has the following property: each set $\{A \mid A : \textit{write}(r) \in \mathbb{W}_i\}$ for $i \neq n$ can be totally ordered (with $\rightsquigarrow$ as the ordering relation). The theorem below is immediate from Corollary 1 under this additional property.

THEOREM 5. *A $\lambda_{wit}^{reg}$ program $e$ is witness race free if and only if every trace graph of $e$ has a read-write pipeline with bottlenecks for every region $r$.*

This additional property helps in designing a static checking algorithm.

### 4.2 From Network Flow to Types

Now our goal is to design an algorithm for statically checking if every trace graph of a $\lambda_{wit}^{reg}$ program $e$ has a read-write pipeline with bottlenecks for every region $r$. Our approach exploits a *network flow* property of read-write pipelines with bottlenecks. Consider a trace graph as a network of nodes with each edge $(A, B)$ able to carry any non-negative flow from $A$ to $B$. (Recall edges are directed.) As usual with network flow, we require that the total incoming flow equal the total outgoing flow for every node in the graph. Now, let us add a *virtual source* node $A_S$ and connect it to every node $B$ by adding an edge $(A_S, B)$. We assign incoming flow 1 to $A_S$. Then it is not hard to see that if there exists a read-write pipeline with bottlenecks for the region $r$ then there exists flow assignments such that every *read*($r$) node and *write*($r$) node gets a positive flow and every $A$ : *write*($r$) $\in \mathbb{W}_i$ for $i \neq n$ gets a flow equal to 1.

It turns out that the converse also holds. That is, given a trace graph, if there is a flow assignment such that each *read*($r$) node and *write*($r$) node gets a positive flow and each $A$ : *write*($r$) such that there exists $B$ : *read*($r$) with $A \rightsquigarrow B$ gets a flow equal to 1, then there is a read-write pipeline with bottlenecks for the region $r$. By Theorem 5, this implies that there exists such a flow assignment for every region $r$ if and only if the trace graph is witness race free. Because edges in a trace graph are traces of witnesses, our idea is to assign a type to a witness such that the type contains flow assignments for each (static) region. We use this idea to design a type system such that a well-typed program is guaranteed to be witness race free.

Formally, a witness type $W$ is a function from the set of *static region identifiers* **RegIDs** to rational numbers in the range $[0, 1]$, i.e., $W$ : **RegIDs** $\to [0, 1]$. The rational number $W(\rho)$ indicates the flow amount for the static region identifier $\rho$ in the witness type $W$. We use the notation $\{\rho_1 \mapsto q_1, \ldots, \rho_n \mapsto q_n\}$ to mean a witness type $W$ such that $W(\rho) = q_i$ if $\rho = \rho_i$ for some $1 \leq i \leq n$ and $W(\rho) = 0$ otherwise. (We use the symbols $q$, $q_i$, $q'$, etc for non-negative rational numbers, including those larger than 1.)

The rest of the types are defined in Figure 7, including types include integer types *int*, function types $\tau \xrightarrow{q} \tau'$, pair types $\tau \otimes \tau'$, reference types *ref* $(\tau, \tau', \rho)$, and region types *reg*($\rho$). The non-negative rational number $q$ in $\tau \xrightarrow{q} \tau'$ represents the number of times the function can be called. We allow the symbols $q, q'$, etc

[5] Regions are traditionally coupled with some semantic meaning such as memory management [14, 5]. It is possible to extend $\lambda_{wit}^{reg}$ to do similar things with its regions.

$$\tau \quad := \quad \mathit{int} \mid \tau \xrightarrow{q} \tau' \mid \tau \otimes \tau' \mid \mathit{ref}\,(\tau, \tau', \rho) \mid \mathit{reg}(\rho) \mid W$$

Figure 7: The type language.

$$\frac{\Gamma; W \vdash e : \tau \qquad \tau \geq \tau'}{\Gamma; W \vdash e : \tau'}\ \textbf{Sub} \qquad \frac{\Gamma(x) = \tau}{\Gamma; W \vdash x : \tau}\ \textbf{Var}$$

$$\frac{}{\Gamma; W \vdash i : \mathit{int}}\ \textbf{Int} \qquad \frac{}{\Gamma; W \vdash \bullet : \emptyset}\ \textbf{Dummy}$$

$$\frac{\Gamma; W_1 \vdash e : W_2}{\Gamma; W_1 + W_3 \vdash e : W_2 + W_3}\ \textbf{Source}$$

$$\frac{\Gamma, x{:}\tau; W \vdash e : \tau'}{\Gamma \times q; W \times q \vdash \lambda x.e : \tau \xrightarrow{q} \tau'}\ \textbf{Abs}$$

$$\frac{\Gamma; W \vdash e : \tau \xrightarrow{q} \tau' \qquad \Gamma'; W' \vdash e' : \tau \qquad q \geq 1}{\Gamma + \Gamma'; W + W' \vdash e\, e' : \tau'}\ \textbf{App}$$

$$\frac{\Gamma; W \vdash e : \tau \qquad \Gamma'; W' \vdash e' : \tau'}{\Gamma + \Gamma'; W + W' \vdash e \otimes e' : \tau \otimes \tau'}\ \textbf{Pair}$$

$$\frac{\Gamma; W \vdash e : \tau_1 \otimes \tau_2}{\Gamma; W \vdash \pi_i(e) : \tau_i}\ \textbf{Proj}$$

$$\frac{\Gamma; W \vdash e : \tau \qquad \Gamma'; W' \vdash e' : \mathit{reg}(\rho)}{\Gamma + \Gamma'; W + W' \vdash \texttt{ref}\, e\, e' : \mathit{ref}\,(\tau, \tau, \rho)}\ \textbf{Ref}$$

$$\frac{\begin{array}{c}\Gamma_1; W_1 \vdash e_1 : \mathit{ref}\,(\tau, \tau', \rho) \\ \Gamma_2; W_2 \vdash e_2 : \tau' \\ \Gamma_3; W_3 \vdash e_3 : W \qquad W(\rho) \geq 1\end{array}}{\Gamma_1 + \Gamma_2 + \Gamma_3; W_1 + W_2 + W_3 \vdash \texttt{write}\, e_1\, e_2\, e_3 : W}\ \textbf{Write}$$

$$\frac{\begin{array}{c}\Gamma; W_1 \vdash e : \mathit{ref}\,(\tau, \tau', \rho) \\ \Gamma'; W_2 \vdash e' : W \qquad W(\rho) > 0\end{array}}{\Gamma + \Gamma'; W_1 + W_2 \vdash \texttt{read}\, e\, e' : W \otimes \tau}\ \textbf{Read}$$

$$\frac{\begin{array}{c}\Gamma, x{:}\mathit{reg}(\rho); W + \{\rho \mapsto q\} \vdash e : \tau \qquad q \leq 1 \\ \rho \notin \mathit{free}(\Gamma) \cup \mathit{free}(W) \cup \mathit{free}(\tau)\end{array}}{\Gamma; W \vdash \texttt{letreg}\ x\ e : \tau}\ \textbf{LetRegion}$$

$$\frac{\Gamma; W_1 \vdash e : W \qquad \Gamma; W_2 \vdash e' : W'}{\Gamma + \Gamma'; W_1 + W_2 \vdash \texttt{join}\, e\, e' : W + W'}\ \textbf{Join}$$

$$\frac{\Gamma; W \vdash e'[(\texttt{let}\ x = e\ \texttt{in}\ x)/x] : \tau \qquad e \in V^+}{\Gamma; W \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : \tau}\ \textbf{LetA}$$

$$\frac{\begin{array}{c}\Gamma, x{:}\tau; W \vdash e : \tau \qquad \Gamma', x{:}\tau; W' \vdash e' : \tau' \\ \tau \geq \tau \times \infty \text{ if } x \in \mathit{free}(e)\end{array}}{\Gamma + \Gamma'; W + W' \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : \tau'}\ \textbf{LetB}$$

Figure 8: Type judgment rules.

to take the valuation $\infty$ to imply that the function can be called arbitrarily many times. We use the following arithmetic relation: $q + \infty = \infty$, $q \times \infty = \infty$ for $q \neq 0$, and $0 \times \infty = 0$.

Figure 8 shows the main type judgment rules. Our type system belongs to the family of substructural type systems, which includes linear types. We discuss the rules from top-to-bottom and left-to-right, except for the rule **Sub** which we defer to the end. The rules **Var** and **Int** are standard. The rule **Dummy** gives a dummy witness $\bullet$ an empty witness type; note that $\emptyset(\rho) = 0$ for any static region identifier $\rho$.

$$\begin{array}{l}
\mathit{reg}(\rho) + \mathit{reg}(\rho) = \mathit{reg}(\rho) \\
\mathit{int} + \mathit{int} = \mathit{int} \\
\tau \xrightarrow{q} \tau' + \tau \xrightarrow{q'} \tau' = \tau \xrightarrow{q+q'} \tau' \\
\tau_1 \otimes \tau_2 + \tau_3 \otimes \tau_4 = (\tau_1 + \tau_3) \otimes (\tau_2 + \tau_4) \\
\mathit{ref}\,(\tau_1, \tau, \rho) + \mathit{ref}\,(\tau_2, \tau, \rho) = \mathit{ref}\,(\tau_1 + \tau_2, \tau, \rho) \\
W + W' = \{\rho \mapsto W(\rho) + W'(\rho) \mid \rho \in \textbf{RegIDs}\} \\
\mathit{reg}(\rho) \times q = \mathit{reg}(\rho) \\
\mathit{int} \times q = \mathit{int} \\
\tau \xrightarrow{q'} \tau' \times q = \tau \xrightarrow{q' \times q} \tau' \\
\tau \otimes \tau' \times q = (\tau \times q) \otimes (\tau' \times q) \\
\mathit{ref}\,(\tau, \tau', \rho) \times q = \mathit{ref}\,(\tau \times q, \tau', \rho) \\
W \times q = \{\rho \mapsto W(\rho) \times q \mid \rho \in \textbf{RegIDs}\}
\end{array}$$

Figure 9: Arithmetic over types.

The rule **Source** uses additive arithmetic over types defined in Figure 9. The rule adds $W_3$ amount of flow from the virtual source nodes (i.e., nodes $A_S$ from the first paragraph of this section) to $W_2$. In the type judgment relation $\Gamma; W \vdash e : \tau$, the witness type $W$ represents the flow the expression $e$ receives from the virtual source nodes. Therefore, the rule **Source** says that assuming that we took $W_1$ flow from the virtual source nodes in the precondition, we are now taking $W_3$ more.

In the rule **Abs**, we multiply the left hand side of the judgments by the number of times that the function can be used. Multiplication over type environments $\Gamma$ is defined as follows:

$$(\Gamma, x{:}\tau) \times q = (\Gamma \times q), x{:}(\tau \times q)$$

So for example, if the $\lambda$ abstraction $\lambda x.e$ captures a witness as a free variable $y$ and that $\Gamma(y) = W$, then $(\Gamma \times q)(y) = W \times q$. Thus if the function body requires $W$ amount of flow in the witness, then we actually require $W \times q$ amount of flow because the function may be called $q$ times.

In the rule **App**, the precondition $q \geq 1$ says that the number of times the function can be used must be at least 1. The left hand side of the two judgments in the precondition are added so that we can compute the combined flow required for the expressions $e$ and $e'$. Addition over type environments is defined as follows:

$$(\Gamma, x{:}\tau) + (\Gamma', x{:}\tau') = (\Gamma + \Gamma'), x{:}(\tau + \tau')$$

The rules **Pair** and **Proj** are self-explanatory.

In a reference type $\mathit{ref}\,(\tau, \tau', \rho)$, the static region identifier $\rho$ identifies the region where the reference belongs while the type $\tau$ is the read type of the reference and the type $\tau'$ is the write type of the reference. Initially the read and write types are the same as seen in the rule **Ref**. The rule **Write** matches the type of the to-be-assigned expression $e_2$ with the write type of the reference while the rule **Read** uses the read type of the reference type. Note that we require $W(\rho) \geq 1$ at **Write** and $W(\rho) > 0$ at **Read**; both correspond to the flow requirement for writes and reads. The reason read-type/write-type separation is subtle. Consider the following expression where the expressions $e_1$ and $e_3$ are witnesses and the expression $e_2$ is a region:

$$\texttt{let}\ x = (\texttt{ref}\ e_1\ e_2)\ \texttt{in let}\ w = (\texttt{write}\ x\ e_3\ \bullet)\ \texttt{in read}\ x\ w$$

Suppose we just have read types so that the type system uses read types at instances **Write** as well as at instances **Read**. Then the type system is unsound (even without **Sub**) for the following reason. The type system may assign some flow $W$ to the occurrence of

$$\frac{}{\tau \geq \tau} \qquad \frac{\tau_1 \leq \tau_1' \quad \tau_2 \geq \tau_2' \quad q \geq q'}{\tau_1 \xrightarrow{q} \tau_2 \geq \tau_1' \xrightarrow{q'} \tau_2'}$$

$$\frac{\tau_1 \geq \tau_1' \quad \tau_2 \geq \tau_2'}{\tau_1 \otimes \tau_2 \geq \tau_1' \otimes \tau_2'}$$

$$\frac{\tau_1 \geq \tau_1' \qquad \tau_2 \leq \tau_2'}{\mathit{ref}\,(\tau_1, \tau_2, \rho) \geq \mathit{ref}\,(\tau_1', \tau_2', \rho)}$$

$$\frac{W(\rho) \geq W(\rho) \text{ for all } \rho \in \textbf{RegIDs}}{W \geq W'}$$

Figure 10: Subtyping.

the variable $x$ at the write and some flow $W'$ to the occurrence of the variable $x$ at the read. But there is no constraint to force $W = W'$, so the type system can let $W' > W$ while keeping the sum $W + W'$ fixed, i.e., we get more flow from a reference than what was assigned to the reference. Separating read and write types prevents this problem because addition and multiplication do not act on write types.

The rule **LetRegion** introduces a fresh static region identifier $\rho$. The witness type $\{\rho \mapsto q\}$ represents the virtual source node for the new region. We constrain $q \leq 1$ to ensure that we do not use more than 1 unit total from the source.

The rule **Join** combines two witnesses by adding their types.

There are two rules, **LetA** and **LetB**, for the expression kind `let` $x$ `=` $e$ `in` $e'$. **LetA** is less conservative and should be used whenever $x$ occurs more than once in $e'$ and $e \in V^+$ where $V^+$ is the smallest set such that $V^+ = V \cup \{\texttt{let } x = e \texttt{ in } x \mid e \in V^+\}$. This rule corresponds to the usual substitution interpretation of let-based predicative polymorphism with the value restriction. **LetB** is used if $e \notin V^+$ or $x$ occurs at most once in $e'$. Here, $\mathit{free}(\tau)$ is the set of static region identifiers in the type $\tau$ where $\mathit{free}(W) = \{\rho \mid W(\rho) \neq 0\}$, and $\mathit{free}(\Gamma) = \bigcup_{\tau \in \mathit{ran}(\Gamma)} \mathit{free}(\tau)$.

Finally, we come return to **Sub**. The subtyping relation is defined in Figure 10. As usual, argument types of function types are contravariant. Write types of reference types are also contravariant; this treatment of reference subtyping is identical to that of a type-based formulation of Andersen's points-to analysis [4]. Intuitively, the rule **Sub** expresses the observation that the flow graph property may be relaxed so that the sum of the outgoing flow can be less than the sum of the incoming flow, i.e., if we could find a flow assignment satisfying the required flow constraints at reads and writes under this relaxed condition, then we still have a read-write pipeline with bottlenecks.

We say that a $\lambda_{wit}^{reg}$ program $e$ is well-typed if $\emptyset; \emptyset \vdash e : \tau$ for some type $\tau$. The following theorem states that the type system is sound.

THEOREM 6. *If a $\lambda_{wit}^{reg}$ program $e$ is well-typed, then it is witness race free.*

The proof, which uses the network flow property, appears in the companion technical report [13].

We point out a few of the positive properties of this type system. If a program contains no reads or writes and can be typed by a standard Hindley-Milner polymorphic type system, then it can also be typed by our type system; for example, we may use the qualifier $\infty$ for all function types and use 0 for all flows. In general, we can give the $\infty$ qualifier to the function type of any function that does not capture a witness (directly or indirectly). We can also assign flow 0 to any flow for a region $r$ that does not flow into a side-effect primitive operating on the region $r$.

$\mathit{Fresh}(\mathit{int}) = \mathit{int}$
$\mathit{Fresh}(\sigma \rightarrow \sigma) = \mathit{Fresh}(\sigma) \xrightarrow{\beta} \mathit{Fresh}(\sigma)$
  where $\beta$ is fresh
$\mathit{Fresh}(\sigma \otimes \sigma') = \mathit{Fresh}(\sigma) \otimes \mathit{Fresh}(\sigma)$
$\mathit{Fresh}(\mathit{ref}\,(\sigma, \sigma', \rho)) = \mathit{ref}\,(\mathit{Fresh}(\sigma), \mathit{Fresh}(\sigma'), \rho)$
$\mathit{Fresh}(\mathit{reg}(\rho)) = \mathit{reg}(\rho)$
$\mathit{Fresh}(I) = \{\rho \mapsto \alpha \mid \rho \in I\}$
  where $\alpha$ is fresh

Figure 11: *Fresh.*

$$\frac{\Gamma, W \vdash_b e^I : \tau, \mathcal{C}}{\Gamma, W + \mathit{Fresh}(I) \vdash_a e^I : \tau + \mathit{Fresh}(I), \mathcal{C}}$$

$$\frac{\Gamma, W \vdash_b e^\sigma : \tau, \mathcal{C} \qquad \sigma \notin \text{type } I}{\Gamma, W \vdash_a e^\sigma : \tau, \mathcal{C}}$$

Figure 12: Type inference $\vdash_a$.

The type system is quite expressive. In particular, it is able to type all of the examples that were used as correct programs up to this point in the paper (with straightforward modification to translate $\lambda_{wit}$ programs into $\lambda_{wit}^{reg}$). In fact, assuming that for each region $r$, $\mathit{write}(r)$ nodes in each collection $\mathbb{W}_i$ are totally ordered, the type system is "complete" for the first-order fragment (i.e., no higher order functions) with no recursion and no storing of witnesses in references. That is, such a program $e$ is witness race free and well-typed by a standard Hindley-Milner type system if and only if it is well-typed by our type system. We also show later in Section 5 that the type system is more expressive than past approaches.

The limitations of the type system are the standard ones: let-based predicative polymorphism, flow-insensitivity of reference types, and unsoundness under call-by-name semantics; the latter is a typical limitation of a non-linear substructural type system. Another limitation is that the type system enforces for each region $r$ that $\mathit{write}(r)$ nodes in every collection $\mathbb{W}_i$ are totally ordered whereas witness race freedom permits an absence of ordering for the case $i = n$; we believe that this is a minor limitation.

### 4.3 Inference

We next present a type inference algorithm. By Theorem 6, this results in an automatic algorithm for statically checking witness race freedom.

At a high-level, our type system is a standard Hindley-Milner type system with some additional rational arithmetic constraints. Therefore we could perform inference by employing a standard type inference technique to solve all type-structural constraints while generating rational arithmetic constraints on the side, and then solving the generated arithmetic constraints separately. Unfortunately, the arithmetic constraints may be non-linear since they involve the multiplication of variables. Because there is no efficient algorithm for solving general non-linear rational arithmetic constraints, we need to dive into lower-level details of the type system.

Let us separate type inference into two phases. The first phase carries out type inference after erasing all rational numbers from the type system. That is, the types inferred in this phase are:

$$\sigma \quad := \quad \mathit{int} \mid \sigma \rightarrow \sigma' \mid \sigma \otimes \sigma' \mid \mathit{ref}\,(\sigma, \sigma', \rho) \mid \mathit{reg}(\rho) \mid I$$

where a type $I$ is a subset of **RegIDs**. Intuitively, each type $I$ represents the non-0 domain of some witness type $W$. The first phase can be carried out by a standard Hindley-Milner type inference, al-

$$\frac{\tau = \mathit{Fresh}(\sigma)}{\{x{:}\tau\}, \emptyset \vdash_b x^{\sigma} : \tau, \emptyset}$$

$$\frac{}{\emptyset, \emptyset \vdash_b i : \mathit{int}, \emptyset} \qquad \frac{}{\emptyset, \emptyset \vdash_b \bullet : \emptyset, \emptyset}$$

$$\frac{\Gamma, W \vdash_a e : \tau, \mathcal{C} \qquad \beta \text{ is fresh}}{\Gamma \times \beta, W \times \beta \vdash_b \lambda x.e : \Gamma(x) \xrightarrow{\beta} \tau, \mathcal{C}}$$

$$\frac{\begin{array}{c}\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2 \\ \tau = \mathit{Fresh}(\sigma) \qquad \tau' = \mathit{Fresh}(\sigma') \qquad \beta \text{ is fresh} \\ \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \geq \tau \xrightarrow{\beta} \tau', \beta \geq 1, \tau_2 \geq \tau\}\end{array}}{\Gamma_1 + \Gamma_2, W_1 + W_2 \vdash_b e_1^{\sigma \to \sigma'}\ e_2 : \tau', \mathcal{C}}$$

$$\frac{\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2}{\Gamma_1 + \Gamma_2, W_1 + W_2 \vdash_b e_1 \otimes e_2 : \tau_1 \otimes \tau_2, \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\frac{\Gamma, W \vdash_a \pi_i(e) : \tau, \mathcal{C} \qquad \tau_1 = \mathit{Fresh}(\sigma) \qquad \tau_2 = \mathit{Fresh}(\sigma')}{\Gamma, W \vdash_b \pi_i(e^{\sigma_1 \otimes \sigma_2}) : \tau_i, \mathcal{C} \cup \{\tau \geq \tau_1 \otimes \tau_2\}}$$

$$\frac{\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2}{\Gamma_1 + \Gamma_2, W_1 + W_2 \vdash_b \texttt{ref}\ e_1\ e_2^{\mathit{reg}(\rho)} : \mathit{ref}(\tau_1, \tau_2, \rho), \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\frac{\begin{array}{c}\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2 \\ \Gamma_3, W_3 \vdash_a e_2 : \tau_3, \mathcal{C}_3 \qquad \tau = \mathit{Fresh}(\sigma) \qquad \tau' = \mathit{Fresh}(\sigma') \\ \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \geq \mathit{ref}(\tau, \tau', \rho), \tau_2 \geq \tau', \tau_3(\rho) \geq 1\} \\ \Gamma = \Gamma_1 + \Gamma_2 + \Gamma_3 \qquad W = W_1 + W_2 + W_3\end{array}}{\Gamma, W \vdash_b \texttt{write}\ e_1^{\mathit{ref}(\sigma, \sigma', \rho)}\ e_2\ e_3 : \tau_3, \mathcal{C}}$$

$$\frac{\begin{array}{c}\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2 \\ \tau = \mathit{Fresh}(\sigma) \qquad \tau' = \mathit{Fresh}(\sigma') \\ \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \geq \mathit{ref}(\tau, \tau', \rho), \tau_2(\rho) > 0\}\end{array}}{\Gamma_1 + \Gamma_2, W_1 + W_2 \vdash_b \texttt{read}\ e_1^{\mathit{ref}(\sigma, \sigma', \rho)}\ e_2 : \tau \otimes \tau_2, \mathcal{C}}$$

$$\frac{\Gamma, W \vdash_a e : \tau, \mathcal{C}}{\Gamma, W \vdash_b \texttt{letreg}\ x^{\mathit{reg}(\rho)}\ e : \tau, \mathcal{C} \cup \{W(\rho) \leq 1\}}$$

$$\frac{\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2}{\Gamma_1 + \Gamma_2, W_1 + W_2 \vdash_b \texttt{join}\ e_1\ e_2 : \tau_1 + \tau_2, \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\frac{\begin{array}{c}\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2 \\ x \notin \mathit{free}(e_1) \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \geq \Gamma_2(x)\}\end{array}}{\Gamma_1 + \Gamma_2, W_1 + W_2 \vdash_b \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau_2, \mathcal{C}}$$

$$\frac{\begin{array}{c}\Gamma_1, W_1 \vdash_a e_1 : \tau_1, \mathcal{C}_1 \qquad \Gamma_2, W_2 \vdash_a e_2 : \tau_2, \mathcal{C}_2 \\ x \in \mathit{free}(e_1) \\ \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \geq \Gamma_1(x) \times \infty, \tau_1 \geq \Gamma_2(x)\}\end{array}}{\Gamma_1 + \Gamma_2, W_1 + W_2 \vdash_b \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau_2, \mathcal{C}}$$

Figure 13: Type inference $\vdash_b$.

beit with regions, which is no harder than type variables. We omit the details of this phase. We may safely reject the program if the first phase fails. Otherwise we annotate each subexpression $e$ by its inferred type $\sigma$: $e^{\sigma}$. In the second phase, we use the annotated program to generate the appropriate rational arithmetic constraints via bottom-up type-inference. Let $e$ be an annotated program. Then the generated constraints for $e$ is $\mathcal{C}$ where $\Gamma, W \vdash_a e : \tau, C$ for some $\Gamma$, $W$, and $\tau$.

The second-phase type inference rules are separated into two kinds, $\vdash_a$ (Figure 12) and $\vdash_b$ (Figure 13), which must occur in strictly interleaving manner. The purpose of $\vdash_a$ is to account for the type judgment rule **Source**, whereas $\vdash_b$ accounts for all other rules.

We should note that, strictly speaking, types $\tau$ appearing in the algorithm are different from the ones in the type judgment rules. That is, instead of rational numbers, the types $\tau$ in the algorithm are qualified by *rational number variables* $\alpha, \beta, \gamma$, etc. Also the domain of a witness type $W$ is not the entire **RegIDs** set but only some subset of it. In other words, a witness type $W$ is a partial function from **RegIDs** to rational number variables. We re-define the addition of witness types as follows to reflect the change:

$$\begin{array}{l} W + W' = \\ \quad \{\rho \mapsto W(\rho) + W'(\rho) \mid \rho \in \mathit{dom}(W) \wedge \rho \in \mathit{dom}(W')\} \\ \quad \cup \{\rho \mapsto W(\rho) \mid \rho \in \mathit{dom}(W) \wedge \rho \notin \mathit{dom}(W')\} \\ \quad \cup \{\rho \mapsto W'(\rho) \mid \rho \in \mathit{dom}(W') \wedge \rho \notin \mathit{dom}(W)\} \end{array}$$

We also re-define the addition of type environments:

$$\begin{array}{l} \Gamma + \Gamma' = \\ \quad \{x{:}\Gamma(x) + \Gamma'(x) \mid x \in \mathit{dom}(\Gamma) \wedge x \in \mathit{dom}(\Gamma')\} \\ \quad \cup \{x{:}\Gamma(x) \mid x \in \mathit{dom}(\Gamma) \wedge x \notin \mathit{dom}(\Gamma')\} \\ \quad \cup \{x{:}\Gamma'(x) \mid x \in \mathit{dom}(\Gamma') \wedge x \notin \mathit{dom}(\Gamma)\} \end{array}$$

Note that we omit annotations when they are not used (i.e., we say $e$ instead of $e^{\sigma}$, etc.). There are only two cases for $\vdash_a$. The first case is for expressions that were given a witness type $I$ in the first phase. In this case, we add $\mathit{Fresh}(I)$ to $\tau$ and $W$ to account for a possible application of **Source**. *Fresh* is defined in Figure 11. The second case is for expressions that were not given a witness type. In this case, we simply pass the result of the subderivation $\vdash_b$ up.

We discuss a few representative $\vdash_b$ rules. Note that $\vdash_b$ rules are syntax directed. In the case of a variable $x^{\sigma}$, we create a fresh $\tau$ from $\sigma$ and pass $\{x{:}\tau\}, \emptyset \vdash_b x^{\sigma} : \tau, \emptyset$ up to the parent derivation. (Recall our type inference is bottom-up.) The case for integers and dummy witnesses are trivial. In the case of an abstraction $\lambda x.e$, we multiply $\Gamma$ and $W$ passed from the subderivation by $\beta$. In the case of a function application $e_1^{\sigma \to \sigma'}\ e_2$, we add the constraints $\{\tau_1 \geq \tau \xrightarrow{\beta} \tau', \beta \geq 1, \tau_2 \geq \tau\}$ to connect arguments and returns as well as requiring $\beta$ to be at least 1. Note that the type rule **Sub** is implicitly incorporated in the constraints. In the case of `write` $e_1^{\mathit{ref}(\sigma, \sigma', \rho)}\ e_2\ e_3$, we add the constraint $\tau_2(\rho) \geq 1$ to match the type rule **Write**. Note that the first phase guarantees that $\rho \in \mathit{dom}(\tau_3)$. In the case `letreg` $x^{\mathit{reg}(\rho)}\ e$, the constraint $W(\rho) \leq 1$ is effective only when $\rho \in \mathit{dom}(W)$ as $\rho \notin \mathit{dom}(W)$ implies that the region was not used at all. Note that there is no case corresponding to the type rule **LetA**. Prior to running the algorithm, we replace each occurrence of the expression `let` $x$ `=` $e$ `in` $e'$ in the program by the expression $e'[($`let` $x$ `=` $e$ `in` $x)/x]$ whenever $e \in V$ and $x$ occurs more than once in $e'$.

As an example, consider the following program (a $\lambda_{wit}^{reg}$ version of the last example from Section 2):

```
letreg r
  let x = ref 1 r in
    let w = write x 2 • in
      let f = λy.read x w in
        let z = (f 0) ⊗ (f 0) in
          let w = write x 3 join π2(π1(z)) π2(π2(z)) in z
```

Suppose the first phase assigns $r$ the type $reg(\rho)$. Assume each `let`-bound variable is treated monomorphically. The second phase generates the following constraints for the `let`-bound expressions (slightly simplified for readability):

$\{r : reg(\rho)\}; \emptyset \vdash_a \texttt{ref}\ 1\ r : ref(int, int, \rho), \emptyset$

$\Gamma; \{\rho \mapsto \gamma_1 + \gamma_2\} \vdash_a \texttt{write}\ x\ 2\ \bullet : \{\rho \mapsto \gamma_1 + \gamma_2\}, \{\gamma_1 \geq 1\}$
where $\Gamma = \{x : ref(int, int, \rho)\}$

$\Gamma; W \vdash_a \lambda y.\texttt{read}\ x\ w : int \xrightarrow{\beta_1} int \otimes \{\rho \mapsto \alpha_1 + \gamma_3\}, \mathcal{C}$
where $\Gamma = \{x : ref(int, int, \rho), w : \{\rho \mapsto \alpha_1 \times \beta_1\}\}$
and $W = \{\rho \mapsto \gamma_3 \times \beta_1\}$
and $\mathcal{C} = \{\alpha_1 + \gamma_3 > 0\}$

$\Gamma; \emptyset \vdash_a (f\ 0) \otimes (f\ 0) : \tau, \mathcal{C}$
where $\Gamma = \{f : int \xrightarrow{\beta_2 + \beta_3} int \otimes \{\rho \mapsto \alpha_2\}\}$
and $\tau = (int \otimes \{\rho \mapsto \alpha_2\}) \otimes (int \otimes \{\rho \mapsto \alpha_2\})$
and $\mathcal{C} = \{\beta_2 \geq 1, \beta_3 \geq 1\}$

$\Gamma; \{\rho \mapsto \gamma_4\} \vdash_a \texttt{write}\ x\ 3 \ldots : \{\rho \mapsto \alpha_3 + \alpha_4 + \gamma_4\}, \mathcal{C}$
where $\Gamma = \{x : ref(int, int, \rho), z : \tau\}$
and $\tau = (int \otimes \{\rho \mapsto \alpha_3\}) \otimes (int \otimes \{\rho \mapsto \alpha_4\})$
and $\mathcal{C} = \{\alpha_3 + \alpha_4 + \gamma_4 \geq 1\}$

The final constraints, after some simplification, is as follows:

$$\gamma_1 \geq 1, 1 \geq \gamma_1 + \gamma_2 + \gamma_3 \times \beta_1 + \gamma_4, \beta_1 \geq \beta_2 + \beta_3,$$
$$\beta_2 \geq 1, \beta_3 \geq 1, \gamma_1 + \gamma_2 \geq \alpha_1 \times \beta_1, \alpha_1 + \gamma_3 > 0,$$
$$\alpha_1 + \gamma_3 \geq \alpha_2, \alpha_3 + \alpha_4 + \gamma_4 \geq 1, \alpha_2 \geq \alpha_3, \alpha_2 \geq \alpha_4$$

Note that the constraints are satisfiable, e.g., by the substitution

$$\beta_1 = 2 \quad \beta_2 = \beta_3 = \gamma_1 = 1$$
$$\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = 0.5$$
$$\gamma_2 = \gamma_3 = \gamma_4 = 0$$

In general, a program $e$ is well-typed if and only if the constraints $\mathcal{C}$ generated by type inference are satisfiable. So it suffices to show that the satisfaction problem for any $\mathcal{C}$ generated by the type inference algorithm can be solved.

To this end, we first note that because of the first phase, any constraint $\tau \geq \tau' \in \mathcal{C}$ can be reduced to a set of rational arithmetic constraints of the form $p \geq p'$ where $p, p'$ are rational polynomials. The troublesome non-linearity comes from $\Gamma \times \beta$ and $W \times \beta$ in the $\lambda x.e$ case. Let us focus our attention on the set $\mathcal{B}$ of variables used in such multiplications. (Note that we have used $\beta$ instead of $\alpha$ just for this case in the pseudo-code to make it clear that these variables are special.) We can show that the following holds:

THEOREM 7. *Let $p \rhd p' \in \mathcal{C}$ where $\rhd \in \{\geq, >\}$. If $\beta \in \mathcal{B}$ occurs in the polynomial p, then it must be the case that $\rhd = \geq$, $p = \beta$, and that the polynomial $p'$ consists only of symbols in the set $\{+, \times, 1, \infty\} \cup \mathcal{B}$.*

Proof (Sketch): Let $\Gamma, W \vdash_a e : \tau, \mathcal{C}$. For any $\tau' \in \Gamma$, $\times$ and $+$ only appear at the top-level, i.e., not within argument and return types of a function type. Secondly, we can show by induction that within the type $\tau$, $\times$ only appears in negative positions. More precisely, for any $p \in Pos(\tau)$, the polynomial $p$ contains no $\times$ where $Pos$ is defined as follows:

$$
\begin{aligned}
&Pos(\tau \xrightarrow{p} \tau') = Neg(\tau) \cup Pos(\tau') \cup \{p\} \\
&Pos(\tau \otimes \tau') = Pos(\tau) \cup Pos(\tau') \\
&Pos(ref(\tau, \tau', \rho)) = Pos(\tau) \cup Neg(\tau') \\
&Pos(W) = ran(W) \\
&Neg(int) = Pos(int) = Neg(reg(\rho)) = Pos(reg(\rho)) = \emptyset \\
&Neg(\tau \xrightarrow{p} \tau') = Pos(\tau) \cup Neg(\tau') \\
&Neg(\tau \otimes \tau') = Neg(\tau) \cup Neg(\tau') \\
&Neg(ref(\tau, \tau', \rho)) = Neg(\tau) \cup Pos(\tau') \\
&Neg(W) = \emptyset
\end{aligned}
$$

Third, for any $+$ that appears in a positive position of $\tau$, i.e. in some $p \in Pos(\tau)$, the polynomial $p$ does not contain any $\beta \in \mathcal{B}$. Then the result follows from inspection of the subtyping rules. □

The theorem implies that we can compute all assignments to the variables in $\mathcal{B}$ by computing the minimum satisfying assignment for $\mathcal{C}' = \{\beta \geq p \mid \beta \in \mathcal{B}\} \subseteq \mathcal{C}$. It is easy to see that such an assignment always exists. (Recall that the range is non-negative.) This problem can be solved in quadratic time by an iterative method in which all variables are initially set to 0, and at each iteration the new values for the variables are computed by taking the maximum of the right hand polynomials evaluated at the current values. It is possible to show that if the minimum satisfying assignment for a variable $\beta$ is some $q < \infty$, then the iterative method finds $q$ for $\beta$ in $2|\mathcal{C}'|$ iterations. Hence any variable changing after the $2|\mathcal{C}'|$th iteration can be safely set to $\infty$. All variables are then guaranteed to converge within $3|\mathcal{C}'|$ iterations. Because each iteration examines every constraint, the overall time is at most quadratic in the size of $\mathcal{C}'$.

Substituting the computed assignments for $\mathcal{B}$ in $\mathcal{C}$ results in linear rational constraints, which can be solved efficiently by a linear programming algorithm.

## 5. Related Work

Adding side-effects to a functional language is an old problem with many proposed solutions. Here we compare our technique against two of the more prominent approaches: linear types [15, 6, 1] and monads [10, 11, 7, 9].

In linear types, there is an explicit *world* program value (or one world per region for languages with regions) that conceptually represents the current program state. By requiring each world have a linear type, the type system ensures that the world can be updated in place.

The linear type system can be expressed in our type system by restricting every flow to 1, every witness to contain only one flow, and designating one dummy witness to serve the role of the "starting" witness (or for regions, one dummy witness per region). Thus our approach is more expressive than such an approach. Note that this also implies that every function type can be restricted to have either 1 or $\infty$ as the qualifier. It is easy to see that the program is well-typed under this restriction if and only if it is well-typed by the linear type system. The restriction limits programs to manipulate witnesses only in a linear fashion. In practice, this implies that there can be no parallel reads, no dead witnesses, no redundant witnesses, and no duplication of values containing witnesses.

Our approach can implement monadic primitives as follows (for concrete comparison, we use *state monads* [9]):

$$\begin{aligned}
&\texttt{newVar} = \lambda x.\lambda y.\texttt{let}\ z = (\texttt{ref}\ x\ \pi_2(y))\ \texttt{in}\ y \otimes z\\
&\texttt{readVar} = \lambda x.\lambda y.\texttt{let}\ z = (\texttt{read}\ x\ \pi_1(y))\ \texttt{in}\\
&\qquad\qquad (\pi_2(z) \otimes \pi_2(y)) \otimes \pi_1(z)\\
&\texttt{writeVar} = \lambda x.\lambda w.\lambda y.\texttt{let}\ z = (\texttt{write}\ x\ w\ \pi_1(y))\ \texttt{in}\\
&\qquad\qquad (z \otimes \pi_2(y)) \otimes w\\
&\texttt{>>=} = \lambda f.\lambda g.\lambda x.\texttt{let}\ y = (f\ x)\ \texttt{in}\ g\ \pi_2(y)\ \pi_1(y)\\
&\texttt{returnST} = \lambda x.\lambda y.y \otimes x\\
&\texttt{runST}\ e = \texttt{letreg}\ x\ \pi_2(e\ (\bullet \otimes x))
\end{aligned}$$

The idea behind these definitions is to implement each state monad of the type $ST(\alpha, \tau)$ as a function that takes a witness and the region $\alpha$ as arguments and returns a witness, the region $\alpha$, and a value of the type $\tau$. It is easy to see that if a state monad program is well-typed by the monadic type system, then it is also well-typed with our type system using the above definitions for the monadic primitives. Thus, our approach is more expressive than monads.

In practice, a monadic approach shares essentially the same limitations as linear types; for example, side-effects are restricted to a linear, sequential order. (In fact, it is not too hard to see that we can actually implement monadic primitives with the linear types restriction with only slightly longer code.) On the other hand, a monadic type system has an engineering advantage as it only requires Hindley-Milner type inference.

In addition to the above technical differences, our approach differs from previous approaches in its design motivation. That is, while our language feature, witnesses, is motivated by a pragmatic observation, understanding the motivation behind linear types and monads (i.e., not just knowing how to use them) arguably requires an appreciation of their underlying theory.

The technique used in our type system has some resemblance to *fractional permissions* [2] which can guarantee non-interference in imperative programs. Indeed, it may be possible to give an "interpretation" of witnesses as some relaxation of permissions or *capabilities* [3] (after promoting permissions or capabilities to first class values) in situations where the program is witness race free.

## 6. Conclusions

We have presented a new approach to adding side-effects in purely functional languages based on witnesses. We have stated a natural semantic correctness condition called witness race freedom and proposed a type-based approach for statically checking witness race freedom.

## References


[1] P. Achten, J. H. G. van Groningen, and R. Plasmeijer. High level specification of I/O in functional languages. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 1–17. Springer-Verlag, 1993.

[2] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis, Tenth International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, San Diego, CA, June 2003. Springer-Verlag.

[3] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, Jan. 1999.

[4] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Montreal, Canada, June 1998.

[5] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[6] J. C. Guzman and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 42–51, Philadelphia, Pennsylvania, June 1990.

[7] S. L. P. Jones and P. Wadler. Imperative functional programming. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 71–84, Charleston, South Carolina, Jan. 1993.

[8] J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, Jan. 1993.

[9] J. Launchbury and A. Sabry. Monadic state: Axiomatization of type safety. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 227–238, Amsterdam, The Netherlands, June 1997.

[10] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[11] M. Odersky, D. Rabin, and P. Hudak. Call by name, assignment, and the lambda calculus. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 43–56, Charleston, South Carolina, Jan. 1993.

[12] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[13] T. Terauchi and A. Aiken. Witnessing side-effects. Technical report, Computer Science Division, EECS Department, University of California, Berkeley.

[14] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value $\lambda$-calculus using a stack of regions. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, Jan. 1994.

[15] P. Wadler. Linear types can change the world! In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 63–74, Baltimore, Maryland, Sept. 1998.