Game Semantics for Type Soundness

Tim Disney
University of California Santa Cruz

Cormac Flanagan
University of California Santa Cruz

***Abstract*—The key idea of game semantics is that a term can interact with its enclosing context via various *events*, such as function calls and returns. A *trace* is a sequence of such interaction events. The meaning of the term is then naturally represented by the set of all event traces that the term can generate. Game semantics allows us to define the meaning of both expressions and types in the same domain which enables an interesting alternative to subject reduction for proving type soundness.**

This paper uses game semantics to define the meaning of and verify type soundness for a sequence of programming languages, starting with a functional sequential language (the call-by-value simply-typed lambda calculus), and then extending that proof with subtyping, side effects, control effects, and concurrency. These proofs are reasonably short and fairly semantic in structure, focusing on the relationship between the meanings of each term and its corresponding type. In particular, we show that the typing and subtyping relations are both conservative approximations of alternating trace containment.

I. Introduction

Over the past 20 years, the syntactic approach [1] has become established as the dominant method for proving type soundness. The flexibility of this approach stems from its exclusive reliance on syntactic methods, in which programs, types, and intermediate computation states are all represented syntactically, and the typing and subtyping relations are defined over syntactic items. The result is an elegant proof technique, but one in which types remain just pieces of syntax, with no associated semantic meaning or *denotation*.

The last two decades have also given birth to a new style of denotational semantics called game semantics, in which the interaction between two modules in a system can be considered a game with alternating moves by the two modules.

This paper attempts to connect these two fields of type systems and games semantics. The motivation for this work is to open up an important application domain for game semantics, while at the same time providing a formal foundation for intuitions about the typing and subtyping relations that are left informal under the syntactic approach. In comparison to the polished syntactic proof machinery developed over the past two decades, we do not propose that the game semantic approach is better (in the sense of being simpler or more efficient). Nevertheless, the game semantic approach does appear to provide different benefits (outlined below), which suggest this approach merits further study.

Our approach uses traces to formalize our game semantics, and the starting point for our semantic development is the call-by-value untyped λ-calculus. In particular, given a program $C[e]$ in this language, we can imagine a remote procedure call mechanism that mediates the interactions between the expression e and its enclosing context $C[\cdot]$ by appropriately routing function calls and returns from e to its context and vice-versa. We use the term *event* to denote a function call or return message sent from e to its context, or vice-versa. Then the semantics of e can be formalized as a (potentially infinite) set of *traces*, denoted $[\![e]\!]_k$, where each trace is a finite sequence of such events. The subscript k in $[\![e]\!]_k$ denotes a continuation channel on which to send the result of e's computation to the context C. Thus, this transformation from program syntax to semantic traces in some ways follows the CPS transform. For simplicity, we assume here that e is closed, although our semantic framework supports open terms.

To incorporate types from the simply typed λ-calculus, we formalize the meaning of each type A as an analogous set of traces $[\![A]\!]_k$, again with respect to a continuation channel k.

Like other denotational semantics, game semantics is compositional, and so, for example, the meaning $[\![e_1\ e_2]\!]_k$ of a function application is defined in terms of the meanings of the direct subexpressions e_1 and e_2. Like operational semantics, game semantics is fairly syntactic in flavor, primarily dealing with sets of (syntactic) traces. Thus, in some sense, game semantics is a compositional syntactic semantics, where the syntax captures behavior (i.e. traces of interactions) rather than state (as in operational semantics).

By defining the meaning of both expressions and types in the same domain of trace sets, this approach enables us to capture the typing judgment $\vdash e : A$ as an appropriate relation on the corresponding trace sets $[\![e]\!]_k$ and $[\![A]\!]_k$. In particular, a trace in $[\![e]\!]_k$ may contain both *send* events (which transfer control from e to its context) and *receive* events (which transfer control from the context to e). If e has type A, then $[\![A]\!]_k$ must permit any send event in $[\![e]\!]_k$ (since any function return value sent by e must be permitted by its type A); conversely by a contravariant argument $[\![e]\!]_k$ must contain any receive event in $[\![A]\!]_k$. Thus, if

$$\vdash e : A$$

then the appropriate relation between the corresponding trace sets is *alternating trace containment* [2], denoted

$$[\![e]\!]_k \ \sqsubset\ [\![A]\!]_k$$

where $[\![A]\!]_k$ contains (non-strictly) more sends and (non-strictly) fewer receives than $[\![e]\!]_k$. Thus, typing conservatively approximates alternating containment on traces.

Theorem 1. *If* $\vdash e : A$ *then* $[\![e]\!]_k \sqsubset [\![A]\!]_k$

We next consider the subtyping relation $A <: B$. Again, we can show that the type B must contain more sends and fewer receives than A, so subtyping also conservatively approximates alternating trace containment.

Theorem 2. *If* $A <: B$ *then* $[\![A]\!]_k \sqsubset [\![B]\!]_k$

One interesting aspect of trace-based type soundness proofs is that they are mostly compositional, in that each type rule can be verified as admissible independent of the other rules in the system. For example, we verify the admissibility of the function application rule:

$$\frac{\vdash e_1 : A \to B \quad \vdash e_2 : A}{\vdash e_1\ e_2 : B}$$

(ignoring the type environment here for simplicity) by proving a corresponding lemma:

$$\begin{array}{lrcl} \textit{if} & [\![e_1]\!]_{k_1} & \sqsubset & [\![A \to B]\!]_{k_1} \\ \textit{and} & [\![e_2]\!]_{k_2} & \sqsubset & [\![A]\!]_{k_2} \\ \textit{then} & [\![e_1\ e_2]\!]_k & \sqsubset & [\![B]\!]_k \end{array}$$

Since $[\![e_1\ e_2]\!]_k$ is defined compositionally in terms of $[\![e_1]\!]_{k_1}$ and $[\![e_2]\!]_{k_2}$ (with respect to the appropriate channels k_1 and k_2), this lemma is independent of the subterms e_1 and e_2 and depends only on the semantics of function application and of function types. Consequently, any extension to the type or term language is safe with respect to the above rule provided it does not modify the meaning of function application or function types.

Our experience to date suggests that this approach provides a helpful semantic foundation for exploring typed programming languages. In particular, some language extensions can be developed and proven type sound independently, where the formalism precludes unintentional cross-cutting interference between language features.

To illustrate this benefit, this paper uses trace semantics to verify type soundness of a sequence of programming languages. Section II first formalizes a calculus for composing and reasoning about trace sets. Section III illustrates our approach by verifying type soundness for the simply typed λ-calculus. We then enrich this language with subtyping (Section IV), first-class continuations (Section V), imperative features (Section VI), and concurrent threads (Section VII), with compositional proofs for all extensions.

In summary, this paper provides the following contributions.

1) It provides a compositional semantic meaning for types and terms as trace sets.
2) The typing relation ($\vdash e : A$) naturally corresponds to alternating trace containment on trace sets ($[\![e]\!]_k \sqsubset [\![A]\!]_k$).
3) The subtyping relation ($A <: B$) also corresponds to alternating trace containment on trace sets ($[\![A]\!]_k \sqsubset [\![B]\!]_k$).
4) Our initial soundness proof for the simply typed λ-calculus scales well to support concurrency, imperative features, and control effects of the term level, as well as subtyping at the type level, since the admissibility of each typing rule depends only on the semantics of types and terms, and is independent of the other rules in the system.

We conjecture that trace semantics might provide helpful insights in the development and verification of other program analyses and type systems. As one example, the unification of typing and subtyping as alternating trace containment relation provides some semantic motivation for recent work on pure subtype systems, which also merge the typing and subtyping relations [3], [4].

Much prior work has explored various aspects of game semantics, including developing fully abstract game semantics for various programming languages [5], [6], [7], [8]. In closely related prior work, Chroboczek used game semantics to elegantly prove type soundness for the call-by-name λ-calculus [9], [10], [11]. In contrast to that work, this paper targets the call-by-value λ-calculus, which then enables us to extend this approach to address imperative features such as call/cc and mutable references.

II. The Trace Calculus

We start by formalizing the semantic domain of trace sets: see Figure 1. A *trace* α is a finite sequence of events. Each *event* is either a *send event* $x!\overline{y}$, which sends the channel list $\overline{y}$ to x, or a *receive event* $x?\overline{y}$, which receives $\overline{y}$ from x. Note that send events $x!\overline{y}$ and receive events $x?\overline{y}$ both bind the argument list $\overline{y}$; these channels are then in scope in the rest of the trace and can be α-renamed in the usual fashion. Thus, for example, we consider the traces $k!y\,.\,y?r$ and $k!x\,.\,x?r$ to be α-equivalent.

The FS and FR functions identify the free sending and receiving channels in a trace, respectively. As an example, if $\alpha = x!y\,.\,y?z\,.\,z!h$ then we have $\mathrm{FS}(\alpha) = \{x\}$ and $\mathrm{FR}(\alpha) = \emptyset$. We consider traces to be equivalent modulo α-renaming, and thus we have for example $x!y\,.\,y?z = x!w\,.\,w?z$.

To provide an initial intuition of how traces capture program semantics, consider the meaning of the higher-order function $[\![\lambda f.\ f\ (\lambda x.\ x)]\!]_k$, where the channel k represents the initial continuation for this code fragment. Since this code fragment can interact with its context in arbitrary ways, it has infinitely many possible traces, but one possible trace is:

$$\alpha = k!r\,.\,r?fh\,.\,f!yh'\,.\,h'?z\,.\,h!z'$$

In this trace:

1) The first event $k!r$ sends a fresh channel r to the context k providing a shared channel for the context to call this function.
2) Next the event $r?fh$ receives from the context two channels; f, which represents the function argument, and h, which represents the continuation for this call.
3) The event $f!yh'$ sends to f a channel y' denoting the identity function $\lambda x.\ x$ and a continuation h'.
4) The call to f immediately returns to its continuation h' via $h'?z$.
5) Finally, the function $\lambda f.\ f\ (\lambda x.\ x)$ returns to its continuation h, passing a fresh channel z'. Later message events

Figure 1: The Trace Calculus

Grammar:

$$
\begin{array}{rcl}
x, y, g, h, k & \in & \mathit{Chan} \\
\pi & \in & \mathit{Event} \quad ::= \quad x!\overline{y} \mid x?\overline{y} \\
\alpha & \in & \mathit{Trace} \quad ::= \quad \pi_1 \,.\, \cdots \,.\, \pi_n \\
P, Q, R & \in & \mathit{TraceSet} = 2^{\mathit{Trace}}
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{FS}(\epsilon) & = & \emptyset \\
\mathrm{FS}(x?\overline{y} \,.\, \alpha) & = & \mathrm{FS}(\alpha) \setminus \{\overline{y}\} \\
\mathrm{FS}(x!\overline{y} \,.\, \alpha) & = & \{x\} \cup \mathrm{FS}(\alpha) \setminus \{\overline{y}\}
\end{array}
\qquad
\begin{array}{rcl}
\mathrm{FR}(\epsilon) & = & \emptyset \\
\mathrm{FR}(x?\overline{y} \,.\, \alpha) & = & \{x\} \cup \mathrm{FR}(\alpha) \setminus \{\overline{y}\} \\
\mathrm{FR}(x!\overline{y} \,.\, \alpha) & = & \mathrm{FR}(\alpha) \setminus \{\overline{y}\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{FC}(\alpha) & = & \mathrm{FS}(\alpha) \cup \mathrm{FR}(\alpha) \\
\mathrm{BC}(x!\overline{y}) & = & \mathrm{BC}(x?\overline{y}) = \{\overline{y}\}
\end{array}
$$

Traceset Operations and Constants:

$$
\begin{array}{rcl}
1 & = & \{\epsilon\} \\
\neg(x!\overline{y}) & = & x?\overline{y} \\
\neg(x?\overline{y}) & = & x!\overline{y} \\
\pi \,.\, P & = & \{\epsilon, \pi \,.\, \alpha \mid \alpha \in P\} \\
P \setminus \pi & = & \{\alpha \mid \pi \,.\, \alpha \in P\} \\
\nu\overline{x}.P & = & \{\alpha \in P \mid \forall x \in \overline{x}.\ x \notin \mathrm{FC}(\alpha)\} \\
P \cup Q & = & \{\alpha \mid \alpha \in P \text{ or } \alpha \in Q\} \\
P \times Q & = & \bigcup_\pi \pi \,.\, ((P \setminus \pi \times Q) \cup (P \times Q \setminus \pi))
\end{array}
$$

$$
\begin{array}{rcl}
P \otimes Q & = & \bigcup_\pi \pi \,.\, ((P \setminus \pi \otimes Q) \cup (P \otimes Q \setminus \pi)) \\
 & \cup & \bigcup_\pi \nu\overline{x}.((P \setminus \pi) \otimes (Q \setminus \neg\pi)) \\
 & & \textit{where } \overline{x} = \mathrm{BC}(\pi) \\
\bigcup_i^n P_i & = & P_1 \cup P_2 \cup \cdots \cup P_n \\
\prod_i^n P_i & = & P_1 \times P_2 \times \cdots \times P_n \\
\coprod_i^n P_i & = & P_1 \otimes P_2 \otimes \cdots \otimes P_n \\
P^n & = & \prod_1^n P \\
*P & = & \bigcup_{i=1}^\infty \prod_{j=1}^i P^j
\end{array}
$$

Properties:

$(\mathit{TraceSet}, \cup, \times, \emptyset, 1)$ is a commutative semiring

1. $P \times Q = Q \times P$
2. $P \times (Q \times R) = (P \times Q) \times R$
3. $P \times 1 = P$
4. $P \times (Q \cup R) = (P \times Q) \cup (P \times R)$
5. $P \times \emptyset = \emptyset$

$(\mathit{TraceSet}, \cup, \otimes, \emptyset, 1)$ is a commutative semiring

6. $P \otimes Q = Q \otimes P$
7. $P \otimes (Q \otimes R) = (P \otimes Q) \otimes R$
8. $P \otimes 1 = P$
9. $P \otimes (Q \cup R) = (P \otimes Q) \cup (P \otimes R)$
10. $P \otimes \emptyset = \emptyset$

$\neg$ distributes over $\cup, \times, \otimes$ and is an involution

11. $\neg(P \cup Q) = \neg P \cup \neg Q$
12. $\neg(P \times Q) = \neg P \times \neg Q$
13. $\neg(P \otimes Q) = \neg P \otimes \neg Q$
14. $\neg\neg P = P$

$*$ distributes over $\times$ and is idempotent

15. $*(P \times Q) = *P \times *Q$
16. $** P = *P$

$\sqsubset$ is reflexive and trasitive; $\cup, \times$ and $*$ are monotonic; $\neg$ is anti-monotonic

17. $P \sqsubset P$
18. $(P \sqsubset Q) \wedge (Q \sqsubset R) \Rightarrow P \sqsubset R$
19. $P \sqsubset P' \Rightarrow (P \cup Q) \sqsubset (P' \cup Q)$
20. $P \sqsubset P' \Rightarrow (P \times Q) \sqsubset (P' \times Q)$
21. $P \sqsubset Q \Rightarrow *P \sqsubset *Q$
22. $P \sqsubset Q \Rightarrow \neg Q \sqsubset \neg P$

Other properties:

23. $(P \cup Q) \setminus \pi = P \setminus \pi \cup Q \setminus \pi$
24. $(P \times Q) \setminus \pi = (P \setminus \pi \times Q) \cup (P \times Q \setminus \pi)$
25. $(*P) \setminus \pi = *P \times (P \setminus \pi)$

26. $\pi \,.\, \emptyset = \emptyset$
27. $P \setminus \pi = \emptyset$ *for* $\pi \notin P$
28. $P \cup \emptyset = P$
29. $\nu\overline{x}.\emptyset = \emptyset$

30. $\pi \,.\, 1 = \pi$
31. $P \cup 1 = P$ *for* $P \neq \emptyset$
32. $\nu\overline{x}.1 = 1$
33. $\neg 1 = 1$

34. $*P = *P \times P$
35. $*P = *P \times *P$

36. $P \sqsubset Q \Leftrightarrow \pi.P \sqsubset \pi.Q$
37. $(P \sqsubset Q) \wedge (P \sqsubset R) \Rightarrow P \sqsubset (Q \cup R)$
38. $\pi \,.\, (S \times Q) \sqsubset S \times \pi \,.\, Q$
39. $S \sqsubset *S$
40. $P \sqsubset Q \Rightarrow P \sqsubset Q \times S$
    if $\mathrm{FC}(P) \cap \mathrm{FC}(S) = \emptyset$

In 38, 39, 40 assume no trace in S starts with a receive event

sent to z' (if any) will be forwarded to z via the copycat proxy (as described in Section II-D below).

A. Operations on Tracesets

We use the term *traceset* to denote a prefix-closed set of traces, and use the metavariables P, Q, R to range over tracesets. We often write tracesets as sets modulo prefix closure for brevity, and thus $\{x?\,.\,y!\}$ abbreviates $\{\epsilon,\ x?,\ x?\,.\,y!\}$. Here $x?$ is a receive event that receives zero arguments. We use the notation $\pi \in P$ to mean that P contains the single-event trace π. For example, $x! \in \{\epsilon, x!, x!\,.\,y?\}$ but $x! \notin \{\epsilon, y!, y!\,.\,x!\}$.

To define the meaning of expressions (and later types) compositionally, we present a collection of operations for defining and composing sets of traces in Figure 1. The constant 1 denotes the singleton set $\{\epsilon\}$ containing the empty trace. At an intuitive level, 1 denotes a computation that does nothing (a no-op), while the empty set $\emptyset$ denotes a computation that should never be executed.

The negation operation $\neg\pi$ swaps send and receives events, and negation extends in a pointwise manner to traces and tracesets.

The operation $\pi\,.\,P$ prefixes each trace in P with the event π. For example, $x!\,.\,\{y?\,.\,z!\} = \{x!\,.\,y?\,.\,z!\}$. Conversely, the operation $P \setminus \pi$ drops an initial event π from each trace in P, and drops traces in P that do not start with π; this operation yields the empty set if no trace in P starts with π. Thus,

$$\{z?,\ x?\,.\,y!\} \setminus x? = \{y!\}$$
$$\{z?,\ x?\,.\,y!\} \setminus y! = \emptyset$$

Note that, by sharing trace prefixes, a traceset P can be viewed as a "trace tree", in which every edge is labelled with an event, and the set of paths from the root to nodes in the tree captures the traces in P. From this perspective, $P \setminus \pi$ corresponds to navigating down the π-labelled edge from the root of the trace tree for P.

The restriction operation $\nu\overline{x}.P$ denotes the traces in P where none of the channels in $\overline{x}$ appear free. Intuitively, this operation introduces fresh channels and avoids channel collisions. For example,

$$\nu z.\{z!,\ x!z\,.\,z?,\ y!\,.\,z?\} = \{x!z\,.\,z?,\ y!\}$$

where traces with free occurrences of z are removed.

The operation $P \cup Q$ performs set union on tracesets, and $\bigcup_{i=1}^{n} P_i$ abbreviates the n-ary union $P_1 \cup \cdots \cup P_n$.

The operation $P \times Q$ denotes the non-deterministic interleaving of traces from P and Q. Thus,

$$\{z!\} \times \{x?\,.\,y!\} = \{z!\,.\,x?\,.\,y!,\ x?\,.\,z!\,.\,y!,\ x?\,.\,y!\,.\,z!\}$$

For each event π, $P \times Q$ contains traces starting with π and followed by traces in $(P \setminus \pi) \times Q$ or in $P \times (Q \setminus \pi)$; that is, it pulls the initial event π from either P or Q. Note that if $\pi \notin P$ (i.e. the single-event trace π does not occur in P) then $P \setminus \pi$ and $P \setminus \pi \times Q$ are both $\emptyset$, and similarly if $\pi \notin Q$.

The operation $P \otimes Q$ generalizes $P \times Q$ by also permitting communication between P and Q, where P may transmit an event $y!\overline{x}$, Q may receive the corresponding event $y?\overline{x}$ (or vice-versa), and computation proceeds with $\nu\overline{x}.((P \setminus y!\overline{x}) \otimes (Q \setminus y?\overline{x}))$. Note we assume that implicit α-renaming is used to match up the bound channels in the send event $y!\overline{x}$ of P with those in the receive event $y?\overline{x}$ of Q. For example,

$$\nu y.(\{y!z\} \otimes \{y?x\,.\,x!\}) = \nu y.(\{y!z\} \otimes \{y?z\,.\,z!\}) = \{z!\}$$

The n-ary operations $\prod_i^n P_i$ and $\coprod_i^n P_i$ generalize interleaving $\times$ and parallel composition $\otimes$, respectively. The operations P^n and $*P$ denote the interleaving of n or arbitrarily many copies of P respectively.

By convention $\nu\overline{x}$ binds as far to the right as possible while $\times$, $\cup$, $\pi\,.\,P$, $P \setminus \pi$, $*$ and $\neg$ bind with decreasing proximity. So for example,

$$\nu\overline{x}.*\pi\,.\,P \times Q \cup R = \nu\overline{x}.(((*(\pi\,.\,P)) \times Q) \cup R)$$

These operations on tracesets are closely related to the π-calculus [12], but with the restriction that send events only transmit fresh channels (i.e. in the trace $y!x\,.\,\alpha$ the channel x is bound in α), which yields a simpler semantic structure. Moreover, tracesets are (potentially infinite) sets of traces, rather than finite pieces of syntax with an associated evaluation semantics.

B. The Alternating Trace Containment Relation

As mentioned in the introduction, tracesets are naturally ordered according to the *alternating trace containment* relation $P \sqsubset Q$, which holds provided every send in P is also in Q, and conversely every receive in Q is also in P. More specifically, if P includes a trace $\alpha\,.\,\pi$ where π is a send, then if Q includes α then Q must also include $\alpha\,.\,\pi$ (and vice versa). This relation allows a program component with traceset P to be used safely in a context that expects a traceset Q.

Furthermore, after sending or receiving matching events, P and Q must continue to satisfy this relation.

To facilitate inductive proofs, we first define the *indexed alternating trace containment relation* $P \sqsubset_n Q$, which holds if $n = 0$ (the base case), or if $n > 0$ and:

1) For all send events $\pi \in P$, then $\pi \in Q$ and $P \setminus \pi \sqsubset_{n-1} Q \setminus \pi$
2) For all receive events $\pi \in Q$, then $\pi \in P$ and $P \setminus \pi \sqsubset_{n-1} Q \setminus \pi$

We then define $P \sqsubset Q$ to hold if and only if $P \sqsubset_n Q$ holds for all n. Thus, for example:

$$\{x?\,.\,z?,\ y?\} \sqsubset \{x?,\ y?\} \sqsubset \{x?\} \sqsubset 1 \sqsubset \{x!\} \sqsubset \{x!\,.\,y!\}$$

C. The Algebra of Traces

These operations on tracesets enjoy a rich algebraic structure, as described in Figure 1. In particular, $(\mathit{TraceSet}, \cup, \times, \emptyset, 1)$ and $(\mathit{TraceSet}, \cup, \otimes, \emptyset, 1)$ are both commutative semirings. Moreover, the operations $\cup$ and $\times$ are monotonic with regard to $\sqsubset$, and so for example if $P \sqsubset Q$ then $P \times R \sqsubset Q \times R$.

Unfortunately, monotonicity does not extend to parallel composition. As a counterexample, consider

$$\begin{array}{lcl} P & = & 1 \\ Q & = & \{x!\} \\ R & = & \{x?\,.\,y?\} \end{array}$$

Then,

$$P \otimes R = R \not\sqsubset Q \otimes R = \{x!\,.\,x?\,.\,y?, x?\,.\,x!\,.\,y?, x?\,.\,y?\,.\,x!, y?\}$$

as the right side includes the receive event $y?$ that is not on the left. Thus, the extra send $x!$ in Q exposes an additional receive $y?$ in $Q \otimes R$ that is not in $P \otimes R$.

Instead, we develop a Compositional Reasoning Lemma for a parallel composition $\nu\overline{x}.(P \otimes Q)$ that requires specifying the protocol R and the channel $\overline{x}$ by which P communicates to Q. If P satisfies the specification $P' \times R$ and Q satisfies the specification $Q' \times \neg R$, then the *parallel* composition $\nu\overline{x}.(P \otimes Q)$ satisfies the *interleaved* specification $P' \times Q'$. We assume that P and Q communicate only via the restricted channels $\overline{x}$, where R mentions only $\overline{x}$ but P' and Q' do not mention $\overline{x}$.

Lemma 1 (Compositional Reasoning)**.** *Suppose:*

$$\begin{array}{r} P \sqsubset P' \times R \\ Q \sqsubset Q' \times \neg R \\ \mathrm{FC}(Q') \cap \overline{x} = \emptyset \\ \mathrm{FC}(P') \cap \overline{x} = \emptyset \\ \mathrm{FC}(R) \subseteq \overline{x} \\ \mathrm{FS}(P) \cap \mathrm{FR}(Q) \subseteq \overline{x} \\ \mathrm{FR}(P) \cap \mathrm{FS}(Q) \subseteq \overline{x} \end{array}$$

Then:

$$\nu\overline{x}.(P \otimes Q) \;\sqsubset\; P' \times Q'$$

Proof. See the extended tech report [13]. □

As we will see, this lemma plays a critical role in our proofs.

D. Copycat Sends

One central choice in our design is that a send event always transmits fresh channels. For example, in the trace $k?y\,.\,x!y$, the y in $x!y$ is distinct from the y in $k?y$. This choice simplifies some parts of our development, but does require that we set up a mechanism to copy behavior from an existing channel to a fresh channel. In particular the *copycat send* abbreviation $x!!y$ sends out a copy of y by:

1) First, sending a fresh channel y' (the copy of y) along x.
2) Receiving on y' some (0, 1, or more) channels $\overline{z}$.
3) A fresh copy $\overline{z'}$ of the channels $\overline{z}$ is then passed along to the original y, where $|\overline{z'}| = |\overline{z}|$.
4) Any further communication is appropriately copied in the same manner.

To precisely define how the copycat send works we introduce a bijection $\theta : \mathit{Chan} \to \mathit{Chan}$ that maps a channel to its copy. We can then define the copycat send abbreviation $x!!\overline{y}$ that transmits (a copy $\theta y_{1\ldots n} = \theta y_1 \cdots \theta y_n$ of) existing channels $\overline{y}$ on channel x.

$$x!!\overline{y} \;\stackrel{\text{def}}{=}\; x!\theta\overline{y}\,.\,*(\bigcup_{y \in \overline{y}} \theta y?\overline{z}\,.\,y!!\overline{z})$$

The key property of a copycat is that is exhibits the same behavior on both its sides, since it simply copies events (with appropriate renaming) from one side to the other. Hence, if P is an appropriate specification for the behavior of one side, then $\neg\theta P$ is a corresponding specification for the other side's behavior, where θ performs appropriate channel renamings and the negation operation ($\neg$) changes receives on one side to sends on the other side, and vice-versa.

One caveat is that since the copycat may buffer events, we require that the specification P is invariant under buffering, which essentially means it does not matter in what order we remove events of the same direction (i.e. sends vs receives). More precisely, a traceset P is *well-formed* if for all events π_1 and π_2 of the same direction, it is the case that $P \setminus \pi_1 \setminus \pi_2 = P \setminus \pi_2 \setminus \pi_1$ and $P \setminus \pi_1$ is also well-formed.

The abbreviation $x!!\overline{y}$ satisfies the copycat lemma: for any well-formed specification P over $\overline{y}$ and any mapping θ from $\overline{y}$ to fresh channels, the abbreviation $x!!\overline{y}$ satisfies the specification $x!\theta\overline{y}\,.\,(P \times \neg\theta P)$.

Lemma 2 (Copycats Preserve Specifications)**.** *If P is well-formed and $\mathrm{FR}(P) = \emptyset$ and $\mathrm{FS}(P) \subseteq \overline{y}$ then:*

$$x!!\overline{y} \;\sqsubset\; x!\theta\overline{y}\,.\,(P \times \neg\theta P)$$

Proof. See the extended tech report [13]. □

III. Type Soundness for the Simply Typed Lambda Calculus

Based on the trace calculus properties and lemmas of the previous section, we are now in a position to study trace-based soundness proofs for a range of programming languages, starting with the simply typed λ-calculus.

A. STLC Syntax and Semantics

We summarize the STLC syntax as follows:

$$\begin{array}{rcl} e \in \mathit{Expr} & ::= & x \mid \lambda x.\,e \mid e\ e \mid \text{unit} \\ A, B \in \mathit{Type} & ::= & \text{Top} \mid \text{Unit} \mid A \to B \\ E \in \mathit{Env} & ::= & \emptyset \mid E, x : A \end{array}$$

We define the meaning $[\![e]\!]_k$ of each expression e with respect to a channel k as the following tracesets:

$$\begin{array}{rcll} [\![\cdot]\!]. & : & \mathit{Expr} \times \mathit{Chan} \to \mathit{Traceset} & \\ [\![x]\!]_k & \stackrel{\text{def}}{=} & k!!x & \\ [\![\lambda x.\,e]\!]_k & \stackrel{\text{def}}{=} & k!a\,.\,*(a?xh\,.\,[\![e]\!]_h) & a, h \notin \mathrm{FV}(e) \\ [\![e_1\ e_2]\!]_k & \stackrel{\text{def}}{=} & \nu k_1.([\![e_1]\!]_{k_1} & k_1, k_2, x_1, x_2 \notin \mathrm{FV}(e_1, e_2) \\ & & \quad \otimes *k_1?x_1\,.\,\nu k_2.([\![e_2]\!]_{k_2} & \\ & & \quad \otimes *k_2?x_2\,.\,x_1!!x_2 k)) & \\ [\![\text{unit}]\!]_k & \stackrel{\text{def}}{=} & k!a\,.\,*(a?\overline{x}\,.\,\mathit{wrong}!) & \end{array}$$

The traceset $[\![x]\!]_k$ simply sends a copy of x to k using a copycat send. We unify variables in programs with channels in traces, and so the terms *variable* and *channel* are synonyms.

The traceset $[\![\lambda x.\, e]\!]_k$ sends to k a fresh channel a, and then repeatedly receives on a an argument x and calling continuation h, and then evaluates e sending the result to h.

The traceset $[\![e_1\ e_2]\!]_k$ evaluates e_1 and receives the result along channel k_1 in x_1, evaluates e_2 and receives the result in x_2, and then sends to x_1 the argument-continuation pair x_2k. (The replicated receives $*k_1?x_1 \ldots$ and $*k_2?x_2 \ldots$ permit subexpressions to return multiple times, to facilitate first-class continuations in Section V.)

We use the expression unit to represent a program "going wrong" if unit is ever applied to a term. The traceset $[\![\text{unit}]\!]_k$ sends a channel a to its continuation, but if it ever receives an event on a it performs a send on the channel *wrong*, signalling that an error occurred. Thus, for example the following program trivially goes wrong.

$$[\![(\text{unit unit})]\!]_k = \{\mathit{wrong}!\}$$

We now address the meaning of types and type environments, starting with the meaning $[\![A]\!]_k$ of a type A with respect to a continuation k, which simply sends a fresh channel a to k, and then stands ready to receive operations on a according to the type A.

$$\begin{array}{rcl} [\![\cdot]\!]_\cdot & : & \mathit{Type} \times \mathit{Chan} \to \mathit{Traceset} \\ [\![A]\!]_k & \stackrel{\text{def}}{=} & *k!a \,.\, \neg[\![a : A]\!] \end{array}$$

Next, we define the meaning of a single-entry environment $[\![x : A]\!]$ by case analysis on A:

$$\begin{array}{rcl} [\![\cdot]\!] & : & \mathit{Env} \to \mathit{Traceset} \\ [\![x : A \to B]\!] & \stackrel{\text{def}}{=} & *x!yk \,.\, (\neg[\![y : A]\!] \times \neg[\![B]\!]_k) \\ [\![x : \text{Top}]\!] & \stackrel{\text{def}}{=} & 1 \\ [\![u : \text{Unit}]\!] & \stackrel{\text{def}}{=} & 1 \end{array}$$

If the environment contains a function binding x, then code in that environment can repeatedly send argument-continuation pairs yk to x, after which the code should be ready to receive (via $\neg$) B-values on k, and also receive (again via $\neg$) requests on y according to its type A. Note that, since $\times$ denotes arbitrary interleaving, requests on y may be received both before and after returns on k.

Our type language includes Top, since there are no operations on values of this type, an environment binding of type Top has the no-op trace 1.

In addition, to prevent well-typed programs from going wrong, the type Unit has no operations and thus is the no-op trace 1.

Note that we use the channel *wrong* only in the meaning of terms, not in types. Thus, if $[\![e]\!]_k \sqsubset [\![A]\!]_k$, then since *wrong* does not appear in $[\![A]\!]_k$, the term e is guaranteed not to go wrong (i.e. send to the channel *wrong*) provided it is used in accordance with its type specification A. Our type soundness theorem in the next section will prove that well-typed terms behave according to their types and thus do not go wrong.

Figure 2: Typing Rules for STLC

$$\frac{x : A \in E}{E \vdash x : A} \quad \text{[T-VAR]}$$

$$\frac{E, x : A \vdash e : B}{E \vdash \lambda x.\, e : A \to B} \quad \text{[T-ABS]}$$

$$\frac{}{E \vdash \text{unit} : \text{Unit}} \quad \text{[T-UNIT]}$$

$$\frac{E \vdash e_1 : A \to B \quad E \vdash e_2 : A}{E \vdash e_1\ e_2 : B} \quad \text{[T-APP]}$$

Note that our traceset meanings for Top and Unit coincide, ($[\![\text{Top}]\!]_k = [\![\text{Unit}]\!]_k = *k!a$), since no operations can be performed on a value of either static type. Despite this traceset equivalence, these two types are still distinct and we will treat them differently when we extend the language with subtyping in section IV. For example, $\text{Unit} <: \text{Top}$ but not vice-versa. Thus, these two types play different useful roles in the type system.

Finally, the meaning of a type environment with multiple bindings is the interleaving of the meanings of each individual binding:

$$[\![x_1 : A_1, \cdots, x_n : A_n]\!] \quad \stackrel{\text{def}}{=} \quad [\![x_1 : A_1]\!] \times \cdots \times [\![x_n : A_n]\!]$$

B. STLC Typing and Type Soundness

If an expression e has type A, then the traceset $[\![e]\!]_k$ should generate at most those output events permitted by $[\![A]\!]_k$, and should receive at least those input events in $[\![A]\!]_k$. Thus, $\vdash e : A$ must imply a corresponding alternating trace containment relation $[\![e]\!]_k \sqsubset [\![A]\!]_k$ on tracesets.

If e contains free variables with types defined by an environment E, then e can also interact with its environment according to the traceset specification $[\![E]\!]$. In this case we have that $E \vdash e : A$ must imply $[\![e]\!]_k \sqsubset [\![A]\!]_k \times [\![E]\!]$

We prove this traceset correspondence property by induction on the typing derivation $E \vdash e : A$. For each typing rule, we show that if this traceset correspondence holds for the antecedents in the rule then it also holds for the conclusion of the rule; in this case we say the rule is *admissible*.

Theorem 3 (Type Soundness)**.** *If $E \vdash e : A$ where each rule in this derivation is admissible, then $[\![e]\!]_k \sqsubset [\![A]\!]_k \times [\![E]\!]$.*

Proof. By induction on the derivation $E \vdash e : A$. □

Figure 2 summarizes the standard STLC typing rules, and the following lemma verifies that all these rules are admissible.

Lemma 3. *The STLC typing rules are admissible.*

Proof.

- Case [T-VAR] where $x : A \in E$ and $E \vdash x : A$. We show $[\![x]\!]_k \sqsubset [\![A]\!]_k \times [\![E]\!]$.

$$\begin{array}{lll} & [\![x]\!]_k & \\ = & k!!x & \\ \sqsubset & k!\theta x\,.\,([\![x : A]\!] \times \neg[\![\theta x : A]\!]) & \text{(Lem 2)} \\ \sqsubset & [\![x : A]\!] \times k!\theta x\,.\,\neg[\![\theta x : A]\!] & \text{(Prop 38)} \\ \sqsubset & [\![x : A]\!] \times *k!\theta x\,.\,\neg[\![\theta x : A]\!] & \text{(Prop 39)} \\ = & [\![x : A]\!] \times [\![A]\!]_k & \text{(def)} \\ \sqsubset & [\![x : A]\!] \times [\![A]\!]_k \times [\![E \setminus (x : A)]\!] & \text{(Prop 40)} \\ = & [\![A]\!]_k \times [\![E]\!] & \text{(since } x : A \in E\text{)} \end{array}$$

- Case [T-UNIT]

$$\begin{array}{lll} [\![\text{unit}]\!]_k = & k!a\,.\,*\,(a?\overline{x}\,.\,\mathit{wrong}!) & \\ \sqsubset & *k!a\,.\,1 & \text{(Prop 39)} \\ = & [\![\text{Unit}]\!]_k & \text{(def)} \\ \sqsubset & [\![\text{Unit}]\!]_k \times [\![E]\!] & \text{(Prop 40)} \end{array}$$

- Case [T-ABS] where $E \vdash \lambda x.\, e : A \to B$ via antecedent $E, x : A \vdash e : B$. We show $[\![\lambda x.\, e]\!]_k \sqsubset [\![A \to B]\!]_k \times [\![E]\!]$.

$$\begin{array}{lll} & [\![\lambda x.\, e]\!]_k & \\ = & k!a\,.\,*\,(a?xk'\,.\,[\![e]\!]_{k'}) & \\ \sqsubset & k!a\,.\,*\,(a?xk'\,.\,([\![B]\!]_{k'} \times [\![E, x : A]\!])) & (*) \\ = & k!a\,.\,*\,(a?xk'\,.\,([\![B]\!]_{k'} \times [\![x : A]\!] \times [\![E]\!])) & \\ \sqsubset & k!a\,.\,*\,(a?xk'\,.\,([\![B]\!]_{k'} \times [\![x : A]\!]) \times [\![E]\!]) & \\ & \text{(Prop 38)} & \\ \sqsubset & k!a\,.\,(*a?xk'\,.\,([\![B]\!]_{k'} \times [\![x : A]\!]) \times *[\![E]\!]) & \\ & \text{(Prop 15)} & \\ = & k!a\,.\,(*a?xk'\,.\,([\![B]\!]_{k'} \times [\![x : A]\!]) \times [\![E]\!]) & \\ & \text{(Prop 16)} & \\ \sqsubset & k!a\,.\,*\,(a?xk'\,.\,([\![B]\!]_{k'} \times [\![x : A]\!])) \times [\![E]\!] & \\ & \text{(Prop 38)} & \\ \sqsubset & *k!a\,.\,*\,(a?xk'\,.\,([\![B]\!]_{k'} \times [\![x : A]\!])) \times [\![E]\!] & \\ & \text{(Prop 39)} & \\ = & [\![A \to B]\!]_k \times [\![E]\!] & \text{(def)} \end{array}$$

  The (*) step is justified because by induction $[\![e]\!]_{k'} \sqsubset [\![B]\!]_{k'} \times [\![E, x : A]\!]$ and both prefix and replication are monotonic (Properties 36 and 21).
- Case [T-APP] where $E \vdash e_1\ e_2 : B$ via antecedents $E \vdash e_1 : A \to B$ and $E \vdash e_2 : A$. We begin by letting:

$$\begin{array}{rl} [\![e_1\ e_2]\!]_k = & \nu k_1.L_1 \otimes R_1 \\ L_1 = & [\![e_1]\!]_{k_1} \\ R_1 = & *k_1?x_1\,.\,\nu k_2.(L_2 \otimes R_2) \\ L_2 = & [\![e_2]\!]_{k_2} \\ R_2 = & *k_2?x_2\,.\,x_1!!x_2k \end{array}$$

  By induction $L_2 \sqsubset [\![A]\!]_{k_2} \times [\![E]\!]$ and by Lemma 2 with $P = [\![x_2 : A]\!] \times [\![B]\!]_k$ we have:

$$\begin{array}{lll} & R_2 & \\ \sqsubset & *k_2?x_2\,.\,(P \times \neg\theta P) & \\ \sqsubset & *k_2?x_2\,.\,(P \times x_1!\theta x_2\theta k\,.\,(\neg\theta P)) & \\ & \text{(Prop 38)} & \\ \sqsubset & *k_2?x_2\,.\,(P \times *x_1!\theta x_2\theta k\,.\,(\neg\theta P)) & \\ & \text{(Prop 39)} & \\ = & *k_2?x_2\,.\,([\![x_2 : A]\!] \times [\![B]\!]_k \times [\![x_1 : A \to B]\!]) & \\ \sqsubset & (*k_2?x_2\,.\,[\![x_2 : A]\!]) \times *[\![B]\!]_k \times *[\![x_1 : A \to B]\!] & \\ & \text{(Prop 38)} & \\ = & (*k_2?x_2\,.\,[\![x_2 : A]\!]) \times [\![B]\!]_k \times [\![x_1 : A \to B]\!] & \\ & \text{(Prop 16)} & \\ = & \neg(*k_2!x_2\,.\,\neg[\![x_2 : A]\!]) \times [\![B]\!]_k \times [\![x_1 : A \to B]\!] & \\ & \text{(Prop 14)} & \\ = & \neg[\![A]\!]_{k_2} \times [\![x_1 : A \to B]\!] \times [\![B]\!]_k & \end{array}$$

  By Lemma 1 $\nu k_2.(L_2 \otimes R_2) \sqsubset [\![x_1 : A \to B]\!] \times [\![B]\!]_k \times [\![E]\!]$. So:

$$\begin{array}{lll} & R_1 & \\ \sqsubset & *k_1?x_1\,.\,([\![x_1 : A \to B]\!] \times [\![B]\!]_k \times [\![E]\!]) & \\ \sqsubset & (*k_1?x_1\,.\,[\![x_1 : A \to B]\!]) \times *[\![B]\!]_k \times *[\![E]\!] & \text{(Prop 38)} \\ = & (*k_1?x_1\,.\,[\![x_1 : A \to B]\!]) \times [\![B]\!]_k \times [\![E]\!] & \text{(Prop 16)} \\ = & \neg(*k_1!x_1\,.\,\neg[\![x_1 : A \to B]\!]) \times [\![B]\!]_k \times [\![E]\!] & \text{(Prop 14)} \\ = & \neg[\![A \to B]\!]_{k_1} \times [\![B]\!]_k \times [\![E]\!] & \end{array}$$

  By induction $L_1 \sqsubset [\![A \to B]\!]_{k_1} \times [\![E]\!]$ so by Lemma 1 we have $[\![e_1\ e_2]\!]_k \sqsubset [\![B]\!]_k \times [\![E]\!]$.

□

This trace-based proof has a fairly "semantic" proof structure that mostly focuses on the syntactic representation of behavior, in contrast to traditional subject reduction proofs, which focus on the syntactic representation of program state. This trace-based proof does depend on the various lemmas and properties of the trace calculus, but those results are not language-specific and so can be reused in a variety of soundness proofs.

Having developed a type soundness proof for STLC, we next explore how well this proof supports extensions to the language or type system.

IV. Type Soundness for Subtyping

As our first extension, we enrich the type system with subtyping by adding the subsumption rule:

$$\frac{E \vdash e : B \quad B <: A}{E \vdash e : A} \quad \text{[T-SUB]}$$

along with the standard subtyping rules defined in Figure 3. As mentioned in the introduction, subtyping conservatively approximates the alternating trace containment relation.

Lemma 4 (Subtyping Implies Alternating Trace Containment)**.** *If $A <: B$ then $[\![A]\!]_k \sqsubset [\![B]\!]_k$.*

Proof. By induction on the subtyping derivation.

- [S-REFL] and [S-TRANS] follow from Properties 17 and 18.

Figure 3: Subtyping rules

$$\frac{}{A <: A} \quad \text{[S-REFL]}$$

$$\frac{A <: B \quad B <: C}{A <: C} \quad \text{[S-TRANS]}$$

$$\frac{}{A <: \text{Top}} \quad \text{[S-TOP]}$$

$$\frac{B_1 <: A_1 \quad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2} \quad \text{[S-ARROW]}$$

- Case $A <: \text{Top}$ via [S-TOP]. Directly from the definition of Top.
- Case $A_1 \to A_2 <: B_1 \to B_2$ via [S-ARROW].
  From the induction hypothesis we have $[\![B_1]\!]_k \sqsubset [\![A_1]\!]_k$ and $[\![A_2]\!]_k \sqsubset [\![B_2]\!]_k$.

$$\begin{array}{lll}
& [\![A_1 \to A_2]\!]_k & \\
= & *k!x\,.\,\neg[\![x : A_1 \to A_2]\!] & \\
= & *k!x\,.\,*\,x?ah\,.\,([\![a : A_1]\!] \times [\![A_2]\!]_h) & \\
\sqsubset & *k!x\,.\,*\,x?ah\,.\,([\![a : A_1]\!] \times [\![B_2]\!]_h) & \\
& \text{(IH, 36, 20, 21)} & \\
= & *k!x\,.\,*\,\neg x!ah\,.\,(\neg[\![a : A_1]\!] \times \neg[\![B_2]\!]_h) & \\
& \text{(Prop 14)} & \\
\sqsubset & *k!x\,.\,*\,\neg x!ah\,.\,(\neg[\![a : B_1]\!] \times \neg[\![B_2]\!]_h) & \\
& \text{(IH, Prop 22)} & \\
= & *k!x\,.\,\neg[\![x : B_1 \to B_2]\!] & \\
= & [\![B_1 \to B_2]\!]_k &
\end{array}$$

□

Since subtyping implies alternating trace containment, it is straightforward to show that the [T-SUB] rule is admissible and thus that STLC with subtyping is still sound.

Theorem 4. *The rule* [T-SUB] *is admissible.*

Proof. Suppose $E \vdash e : A$ via [T-SUB] from $E \vdash e : B$ and $B <: A$. By Lemma 4 $[\![B]\!]_k \sqsubset [\![A]\!]_k$ and by assumption $[\![e]\!]_k \sqsubset [\![B]\!]_k \times [\![E]\!]$. Thus by Property 20 we get $[\![e]\!]_k \sqsubset [\![A]\!]_k \times [\![E]\!]$. □

V. TYPE SOUNDNESS FOR CALL/CC

We add control-effects to the language in the form of first-class continuations.

$$e ::= \ldots \mid \text{call/cc}$$

The operation (call/cc f) calls the function f passing the current continuation k as an argument. The function f may either return a value of some type A or may call k passing an argument of type A; in either case call/cc returns a value of type A to its continuation. Thus, the type rule for call/cc is as follows, where the unconstrained type B indicates that the continuation function k never returns.

$$\frac{}{E \vdash \text{call/cc} : ((A \to B) \to A) \to A} \quad \text{[T-CALL/CC]}$$

The semantics for call/cc receives any call/cc invocation $a?fh$ and immediately calls f via $f!gh'$ passing a function g and a continuation h'. Values x sent to g or h' are then copycat sent to the original continuation h:

$$[\![\text{call/cc}]\!]_k \stackrel{\text{def}}{=} k!a\,.\,*\,(a?fh\,.\,f!gh'\,.\,(*\,(g?xk'\,.\,h!!x) \times *\,(h'?x\,.\,h!!x)))$$

Theorem 5. *The rule* [T-CALL/CC] *is admissible.*

Proof. By Lemma 2 with $P = [\![x : A]\!]$:

$$\begin{array}{lll}
& h!!x & \\
\sqsubset & h!\theta x\,.\,([\![x : A]\!] \times \neg[\![\theta x : A]\!]) & \\
\sqsubset & [\![x : A]\!] \times h!\theta x\,.\,\neg[\![\theta x : A]\!] & \text{(Prop 38)} \\
\sqsubset & [\![x : A]\!] \times *h!\theta x\,.\,\neg[\![\theta x : A]\!] & \text{(Prop 39)} \\
= & [\![x : A]\!] \times [\![A]\!]_h & \text{(*)} \\
\sqsubset & [\![x : A]\!] \times [\![A]\!]_h \times [\![B]\!]_{k'} & \text{(Prop 40, **)}
\end{array}$$

From (**) and Prop 36:

$$\begin{array}{lll}
& *g?xk'\,.\,h!!x & \\
\sqsubset & *g?xk'\,.\,([\![x : A]\!] \times [\![A]\!]_h \times [\![B]\!]_{k'}) & \\
\sqsubset & *g?xk'\,.\,([\![x : A]\!] \times [\![B]\!]_{k'}) \times *[\![A]\!]_h & \text{(Prop 38)} \\
= & \neg[\![g : A \to B]\!] \times *[\![A]\!]_h & \\
= & \neg[\![g : A \to B]\!] \times [\![A]\!]_h & \text{(Prop 16)}
\end{array}$$

From (*) and Prop 36:

$$\begin{array}{lll}
& *h'?x\,.\,h!!x & \\
\sqsubset & *h'?x\,.\,([\![x : A]\!] \times [\![A]\!]_h) & \\
\sqsubset & *h'?x\,.\,[\![x : A]\!] \times *[\![A]\!]_h & \text{(Prop 38)} \\
= & \neg[\![A]\!]_{h'} \times *[\![A]\!]_h & \\
= & \neg[\![A]\!]_{h'} \times [\![A]\!]_h & \text{(Prop 16)}
\end{array}$$

So we have:

$$\begin{array}{lll}
& f!gh'\,.\,(*(g?xk'\,.\,h!!x) \times *(h'?x\,.\,h!!x)) & \\
\sqsubset & f!gh'\,.\,(\neg[\![g : A \to B]\!] \times [\![A]\!]_h \times [\![A]\!]_h \times \neg[\![A]\!]_{h'}) & \\
= & f!gh'\,.\,(\neg[\![g : A \to B]\!] \times \neg[\![A]\!]_{h'} \times [\![A]\!]_h)) & \text{(Prop 35)} \\
\sqsubset & f!gh'\,.\,(\neg[\![g : A \to B]\!] \times \neg[\![A]\!]_{h'}) \times [\![A]\!]_h & \text{(Prop 38)} \\
\sqsubset & *f!gh'\,.\,(\neg[\![g : A \to B]\!] \times \neg[\![A]\!]_{h'}) \times [\![A]\!]_h & \text{(Prop 39)} \\
= & [\![f : (A \to B) \to A]\!] \times [\![A]\!]_h &
\end{array}$$

Thus:

$$\begin{array}{lll}
& [\![\text{call/cc}]\!]_k & \\
\sqsubset & k!a\,.\,*\,a?fh\,.\,([\![f : (A \to B) \to A]\!] \times [\![A]\!]_h) & \\
\sqsubset & *k!a\,.\,*\,a?fh\,.\,([\![f : (A \to B) \to A]\!] \times [\![A]\!]_h) & \text{(Prop 39)} \\
= & [\![((A \to B) \to A) \to A]\!]_k & \\
\sqsubset & [\![((A \to B) \to A) \to A]\!]_k \times [\![E]\!] & \text{(Prop 40)}
\end{array}$$

□

VI. Type Soundness for Reference Cells

We next introduce side-effects, in the form of mutable, dynamically allocated reference cells.

$$e ::= \ldots \mid \text{ref}$$

We take an "interface-oriented" view to reference cells, as proposed by Reynolds [14], whereby a reference cell of type $\text{Ref } C$ is encoded as a pair of a getter function (of type $\text{Unit} \to C$) and a setter function (of type $C \to \text{Unit}$) for reading and updating the reference cell, respectively. For simplicity, we use a Church-like encoding of pairs so the full type of a reference cell is:

$$\begin{aligned}\text{Ref } C &= \text{Pair } (\text{Unit} \to C)\ (C \to \text{Unit}) \\ &= ((\text{Unit} \to C) \to (C \to \text{Unit}) \to C) \to C\end{aligned}$$

The new primitive operation ref is a function that takes a value of type C and returns a new reference cell of type $\text{Ref } C$:

$$\frac{}{E \vdash \text{ref} : C \to \text{Ref } C} \quad \text{[T-Ref]}$$

To help use these interface-oriented reference cells, we introduce the abbreviations:

$$\begin{aligned}\text{let } x = e_1 \text{ in } e_2 &\stackrel{\text{def}}{=} (\lambda x.\, e_2)\ e_1 \\ e_1; e_2 &\stackrel{\text{def}}{=} (\lambda x.\, e_2)\ e_1 \qquad x \notin \text{FV}(e_2) \\ !e &\stackrel{\text{def}}{=} e\ (\lambda gs.\, g\ \text{unit}) \\ e_1 := e_2 &\stackrel{\text{def}}{=} e_1\ (\lambda gs.\, \text{let } t = e_2 \text{ in } s\ t; t)\end{aligned}$$

Thus, for example, the following code fragment yields the expected behavior:

$$\begin{aligned}&\text{let } r = \text{ref } x \text{ in} \\ &\quad r := y; \\ &\quad !r;\end{aligned}$$

As a starting point for defining the semantics of ref, we first define a *reference cell traceset* R_x that can receive and process events on the channels get and set.

$$\begin{aligned}R_x = {} & get?uk \,.\, (k!!x \times R_x) \\ & \cup\, set?yk \,.\, (k!u \times R_y)\end{aligned}$$

The event $get?uk$ causes R_x to copycat send x to the continuation k, and then continue behaving as R_x. The event $set?yk$ causes R_x to send a dummy unit value to k, and continue as R_y so that subsequent get events receive y rather than x.

The traceset of ref then essentially wraps R_x in the appropriate interface.

$$\begin{aligned}&[\![\text{ref}]\!]_k \\ \stackrel{\text{def}}{=}\ & k!a \,.\, *\,(a?xh \,.\, h!p \,.\, \nu set, get. \\ & \qquad (R_x \mid * p?f_1k_1 \,.\, \nu k_2 \,.\, (f_1!!get, k_2 \\ & \qquad\qquad \mid * k_2?g \,.\, g!!set, k_1)))\end{aligned}$$

This traceset sends a to the ref continuation and then repeatedly receives requests $a?xh$ to create a new reference cell with an initial value of x. It returns a channel p (of type $\text{Ref } C$) to h, and initializes a traceset R_x, with channels get and set, to record the current value of the reference cell. When p receives a function f_1 of type $(\text{Unit} \to C) \to (C \to \text{Unit}) \to C$, it simply sends get and set to f_1.

The following lemma shows that the traceset R_x is approximated by the types of the exported get and set functions, and of the imported variable x.

Lemma 5 (Reference Cell Specification)**.**

$$R_x \sqsubset \neg[\![get : \text{Unit} \to C, set : C \to \text{Unit}]\!] \times [\![x : C]\!]$$

Proof. Let $\text{RHS} = \neg[\![get : \text{Unit} \to C, set : C \to \text{Unit}]\!] \times [\![x : C]\!]$. We prove by induction on n that

$$R_x \ \sqsubset_n\ \text{RHS}$$

Note that there are no sends in R_x. We have two receive events in RHS to consider:

- $get?uk \in \text{RHS}$. We have:

$$\begin{aligned}& R_x \setminus get?uk \\ = {} & (k!!x \times R_x) \\ \sqsubset {} & k!x' \,.\, ([\![x : C]\!] \times \neg[\![x' : C]\!]) \times R_x \quad \text{(Lem 2)} \\ \sqsubset {} & *k!x' \,.\, \neg[\![x' : C]\!] \times R_x\end{aligned}$$

$$\begin{aligned}& \text{RHS} \setminus get?uk \\ = {} & \text{RHS} \times [\![u : \text{Unit}]\!] \times [\![C]\!]_k \\ = {} & \text{RHS} \times *k!x' \,.\, \neg[\![x' : C]\!]\end{aligned}$$

  Since, by induction $R_x \sqsubset_{n-1} \text{RHS}$ we have $R_x \setminus get?uk \sqsubset_{n-1} \text{RHS} \setminus get?uk$.
- $set?yk \in \text{RHS}$. We have:

$$\begin{aligned}R_x \setminus set?yk &= k!u \times R_y \\ &\sqsubset *k!u \times R_y\end{aligned}$$

$$\begin{aligned}\text{RHS} \setminus set?yk &= \text{RHS} \times [\![y : C]\!] \times [\![\text{Unit}]\!]_k \\ &= \text{RHS} \times [\![y : C]\!] \times *k!u \,.\, \neg[\![u : \text{Unit}]\!] \\ &= \text{RHS} \times [\![y : C]\!] \times *k!u\end{aligned}$$

  Since, by induction $R_y \sqsubset_{n-1} \neg[\![get : \text{Unit} \to C, set : C \to \text{Unit}]\!] \times [\![y : C]\!]$ we have $R_x \setminus \pi \sqsubset_{n-1} \text{RHS} \setminus \pi$

□

With this lemma we show that the type rule for reference cells is admissible.

Theorem 6. *The* [T-Ref] *rule is admissible.*

Proof. Let $P = [\![get : \text{Unit} \to C]\!] \times [\![(C \to \text{Unit}) \to C]\!]_{k_2}$. Then:

$$\begin{aligned}& f_1!!get, k_2 \\ \sqsubset {} & f_1!\theta get, \theta k_2 \,.\, (P \times \neg\theta P) \\ \sqsubset {} & *f_1!\theta get, \theta k_2 \,.\, (P \times \neg\theta P) \\ \sqsubset {} & [\![f_1 : ((\text{Unit} \to C) \to (C \to \text{Unit}) \to C)]\!] \\ & \times [\![get : \text{Unit} \to C]\!] \times [\![(C \to \text{Unit}) \to C]\!]_{k_2}\end{aligned}$$

Let $Q = [\![set : C \to \text{Unit}]\!] \times [\![C]\!]_{k_1}$. Then:

$$\begin{array}{ll} & g!!set, k_1 \\ \sqsubset & g!\theta set, \theta k_1 \,.\, (Q \times \neg\theta Q) \\ \sqsubset & *g!\theta set, \theta k_1 \,.\, (Q \times \neg\theta Q) \\ \sqsubset & [\![g : (C \to \text{Unit}) \to C]\!] \times [\![set : C \to \text{Unit}]\!] \times [\![C]\!]_{k_1} \end{array}$$

$$\begin{array}{ll} & *k_2?g \,.\, g!!set, k_1 \\ \sqsubset & *k_2?g \,.\, ([\![set : C \to \text{Unit}]\!] \times [\![C]\!]_{k_1} \\ & \qquad \times [\![g : (C \to \text{Unit}) \to C]\!]) \\ \sqsubset & \neg[\![(C \to \text{Unit}) \to C]\!]_{k_2} \times [\![set : C \to \text{Unit}]\!] \times [\![C]\!]_{k_1} \end{array}$$

From this we have:

$$\begin{array}{ll} & *p?f_1k_1 \,.\, \nu k_2.(f_1!!get, k_2) \otimes (*k_2?g \,.\, g!!set, k_1) \\ \sqsubset & *p?f_1k_1 \,.\, ([\![get : \text{Unit} \to C]\!] \times [\![set : C \to \text{Unit}]\!] \times [\![C]\!]_{k_1} \\ & \qquad \times [\![f_1 : ((\text{Unit} \to C) \to (C \to \text{Unit}) \to C)]\!]) \\ \sqsubset & \neg[\![p : \text{Ref } C]\!] \times [\![get : \text{Unit} \to C, set : C \to \text{Unit}]\!] \end{array}$$

By Lemma 5, $R_x \sqsubset [\![get : \text{Unit} \to C, set : C \to \text{Unit}]\!] \times [\![x : C]\!]$. Thus:

$$\begin{array}{lll} & [\![\text{ref}]\!]_k & \\ \sqsubset & k!a \,.\, *(a?xh \,.\, h!p \,.\, ([\![x : C]\!] \times \neg[\![p : \text{Ref } C]\!])) & \text{(Lem 1)} \\ \sqsubset & k!a \,.\, *(a?xh.([\![x : C]\!] \times h!p \,.\, \neg[\![p : \text{Ref } C]\!])) & \\ \sqsubset & k!a \,.\, *(a?xh \,.\, ([\![x : C]\!] \times [\![\text{Ref } C]\!]_h)) & \\ = & k!a \,.\, [\![a : C \to \text{Ref } C]\!] & \\ \sqsubset & *k!a \,.\, [\![a : C \to \text{Ref } C]\!] & \\ = & [\![C \to \text{Ref } C]\!]_k & \\ \sqsubset & [\![C \to \text{Ref } C]\!]_k \times [\![E]\!] & \end{array}$$

□

VII. Type Soundness for Fork

Our final language extension adds multiple concurrent threads, via an operation (fork f) that evaluates the thunk f in a new thread of control. As we will see, even though concurrency (like side-effects) is a significant language extension, it requires only local extensions to the language semantics. The syntactic extension and corresponding type rule for fork are straightforward:

$$e ::= \ldots \mid \text{fork}$$

$$\frac{}{E \vdash \text{fork} : (\text{Unit} \to \text{Unit}) \to \text{Unit}} \quad \text{[T-Fork]}$$

Rather surprisingly, extending the language semantics with concurrency is also straightforward:

$$[\![\text{fork}]\!]_k \stackrel{\text{def}}{=} k!a \,.\, *a?fh \,.\, h!u \,.\, f!uh \,.\, h?y \,.\, 1$$

Here, the channel a (representing the fork value) is immediately returned to fork's continuation. When a later receives a fork request $a?fh$, it immediately returns a unit channel u to the continuation h, but also calls the given thunk f. Thus, the two consecutive send events performed by fork are sufficient to initiate concurrent evaluation. Finally, if f later returns via $h?y$ its result is discarded and its thread is terminated.

We can use reference cells to implement inter-thread synchronization primitives such as semaphores, since read and write operations on reference cells execute atomically. The following proof shows that this language extension with concurrency preserves type soundness.

Theorem 7. *The rule* [T-Fork] *is admissible.*

Proof.

$$\begin{array}{lll} & [\![\text{fork}]\!]_k & \\ = & k!a \,.\, *a?xh \,.\, h!u \,.\, x!uh \,.\, h?y \,.\, 1 & \\ \sqsubset & k!a \,.\, *a?xh \,.\, h!u \,.\, x!uh \,.\, (\neg[\![u : \text{Unit}]\!] \times \neg[\![\text{Unit}]\!]_h) & \\ \sqsubset & k!a \,.\, *a?xh \,.\, h!u \,.\, [\![x : \text{Unit} \to \text{Unit}]\!] & \\ \sqsubset & k!a \,.\, *a?xh \,.\, h!u \,.\, ([\![x : \text{Unit} \to \text{Unit}]\!] \times \neg[\![u : \text{Unit}]\!]) & \\ \sqsubset & k!a \,.\, *a?xh \,.\, ([\![x : \text{Unit} \to \text{Unit}]\!] \times h!u \,.\, \neg[\![u : \text{Unit}]\!]) & \\ \sqsubset & k!a \,.\, *a?xh \,.\, ([\![x : \text{Unit} \to \text{Unit}]\!] \times *h!u \,.\, \neg[\![u : \text{Unit}]\!]) & \\ = & k!a \,.\, *a?xh \,.\, ([\![x : \text{Unit} \to \text{Unit}]\!] \times [\![\text{Unit}]\!]_h) & \\ \sqsubset & [\![(\text{Unit} \to \text{Unit}) \to \text{Unit}]\!]_k \times [\![E]\!] & \end{array}$$

□

VIII. Related and Future Work

Wright and Felleisen [1] introduced *subject reduction* as a technique for proving soundness of type systems by showing that evaluation preserves typing: if a program state S is well-typed $\vdash S$ and S evaluates to S' (written $S \to S'$) then S' is also well-typed $\vdash S'$. This proof technique has proven highly flexible, in large part due to the global nature of the evaluation relation $\to$, which can observe or mutate any part of the program state. For example, side-effects and control-effects manipulate the global store and evaluation context, respectively [15], [16], [17].

Before subject reduction, many type soundness proofs were based on denotational semantics [18], [19], [20], [21], [22], typically with different domain equations or different proof techniques. Even when two soundness proofs addressed extensions of a common language, it was not clear whether or how different proofs could be merged to yield a proof for the combined system. By using the semantic framework of rewriting-based operational semantics, subject reduction provided a common proof structure that could accommodate a wide range of languages and type systems. This paper takes this work one step further—by formalizing types (A), terms (e), and typing judgments ($\vdash e : A$) all in the common framework of tracesets, the admissibility of each typing rule can now be verified independently. Thus, we adapt the ideas of subject reduction to focus on syntactic representations of behavior (formalized as tracesets) rather than on syntactic representations of program states.

Much prior work has studied the denotational semantics of higher-order languages, often with the goal of developing fully abstract denotational semantics [5], [6], in which observable equivalence implies denotational equivalence. Game semantics has emerged as an appealing foundation for developing fully abstract denotational models. For example, fully abstract game semantics have been developed for PCF [7], [8] or for languages with features such as call-by-value [23], general references [24], and exceptions [25], [26] to name just a few. Game semantics has also been used as a foundation for

language design [27], [28]. Compositional game semantics also facilitate compositional verification [29].

As mentioned earlier, our trace calculus notably resembles the π-calculus [30], [31], [32], but with some differences. The trace calculus consists of a collection of operators and relations over tracesets, with associated axioms, rather than syntactic constructors. Moreover, traces support negation since send and receive events both bind their argument channels, which allow us to express contravariance in types as negation on tracesets. Nonetheless, this connection deserves further exploration, and perhaps existing results from the π-calculus could facilitate or simplify our type soundness proofs.

A number of type systems have been developed for the π-calculus [33], [34], [35], [36]. For our purposes, tracesets themselves are sufficient both for describing implementations (e.g. $[\![e]\!]_k$) and also specifications (e.g. $[\![A]\!]_k$), and thus we have not needed an extra type specification language for traces.

In this work we give a semantics for untyped terms ($\lambda x.\, e$) but a clear topic for future work is to give a traceset semantics for typed terms ($\lambda x\colon A.\, e$) and dependent types ($\Pi x\colon A.\, B$), and to extend this proof technique to additional language constructs (e.g. constants, primitive operations, and data constructors) and to richer type systems (e.g. with polymorphism, bounded quantification, dependent types, etc.). Several interesting questions immediately arise, for example, what is the trace semantic meaning $[\![\forall X.A]\!]_k$ of a polymorphic type?

Another important direction is the relationship between higher-order dynamic contracts [37], [38] (which filter behaviors) and static types (which specify behavior), and perhaps expressing both in the common framework of tracesets could help elucidate this relationship.

Acknowledgments We thank Philippa Gardner, Scott Smith, DeLesley Hutchins, Philip Wadler, Jeremy Siek, and Martin Abadi for helpful conversations on this work

REFERENCES

[1] A. Wright and M. Felleisen, "A syntactic approach to type soundness," *Info. Comput.*, vol. 115, no. 1, pp. 38–94, 1994.

[2] R. Alur, T. Henzinger, O. Kupferman, and M. Vardi, "Alternating refinement relations," *CONCUR'98 Concurrency Theory*, pp. 163–178, 1998. [Online]. Available: http://dx.doi.org/10.1007/BFb0055622

[3] D. Hutchins, "Pure subtype systems: A type theory for extensible software," 2009.

[4] ——, "Pure subtype systems," in *Symposium on Principles of Programming Languages*, vol. 45, no. 1. ACM, 2010, pp. 287–298.

[5] R. Cartwright and M. Felleisen, "Observable sequentiality and full abstraction," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1992, pp. 328–342.

[6] D. Hopkins and C. Ong, "Homer: A higher-order observational equivalence model checker," in *Computer Aided Verification*. Springer, 2009, pp. 654–660.

[7] S. Abramsky, R. Jagadeesan, and P. Malacaria, "Full abstraction for PCF," *Information and Computation*, vol. 163, pp. 409–470, 1996.

[8] J. M. E. Hyland and C.-H. L. Ong, "On full abstraction for PCF: I, II, and III," *Inf. Comput.*, vol. 163, no. 2, pp. 285–408, 2000.

[9] J. Chroboczek, "Game semantics and subtyping," in *Logic in Computer Science, 2000. Proceedings. 15th Annual IEEE Symposium on*. IEEE, 2000, pp. 192–203.

[10] ——, "Subtyping recursive games," in *Typed Lambda Calculi and Applications*. Springer, 2001, pp. 61–75.

[11] ——, "Game semantics and subtyping," Ph.D. dissertation, 2003.

[12] R. Milner, *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.

[13] T. Disney and C. Flanagan, "Game Semantics for Type Systems," University of California Santa Cruz, Tech. Rep., May 2015.

[14] J. C. Reynolds, *The essence of ALGOL*. Cambridge, MA, USA: Birkhauser Boston Inc., 1997, pp. 67–88.

[15] M. Felleisen and D. P. Friedman, "Control operators, the SECD-machine, and the lambda-calculus," in *3rd Working Conference on the Formal Description of Programming Concepts*, 1986, pp. 193–219.

[16] ——, "A syntactic theory of sequential state," Indiana University, Bloomington, Indiana, Computer Science Dept. Technical Report 230, 1987.

[17] M. Felleisen and R. Hieb, "The revised report on the syntactic theories of sequential control and state," *Theoretical computer science*, vol. 103, no. 2, pp. 235–271, 1992.

[18] Milner, R, "A theory of type polymorphism in programming," *J. Comput. Syst. Sci.*, vol. 17, pp. 348–375, 1978.

[19] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1982, pp. 207–212.

[20] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin, "Dynamic typing in a statically-typed language," in *Symposium on Principles of Programming Languages*, 1989, pp. 213–227.

[21] L. M. M. Damas, "Type assignment in programming languages," Ph.D. dissertation, University of Edinburgh, 1985.

[22] B. Duba, R. Harper, and D. MacQueen, "Typing first-class continuations in ml," in *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1991, pp. 163–173.

[23] S. Abramsky and G. McCusker, "Call-by-value games," in *CSL*, 1997, pp. 1–17.

[24] S. Abramsky, K. Honda, and G. McCusker, "A fully abstract game semantics for general references," in *LICS*, 1998, pp. 334–344.

[25] R. Cartwright, P.-L. Curien, and M. Felleisen, "Fully abstract semantics for observably sequential languages," *Inf. Comput.*, vol. 111, no. 2, pp. 297–401, 1994.

[26] J. Laird, "A fully abstract game semantics of local exceptions," in *Logic in Computer Science*, Washington, DC, USA, 2001. [Online]. Available: http://portal.acm.org/citation.cfm?id=871816.871866

[27] J. Longley and N. Wolverson, "Eriskay: a programming language based on game semantics," in *Games for Logic and Programming Languages III Workshop*. Citeseer, 2008.

[28] N. Wolverson, "Game semantics for an object-oriented language," 2009.

[29] S. Abramsky, D. R. Ghica, A. S. Murawski, and C.-H. L. Ong, "Applying game semantics to compositional software modeling and verification," in *TACAS*, 2004, pp. 421–435.

[30] R. Milner, "The polyadic π-calculus: A tutorial," *Logic and Algebra of Specification*, vol. 94, 1991.

[31] B. Pierce, "Foundational calculi for programming languages," *Handbook of Computer Science and Engineering*, pp. 2190–2207, 1995.

[32] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, I," *Information and computation*, vol. 100, no. 1, pp. 1–40, 1992.

[33] B. Pierce and D. Sangiorgi, "Typing and subtyping for mobile processes," in *Logic in Computer Science, 1993. LICS'93., Proceedings of Eighth Annual IEEE Symposium on*. IEEE, 1993, pp. 376–385.

[34] Y. Deng and D. Sangiorgi, "Towards an algebraic theory of typed mobile processes," *Automata, Languages and Programming*, pp. 445–456, 2004.

[35] N. Kobayashi, B. Pierce, and D. Turner, "Linearity and the π-calculus," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 5, pp. 914–947, 1999.

[36] J. Laird, "A game semantics of the asynchronous π-calculus," *CONCUR 2005–Concurrency Theory*, pp. 51–65, 2005.

[37] R. B. Findler and M. Felleisen, "Contracts for higher-order functions," in *Proceedings of the International Conference on Functional Programming*, 2002, pp. 48–59.

[38] R. Back and J. Von Wright, "Contracts, games, and refinement," *Information and Computation*, vol. 156, no. 1, pp. 25–45, 2000.