Witnessing Side Effects

TACHIO TERAUCHI
Tohoku University
and
ALEX AIKEN
Stanford University

We present a new approach to the old problem of adding global mutable state 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 somewhat surprising static checking algorithm that makes use of a network flow property equivalent to the semantic condition via reduction to a satisfaction problem for a system of linear inequalities.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*; F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program analysis*; F.3.3 [**Logics and Meaning of Programs**]: Studies of Program Constructs—*Type structure*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Mutable state, Side effects

1. INTRODUCTION

Adding global mutable state to a purely functional language is a well-known problem with a number of solutions [Peyton Jones and Wadler 1993; Launchbury and Sabry 1997; Odersky et al. 1993; Wadler 1990; Guzman and Hudak 1990; Achten et al. 1993] with monads [Moggi 1991; Odersky et al. 1993; Peyton Jones and Wadler 1993; Launchbury and Sabry 1997] being arguably the most popular. This paper proposes 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 from global mutable states. Witnesses by themselves do not guarantee *correctness* (i.e., that the program can be viewed as a purely functional program). We give a natural semantic condition sufficient to guarantee correctness and give a static checking algorithm. Our static checking algorithm is a non-standard type-based algorithm that converts the problem to solving a system of linear inequalities. The result is a new framework for guaranteeing correctness

This is a revised and extended version of a paper presented at the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP 2005), pp.105-115, 2005.

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, use of side effects, namely manual destructive memory/resource updates, is 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* for ordering 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. (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.

This article is a revised and extended version of a conference paper with the same title [Terauchi and Aiken 2005]. The main additions are the proofs, corrections to the type inference algorithm (the conference version has an error where the types in the generated constraints do not have enough "fresh" variables and hence are over-constrained), and a more thorough comparison with related work.

2. PRELIMINARIES

We need a precise definition of what it means for side effects to be "correct" within a functional language. The idea is to show that a program's semantics is independent of a class of "functional" program transformation rules. To this end, we fix a set of program transformations expressive enough to model different functional reduction strategies, including call-by-value, call-by-need (i.e., lazy-evaluation), and

[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.

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

Fig. 1. The syntax of the language λ_{wit}.

parallel evaluation. 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, modulo termination (for the same reason untyped pure lambda calculus is confluent but not strongly normalizing.)

For ease of exposition, we include the aforementioned 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.

Figure 1 gives the syntax of λ_{wit}, a simple functional language with side effects and witnesses. Note λ_{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 may appear in 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 the reference e is not read until seeing 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 seeing 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 λ_{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 λ_{wit}, we describe novel properties of λ_{wit} informally by examples.

Programs in λ_{wit} can use witnesses to order side effects. For example, the following program returns $2 \otimes w$ 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, λ_{wit} does not guarantee correctness. For example, the following λ_{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

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

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

Fig. 2. The semantics of λ_{wit}.

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 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.

In the rest of this section, we describe the semantics of λ_{wit} so that we can formally define when a λ_{wit} program is correct, i.e., when it is confluent. Figure 2 shows the semantics of λ_{wit} 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* ℓ 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.[2] The reduction rules are parametrized by the evaluation contexts E. For an expression e, $free(e)$ is the set of free variables, ports, and reference locations of e. For an expression store D, $free(D) = dom(D) \cup \bigcup_{e \in ran(D)} free(e)$.

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 $dom(F) \cap dom(F') = \emptyset$ and is undefined otherwise. **App** creates a fresh port a and stores e' at a. **Let** is similar. **Pair** projects the ith element of the pair. **Write** creates a fresh port a, stores the expression e' at the port a, and stores the port a at the reference location ℓ. We use $S[\ell \mapsto a]$ as a shorthand for $\{\ell' \mapsto S(\ell') \mid \ell' \in dom(S) \wedge \ell' \neq \ell\} \cup \{\ell \mapsto a\}$. Note that the dummy witness symbol $\bullet$ is used as the run-time representation of any witness. That is, seman-

[2] In the literature [Launchbury 1993; Ariola et al. 1995], top-level `let`-bound variables often double as variables and ports.

tically, all witnesses are identical and the only important question is whether a witness has been reduced to normal form. Intuitively, a witness is like a dataflow token in dataflow machines. **Read** reads from the reference location ℓ and, as noted above, returns the value paired with a witness. **Ref** creates a fresh location ℓ and a fresh port a, initializes a to the expression e and ℓ to the port a. **Join** takes two witnesses and returns one witness.

Arrive is somewhat non-standard. v, defined in Figure 2, is called *value*. Intuitively, values are "safe to duplicate" expressions. **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 most operational semantics for functional languages [Plotkin 1975; Launchbury 1993] implicitly combine **Arrive** with other rules, we separate **Arrive** for increased freedom in the evaluation order. This separation also appears in the lazy calculus studied by Ariola and Felleisen [Ariola and Felleisen 1997]. Lastly, the rule **GC** garbage-collects unreachable (from the root port) portions of the expression store.

Here is an example of a λ_{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:

$$(\emptyset, \{\diamond \mapsto (\lambda x.\texttt{read}\, x\, \bullet)\ (\texttt{ref}\ 1)\}) \Rightarrow (\emptyset, \{\diamond \mapsto \texttt{read}\, a\, \bullet, a \mapsto \texttt{ref}\ 1\})\ \ \textbf{App}$$

Before defining confluence, we point out several important properties of this semantics. The evaluation contexts E do not extend to subexpressions of a λ abstraction, i.e., we do not reduce under λ. 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 λ_{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

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

Strictly speaking, our calculus is not quite call-by-need because a call-by-need evaluation treats pairs as non-values and shares them whereas our calculus treats pairs as values and copies them.

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

$$E := \ldots \mid (\lambda x.e)\ E \mid E \otimes e \mid v \otimes E \mid \texttt{write}\ \ell\ E\ e \mid \texttt{write}\ \ell\ v\ E \mid \texttt{ref}\ E \mid \texttt{let}\ x = E\ \texttt{in}\ e$$

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 $e \in V$.[3] Note that both lazy writes and strict writes are possible in λ_{wit}. The semantics of λ_{wit} has more freedom than that of Scheme [Matthews and Findler 2005] because choice of evaluation context is totally non-deterministic.

Note 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 λ_{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 $\Rightarrow$.

Definition 2.1 *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, λ_{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.)

Note that while confluence implies that if two evaluation orders terminate they will terminate with the same value, it does not imply that all orders of evaluation will terminate. This is indeed the case with pure lambda calculus, and is also case with our system presented in this paper.

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

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.

Intuitively, our goal is to 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.

A *trace graph* is a program trace with all information other than reads, writes, and witnesses elided. There are three kinds of nodes in a trace graph: read nodes *read*(ℓ), write nodes *write*(ℓ), and the join node *join*. Read and write nodes are parametrized by a reference location ℓ. 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:

$$\begin{array}{ll}\textbf{Write} & (S, E[\texttt{write } \ell\ e\ A]) \Rightarrow (S[\ell \mapsto a], E[B] \uplus \{a \mapsto e\})\\ & \quad \mathbb{V} := \mathbb{V} \cup \{B\} \text{ where } B \text{ is a new } \textit{write}(\ell) \text{ node}\\ & \quad \mathbb{E} := \mathbb{E} \cup \{(A, B)\}\\ \textbf{Read} & (S, E[\texttt{read } \ell\ A]) \Rightarrow (S, E[S(\ell) \otimes B])\\ & \quad \mathbb{V} := \mathbb{V} \cup \{B\} \text{ where } B \text{ is a new } \textit{read}(\ell) \text{ node}\\ & \quad \mathbb{E} := \mathbb{E} \cup \{(A, B)\}\\ \textbf{Join} & (S, E[\texttt{join } A\ B]) \Rightarrow (S, E[C])\\ & \quad \mathbb{V} := \mathbb{V} \cup \{C\} \text{ where } C \text{ is a new } \textit{join} \text{ node}\\ & \quad \mathbb{E} := \mathbb{E} \cup \{(A, C), (B, C)\}\end{array}$$

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. Note 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 λ_{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 3.1. 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 3.2. *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. We 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.

We write $A : \mathit{nodetype}$ to mean a node A of type $\mathit{nodetype}$. If we have $A : \mathit{read}(\ell)$ and $B : \mathit{write}(\ell)$, 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 $A : \mathit{read}(\ell)$, if there are two nodes $B_1, B_2 : \mathit{write}(\ell)$ 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 $C : \mathit{write}(\ell)$ 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 formalize this discussion. For $B : \mathit{write}(\ell)$, we use the shorthand $B \overset{!}{\rightsquigarrow} A$ if for any $C : \mathit{write}(\ell)$ such that $C \rightsquigarrow A$, we have $C \rightsquigarrow B$. Note that for any $A : \mathit{read}(\ell)$, there exists at most one $B : \mathit{write}(\ell)$ such that $B \overset{!}{\rightsquigarrow} A$. Note that the second condition above is equivalent to requiring that for any $A : \mathit{read}(\ell)$, either there is no $B : \mathit{write}(\ell)$ such that $B \rightsquigarrow A$ or there is a $B : \mathit{write}(\ell)$ such that $B \overset{!}{\rightsquigarrow} A$

Definition 3.3 *Witness Race Freedom.* We say that a trace graph $(\mathbb{V}, \mathbb{E})$ is witness race free if for every location ℓ,

—for every $A : \mathit{read}(\ell) \in \mathbb{V}$ and $B : \mathit{write}(\ell) \in \mathbb{V}$, either $A \rightsquigarrow B$ or $B \rightsquigarrow A$, and

—for every $A : \mathit{read}(\ell) \in \mathbb{V}$, either there is no $B : \mathit{write}(\ell) \in \mathbb{V}$ such that $B \rightsquigarrow A$ or there is a $B : \mathit{write}(\ell) \in \mathbb{V}$ 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 3.4. *If e is witness race free then e is confluent.*

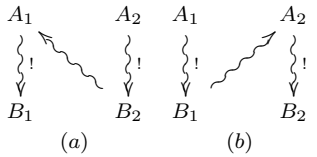

Fig. 3. Possible orderings between pairs of reads and writes in a witness race free trace graph.

The proof appears in Appendix A.

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 λ_{wit}, expressions are not reduced under λ 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.

Because witness race freedom is an entirely semantic condition, the result in this section can be extended to most other functional program transformations. 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 λ_{wit}. For example, the checking algorithm is unsound for call-by-name.

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 λ_{wit} program is witness race free. Witness race freedom may be checked directly by checking every program trace, which is obviously statically infeasible. Instead, we exploit a special property of witness-race-free trace graphs to design a sound algorithm that can verify a large subset of witness-race-free λ_{wit} programs.

The key observation is that any witness-race-free trace graph contains for each reference location ℓ a subgraph that we shall call 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 $A_1, A_2 : \mathit{write}(\ell)$ and $B_1, B_2 : \mathit{read}(\ell)$ 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 free-

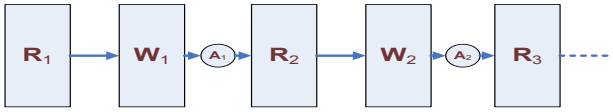

Fig. 4. A read-write pipeline with bottlenecks for a reference location ℓ.

dom 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 ℓ, 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 $\mathit{read}(\ell)$ nodes and $\mathit{write}(\ell)$ 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 $\mathit{write}(\ell)$ node and no $\mathbb{W}_i$ contains a $\mathit{read}(\ell)$ nodes. Each A_i is one $\mathit{write}(\ell)$ node. An arrow from X to Y means that there is a path from each $\mathit{write}(\ell)$ node or $\mathit{read}(\ell)$ node in X to each $\mathit{write}(\ell)$ node or $\mathit{read}(\ell)$ node in Y, except if a $\mathbb{W}_i$ contains no such node, then there is a path from each $\mathit{read}(\ell) \in \mathbb{R}_i$ to A_i. Each $\mathbb{R}_i$ for $i \neq 1$ must contain at least one $\mathit{read}(\ell)$. 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 4.1. Given a trace graph $(\mathbb{V}, \mathbb{E})$ and a reference location ℓ, we call its subgraph G_ℓ a read-write pipeline with bottlenecks if G_ℓ 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 $\mathit{write}(\ell)$ nodes and $\mathit{read}(\ell)$ 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 ℓ as a subgraph

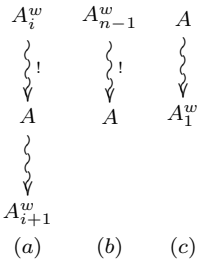

Fig. 5. The three cases for partitioning read nodes with respect to A_i^w nodes.

G_ℓ (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 4.2. *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 ℓ.*

PROOF. If
Suppose $(\mathbb{V}, \mathbb{E})$ has a read-write pipeline with bottlenecks for every ℓ. Let $\mathbb{R}_1, \mathbb{R}_2, \ldots, \mathbb{R}_n$ and $\mathbb{W}_1, \mathbb{W}_2, \ldots, \mathbb{W}_n$ be the collection of the nodes making up the read-write pipeline with bottlenecks for ℓ. Let $A : \mathit{read}(\ell), B : \mathit{write}(\ell) \in \mathbb{V}$. Then $A \in \mathbb{R}_i$ and $B \in \mathbb{W}_j$ for some i, j. If $i \leq j$, then $A \rightsquigarrow B$. Otherwise, $i > j$ and $B \rightsquigarrow A$. Let $A : \mathit{read}(\ell), B : \mathit{write}(\ell) \in \mathbb{V}$ and $B \rightsquigarrow A$. Then $A \in \mathbb{R}_i$ for some $i \neq 0$. So there is $C : \mathit{write}(\ell) \in \mathbb{R}_{i-1}$ such that $C \overset{!}{\rightsquigarrow} A$.

Only If
Suppose $(\mathbb{V}, \mathbb{E})$ is witness race free. Let ℓ be a reference location. Let $A_1^w, A_2^w, \ldots, A_{n-1}^w$ be the $\mathit{write}(\ell)$ nodes such that for each A_i^w, there is some $B : \mathit{read}(\ell)$ such that $A_i^w \overset{!}{\rightsquigarrow} B$. As discussed above, $A_1^w, A_2^w, \ldots, A_{n-1}^w$ are in some total order. Without loss of generality, we assume that $A_1^w \rightsquigarrow A_2^w \rightsquigarrow \ldots \rightsquigarrow A_{n-1}^w$.

We construct the collections $\mathbb{R}_1, \mathbb{R}_2, \ldots, \mathbb{R}_n$ and $\mathbb{W}_1, \mathbb{W}_2, \ldots, \mathbb{W}_n$ as follows. First, we add the node A_i^w to the collection $\mathbb{W}_i$ for each $i \neq n$. Let $A : \mathit{read}(\ell)$. Suppose there is some $B : \mathit{write}(\ell)$ where $B \rightsquigarrow A$. Then $A_i^w \overset{!}{\rightsquigarrow} A$ for some i. As discussed above, it must be the case that $A \rightsquigarrow A_{i+1}^w$ if $i \neq n-1$. So we put A in the collection $\mathbb{R}_{i+1}$. Figure 5 depicts this case when (a) $i \neq n-1$ and (b) $i = n-1$.

Otherwise, there is no $B : \mathit{write}(\ell)$ such that $B \rightsquigarrow A$. Hence $A \rightsquigarrow C$ for any $C : \mathit{write}(\ell)$. In particular, $A \rightsquigarrow A_1^w$. So we put A in the collection $\mathbb{R}_1$. This case corresponds to the diagram (c) in Figure 5.

At this point, we have successfully partitioned $\mathit{read}(\ell)$ nodes with respect to A_i^w's. What remains are the $\mathit{write}(\ell)$ nodes that are not among the A_i^w's. Let $A : \mathit{write}(\ell)$ be such a node. Suppose that there is no $B : \mathit{read}(\ell)$ such that $A \rightsquigarrow B$. Then it must be the case that for all $B : \mathit{read}(\ell)$, $B \rightsquigarrow A$. In particular, for all $B : \mathit{read}(\ell) \in \mathbb{R}_n$, $B \rightsquigarrow A$. So we put such a node A in the collection $\mathbb{W}_n$. Otherwise, there exists $B : \mathit{read}(\ell)$ such that $A \rightsquigarrow B$. Let i be the largest such that there is no $C : \mathit{read}(\ell) \in \mathbb{R}_i$ with $A \rightsquigarrow C$. Note that such i always exists since

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

Fig. 6. The syntax of λ^{reg}_{wit}.

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

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

Fig. 7. The semantics of λ^{reg}_{wit}.

no *read*(ℓ) nodes in the collection $\mathbb{R}_1$ can be reached from a *write*(ℓ) node. So, for each C : *read*(ℓ) $\in \mathbb{R}_i$, we have $C \rightsquigarrow A$. And since i is the largest, there exists D : *read*(ℓ) $\in \mathbb{R}_{i+1}$ such that $A \rightsquigarrow D$. Therefore, $A \rightsquigarrow A^w_i$. Hence we can put the node A in the collection $\mathbb{W}_i$. □

Corollary 4.3. *A λ_{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 ℓ.*

4.1 Regions

Corollary 4.3 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 ℓ. Therefore it suffices to design an algorithm for solving the latter problem. But before we do so, we make a slight change to λ_{wit} to make the problem more tractable. In λ_{wit}, there is a read-write pipeline with bottlenecks for each reference location ℓ, 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 6 shows λ^{reg}_{wit}, λ_{wit} extended with regions. The syntax contains two new expression kinds: $\texttt{letreg}\ x\ e$ which creates a new region and $\texttt{ref}\ e\ e'$ which places the newly created reference in region e'; $\texttt{ref}\ e\ e'$ replaces $\texttt{ref}\ e$. Figure 7 gives the semantics of λ^{reg}_{wit} which differs from λ_{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 duplicate, 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 $\texttt{letreg}\ x\ e$. Note that semantics

of `letreg` $x\ e$ does not impose any scoping restriction on the generated region (i.e., the region can be used outside of e). However, the type system does impose the scoping restriction to make type inference simple (see Section 4.2). The type system also imposes restriction on regions that are standard for static region systems, such as forcing two regions r and r' to be equal in

$$\lambda f.f\ r \otimes f\ r'$$

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 ℓ belongs to the region r. Note that the semantics does not actually associate the reference location ℓ and the region r, and therefore grouping of reference locations via regions is entirely conceptual. [4] The reader may find it helpful to look upon this construction as encoding the results of some static may-alias analysis in the run-time state. Indeed, an alternative approach is to use an automatic may-alias analysis instead of explicit regions, i.e., use inferred alias sets as regions. A disadvantage of such an approach is that the programmer may lose control over which reference belongs to which alias set.

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*(ℓ) nodes and *write*(ℓ) nodes, we use *read*(r) nodes and *write*(r) nodes. Formally, a trace graph for λ_{wit}^{reg} is constructed by the following graph construction below.

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.

Write $(R, K, S, E[\texttt{write}\,\ell\,e\,A]) \Rightarrow (R, K, S[\ell \mapsto a], E[B] \uplus \{a \mapsto e\})$
$\mathbb{V} := \mathbb{V} \cup \{B\}$ 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 $(R, K, S, E[\texttt{ref}\,e\,r]) \Rightarrow (R, K \uplus \{\ell \mapsto r\}, S \uplus \{\ell \mapsto a\}, E[\ell] \uplus \{a \mapsto e\})$

Since there is less information available in a λ_{wit}^{reg} trace graph than in a λ_{wit} trace graph, the witness race freedom condition is more conservative for λ_{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 λ_{wit}, which only

[4] Regions are traditionally coupled with some semantic meaning such as memory management [Tofte and Talpin 1994; Grossman et al. 2002]. It is possible to extend λ_{wit}^{reg} to do similar things with its regions.

requires some $C : \mathit{write}(r)$ 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 λ^{reg}_{wit} whether B_1 and B_2 both write to the same reference location.

Formally, witness race freedom for λ^{reg}_{wit} can be defined as follows.

Definition 4.4 *Witness Race Freedom for* λ^{reg}_{wit}. We say that a λ^{reg}_{wit} trace graph $(\mathbb{V}, \mathbb{E})$ is witness race free if for every region r,

—for every $A : \mathit{read}(r) \in \mathbb{V}$ and $B : \mathit{write}(r) \in \mathbb{V}$, either $A \rightsquigarrow B$ or $B \rightsquigarrow A$, and
—for every $A : \mathit{read}(r) \in \mathbb{V}$ and $B_1, B_2 : \mathit{write}(r) \in \mathbb{V}$ 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 4.5. *If a* λ^{reg}_{wit} *program* e *is witness race free, then* e *is confluent.*

PROOF. For any evaluation of e, carry out the same reduction sequence with the trace graph building action of λ_{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 3.4. □

It is easy to see that Definition 4.4 is the weakest possible restriction to the original witness race freedom under the region abstraction because for any λ^{reg}_{wit} 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 λ^{reg}_{wit}, 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 $\{A \mid A : \mathit{write}(r) \in \mathbb{W}_i\}$ for $i \neq n$ can be totally ordered (with $\rightsquigarrow$ as the ordering relation). The theorem below is immediate from Corollary 4.3 under this additional property.

THEOREM 4.6. *A* λ^{reg}_{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 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 λ^{reg}_{wit} 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 $\mathit{read}(r)$ node and $\mathit{write}(r)$ node gets a positive flow and every $A : \mathit{write}(r) \in \mathbb{W}_i$ 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 $\mathit{read}(r)$ node and $\mathit{write}(r)$ node gets a positive flow and each $A : \mathit{write}(r)$ that has some $B : \mathit{read}(r)$ such that $B \not\rightsquigarrow A$ gets a flow

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

Fig. 8. The type language.

equal to 1, then there is a read-write pipeline with bottlenecks for the region r. By Theorem 4.6, 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., $W : \mathbf{RegIDs} \to [0, 1]$. The rational number $W(\rho)$ indicates the flow amount for the static region identifier ρ 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.)

Before we present the type system formally, we informally describe the general idea. The type judgment has the following form $\Gamma, W \vdash e : \tau$ where W is the witness type described above, Γ is a type environment, i.e., a mapping from variables to types. Γ and τ are as usual, i.e., it means that e has the the type τ under the environment Γ, whereas W intuitively represents the amount of flow from virtual source nodes available for use by e. We need to be careful with W's, e.g., we must avoid copying it freely. For example, a rule for pairs looks as follows.

$$\frac{\Gamma; W \vdash e : \tau \quad \Gamma'; W' \vdash e' : \tau'}{\Gamma + \Gamma'; W + W' \vdash e \otimes e' : \tau \otimes \tau'}$$

Intuitively, we can read $W + W'$ as though the amount of flow usable by $e \otimes e'$ is "split up" between W, which is used by e, and W', which is used by e'. Note that Γ could contain witness types, and must also be split up in a similar fashion.

We now introduce the type system formally.

The rest of the types are defined in Figure 8. Types include integer types *int*, function types $\tau \xrightarrow{q} \tau'$, pair types $\tau \otimes \tau'$, reference types $\mathit{ref}\,(\tau, \tau', \rho)$, and region types $\mathit{reg}(\rho)$. 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 to take the valuation ∞ 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 9 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 **Sub** which we defer to the end. **Var** and **Int** are standard. **Dummy** gives a dummy witness $\bullet$ an empty witness type; note that $\emptyset(\rho) = 0$ for any static region identifier ρ.

Source uses additive arithmetic over types defined in Figure 10. Recall that since $\emptyset(\rho) = 0$ for all ρ, we have $\emptyset \times q = \emptyset$ and $\emptyset + W = W$ for any q and W. The **Source** rule adds W_3 amount of flow from the virtual source nodes (i.e., nodes A_S

$$\frac{\Gamma; W \vdash e : \tau \quad \tau \geq \tau'}{\Gamma; W \vdash e : \tau'} \ \textbf{Sub}$$

$$\frac{\Gamma(x) = \tau}{\Gamma; W \vdash x : \tau} \ \textbf{Var}$$

$$\frac{}{\Gamma; W \vdash i : \mathit{int}} \ \textbf{Int}$$

$$\frac{}{\Gamma; W \vdash \bullet : \emptyset} \ \textbf{Dummy}$$

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

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma; W \vdash e'[v/x] : \tau \quad x \notin \mathit{free}(v)}{\Gamma; W \vdash \texttt{let}\ x = v\ \texttt{in}\ e' : \tau} \ \textbf{LetA}$$

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

Fig. 9. Type judgment rules.

from the first paragraph of this section) to W_2. As we noted earlier informally, in the type judgment relation $\Gamma; W \vdash e : \tau$, the witness type W represents the portion

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

Fig. 10. Arithmetic over types.

of flow from virtual source nodes the expression e may use. Therefore, **Source** says that assuming that we took W_1 flow from virtual source nodes in the precondition, we are now taking W_3 more. We shall see later in the type rule for reads and writes that the **Source** rule is needed to give the required amount of flow to the read and write nodes.

In **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 Γ is defined as follows:

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

So for example, if $\lambda x.e$ 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. It is easy to see this multiplication rule as a natural extension of the usual used once versus used arbitrary many times rule of linear type systems [Turner et al. 1995] in that the function used only once means $q = 1$, and arbitrary many times means $q = \infty$.

In **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:

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

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

In a reference type $ref(\tau, \tau', \rho)$, the static region identifier ρ identifies the region where the reference belongs while the type τ is the read type of the reference and the type τ' is the write type of the reference. Initially the read and write types are the same as seen in **Ref**. **Write** matches the type of the to-be-assigned expression e_2 with the write type of the reference while **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 for 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:

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

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

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

$$\frac{\tau_1 \geq \tau_1' \quad \tau_2 \leq \tau_2'}{\mathit{ref}(\tau_1, \tau_2, \rho) \geq \mathit{ref}(\tau_1', \tau_2', \rho)} \qquad \frac{W(\rho) \geq W(\rho) \text{ for all } \rho \in \mathbf{RegIDs}}{W \geq W'}$$

Fig. 11. Subtyping.

occurrence of 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.

LetRegion introduces a fresh static region identifier ρ. 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. Here, $\mathit{free}(\tau)$ is the set of static region identifiers in the type τ. Here $\mathit{free}(W) = \{\rho \mid W(\rho) \neq 0\}$ and $\mathit{free}(\Gamma) = \bigcup_{\tau \in ran(\Gamma)} \mathit{free}(\tau)$.

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 is a safe to duplicate, i.e., e is some value 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'. where $\mathit{free}(W) = \{\rho \mid W(\rho) \neq 0\}$, and $\mathit{free}(\Gamma) = \bigcup_{\tau \in ran(\Gamma)} \mathit{free}(\tau)$.

Finally, we return to **Sub**. The subtyping relation is defined in Figure 11. 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 [Fähndrich et al. 1998]. 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 λ^{reg}_{wit} program e is well-typed if $\emptyset; \emptyset \vdash e : \tau$ for some type τ. The following theorem states that the type system is sound.

THEOREM 4.7. *If a λ^{reg}_{wit} program e is well-typed, then e is witness race free.*

The proof appears in Appendix B.

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; we may use the qualifier ∞ for all function types and use 0 for all flows. In general, we can give the ∞ qualifier to the function type of any function that does not capture a witness. We can also assign 0 to any flow for a region r that does not flow into a

side-effect primitive operating on the region r.

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 λ_{wit} programs into λ_{wit}^{reg}). In fact, assuming that $write(r)$ nodes in each collection $\mathbb{W}_i$ are totally ordered for each r, 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. We also show 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, and flow-insensitivity of reference types. Another limitation is that the type system enforces $write(r)$ nodes in every collection $\mathbb{W}_i$ to be totally ordered for each r whereas witness race freedom permits an absence of ordering for the case $i = n$; we believe that this is a minor limitation. For example, the program below is witness race free (and hence confluent) but fails to type check.

$$\begin{array}{l}\texttt{letreg}\, r\\ \texttt{let}\, q = \texttt{ref}\, 1\, r\, \texttt{in}\\ \texttt{write}\, q\, 1\, r \otimes \texttt{write}\, q\, 2\, r\end{array}$$

As briefly discussed at the end of Section 3, our type system is unsound with call-by-name reduction as is typical of a non-linear substructural type system. For example, the following program type checks but is not confluent.

$$\begin{array}{l}\texttt{letreg}\, r\\ \texttt{let}\, q = \texttt{ref}\, 1\, r\, \texttt{in}\\ \texttt{let}\, c = (\texttt{let}\, x = \texttt{read}\, q \bullet \texttt{in}\, \texttt{write}\, q\, \pi_1(x) + 1\, \pi_2(x))\, \texttt{in}\\ c \otimes c\end{array}$$

As observed by Sabry [Sabry 1998], essentially the same deficiency exists for implementations of monads.

Although we do not prove formally as it is not the focus of this paper, it is easy to prove via the standard syntactic method [Wright and Felleisen 1994; Grossman et al. 2002; Terauchi and Aiken 2004] that our type system is memory safe in the sense that a well-typed program does not access dangling pointers.

4.3 Inference

We next present the type inference algorithm. By Theorem 4.7, 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

$$
\begin{array}{l}
\mathit{Fresh}(\mathit{int}) = \mathit{int} \\
\mathit{Fresh}(\sigma{\to}\sigma) = \mathit{Fresh}(\sigma) \xrightarrow{\beta} \mathit{Fresh}(\sigma) \text{ where } \beta \text{ is fresh} \\
\mathit{Fresh}(\sigma \otimes \sigma') = \mathit{Fresh}(\sigma) \otimes \mathit{Fresh}(\sigma) \\
\mathit{Fresh}(\mathit{ref}(\sigma, \sigma', \rho)) = \mathit{ref}(\mathit{Fresh}(\sigma), \mathit{Fresh}(\sigma'), \rho) \\
\mathit{Fresh}(\mathit{reg}(\rho)) = \mathit{reg}(\rho) \\
\mathit{Fresh}(I) = \{\rho \mapsto \alpha \mid \rho \in I\} \text{ where } \alpha \text{ is fresh}
\end{array}
$$

Fig. 12. *Fresh*.

$$
\frac{\Gamma, W \vdash_b e^I : \tau, \mathcal{C}}{\Gamma, W + \mathit{Fresh}(I) \vdash_a e^I : \tau + \mathit{Fresh}(I), \mathcal{C}}
\qquad
\frac{\Gamma, W \vdash_b e^\sigma : \tau, \mathcal{C} \quad \sigma \text{is not a witness type}}{\Gamma, W \vdash_a e^\sigma : \tau, \mathcal{C}}
$$

Fig. 13. Type inference $\vdash_a$.

types inferred in this phase are:

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

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, albeit 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 σ: e^σ. 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 Γ, W, and τ.

The second-phase type inference rules are separated into two kinds, $\vdash_a$ (Figure 13) and $\vdash_b$ (Figure 14), 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 τ appearing in the algorithm are different from the ones in the type judgment rules. That is, instead of rational numbers, the types τ in the algorithm are qualified by *rational number variables* α, β, γ, 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:

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

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

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

$$\frac{}{\emptyset, \emptyset \vdash_b \bullet : \emptyset, \emptyset}\ \mathit{Dummy}$$

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

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

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

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

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

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

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

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

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

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

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

Fig. 14. Type inference $\vdash_b$.

Note that we omit annotations when they are not used (i.e., we say e instead of e^σ, 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 τ and W to account for a possible application of **Source**. *Fresh* is defined in Figure 12. 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^σ, we apply the rule *Var* to create a fresh τ from σ 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 (the rule *Int*) and dummy witnesses (the rule *Dummy*) are trivial. In the case of an abstraction $\lambda x.e$ (*Abs*), we multiply Γ and W passed from the subderivation by β. In the case of a function application $e_1^{\sigma\to\sigma'}\ e_2$, we apply the rule *App* to 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 β to be at least 1. Note that the type rule **Sub** is implicitly incorporated in the constraints. In the case of $\texttt{write}\ \mathit{ref}(\sigma, \sigma', \rho)\ e_2\ e_3$, we apply the type inference rule *Write* to add the constraint $\tau_2(\rho) \geq 1$ to match the type checking rule **Write**. Note that the first phase guarantees that $\rho \in \mathit{dom}(\tau_3)$. In the case $\texttt{letreg}\ x^{\mathit{reg}(\rho)}\ e$ we apply the rule *Read*. Note that 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 $\texttt{let}\ x = e\ \texttt{in}\ e'$ in the program by the expression $e'[e/x]$ whenever $e \in V$, $x \notin \mathit{free}(e)$, and x occurs more than once in e'.

As an example, consider the program shown below (a λ^{reg}_{wit} version of the last example from Section 2):

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

Suppose the first phase assigns r the type $\mathit{reg}(\rho)$. Assume each `let`-bound variable is treated monomorphically. Figure 15 shows the constraints for the `let`-bound expressions generated by the second phase (slightly simplified for readability). The final constraints, after some simplification, is as follows:

$$
\begin{array}{l}
\gamma_1 \geq 1, 1 \geq \gamma_1 + \gamma_2 + \gamma_3 \times \beta_1 + \gamma_4, \beta_1 \geq \beta_2 + \beta_3, \\
\beta_2 \geq 1, \beta_3 \geq 1, \gamma_1 + \gamma_2 \geq \alpha_1 \times \beta_1, \alpha_1 + \gamma_3 > 0, \\
\alpha_1 + \gamma_3 \geq \alpha_2, \alpha_3 + \alpha_4 + \gamma_4 \geq 1, \alpha_2 \geq \alpha_3, \alpha_2 \geq \alpha_4
\end{array}
$$

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

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

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.

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

$\Gamma; \{\rho \mapsto \gamma_1 + \gamma_2\} \vdash_a \texttt{write}\ x\ 2\ \bullet : \{\rho \mapsto \gamma_1 + \gamma_2\}, \{\gamma_1 \geq 1\}$
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 $\mathcal{C} = \{\alpha_3 + \alpha_4 + \gamma_4 \geq 1\}$

Fig. 15. Constraints generation example

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 β instead of α 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 4.8. *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. 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 τ, $\times$ 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:

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

Intuitively, this follows from the fact that *Var* uses *Fresh*σ and *Abs* multiplies

$\Gamma(x)$'s.

Third, for any $+$ that appears in a positive position of τ, 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 β is some $q < \infty$, then the iterative method finds q for β in $2|\mathcal{C}'|$ iterations. Hence any variable changing after the $2|\mathcal{C}'|$th iteration can be safely set to ∞. 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 a system of linear inequalities, which can be solved efficiently by a linear programming algorithm, such as the simplex 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 [Wadler 1990; Guzman and Hudak 1990; Achten et al. 1993] and monads [Moggi 1991; Odersky et al. 1993; Peyton Jones and Wadler 1993; Launchbury and Sabry 1997].

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 ∞ 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* [Launchbury and Sabry 1997; Semmelroth and Sabry

1999; Moggi and Sabry 2001]):

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

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 α as arguments and returns a witness, the region α, and a value of the type τ. This implementation has been formally investigated [Ariola and Sabry 1998]. 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 is related to Boyland's *fractional permissions* [Boyland 2003] which can guarantee non-interference in imperative programs. However, unlike in Boyland's work which is purely static and can be cast as an extension of *capability calculus* [Crary et al. 1999] which was introduced as a system for reasoning about resource usage in imperative programs, witnesses are first class values that can be manipulated explicitly by the programmer to order side effects in non-strict languages. However, our static system closely resembles that of Boyland's in that they both use rational numbers (fractions). In this paper, we have shown that not only do fractions allow more flexibility (e.g., concurrent reads) than more rigid set-oriented capabilities in the original capability calculus, but are more convenient to reason with because splitting and joining are defined by simple linear arithmetic and the type inference can be converted to the problem of solving a system of linear inequalities. Recently, we have extended the fractional capabilities framework to check (partial) determinism of concurrent programs (more specifically, programs consisting of concurrently running sequential processes communicating via buffered and unbuffered channels and reference cells) [Terauchi and Aiken 2006]. The checking algorithm is similar to the type inference algorithm in this paper.

While our *trace graph* has some resemblance to Sestoft's *path semantics* [Sestoft 1989] in that they both record when variables are read or written, they are actually quite different because *path semantics* is only defined for strict left-to-right call-by-

value languages, and has a rather different goal: compiler optimization to reduce stack space usage of purely applicable languages where global mutable state are not directly accessible by the programmer.

6. CONCLUSIONS

We have presented a new approach to adding global mutable state 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

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

Ariola, Z. M. and Felleisen, M. 1997. The call-by-need lambda calculus. *Journal of Functional Programming 7,* 3, 265–301.

Ariola, Z. M., Maraist, J., Odersky, M., Felleisen, M., and Wadler, P. 1995. A call-by-need lambda calculus. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California, 233–246.

Ariola, Z. M. and Sabry, A. 1998. Correctness of monadic state: an imperative call-by-need calculus. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, California, 62–74.

Boyland, J. 2003. Checking interference with fractional permissions. In *Static Analysis, Tenth International Symposium*. San Diego, CA, 55–72.

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

Fähndrich, M., Foster, J. S., Su, Z., and Aiken, A. 1998. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Montreal, Canada, 85–96.

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

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

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

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

Matthews, J. and Findler, R. B. 2005. An operational semantics for r5rs scheme. In *2005 Workshop on Scheme and Functional Programming*.

Moggi, E. 1991. Notions of computation and monads. *Information and Computation 93,* 1, 55–92.

Moggi, E. and Sabry, A. 2001. Monadic encapsulation of effects: a revised approach (extended version). *Journal of Functional Programming 11,* 6, 591–627.

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

App	$(L, E[(\lambda x.e)\ e']) \Rightarrow_{alt} (L, E[e[a/x]] \uplus \{a \mapsto e'\})$
Let	$(L, E[\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e']) \Rightarrow_{alt} (L, E[e'[a/x]] \uplus \{a \mapsto e[a/x]\})$
Pair	$(L, E[\pi_i(e_1 \otimes e_2)]) \Rightarrow_{alt} (L, E[e_i])$
Write	$(L, E[\texttt{write}\ \langle \ell, a' \rangle\ e\ T]) \Rightarrow_{alt} (L, E[\mathbf{write}(T, \ell, a)] \uplus \{a \mapsto e\})$
Read	$(L, E[\texttt{read}\ \langle \ell, a \rangle\ T]) \Rightarrow_{alt} (L, E[\mathbf{read}(T, \ell, a) \otimes T])$
Ref	$(L, E[\texttt{ref}\ e]) \Rightarrow_{alt} (L \uplus \{\ell\}, E[\langle \ell, a \rangle] \uplus \{a \mapsto e\})$
Join	$(L, E[\texttt{join}\ T\ T']) \Rightarrow_{alt} (L, E[\mathbf{join}(T, T')])$
Arrive	$(L, E[a] \uplus \{a \mapsto e\}) \Rightarrow_{alt} (L, E[e] \uplus \{a \mapsto e\})$ where $e \in V$
GC	$(L, D \uplus D') \Rightarrow_{alt} (L, D)$ where $\diamond \notin dom(D') \wedge dom(D') \cap free(D) = \emptyset$

Fig. 16. The alternative semantics for λ_{wit}.

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

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

Sabry, A. 1998. What is a purely functional language? *Journal of Functional Programming 8,* 1, 1–22.

Semmelroth, M. and Sabry, A. 1999. Monadic encapsulation in ml. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming*. Paris, France, 8–17.

Sestoft, P. 1989. Replacing function parameters by global variables. In *FPCA '89 Conference on Functional Programming Languages and Computer Architecture*. London, United Kingdom.

Terauchi, T. and Aiken, A. 2004. Memory Management with Use-counted Regions. Tech. Rep. UCB//CSD-04-1314, University of California, Berkeley. Mar.

Terauchi, T. and Aiken, A. 2005. Witnessing side-effects. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*. Tallinn, Estonia, 105–115.

Terauchi, T. and Aiken, A. 2006. A capability calculus for concurrency and determinism. In *CONCUR 2006 - Concurrency Theory, 17th International Conference*. Bonn, Germany, 218–232.

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

Turner, D. N., Wadler, P., and Mossin, C. 1995. Once upon a type. In *FPCA '95 Conference on Functional Programming Languages and Computer Architecture*. La Jolla, California, 1–11.

Wadler, P. 1990. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, M. Broy and C. Jones, Eds. North Holland, 347–359.

Wright, A. K. and Felleisen, M. 1994. A Syntactic Approach to Type Soundness. *Information and Computation 115,* 1, 38–94.

A. PROOF OF THEOREM 3.4

Theorem 3.4. *If e is witness race free then e is confluent.*

We prove the theorem by showing that the λ_{wit} semantics is equivalent to an alternative semantics provided that e is witness race free. The alternative semantics is confluent for any program, witness-race-free or not. Hence it follows that e is confluent with the λ_{wit} semantics.

Figure 16 shows this alternative semantics. Evaluation contexts E are unchanged. L is some set of reference locations. The new expression kind $\langle \ell, a \rangle$ is merely a pair of a reference location and a port, and is also a value (i.e., $\langle \ell, a \rangle \in V$). A *table* T is a function from reference locations to ports. The symbol $\bullet$ is interpreted as a table

such that $\bullet(\ell) = \bot$ for all ℓ. Operations on tables are defined as follows:

$$\begin{aligned}
\mathbf{write}(T, \ell, a) &= \{\ell' \mapsto a' \mid \ell' \mapsto a' \in T \wedge \ell' \neq \ell\} \cup \{\ell \mapsto a\} \\
\mathbf{read}(T, \ell, a) &= \begin{cases} a & \text{if } T(\ell) = \bot \\ T(\ell) & \text{if } T(\ell) \neq \bot \end{cases} \\
\mathbf{join}(T, T') &= \{\ell \mapsto T(\ell) \mid T'(\ell) = T(\ell) \vee T'(\ell) = \bot\} \\
&\quad \cup \{\ell \mapsto T'(\ell) \mid T(\ell) = \bot\} \\
&\quad \cup \{\ell \mapsto \bot \mid T(\ell) = T'(\ell) = \bot \vee T(\ell) \neq T'(\ell)\}
\end{aligned}$$

Two states (L, D) and (L', D') are defined to be observationally equivalent if $\mathbf{Erase}(D) \approx \mathbf{Erase}(D')$ where $\mathbf{Erase}(D)$ is D but with each occurrence of a table replaced by $\bullet$ and each occurrence of $\langle \ell, a \rangle$ replaced by ℓ. Note that there are no reference stores in the alternative semantics, i.e., the alternative semantics is trivially side effect free. Therefore, while we will not prove it formally, it is not hard to see that the alternative semantics is always confluent, i.e., for any states (L, D), (L_1, D_1) and (L_2, D_2) such that $(L, D) \Rightarrow_{alt}^* (L_1, D_1)$ and $(L, D) \Rightarrow_{alt}^* (L_2, D_2)$, there exist states (L_1', D_1') and (L_2', D_2') such that $(L_1, D_1) \Rightarrow_{alt}^* (L_1', D_1')$, $(L_2, D_2) \Rightarrow_{alt}^* (L_2', D_2')$, and $\mathbf{Erase}(D_1') \approx \mathbf{Erase}(D_2')$. Hence, it suffices to prove that if a program e is witness race free then for any evaluation $(\emptyset, D = \{\diamond \mapsto e\}) \Rightarrow^* (S, D_1)$ there is an evaluation $(\emptyset, D) \Rightarrow_{alt}^* (L, D_2)$ such that for any evaluation $(L, D_2) \Rightarrow_{alt}^* (L', D_2')$ there is an evaluation $(S, D_1) \Rightarrow^* (S', D_1')$ such that $D_1' \approx \mathbf{Erase}(D_2')$.

We augment the graph constructing semantics slightly by adding information about the written port to each *write*(ℓ) node:

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

For observational equivalence, we ignore this information, i.e., formally, each node is replaced by $\bullet$.

Our idea is to show that for a witness-race-free program e, $\Rightarrow$ and $\Rightarrow_{alt}$ can simulate each other while maintaining the following relationship.

Definition A.1. $(S, D_1) \sim_{(\mathbb{V}, \mathbb{E})} (L, D_2)$ if

—$L = dom(S)$,
—$D_1 = \boldsymbol{Erase}(D_2)$,
—if ℓ has not been written, i.e., there is no *write*(ℓ) nodes in $\mathbb{V}$, then for any occurrence of $\langle \ell, a \rangle$ in D_2, $a = S(\ell)$, and
—for any table T occurring in D_2, $T(\ell) = a \neq \bot$ iff there exists a node B : *write*(ℓ, a) such that $B \overset{!}{\rightsquigarrow} A$ where A is the node associated with T.

The phrase *node associated with a table* used in the last sentence is defined as follows: the table T at the expression store $D_2 = E[T]$ is associated with the node A at the expression store $D_1 = \mathbf{Erase}(E)[A]$.

We now state the main claim which carries out the aforementioned simulation.

Lemma A.2. *Let a program e be witness race free. Then there exists a set Ω_e such that $((\emptyset, \{\diamond \mapsto e\}), (\emptyset, \emptyset), (\emptyset, \{\diamond \mapsto e\})) \in \Omega_e$ and for any triple $((S, D_1), (\mathbb{V}, \mathbb{E}), (L, D_2)) \in \Omega_e$,*

—$(S, D_1) \sim_{(\mathbb{V},\mathbb{E})} (L, D_2)$,

—$(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^* (S, D_1)$ *with the trace graph* $(\mathbb{V}, \mathbb{E})$,

—*if* $(S, D_1) \Rightarrow (S', D_1')$ *with the corresponding trace graph action updating the trace graph from the state* $(\mathbb{V}, \mathbb{E})$ *to the state* $(\mathbb{V}', \mathbb{E}')$, *then there exists a state* (L', D_2') *such that* $(L, D_2) \Rightarrow_{alt} (L', D_2')$ *and* $((S', D_1'), (\mathbb{V}', \mathbb{E}'), (L', D_2')) \in \Omega_e$, *and*

—*if* $(L, D_2) \Rightarrow_{alt} (L', D_2')$, *then there exists a state* (S', D_1') *and a trace graph* $(\mathbb{V}', \mathbb{E}')$ *such that* $(S, D_1) \Rightarrow (S', D_1')$ *with the corresponding trace graph action updating the trace graph from the trace graph* $(\mathbb{V}, \mathbb{E})$ *to the trace graph* $(\mathbb{V}', \mathbb{E}')$ *and* $((S', D_1'), (\mathbb{V}', \mathbb{E}'), (L', D_2')) \in \Omega_e$.

The first condition says that the two states in a triple in Ω_e are $\sim$-related with the trace graph in the triple. The second condition says that these states can be reached while generating the trace graph. The third and fourth conditions are the simulation steps, i.e., showing that a step in $\Rightarrow$ can be simulated by a step in $\Rightarrow_{alt}$ and vice versa.

Proof. Our proof constructs Ω_e inductively. For the base case, it is easy to see that $(\emptyset, \{\diamond \mapsto e\}) \sim_{(\emptyset,\emptyset)} (\emptyset, \{\diamond \mapsto e\})$ and $(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^* (\emptyset, \{\diamond \mapsto e\})$ with the trace graph $(\emptyset, \emptyset)$. Therefore, we may set $\Omega_e = \{((\emptyset, \{\diamond \mapsto e\}), (\emptyset, \emptyset), (\emptyset, \{\diamond \mapsto e\}))\}$ initially.

The inductive case is split into case by reduction kinds. Pick $((S, D_1), (\mathbb{V}, \mathbb{E}), (L, D_2)) \in \Omega_e$.

App, **Let**, **Pair**, **Arrive**, **GC**

Suppose $D_1 = E_1[(\lambda x.e)\ e']$ and we took a **App** step so that

$$(S, E_1[(\lambda x.e_1)\ e_1']) \Rightarrow (S, E_1[e_1[a/x]] \uplus \{a \mapsto e_1'\})$$

Note that the trace graph $(\mathbb{V}, \mathbb{E})$ is not updated by this reduction. Let us take a $\Rightarrow_{alt}$ version of **App** from the state (L, D_2) so that

$$(L, D_2 = E_2[(\lambda x.e_2)\ e_2']) \Rightarrow_{alt} (L, E_2[e_2[a/x]] \uplus \{a \mapsto e_2'\})$$

where $\mathbf{Erase}(E_2) = E_1$. Such E_2 exists since $D_1 = \mathbf{Erase}(D_2)$. Also, it is easy to see that

$$(S, E_1[e_1[a/x]] \uplus \{a \mapsto e_1'\}) \sim_{(\mathbb{V},\mathbb{E})} (L, E_2[e_2[a/x]] \uplus \{a \mapsto e_2'\})$$

So we add $((S, E_1[e_1[a/x]] \uplus \{a \mapsto e_1'\}), (\mathbb{V}, \mathbb{E}), (L, E_2[e_2[a/x]] \uplus \{a \mapsto e_2'\}))$ to Ω_e. The converse case where we take the **App** step from the state (L, D_2) is analogous. A similar argument works for the case a **Let** step, a **Pair** step, an **Arrive** step, or a **GC** step is taken from the state (S, D_1) or the state (L, D_2).

Write

Suppose $D_1 = E_1[\texttt{write}\ \ell\ e_1\ A]]$ and we took a **Write** step so that

$$(S, E_1[\texttt{write}\ \ell\ e_1\ A]]) \Rightarrow (S[\ell \mapsto a], D_1' = (E_1[B] \uplus \{a_1 \mapsto e\}))$$

with $\mathbb{V}' = \mathbb{V} \cup \{B\}$ and $\mathbb{E}' = \mathbb{E} \cup \{(A, B)\}$ where $B : \mathit{write}(\ell, a)$. Let us take a $\Rightarrow_{alt}$ version of **Write** from (L, D_2) so that

$$(L, D_2 = E_2[\texttt{write}\ \langle \ell, a' \rangle\ e_2\ T]) \Rightarrow_{alt} (L, D_2' = (E_2[\mathbf{write}(T, \ell, a)] \uplus \{a \mapsto e_2\}))$$

where $\mathbf{Erase}(E_2) = E_1$. Clearly, $D_1' = \mathbf{Erase}(D_2')$. For any table T in the expression store D_2 other than the table $T' = \mathbf{write}(T, \ell, a)$ at the context $E_2[\ ]$, it is easy to see that the fourth condition from the definition of $\sim_{(\mathbb{V}', \mathbb{E}')}$ holds since there is no node reachable from the newly added node B. So consider the port $T'(\ell')$. If $\ell' = \ell$, then $T'(\ell) = a$, which is consistent with the condition since $B \overset{!}{\rightsquigarrow} B$ and $B : \mathit{write}(\ell, a)$. On the other hand, if $\ell' \neq \ell$, then $T'(\ell') = T(\ell)$, and again this is consistent with the condition because the node A is associated with the table T and for any $C : \mathit{write}(\ell', a')$, $C \overset{!}{\rightsquigarrow} B$ if and only if $C \overset{!}{\rightsquigarrow} A$. Therefore, $(S[\ell \mapsto a], D_1) \sim_{(\mathbb{V}', \mathbb{E}')} (L, D_2)$. So we add the triple $((S[\ell \mapsto a], D_1), (\mathbb{V}', \mathbb{E}'), (L, D_2))$ to the set Ω_e. The converse case where we take a **Write** step from the state (L, D_2) is analogous.

Ref

Suppose $D_1 = E_1[\mathtt{ref}\ e_1]$ and we took a **Ref** step so that

$$(S, D_1 = E_1[\mathtt{ref}\ e_1]) \Rightarrow (S \uplus \{\ell \mapsto a\}, D_1' = (E_1[\ell] \uplus \{a \mapsto e_1\}))$$

Note that the trace graph $(\mathbb{V}, \mathbb{E})$ is not updated by this reduction. Let us take a $\Rightarrow_{alt}$ version of **Ref** from the state (L, D_2) so that

$$(L, D_2 = E_2[\mathtt{ref}\ e_2]) \Rightarrow_{alt} (L \uplus \{\ell\}, D_2' = (E_2[\langle \ell, a \rangle] \uplus \{a \mapsto e_2\}))$$

where $\mathbf{Erase}(E_2) = E_1$. Clearly, $D_1' = \mathbf{Erase}(D_2')$. Also, $L \uplus \{\ell\} = \mathit{dom}(S \uplus \{\ell \mapsto a\})$. The reference location ℓ has not been written and $a = (S \uplus \{\ell \mapsto a\})(\ell)$. Therefore $(S \uplus \{\ell \mapsto a\}, D_1) \sim_{(\mathbb{V}', \mathbb{E}')} (L \uplus \{\ell\}, D_2)$. So we add the triple $((S \uplus \{\ell \mapsto a\}, D_1), (\mathbb{V}, \mathbb{E}), (L \uplus \{\ell\}, D_2))$ to Ω_e. The converse case where we take a **Ref** step from the state (L, D_2) is analogous.

Read

Suppose $D_1 = E_1[\mathtt{read}\ \ell\ A]$ and we took a **Read** step so that

$$(S, D_1 = E_1[\mathtt{read}\ \ell\ A]) \Rightarrow (S, D_1' = E_1[S(\ell) \otimes B])$$

with $\mathbb{V}' = \mathbb{V} \cup \{B\}$ and $\mathbb{E}' = \mathbb{E} \cup \{(A, B)\}$ where $B : \mathit{read}(\ell)$. Let us take a $\Rightarrow_{alt}$ version of **Read** from the state (L, D_2) such that

$$(L, D_2 = E_2[\mathtt{read}\ \langle \ell, a \rangle\ T]) \Rightarrow_{alt} (L, D_2' = E_2[\mathbf{read}(T, \ell, a) \otimes T])$$

where $\mathbf{Erase}(E_2) = E_1$. Now consider the table $\mathbf{read}(T, \ell, a)$. If the reference location ℓ has not been written, i.e., there are no $\mathit{write}(\ell)$ nodes in the vertex set $\mathbb{V}$, then we have $S(\ell) = a$. Also, $T(\ell) = \bot$ since otherwise there is some $C : \mathit{write}(\ell)$ such that $C \overset{!}{\rightsquigarrow} A$, which contradicts the statement we just made. Hence $\mathbf{read}(T, \ell, a) = a$. Otherwise, the reference location ℓ has been written, i.e., there exists $C : \mathit{write}(\ell)$ in the vertex set $\mathbb{V}$. By witness race freedom, it must be the case that either $C \rightsquigarrow B$ or $B \rightsquigarrow C$, but since B is a newly added node, it must be the case that $C \rightsquigarrow B$. Therefore, again by witness race freedom, it must be the case that there exists a node $C' : \mathit{write}(\ell, a')$ for some port a' such that $C' \overset{!}{\rightsquigarrow} B$. Obviously, $C' \overset{!}{\rightsquigarrow} A$. Therefore $\mathbf{read}(T, \ell, a) = a'$. Suppose for contradiction that $S(\ell) = a'' \neq a'$. Then there must be a node $C'' : \mathit{write}(\ell, a'')$ such that this node

was added after C' was added. But by witness race freedom, it must be the case that $C'' \rightsquigarrow B$. Hence $C'' \rightsquigarrow C'$ by the definition of $\overset{!}{\rightsquigarrow}$. But this implies that C'' was added before C' was added, a contradiction. Hence $S(\ell) = a$ and $D'_1 = \mathbf{Erase}(D'_2)$. Lastly, for any C, $C \overset{!}{\rightsquigarrow} A$ if and only if $C \overset{!}{\rightsquigarrow} B$. Therefore $(S, D_1) \sim_{(\mathbb{V}', \mathbb{E}')} (L, D_2)$. So we add the triple $((S, D_1), (\mathbb{V}', \mathbb{E}'), (L, D_2))$ to the set Ω_e. The converse case where we take a **Read** step from the state (L, D_2) is analogous.

Join

Suppose $D_1 = E_1[\texttt{join}\, A\, B]$ and we took a **Join** step so that

$$(S, E_1[\texttt{join}\, A\, B]) \Rightarrow (S, E_1[C])$$

with $\mathbb{V}' = \mathbb{V} \cup \{C\}$ and $\mathbb{E}' = \mathbb{E} \cup \{(A, C), (B, C)\}$ where $C : \mathit{join}$. Let us take a $\Rightarrow_{alt}$ version of **Join** from the state (L, D_2) so that

$$(L, D_2 = E_2[\texttt{join}\, T\, T']) \Rightarrow_{alt} (L, D'_2 = E_2[\mathbf{join}(T, T')])$$

where $\mathbf{Erase}(E_2) = E_1$. Clearly, $D'_1 = \mathbf{Erase}(D'_2)$.

Consider the table $\mathbf{join}(T, T')(\ell)$. Because C is a new node, there exists $C' : \mathit{write}(\ell, a)$ such that $C' \overset{!}{\rightsquigarrow} C$ iff either

(1) $C' \overset{!}{\rightsquigarrow} A$ and $C' \overset{!}{\rightsquigarrow} B$,

(2) $C' \overset{!}{\rightsquigarrow} A$ and there exists no $C'' : \mathit{write}(\ell, a)$ such that $C'' \neq C'$ and $C'' \overset{!}{\rightsquigarrow} B$, or

(3) $C' \overset{!}{\rightsquigarrow} B$ and there exists no $C'' : \mathit{write}(\ell, a)$ such that $C'' \neq C'$ and $C'' \overset{!}{\rightsquigarrow} A$.

For case 1, $T(\ell) = T'(\ell) = a$. For case 2, $T(\ell) = a$ and $T'(\ell) = \bot$. For case 3, $T(\ell) = \bot$ and $T'(\ell) = a$. In all three cases, $\mathbf{join}(T, T')(\ell) = a$. Therefore $(S, D_1) \sim_{(\mathbb{V}', \mathbb{E}')} (L, D_2)$. So we add the triple $((S, D_1), (\mathbb{V}', \mathbb{E}'), (L, D_2))$ to the set Ω_e. The converse case where we take a **Join** step from the state (L, D_2) is analogous. □

The preceding lemma implies that for a witness-race-free e, any evaluation $(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^* (S, D)$ has a corresponding simulation $(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^*_{alt} (L, D_{alt})$ such that any evaluation $(L, D_{alt}) \Rightarrow^*_{alt} (L', D'_{alt})$ has a corresponding simulation $(S, D) \Rightarrow^* (S', D')$ such that $D' \approx \mathbf{Erase}(D'_{alt})$.[5]

So suppose $(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^* (S_1, D_1)$ and $(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^* (S_2, D_2)$. Then there are $(L_1, D_{1,alt})$ and $(L_2, D_{2,alt})$ such that

—$(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^*_{alt} (L_1, D_{1,alt})$,

—$(\emptyset, \{\diamond \mapsto e\}) \Rightarrow^*_{alt} (L_2, D_{2,alt})$,

—for any evaluation $(L_1, D_{1,alt}) \Rightarrow^*_{alt} (L'_1, D'_{1,alt})$, there is (S'_1, D'_1) such that $(S_1, D_1) \Rightarrow^* (S'_1, D'_1)$ and $D'_1 \approx \mathbf{Erase}(D'_{1,alt})$, and

—for any evaluation $(L_2, D_{2,alt}) \Rightarrow^*_{alt} (L'_2, D'_{2,alt})$, there is (S'_2, D'_2) such that $(S_2, D_2) \Rightarrow^* (S'_2, D'_2)$ and $D'_2 \approx \mathbf{Erase}(D'_{2,alt})$.

[5]In fact, Lemma A.2 implies that there is one that maintains $\approx$ relation at every step of the simulation. But $\approx$ at the end of the simulation is sufficient to prove the theorem.

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

App $(R, K, S, E[(\lambda x.e)\ e']) \Rightarrow_{flow} (R, K, S, E[e[a/x]] \uplus \{a \mapsto e'\})$

Let $(R, K, S, E[\texttt{let}\ x = e\ \texttt{in}\ e']) \Rightarrow_{flow} (R, K, S, E[e'[a/x]] \uplus \{a \mapsto e[a/x]\})$

Pair $(R, K, S, E[\pi_i(e_1 \otimes e_2)]) \Rightarrow_{flow} (R, K, S, E[e_i])$

Write $(R, K, S, E[\texttt{write}\ \ell\ e\ \langle A, P\rangle] \Rightarrow_{flow} (R, K, S[\ell \mapsto a], E[\langle B, P\rangle] \uplus \{a \mapsto e\})$
$\mathbb{V} := \mathbb{V} \cup \{B\}$ where B is a new $write(K(\ell))$ node
$\mathbb{E} := \mathbb{E} \cup \{(A, B)\}$
for each r flow $P(r)$ from A to B

Read $(R, K, S, E[\texttt{read}\ \ell\ \langle A, P\rangle]) \Rightarrow_{flow} (R, K, S, E[S(\ell) \otimes \langle B, P\rangle])$
$\mathbb{V} := \mathbb{V} \cup \{B\}$ where B is a new $read(K(\ell))$ node
$\mathbb{E} := \mathbb{E} \cup \{(A, B)\}$
for each r flow $P(r)$ from A to B

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

Join $(R, K, S, E[\texttt{join}\ \langle A, P\rangle\ \langle B, P'\rangle]) \Rightarrow_{flow} (R, K, S, E[\langle C, P + P'\rangle])$
$\mathbb{V} := \mathbb{V} \cup \{C\}$ where C is a new $join$ node
$\mathbb{E} := \mathbb{E} \cup \{(A, C), (B, C)\}$
for each r flow $P(r)$ from A to C
for each r flow $P'(r)$ from B to C

LetReg $(R, K, S, E[\texttt{letreg}\ x\ e]) \Rightarrow_{flow} (R \uplus \{r \mapsto 1\}, K, S, E[e[a/x]] \uplus \{a \mapsto r\})$

Arrive $(R, K, S, E[a] \uplus \{a \mapsto e\}) \Rightarrow_{flow} (R, K, S, E[e_1] \uplus \{a \mapsto e_2\})$ where $e \in V \wedge e = e_1 + e_2$

GC $(R, K, S, D \uplus D') \Rightarrow_{flow} (R, K, S, D)$ where $\diamond \notin dom(D') \wedge dom(D') \cap free(D) = \emptyset$

Source $(R + P', K, S, E[\langle A, P\rangle]) \Rightarrow_{flow} (R, K, S, E[\langle A, P + P'\rangle])$
for each r flow $P(r)$ from the source node to A

Fig. 17. Flow annotated λ^{reg}_{wit} semantics.

$$
\begin{array}{l}
e + e = e \text{ where } e = x, i, a, r, \ell, \text{ or } \lambda x.e \\
e_1 \otimes e_2 + e'_1 \otimes e'_2 = (e_1 + e_2) \otimes (e'_1 + e'_2) \\
\langle A, P_1\rangle + \langle A, P_2\rangle = \langle A, P_1 + P_2\rangle \\
P_1 + P_2 = \{r \mapsto P_1(r) + P_2(r) \mid r \in \textbf{Regions}\}
\end{array}
$$

Fig. 18. Additive arithmetic of $e \in V$.

But since $\Rightarrow_{alt}$ is confluent, there are $(L'_1, D'_{1,alt})$ and $(L'_2, D'_{2,alt})$ such that $\textbf{Erase}(D'_{1,alt}) \approx \textbf{Erase}(D'_{2,alt})$. Hence there are (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 $\textbf{Erase}(D'_1) \approx \textbf{Erase}(D'_2)$, i.e., e is confluent even with $\Rightarrow$.

B. PROOF OF THEOREM 4.7

THEOREM 4.7. *If a λ^{reg}_{wit} program e is well-typed, then e is witness race free.*

We prove the result by showing that any evaluation of e generates a witness-race-free trace graph. By Theorem 4.2, it suffices to show that the evaluation builds a read-write pipeline with bottlenecks for every r. Then, as discussed in Section 4.2, it suffices to show that for each r there exists flow assignments to the trace graph with a source node having flow 1 such that every $read(r)$ node gets a positive flow (i.e., flow > 0) and every $write(r)$ node gets a flow equal to 1.

To this end, we annotate the semantics with flow information such that generated trace graph is constructed together with flow assignments. The resulting seman-

tics $\Rightarrow_{flow}$ shown in Figure 17 has the following differences from the unannotated version.

(1) R's are now mapping from some set of regions to rational numbers in the range $[0, 1]$. Intuitively, $R(r)$ represents the amount of flow remaining in the source node for r.
(2) Each witness (i.e., a graph node) is paired with a *packet* P. A packet P is a function from regions to rational numbers in the range $[0, 1]$. Intuitively, the pair $\langle A, P\rangle$ implies that $P(r)$ amount of flow for region r is flowing from the node A.
(3) **Arrive** "splits" the expression e into expressions e_1 and e_2 instead of duplicating e. Splitting is defined using the additive arithmetic $e_1 + e_2 = e$ given in Figure 18.[6] Note that the addition $P_1 + P_2$ is also used at the **Join** rule to combine two packets.
(4) Flow assignments are made at **Read**, **Write**, and **Join**.
(5) The new reduction rule **Source** is introduced to account for flow from the source node to an arbitrary node.

Any $\Rightarrow_{flow}$ evaluation sequence has a corresponding unannotated evaluation sequence, i.e., replace packets by $\bullet$ and bypass **Source** steps. Conversely, any $\Rightarrow$ evaluation sequence has at least one corresponding flow-annotated evaluation sequence. However, this correspondence is not a bijection since different $\Rightarrow_{flow}$ evaluation sequences may correspond to the same $\Rightarrow$ evaluation sequence. In particular, even for a well-typed-program, there may be a $\Rightarrow_{flow}$ evaluation sequence which does not have the proper flow requirement, i.e., some $read(r)$ node gets flow ≤ 0 or some $write(r)$ node gets flow < 1.

Our goal is to show that for any $\Rightarrow$ evaluation sequence of a well-typed program e, there is a corresponding $\Rightarrow_{flow}$ evaluation sequence with proper flow assignments. The proof is by subject reduction, i.e., a $\Rightarrow$ step from a *well-typed state* X always has (a) corresponding $\Rightarrow_{flow}$ step(s) taking X to another well-typed state Y. As we shall see, we can always assign proper flow as long as we are in well-typed states. But before we define well-typed states, we need to extend the type system to type non-source expression kinds:

$$\frac{\text{for each } r \in dom(\Gamma), P(r) = W'(\rho) \text{ where } \Gamma(r) = reg(\rho)}{\Gamma; W \vdash \langle A, P\rangle : W'} \textbf{ Packet}$$

$$\frac{\Gamma(r) = reg(\rho)}{\Gamma; W \vdash r : reg(\rho)} \textbf{ Region} \qquad \frac{\Gamma(\ell) = ref\,(\tau, \tau', \rho)}{\Gamma; W \vdash \ell : ref\,(\tau, \tau', \rho')} \textbf{ Loc}$$

$$\frac{\Gamma(a) = \tau}{\Gamma; W \vdash a : \tau} \textbf{ Port}$$

Also, we need to extend arithmetic over type environments such that

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

[6] We do not need to split below the body of $\lambda x.e$ since there cannot be any packets captured in a λ abstraction.

and

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

for any expression e that is either a port, a variable, a reference location, or a region.

We are now ready to define well-typed states.

Definition B.1. The state (R, K, S, D) is well-typed under the environment $\Gamma; W$ (written $\Gamma; W \vdash (R, S, K, D)$) if

—$dom(R) = dom(\Gamma) \cap \boldsymbol{Regions}$,

—$dom(S) = dom(K) = dom(\Gamma) \cap \boldsymbol{Locations}$,

—$dom(D) = dom(\Gamma) \cap \boldsymbol{Ports}$,

—For any region $r \in dom(R)$, $R(r) \geq W(\rho)$ where $\Gamma(r) = reg(\rho)$,

—For any reference location $\ell \in dom(K)$, $\Gamma(K(\ell)) = reg(\rho)$ where $\Gamma(\ell) = ref(\tau, \tau', \rho)$ for some τ and τ', and

—Suppose $D = \{a_1 \mapsto e_1, \ldots, a_n \mapsto e_n\}$ and $S = \{\ell_1 \mapsto a'_1, \ldots, \ell_m \mapsto a'_m\}$, then there exists environments $\Gamma_1; W_1, \ldots, \Gamma_n; W_n, \Gamma'_1; W'_1, \ldots, \Gamma'_m; W'_m$ such that
  —$\Gamma = \sum_{i=1}^{n} \Gamma_i + \sum_{i=1}^{m} \Gamma'_i$,
  —$W = \sum_{i=1}^{n} W_i + \sum_{i=1}^{m} W'_i$,
  —for each port a_i, $\Gamma_i; W_i \vdash D(a_i) : \Gamma(a_i)$, and
  —for each reference location ℓ_i, $\Gamma'_i; W'_i \vdash S(\ell_i) : \tau$ where $\Gamma(\ell_i) = ref(\tau, \tau', \rho)$ for some types τ and τ' such that $\tau \leq \tau'$ and a static region identifier ρ such that $\Gamma(K(\ell)) = reg(\rho)$.

It becomes cumbersome later when we repeatedly pick just one $\Gamma_i; W_i$ or $\Gamma'_i; W'_i$, and so for convenience, given $\Gamma; W \vdash (R, S, K, D \uplus \{a \mapsto e\})$, we often say "$\Gamma'; W'$ is the environment for $\{a \mapsto e\}$" to mean that the environment $\Gamma'; W'$ is one of the environments $\Gamma_i; W_i$ from the definition above such that $\Gamma_i; W_i \vdash e : \Gamma(a)$. Similarly, given $\Gamma; W \vdash (R, S \uplus \{\ell \mapsto a\}, K, D)$, we often say "$\Gamma'; W'$ is the environment for $\{\ell \mapsto a\}$" to mean that the environment $\Gamma'; W'$ is one of the environments $\Gamma'_i; W'_i$ from the definition above such that $\Gamma'_i; W'_i \vdash a : \Gamma(\ell)$.

Before we prove the main subject reduction lemma (Lemma B.6), we need a few side lemmas. The following lemma is needed to "convert" an application of a **Source** type rule to a step of a **Source** $\Rightarrow_{flow}$ rule.

Lemma B.2. *Suppose* $\Gamma; W + W' \vdash (R, K, S, D \uplus \{a \mapsto E[\langle A, P\rangle]\})$ *such that one application of the type rule* **Source** *with* W' *is applied at the context* $E[\,]$ *following the type rule* **Packet** *in the type judgment for the expression* $E[\langle A, P\rangle]$*, i.e.,*

$$\dfrac{\dfrac{\cdots}{\Gamma'; W'' \vdash \langle A, P\rangle : W_P}\ \textbf{Packet}}{\Gamma'; W'' + W' \vdash \langle A, P\rangle : W_P + W'}$$

appears in the judgment for the expression $E[\langle A, P\rangle]$ *in* $\Gamma; W + W' \vdash (R, K, S, D \uplus \{a \mapsto E[\langle A, P\rangle]\})$*. Then there exists* P' *such that* $\Gamma; W \vdash (R', S, K, D \uplus \{a \mapsto E[\langle A, P + P'\rangle]\})$ *where* $R = R' + P'$ *without using the type rule* **Source** *at the context* $E[\,]$.

PROOF. Choose a packet P' such that for each region r, $P'(r) = W'(\rho)$ where $\Gamma(r) = \mathit{reg}(\rho)$ and $P'(r) = 0$ if $r \notin \mathit{dom}(\Gamma)$. Suppose the environment $\Gamma_i; W_i$ is the environment for $\{a \mapsto E[\langle A, P\rangle]\}$ in the environment $\Gamma; W + W'$, i.e., $\Gamma_i; W_i \vdash E[\langle A, P\rangle] : \tau$ for some type τ. Then it is easy to see that $\Gamma_i; W_i - W' \vdash E[\langle A, P + P'\rangle] : \tau$ by not using the type rule **Source** at the context $E[\ ]$. Hence $\Gamma; W \vdash (R', S, K, D \uplus \{a \mapsto E[\langle A, P + P'\rangle]\})$. □

The following side lemma shows that any expression $e \in V$ can be "split" according to a splitting of its types. This lemma is used only to prove Lemma B.4.

LEMMA B.3. *Suppose $e \in V$, $\Gamma; W \vdash e : \tau$, and $\tau = \tau_1 + \tau_2$. Then there exists expressions e_1 and e_2, and environments $\Gamma_1; W_1$ and $\Gamma_2; W_2$ such that $e = e_1 + e_2$, $\Gamma = \Gamma_1 + \Gamma_2$, $W = W_1 + W_2$, $\Gamma_1; W_1 \vdash e_1 : \tau_1$, and $\Gamma_2; W_2 \vdash e_2 : \tau_2$.*

PROOF. By structural induction on the expression e. The case $e = i$ is trivial. For the case the expression e is x, a, ℓ, or r, we let $e_1 = e_2 = e$ and split the type $\Gamma(e)$ accordingly (the rest of the environment Γ may be split in any way), and if $\Gamma(e) = W'$ for some witness type W', then we also split the witness W in case the type rule **Source** was used in the judgment $\Gamma; W \vdash e : \tau$.

For the case $e = \langle A, P\rangle$, τ_1 and τ_2 are witness types. Therefore there exist packets P_1 and P_2 such that $P = P_1 + P_2$, $\Gamma_1; W_1 \vdash \langle A, P_1\rangle : \tau_1$, and $\Gamma_2; W_2 \vdash \langle A, P_2\rangle : \tau_2$ where $W_1 + W_2 = W$ and $\Gamma_1 + \Gamma_2 = \Gamma$. Here, W_1 and W_2 are split according to the uses of **Source** in typing τ_1 and τ_2. Γ_1 and Γ_2 can be split in any way.

The case $e = v \otimes v'$ follows from the induction and the type rule **Pair**. I.e., since $\tau = \tau' \otimes \tau''$ for some τ' and τ'', we split v and v' according to how τ' and τ'' are split, and add the environments of each half to obtain $\Gamma_1; W_1$ and $\Gamma_2; W_2$. Details are omitted but are straightforward.

For the case $e = \lambda x.e'$, as discussed before, we only consider the case when e' contains no packets because no evaluation produces a λ abstraction containing a packet. So we may let $e_1 = e_2 = \lambda x.e$. Since $\tau_1 = \tau' \xrightarrow{q_1} \tau''$ and $\tau_2 = \tau' \xrightarrow{q_2} \tau''$ for some τ', τ'', q_1, and q_2, we may split Γ into $\Gamma_1 = \Gamma \times q_1$ and $\Gamma_2 = \Gamma \times q_2$, and split W into $W_1 = W \times q_1$ and $W_2 = W \times q_2$. Checking the arithmetic is straightforward. □

The following side lemma is required when we take **Arrive** steps in the subject reduction argument.

LEMMA B.4. *Suppose $\Gamma; W \vdash (R, S, K, E[a] \uplus \{a \mapsto e\})$ and $e \in V$, then there exist expressions e_1, e_2 and a type environment Γ' such that $e = e_1 + e_2$ and $\Gamma'; W \vdash (R, S, K, E[e_1] \uplus \{a \mapsto e_2\})$.*

PROOF. Let $D = E[a] \uplus \{a \mapsto e\}$. Let environments $\Gamma_1; W_1, \ldots, \Gamma_n; W_n, \Gamma'_1; W'_1, \ldots, \Gamma'_m; W'_m$ be such that $\Gamma = \sum_{i=1}^{n} \Gamma_i + \sum_{i=1}^{m} \Gamma'_i$, $W = \sum_{i=1}^{n} W_i + \sum_{i=1}^{m} W'_i$, for each a_i, $\Gamma_i; W_i \vdash D(a_i) : \Gamma(a_i)$, and for each ℓ_i, $\Gamma'_i; W'_i \vdash S(\ell_i) : \Gamma(\ell_i)$. Suppose $a = a_i$ and $E[\ ] = \{a_j \mapsto E'[\ ]\} \cup D'$. Note that $i \neq j$.

Then by Lemma B.3 and by induction on the structure of E and the type judgment rules, it follows that there exist expressions e_1 and e_2 such that $D(a_i) = e_1 + e_2$ where there exist $\Gamma_{i_1}, \Gamma_{i_2}, \tau_1, \tau_2, W_{i_1}$, and W_{i_2} such that

—$\Gamma_i = \Gamma_{i_1} + \Gamma_{i_2}$,

—$W_i = W_{i_1} + W_{i_2}$,
—$\Gamma_{i_1}; W_{i_1} \vdash e_1 : \tau - \tau_2$, and
—$\Gamma_j - \{a{:}\tau_2\} + \Gamma_{i_2}; W_j + W_{i_2} \vdash E'[e_2] : \Gamma(a_j)$

Let $\Gamma' = \Gamma - \Gamma_i - \Gamma_j + \Gamma_{i_1} + (\Gamma_j - \{a{:}\tau_2\} + \Gamma_{i_2}) = \Gamma - \{a{:}\tau_2\}$. Then it follows that

$$\Gamma'; W \vdash (R, S, K, E[e_1] \uplus \{a \mapsto e_2\})$$

□

The following lemma allows us to elide the type checking rule **LetA** when proving the main lemma.

LEMMA B.5. *Let e and e' be programs such that e' is identical to e except that an occurrence of the expression* `let` x `=` e_1 `in` e_2 *is replaced by the expression $e_2[e_1/x]$ where $e_1 \in V$. Then, there is an evaluation of e' that generates a trace graph G iff there is an evaluation of e that generates G.*

PROOF. By inspection of the reduction rules. Recall that in application of **LetA**, x is not free in e_1. □

Based on Lemma B.5, we assume in the next lemma that no type judgment uses **LetA**. Let **Erase** be an operation that replaces each occurrence of $\langle A, P\rangle$ by A. We are now ready to state and prove the main lemma.

LEMMA B.6. *Suppose $\Gamma; W \vdash (R, S, K, D)$ and $(dom(R), S, \mathbf{Erase}(D)) \Rightarrow (R_o, S_o, D_o)$. Then there exist R_f, K_f, S_f, D_f, Γ', and W' such that*

—$(R, S, K, D) \Rightarrow_{flow} (R_f, S_f, K_f, D_f)$,
—$\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$,
—$dom(R_f) = R_o$,
—$S_o = S_f$, *and*
—$\mathbf{Erase}(D_o) = D_f$.

Furthermore, if the above $\Rightarrow_{flow}$ step is a **Write** *step writing to location ℓ then $P(K_f(\ell)) \geq 1$ where P is the packet at the write, and if it is a* **Read** *step reading from ℓ then $P(K_f(\ell)) > 0$ where P is the packet at the read.*

Note that this lemma implies that there is a proper flow assignment for a trace graph constructed from reducing a well-typed state.

PROOF. We prove the lemma by case analysis on $\Rightarrow$.

App

We have

$$(dom(R), S, \mathbf{Erase}(\{a' \mapsto E[(\lambda x.e)\ e']\} \uplus D')) \Rightarrow (R_o, S_o, D_o)$$

where $\{a' \mapsto E[(\lambda x.e)\ e']\} \uplus D') = D$, $R_o = dom(R)$, $S_o = S$, and $D_o = \mathbf{Erase}(\{a' \mapsto E[e[a/x]]\} \uplus D' \uplus \{a \mapsto e'\})$. Pick a **App** step for $\Rightarrow_{flow}$ such that

$$(R, K, S, D = \{a' \mapsto E[(\lambda x.e)\ e'] \uplus D'\}) \Rightarrow_{flow} (R_f, K_f, S_f, D_f)$$

where $R_f = R$, $K_f = K$, $S_f = S$, $D_f = (\{a' \mapsto E[e[a/x]]\} \uplus D' \uplus \{a \mapsto e'\})$. It is easy to see that $dom(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Let $\Gamma_1 + \Gamma_2 + \Gamma_3; W_1 + W_2 + W_3$ be the environment for $\{a' \mapsto E[(\lambda x.e)\ e']\}$ in $\Gamma; W \vdash (R, S, K, D)$ such that

$$\frac{\Gamma_2; W_2 \vdash \lambda x.e : \tau \xrightarrow{q} \tau' \quad \Gamma_3; W_3 \vdash e' : \tau \quad q \geq 1}{\Gamma_2 + \Gamma_3; W_2 + W_3 \vdash (\lambda x.e)\ e' : \tau'}$$

appears in the subderivation at the context $E[\,]$. Then because $q \geq 1$, by inspection of the type checking rules, it follows that $\Gamma_2, a{:}\tau; W_2 \vdash e[a/x] : \tau'$.

We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S the same as the environment $\Gamma; W$ and by using the environment $\Gamma_1 + \Gamma_2, a{:}\tau; W_1 + W_2$ for $\{a' \mapsto E[e[a/x]]\}$ and by using the environment $\Gamma_3; W_3$ for $\{a \mapsto e'\}$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$.

Let

We have

$$(dom(R), S, \mathbf{Erase}(D = (\{a' \mapsto E[\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e']\} \uplus D'))) \Rightarrow (R_o, S_o, D_o)$$

where $R_o = dom(R)$, $S_o = S$, and $D_o = \mathbf{Erase}(\{a' \mapsto E[e'[a/x]]\} \uplus D' \uplus \{a \mapsto e[a/x]\})$. The case $x \notin free(e)$ identical to the **App** case. So suppose $x \in free(e)$. Pick a **Let** step for $\Rightarrow_{flow}$ such that

$$(R, K, S, D = (\{a' \mapsto E[\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e']\} \uplus D')) \Rightarrow_{flow} (R_f, K_f, S_f, D_f)$$

where $R_f = R$, $K_f = K$, $S_f = S$, $D_f = (\{a' \mapsto E[e'[a/x]]\} \uplus D' \uplus \{a \mapsto e[a/x]\})$. It is easy to see that $dom(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Since $x \in free(e)$, Let $\Gamma_1 + \Gamma_2 + \Gamma_3; W_1 + W_2 + W_3$ be the environment for $\{a' \mapsto E[\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e']\}$ in $\Gamma; W \vdash (R, S, K, D)$ such that

$$\frac{\Gamma_2, x{:}\tau; W_2 \vdash e : \tau \quad \Gamma_3, x{:}\tau; W_3 \vdash e' : \tau' \quad \tau \geq \tau \times \infty}{\Gamma_2 + \Gamma_3; W_2 + W_3 \vdash \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' : \tau'}$$

appears in the subderivation at the context $E[\,]$. Then by inspection of the type checking rules, it follows that $\Gamma_2, a{:}\tau; W_2 \vdash e[a/x] : \tau$ and $\Gamma_3, a{:}\tau; W_3 \vdash e'[a/x] : \tau'$. We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S the same as the environment $\Gamma; W$ and by using the environment $\Gamma_2, a{:}\tau; W_2$ for $\{a \mapsto e[a/x]\}$ and using the environment $\Gamma_1 + \Gamma_3, a{:}\tau; W_1 + W_3$ for $\{a' \mapsto E[e'[a/x]]\}$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$ since $\tau \geq \tau \times \infty$ implies that $\tau + \tau = \tau$.

Pair

We have

$$(dom(R), S, \mathbf{Erase}(D = (\{a \mapsto E[\pi_i(e_1 \otimes e_2)]\} \uplus D'))) \Rightarrow (R_o, S_o, D_o)$$

where $R_o = dom(R)$, $S_o = S$, and $D_o = \mathbf{Erase}(\{a \mapsto E[e_i]\} \uplus D')$. Pick a **Pair** step for $\Rightarrow_{flow}$ such that

$$(R, K, S, D = \{a \mapsto E[\pi_i(e_1 \otimes e_2)] \uplus D'\}) \Rightarrow_{flow} (R_f, K_f, S_f, D_f)$$

where $R_f = R$, $K_f = K$, $S_f = S$, $D_f = (\{a \mapsto E[e_i]\} \uplus D')$. It is easy to see that $dom(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Let $\Gamma_1 + \Gamma_2; W_1 + W_2$ be the environment for $\{a \mapsto E[\pi_i(e_1 \otimes e_2)]\}$ in $\Gamma; W \vdash (R, S, K, D)$ such that

$$\frac{\Gamma_2, W_2 \vdash e_1 \otimes e_2 : \tau_1 \otimes \tau_2}{\Gamma_2; W_2 \vdash \pi_i(e_1 \otimes e_2) : \tau_i}$$

appears in the subderivation at the context $E[\ ]$. Then by inspection of the type checking rules, it follows that $\Gamma_2; W_2 \vdash e_i : \tau_i$. We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S the same as $\Gamma; W$ and by using the environment $\Gamma_1 + \Gamma_2; W_1 + W_2$ for $\{a \mapsto E[e_i]\}$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$.

Write

We have

$$(dom(R), S = (S' \uplus \{\ell \mapsto a''\}), \mathbf{Erase}(D = (\{a' \mapsto E[\mathtt{write}\ \ell\ e\ \langle A, P\rangle]\} \uplus D'))) \\ \Rightarrow (R_o, S_o, D_o)$$

where $R_o = dom(R)$, $S_o = (S' \uplus \{\ell \mapsto a\})$, and $D_o = \mathbf{Erase}(\{a' \mapsto E[\langle B, P\rangle]\} \uplus D' \uplus \{a \mapsto e\})$.

Suppose in $\Gamma; W \vdash (R, S, K, D = (\{a' \mapsto E[\mathtt{write}\ \ell\ e\ \langle A, P\rangle]\} \uplus D'))$ the type rule **Source** is applied at $E[\mathtt{write}\ \ell\ e\ [\ ]]$ with some witness type W_6. Then by Lemma B.2, there exists P' such that $\Gamma; W - W_6 \vdash (R - P', S, K, \{a' \mapsto E[\mathtt{write}\ \ell\ e\ \langle A, P + P'\rangle]\} \uplus D')$. Pick a **Source** step followed by a **Write** step for $\Rightarrow_{flow}$ such that

$$(R, K, S = (S' \uplus \{\ell \mapsto a''\}), D = (\{a' \mapsto E[\mathtt{write}\ \ell\ e\ \langle A, P\rangle]\} \uplus D')) \\ \Rightarrow_{flow} (R - P', S' \uplus \{\ell \mapsto a''\}, K, \{a' \mapsto E[\mathtt{write}\ \ell\ e\ \langle A, P + P'\rangle]\} \uplus D') \\ \Rightarrow_{flow} (R_f, K_f, S_f, D_f)$$

where $R_f = R - P'$, $K_f = K$, $S_f = (S' \uplus \{\ell \mapsto a\})$, $D_f = (\{a' \mapsto E[\langle B, P + P'\rangle]\} \uplus D' \uplus \{a \mapsto e\})$. It is easy to see that $dom(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Let $\Gamma_1 + \Gamma_2 + \Gamma_3 + \Gamma_4; W_1 + W_2 + W_3 + W_4$ be the environment for $\{a' \mapsto E[\mathtt{write}\ \ell\ e\ \langle A, P + P'\rangle]\}$ in $\Gamma; W - W_6 \vdash (R - P', S, K, \{a' \mapsto E[\mathtt{write}\ \ell\ e\ \langle A, P + P'\rangle]\} \uplus D')$ such that

$$\frac{\Gamma_2; W_2 \vdash \ell : ref(\tau, \tau', \rho) \quad \Gamma_3; W_3 \vdash e : \tau' \Gamma_4; W_4 \vdash \langle A, P + P'\rangle : W_5 \quad W_5(\rho) \geq 1}{\Gamma_2 + \Gamma_3 + \Gamma_4; W_2 + W_3 + W_4 \vdash \mathtt{write}\ \ell\ e\ \langle A, P + P'\rangle : W_5}$$

appears in the subderivation at the context $E[\ ]$. We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S' the same as the environment $\Gamma; W - W_6$ and by using the environment $\Gamma_3; W_3$ for $\{a \mapsto e\}$, the environment $\Gamma_1 + \Gamma_4; W_1 + W_4$ for $\{a' \mapsto E[\langle B, P + P'\rangle]\}$, and the environment $a{:}\tau; \emptyset$ for $\{\ell \mapsto a\}$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$ since $\tau' \geq \tau$. Also, since we eliminated the type rule **Source** for the expression $\langle A, P + P'\rangle$ at the context $E[\mathtt{write}\ \ell\ e\ [\ ]]$, it must be the case that $(P + P')(r) \geq 1$ where $\Gamma(r) = reg(\rho)$. Since $\Gamma_2; W_2 \vdash \ell : ref(\tau, \tau', \rho)$ implies that $K_f(\ell) = r$, we have $(P + P')(K_f(\ell)) \geq 1$ as required.

Read

We have

$$(dom(R), S = (S' \uplus \{\ell \mapsto a\}), \mathbf{Erase}(D = (\{a' \mapsto E[\mathtt{read}\,\ell\,\langle A, P\rangle]\} \uplus D'))) \\ \Rightarrow (R_o, S_o, D_o)$$

where $R_o = dom(R)$, $S_o = S$, and $D_o = \mathbf{Erase}(\{a' \mapsto E[a \otimes \langle B, P\rangle]\} \uplus D')$.

Suppose in $\Gamma; W \vdash (R, S, K, D = (\{a' \mapsto E[\mathtt{read}\,\ell\,\langle A, P\rangle]\} \uplus D'))$, the type rule **Source** is applied at the context $E[\mathtt{read}\,\ell\,[\,]]$ with some witness type W_5. Then by Lemma B.2, there exists a packet P' such that $\Gamma; W - W_5 \vdash (R - P', S, K, \{a' \mapsto E[\mathtt{read}\,\ell\,\langle A, P + P'\rangle]\} \uplus D')$. Pick a **Source** step followed by a **Read** step for $\Rightarrow_{flow}$ such that

$$(R, K, S = (S' \uplus \{\ell \mapsto a\}), D = (\{a' \mapsto E[\mathtt{read}\,\ell\,\langle A, P\rangle]\} \uplus D')) \\ \Rightarrow_{flow} (R - P', S, K, \{a' \mapsto E[\mathtt{read}\,\ell\,\langle A, P + P'\rangle]\} \uplus D') \\ \Rightarrow_{flow} (R_f, K_f, S_f, D_f)$$

where $R_f = R - P'$, $K_f = K$, $S_f = S$, $D_f = (\{a' \mapsto E[a \otimes \langle B, P + P'\rangle]\} \uplus D')$. It is easy to see that $dom(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Let $\Gamma_1 + \Gamma_2 + \Gamma_3; W_1 + W_2 + W_3$ be the environment for $\{a' \mapsto E[\mathtt{read}\ell\langle A, P + P'\rangle]\}$ in $\Gamma; W - W_5 \vdash (R - P', S, K, \{a' \mapsto E[\mathtt{read}\,\ell\,\langle A, P + P'\rangle]\} \uplus D')$ such that

$$\frac{\Gamma_2; W_2 \vdash \ell : ref(\tau, \tau', \rho) \quad \Gamma_3; W_3 \vdash \langle A, P + P'\rangle : W_4 \quad W_4(\rho) > 0}{\Gamma_2 + \Gamma_3; W_2 + W_3 \vdash \mathtt{read}\,\ell\,\langle A, P + P'\rangle : W_4 \otimes \tau}\ \mathbf{Read}$$

appears in the subderivation at the context $E[\ ]$. We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S' the same as the environment $\Gamma; W - W_5$. Let $\Gamma_6; W_6$ be the environment for $\{\ell \mapsto a\}$ in $\Gamma; W - W_5 \vdash (R - P', S, K, \{a' \mapsto E[\mathtt{read}\,\ell\,\langle A, P + P'\rangle]\} \uplus D')$. Then in the environment $\Gamma'; W'$, we use the environment $\Gamma_6 - \{a{:}\tau\}; W_6$ for $\{\ell \mapsto a\}$ and the environment $\Gamma_1 + \Gamma_3 + \{a : \tau\}; W_1 + W_3$ for $\{a' \mapsto E[a \otimes \langle B, P + P'\rangle]\}$. This implies that $\Gamma'(\ell) = (\Gamma - \Gamma_2)(\ell) \geq ref\,(\tau_o - \tau, \tau'_o, \rho)$ where $\Gamma(\ell) = ref\,(\tau_o, \tau'_o, \rho)$. Therefore,$\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$. Also, since we eliminated the type rule **Source** at the expression $\langle A, P + P'\rangle$ at the context $E[\mathtt{read}\,\ell\,[\,]]$, it must be the case that $(P + P')(r) > 0$ where $\Gamma(r) = reg(\rho)$. Since $\Gamma_2; W_2 \vdash \ell : ref(\tau, \tau', \rho)$ implies that $K_f(\ell) = r$, we have $(P + P')(K_f(\ell)) \geq 1$ as required.

Ref

We have

$$(dom(R), S, \mathbf{Erase}(D = (\{a' \mapsto E[\mathtt{ref}\ e\ r]\} \uplus D'))) \Rightarrow (R_o, S_o, D_o)$$

where $R_o = dom(R)$, $S_o = (S \uplus \{\ell \mapsto a\})$, and $D_o = \mathbf{Erase}(\{a' \mapsto E[\ell]\} \uplus D' \uplus \{a \mapsto e\})$. Pick a **Ref** step for $\Rightarrow_{flow}$ such that

$$(R, K, S, D = (\{a' \mapsto E[\mathtt{ref}\ e\ r]\} \uplus D')) \Rightarrow_{flow} (R_f, K_f, S_f, D_f)$$

where $R_f = R$, $K_f = (K \uplus \{\ell \mapsto r\})$, $S_f = (S \uplus \{\ell \mapsto a\})$, $D_f = (\{a' \mapsto E[\ell]\} \uplus D' \uplus \{a \mapsto e\})$. It is easy to see that $dom(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Let $\Gamma_1 + \Gamma_2 + \Gamma_3; W_1 + W_2 + W_3$ be the environment for $\{a' \mapsto E[\mathtt{ref}\ e\ r]\}$ in

$\Gamma; W \vdash (R, S, K, \{a' \mapsto E[\texttt{ref}\ e\ r]\} \uplus D')$ such that

$$\frac{\Gamma_2; W_2 \vdash e : \tau \quad \Gamma_3; W_3 \vdash r : \mathit{reg}(\rho)}{\Gamma_2 + \Gamma_3; W_2 + W_3 \vdash \texttt{ref}\ e\ r : \mathit{ref}(\tau, \tau, \rho)}$$

appears in the subderivation at the context $E[\ ]$. We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S the same as the environment $\Gamma; W$ and by using the environment $\Gamma_2; W_2$ for $\{a \mapsto e\}$, the environment $\Gamma_1, \ell : \mathit{ref}(\tau, \tau, \rho); W_1$ for $\{a' \mapsto E[\ell]\}$, and the environment $a{:}\tau; \emptyset$ for $\{\ell \mapsto a\}$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$.

Join

We have

$$(\mathit{dom}(R), S, \mathbf{Erase}(D = (\{a \mapsto E[\texttt{join}\ \langle A, P\rangle\ \langle B, P'\rangle]\} \uplus D'))) \Rightarrow (R_o, S_o, D_o)$$

where $R_o = \mathit{dom}(R)$, $S_o = S$, and $D_o = \mathbf{Erase}(\{a \mapsto E[\langle C, P + P'\rangle]\} \uplus D')$. Pick a **Join** step for $\Rightarrow_{\mathit{flow}}$ such that

$$(R, K, S, D = (\{a \mapsto E[\texttt{join}\ \langle A, P\rangle\ \langle B, P'\rangle]\} \uplus D')) \Rightarrow_{\mathit{flow}} (R_f, K_f, S_f, D_f)$$

where $R_f = R$, $K_f = K$, $S_f = S$, $D_f = (\{a \mapsto E[\langle C, P + P'\rangle]\} \uplus D')$. It is easy to see that $\mathit{dom}(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Let $\Gamma_1 + \Gamma_2 + \Gamma_3; W_1 + W_2 + W_3$ be the environment for $\{a \mapsto E[\texttt{join}\langle A, P\rangle\langle B, P'\rangle]\}$ in $\Gamma; W \vdash (R, S, K, \{a \mapsto E[\texttt{join}\ \langle A, P\rangle\ \langle B, P'\rangle]\} \uplus D')$ such that

$$\frac{\Gamma_2; W_2 \vdash \langle A, P\rangle : W_4 \quad \Gamma_2; W_3 \vdash \langle B, P'\rangle : W_5}{\Gamma_2 + \Gamma_3; W_2 + W_3 \vdash \texttt{join}\ \langle A, P\rangle\ \langle B, P'\rangle : W_4 + W_5}$$

appears in the subderivation at the context $E[\ ]$. We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S the same as the environment $\Gamma; W$ and by using the environment $\Gamma_1 + \Gamma_2 + \Gamma_3; W_1 + W_2 + W_3$ for $\{a \mapsto E[\langle C, P + P'\rangle]\}$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$.

LetReg

We have

$$(\mathit{dom}(R), S, \mathbf{Erase}(D = (\{a' \mapsto E[\texttt{letreg}\ x\ e]\} \uplus D'))) \Rightarrow (R_o, S_o, D_o)$$

where $R_o = \mathit{dom}(R \uplus \{r \mapsto 1\})$, $S_o = S$, and $D_o = \mathbf{Erase}(\{a' \mapsto E[e[a/x]]\} \uplus D' \uplus \{a \mapsto r\})$. Pick a **LetReg** step for $\Rightarrow_{\mathit{flow}}$ such that

$$(R, K, S, D = (\{a' \mapsto E[\texttt{letreg}\ x\ e]\} \uplus D')) \Rightarrow_{\mathit{flow}} (R_f, K_f, S_f, D_f)$$

where $R_f = R \uplus \{r \mapsto 1\}$, $K_f = K$, $S_f = S$, $D_f = (\{a' \mapsto E[e[a/x]]\} \uplus D' \uplus \{a \mapsto r\})$. It is easy to see that $\mathit{dom}(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

Let $\Gamma_1 + \Gamma_2; W_1 + W_2$ be the environment for $\{a \mapsto E[\texttt{letreg}\ x\ e]\}$ in $\Gamma; W \vdash (R, S, K, \{a \mapsto E[\texttt{letreg}\ x\ e]\} \uplus D')$ such that

$$\frac{\Gamma_2, x{:}\mathit{reg}(\rho); W_2 + \{\rho \mapsto q\} \vdash e : \tau \quad q \leq 1 \quad \rho \notin \mathit{free}(\Gamma_2, W_2, \tau)}{\Gamma_2; W_2 \vdash \texttt{letreg}\ x\ e : \tau}$$

appears in the subderivation at the context $E[\ ]$. We construct an environment $\Gamma'; W'$ by keeping the portions for the expression store D' and the reference location store S the same as the environment $\Gamma; W$ and by using the environment $\Gamma_1 + \Gamma_2 + \{a : \mathit{reg}(\rho)\}; W_1 + W_2 + \{\rho \mapsto 1\}$ for $\{a' \mapsto E[e[a/x]]\}$ and the environment $r : \mathit{reg}(\rho); \emptyset$ for $\{a \mapsto r\}$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$ since $q \leq 1$.

Arrive
We have

$$(\mathit{dom}(R), S, \mathbf{Erase}(D = (\{a' \mapsto E[a]\} \uplus \{a \mapsto e\} \uplus D'))) \Rightarrow (R_o, S_o, D_o)$$

where $e \in V$, $R_o = \mathit{dom}(R \uplus \{r \mapsto 1\})$, $S_o = S$, and $D_o = \mathbf{Erase}(\{a' \mapsto E[e]\} \uplus \{a \mapsto e\} \uplus D')$.

By Lemma B.4, there exist expressions e_1, e_2, and a type environment Γ' such that $\Gamma'; W \vdash (R, S, K, \{a' \mapsto E[e_1]\} \uplus \{a \mapsto e_2\} \uplus D')$. Pick an **Arrive** step for $\Rightarrow_{\mathit{flow}}$ with the above e_1 and e_2, i.e.,

$$(R, K, S, D = (\{a' \mapsto E[a]\} \uplus \{a \mapsto e\} \uplus D')) \Rightarrow_{\mathit{flow}} (R_f, K_f, S_f, D_f)$$

where $R_f = R$, $K_f = K$, $S_f = S$, $D_f = (\{a' \mapsto E[e_1]\} \uplus \{a \mapsto e_2\} \uplus D')$. It is easy to see that $\mathit{dom}(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$. Let $W' = W$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$.

GC
We have

$$(\mathit{dom}(R), S, \mathbf{Erase}(D = (D' \uplus D''))) \Rightarrow (R_o, S_o, D_o)$$

where $\diamond \notin \mathit{dom}(D'')$, $\mathit{dom}(D'') \cap \mathit{free}(D') = \emptyset$, $R_o = \mathit{dom}(R)$, $S_o = S$, and $D_o = \mathbf{Erase}(D')$. Pick a **GC** step for $\Rightarrow_{\mathit{flow}}$ such that

$$(R, K, S, D = (D' \uplus D'')) \Rightarrow_{\mathit{flow}} (R_f, K_f, S_f, D_f)$$

where $R_f = R$, $K_f = K$, $S_f = S$, $D_f = D'$. It is easy to see that $\mathit{dom}(R_f) = R_o$, $S_o = S_f$, and $\mathbf{Erase}(D_o) = D_f$.

We construct an environment $\Gamma'; W'$ by subtracting the portions for the expression store D'' from the environment $\Gamma; W$. Then it follows that $\Gamma'; W' \vdash (R_f, S_f, K_f, D_f)$. □

Finally, to start off subject reduction, we need the initial state to be well-typed.

LEMMA B.7. *If e is a well-typed program, then $\emptyset; \emptyset \vdash (\emptyset, \emptyset, \emptyset, \{\diamond \mapsto e\})$. That is, the initial state is well-typed.*

Combining Lemma B.6 and Lemma B.7, it follows that any trace graph of a well-typed program has a proper flow assignment.