Trace-Based Program Analysis

Christopher Colby
Carnegie Mellon University
work done while on leave at École Polytechnique

joint work with Peter Lee at CMU

paper draft: `http://lix.polytechnique.fr/~colby`

Semantics-based program analysis (e.g., [CC77])

Goals:

- compiler optimization
- static code transformation
- automated program verification
- static debugging

A subclass: trace-based analysis

Solves some common difficulties of semantics-based analysis:

- determines meaningful properties of **loops**
- analysis of separate code fragments can be **composed**

Outline

- introduction: why loops and compositionality are difficult
- limitations of state-based analyses and the intuition of trace-based analyses
- transfer transition systems: the vehicle of trace-based analysis
- some example applications:
  - software pipelining (a code transformation on loops)
  - loop-invariant removal
  - a compositional alias analysis

Loops are difficult

Analyzing loops is important for optimizing compilers [AHU86].
Typically involves *approximation* of infinite structures:

- Denotational semantics: limits of infinite chains.
- Operational semantics: unbounded transition sequences.

Typical semantics-based methods can incur a fatal loss of precision.
In practice, dataflow analysis [MJ81] is used (not semantics-based).
Trace-based analysis can analyze loops “without approximation”.

Demonstration: software pipelining.
Becomes loop-invariant removal by changing 3 *simple* equations!

Compositionality is difficult

Modular analysis needed for modular optimization, static debugging, and verification.

The quest for a compositional analysis of intensional properties:

- Denotational models: compositional but lack intensional detail.
- Operational models: intensional but usually non-compositional.

Demonstration: compositional alias analysis.
Uncovers unexpected behavior in even tiny (3-line) fragments!

The usual analysis paradigm: state-based analysis

Most program analyzers return a function $\mathcal{S}$ that maps descriptions of run-time points to the *state properties* of interest:

$\mathcal{S} \in \mathit{Label} \to \mathit{Property}[\mathit{State}]$ "monovariant"

$\mathcal{S} \in \mathit{Label} \times \mathit{Occurrence} \to \mathit{Property}[\mathit{State}]$ "polyvariant"

Label: finite set of program labels.
Occurrence: finite partition of run-time occurrences of labels.
Property[*State*]: state properties (corresponding to sets of states).

But state properties just *aren't good enough* for some applications.

Why state-based analysis is insufficient

Consider the following assembly-like program:

$$\begin{aligned} &\mathsf{L}_1 : n \leftarrow 0 \\ &\mathsf{L}_2 : n \leftarrow n + 1 \\ &\mathsf{L}_3 : \textbf{goto}\ \mathsf{L}_2 \end{aligned}$$

Assume a *state* is an integer representing n. Evaluation from state k:

$$\mathsf{L}_1 \!:\! k \longmapsto \mathsf{L}_2 \!:\! 0 \longmapsto \mathsf{L}_3 \!:\! 1 \longmapsto \mathsf{L}_2 \!:\! 1 \longmapsto \mathsf{L}_3 \!:\! 2 \longmapsto \mathsf{L}_2 \!:\! 2 \longmapsto \mathsf{L}_3 \!:\! 3 \longmapsto \cdots$$

The *optimal* "monovariant" state-based analysis of this program is:

$$\begin{aligned} \mathcal{S}(\mathsf{L}_1) &= \{k\} \\ \mathcal{S}(\mathsf{L}_2) &= \{0,1,2,\ldots\} = \mathbb{N} \\ \mathcal{S}(\mathsf{L}_3) &= \{1,2,3,\ldots\} = \mathbb{N} - \{0\} \end{aligned}$$

A way to represent better properties: "polyvariance"

Our program again:

$$\begin{aligned} &\mathsf{L}_1 : n \leftarrow 0 \\ &\mathsf{L}_2 : n \leftarrow n + 1 \\ &\mathsf{L}_3 : \textbf{goto}\ \mathsf{L}_2 \end{aligned}$$

Split labels by some *finite* scheme, such as loop entry (calling) point:

$$\begin{aligned} \mathcal{S}(\mathsf{L}_1) &\supseteq \{k\} \\ \mathcal{S}(\mathsf{L}_2, \mathsf{L}_1) &\supseteq \{0\} \\ \mathcal{S}(\mathsf{L}_3, \mathsf{L}_1) &\supseteq \{1\} \\ \mathcal{S}(\mathsf{L}_2, \mathsf{L}_3) &\supseteq \{1,2,3,\ldots\} = \mathbb{N} - \{0\} \\ \mathcal{S}(\mathsf{L}_3, \mathsf{L}_3) &\supseteq \{2,3,4,\ldots\} = \mathbb{N} - \{0,1\} \end{aligned}$$

$$
\begin{aligned}
&\mathsf{L}_1 : n \leftarrow 0\\
&\mathsf{L}_2 : n \leftarrow n+1\\
&\mathsf{L}_3 : \textbf{goto}\ \mathsf{L}_2
\end{aligned}
$$

or by iteration equivalence class (still *finite*):

$$
\begin{array}{lcl}
\mathcal{S}(\mathsf{L}_1) & \supseteq & \{k\} \\
\mathcal{S}(\mathsf{L}_2, \text{odd}) & \supseteq & \{0, 2, 4, \ldots\} = \{i \mid 0 \geq i \equiv 0 \pmod 2\} \\
\mathcal{S}(\mathsf{L}_3, \text{odd}) & \supseteq & \{1, 3, 5, \ldots\} = \{i \mid 0 \geq i \equiv 1 \pmod 2\} \\
\mathcal{S}(\mathsf{L}_2, \text{even}) & \supseteq & \{1, 3, 5, \ldots\} = \{i \mid 1 \geq i \equiv 1 \pmod 2\} \\
\mathcal{S}(\mathsf{L}_3, \text{even}) & \supseteq & \{2, 4, 6, \ldots\} = \{i \mid 1 \geq i \equiv 0 \pmod 2\}
\end{array}
$$

The problem and the solution

No matter how clever the "polyvariance", there are still interesting properties of this program that seemingly cannot be represented:

- The variable *n* is always increasing.
- More specifically, each iteration increments *n* by 1.

The essential problem: A *finite* collection of state properties cannot represent an *infinitely*-repeating pattern between iterations.

But a finite collection of *trace properties* can!

The solution: *Trace-based analysis*:

$$\mathcal{T} \in \mathit{Label} \rightarrow \mathit{Property}[\mathit{Trace}] \qquad \text{where } \mathit{Trace} = (\mathit{Label} \times \mathit{State})^*$$

A trace-based analysis of our example program

Recall our example program and its evaluation from state k:

$$
\begin{aligned}
&\mathsf{L}_1 : n \leftarrow 0\\
&\mathsf{L}_2 : n \leftarrow n+1 \qquad \mathsf{L}_1\!:\!k \longmapsto \mathsf{L}_2\!:\!0 \longmapsto \mathsf{L}_3\!:\!1 \longmapsto \mathsf{L}_2\!:\!1 \longmapsto \mathsf{L}_3\!:\!2 \longmapsto \cdots\\
&\mathsf{L}_3 : \textbf{goto}\ \mathsf{L}_2
\end{aligned}
$$

A trace-based analysis of this program produces a function $\mathcal{T}$:

$$
\begin{array}{lcl}
\mathcal{T}(\mathsf{L}_1) & \supseteq & \{\mathsf{L}_1\!:\!k \mapsto \mathsf{L}_2\!:\!0 \mapsto \mathsf{L}_3\!:\!1 \mapsto \mathsf{L}_2\!:\!1 \mapsto \mathsf{L}_3\!:\!2 \mapsto \cdots\} \\
\mathcal{T}(\mathsf{L}_2) & \supseteq & \{\mathsf{L}_2\!:\!0 \mapsto \mathsf{L}_3\!:\!1 \mapsto \mathsf{L}_2\!:\!1 \mapsto \mathsf{L}_3\!:\!2 \mapsto \cdots, \\
& & \;\; \mathsf{L}_2\!:\!1 \mapsto \mathsf{L}_3\!:\!2 \mapsto \mathsf{L}_2\!:\!2 \mapsto \mathsf{L}_3\!:\!3 \mapsto \cdots, \\
& & \;\; \ldots\} \\
\mathcal{T}(\mathsf{L}_3) & \supseteq & \{\mathsf{L}_3\!:\!1 \mapsto \mathsf{L}_2\!:\!1 \mapsto \mathsf{L}_3\!:\!2 \mapsto \mathsf{L}_2\!:\!2 \mapsto \cdots, \\
& & \;\; \mathsf{L}_3\!:\!2 \mapsto \mathsf{L}_2\!:\!2 \mapsto \mathsf{L}_3\!:\!3 \mapsto \mathsf{L}_2\!:\!3 \mapsto \cdots, \\
& & \;\; \ldots\}
\end{array}
$$

In other words:

$$
\begin{array}{lcl}
\mathcal{T}(\mathsf{L}_1) & \supseteq & \text{(as before)} \\
\mathcal{T}(\mathsf{L}_2) & \supseteq & \{\mathsf{L}_2\!:\!i \mapsto \mathsf{L}_3\!:\!(i+1) \mapsto \mathsf{L}_2\!:\!(i+1) \mapsto \mathsf{L}_3\!:\!(i+2) \mapsto \cdots \mid i \geq 0\} \\
\mathcal{T}(\mathsf{L}_3) & \supseteq & \{\mathsf{L}_3\!:\!i \mapsto \mathsf{L}_2\!:\!i \mapsto \mathsf{L}_3\!:\!(i+1) \mapsto \mathsf{L}_2\!:\!(i+1) \cdots \mid i \geq 1\}
\end{array}
$$

The above *best* properties are finitely representable with *no* loss of information and reveal the "increment by 1" property:

$$
\begin{array}{lcl}
\mathcal{T}(\mathsf{L}_1) & = & \text{“}\mathsf{L}_1\!:\!k \mapsto \mathsf{L}_2\!:\!0\text{”} \\
\mathcal{T}(\mathsf{L}_2) & = & \text{“}\mathsf{L}_2\!:\!I \mapsto \mathsf{L}_3\!:\!(I+1) \mapsto \mathsf{L}_2\!:\!(I+1) \text{ where } I \geq 0\text{”} \\
\mathcal{T}(\mathsf{L}_3) & = & \text{“}\mathsf{L}_3\!:\!I \mapsto \mathsf{L}_2\!:\!I \mapsto \mathsf{L}_3\!:\!(I+1) \text{ where } I \geq 1\text{”}
\end{array}
$$

Consider these properties as finite trace descriptors that can be "composed" with themselves. (We are informal for the moment.)

Outline

- **introduction: why loops and compositionality are difficult**
- **limitations of state-based analyses and the intuition of trace-based analyses**
- transfer transition systems: the vehicle of trace-based analysis
- some example applications:
  - software pipelining (a code transformation on loops)
  - loop-invariant removal
  - a compositional alias analysis

Transition systems and small-step semantics

A *transition system* is a pair $(\mathit{Config}, \delta)$.
Config: the set of configurations.
$\delta \subseteq \mathit{Config} \times \mathit{Config}$: the transitions ($c\,\delta\,c'$ means $(c, c') \in \delta$).

A *small-step semantics* is a transition system $(\mathit{Label} \times \mathit{State}, \longmapsto)$ satisfying the following condition, where $l\!:\!s$ denotes (l, s):

$$\forall l, l' \in \mathit{Label}.\ \forall s \in \mathit{State}.\ (l\!:\!s \longmapsto l'\!:\!s' \ \wedge\ l\!:\!s \longmapsto l'\!:\!s'') \Rightarrow s' = s''$$

This says that "all nondeterminism must go to different labels".

Transfer functions

A *transfer function* $\Delta \in \mathit{State} \rightharpoonup \mathit{State}$ is a partial function that describes how a state evolves during program execution.

Think of a transfer function as a "computation".

Given a small-step semantics $(\mathit{Label} \times \mathit{State}, \longmapsto)$, we can define a *single-step* transfer function for each pair of labels:

$$\Delta^{l}_{l'}(s) = \begin{cases} s' & \text{if } l\!:\!s \longmapsto l'\!:\!s' \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note: Many transitions represented by a single function.

Representing semantic traces with transfer functions

We have that

$$l_1\!:\!s_1 \longmapsto l_2\!:\!s_2 \longmapsto \cdots \longmapsto l_n\!:\!s_n$$

if and only if

$$(\Delta^{l_{i-1}}_{l_i} \circ \cdots \circ \Delta^{l_1}_{l_2})(s_1) = s_i \text{ for all } 2 \leq i \leq n.$$

This inspires the *derivation* of a new transition system—the *transfer transition system*—from the small-step semantics.

Transfer transition systems

Given a small-step semantics ($\mathit{Label} \times \mathit{State}, \longmapsto$), we define the *transfer transition system* ($\mathit{Label} \times (\mathit{State} \rightharpoonup \mathit{State}), \Longrightarrow$) as

$$l\!:\!\Delta \Longrightarrow l'\!:\!(\Delta^{l}_{l'} \circ \Delta) \text{ for all } \Delta \in \mathit{State} \rightharpoonup \mathit{State} \quad \text{(partial fns).}$$

Lemma 1 (correspondence) *If*

$$l_1\!:\!\Delta_1 \Longrightarrow \cdots \Longrightarrow l_n\!:\!\Delta_n \quad \textit{and} \quad \Delta_1(s) = s_1$$

then

$$l_1\!:\!s_1 \longmapsto \cdots \longmapsto l_n\!:\!s_n \quad \textit{iff} \quad (\forall 2 \leq i \leq n.\ \Delta_i(s) = s_i).$$

Multiple semantic traces via a single transfer trace

A single transfer trace

$$l_1\!:\!\Delta_1 \Longrightarrow \cdots \Longrightarrow l_n\!:\!\Delta_n$$

can represent multiple semantic traces

$$\begin{array}{lcl}
l_1\!:\!s_1 \longmapsto \cdots \longmapsto l_n\!:\!s_n & = & l_1\!:\!\Delta_1(s) \longmapsto \cdots \longmapsto l_n\!:\!\Delta_n(s) \\
l_1\!:\!s'_1 \longmapsto \cdots \longmapsto l_n\!:\!s'_n & = & l_1\!:\!\Delta_1(s') \longmapsto \cdots \longmapsto l_n\!:\!\Delta_n(s') \\
l_1\!:\!s''_1 \longmapsto \cdots \longmapsto l_n\!:\!s''_n & = & l_1\!:\!\Delta_1(s'') \longmapsto \cdots \longmapsto l_n\!:\!\Delta_n(s'') \\
\vdots & & \vdots
\end{array}$$

where $\Delta_1(s) = s_1, \quad \Delta_1(s') = s'_1, \quad \Delta_1(s'') = s''_1, \quad \ldots.$

The utility of finite transfer traces

A trace property usually includes an *infinite* set of *infinite* traces:

$$\begin{array}{l}
\mathcal{T}(\mathsf{L}_2) \supseteq \{\mathsf{L}_2\!:\!0 \mapsto \mathsf{L}_3\!:\!1 \mapsto \mathsf{L}_2\!:\!1 \mapsto \mathsf{L}_3\!:\!2 \mapsto \mathsf{L}_2\!:\!2 \mapsto \cdots, \\
\qquad\qquad \mathsf{L}_2\!:\!1 \mapsto \mathsf{L}_3\!:\!2 \mapsto \mathsf{L}_2\!:\!2 \mapsto \mathsf{L}_3\!:\!3 \mapsto \mathsf{L}_2\!:\!3 \mapsto \cdots, \ldots\}
\end{array}$$

Those traces are **composed** from an infinite collection of finite traces

$$\begin{array}{l}
\{\mathsf{L}_2\!:\!0 \mapsto \mathsf{L}_3\!:\!1 \mapsto \mathsf{L}_2\!:\!1, \\
\ \mathsf{L}_2\!:\!1 \mapsto \mathsf{L}_3\!:\!2 \mapsto \mathsf{L}_2\!:\!2, \\
\ \mathsf{L}_2\!:\!2 \mapsto \mathsf{L}_3\!:\!3 \mapsto \mathsf{L}_2\!:\!3, \ldots\}
\end{array}$$

which is represented by the single *finite* transfer trace

$$\mathsf{L}_2\!:\!\lambda i.i \Longrightarrow \mathsf{L}_3\!:\!\lambda i.(i+1) \Longrightarrow \mathsf{L}_2\!:\!\lambda i.(i+1) \quad [\text{defined when } i \geq 0].$$

Composing transfer traces together

The range $\mathrm{Rng}(\Delta) \subseteq \mathit{State}$ is the "output state property" of Δ (states to which Δ might correspond).

The set $\{s \in \mathit{State} \mid \Delta(s) = s\}$ is the "input state property" of Δ (states on which Δ performs no effective computation).

Suppose the output property of a trace ending with Δ is stronger than the input property of a trace beginning with Δ'.

Then Δ (the net computation of the first trace) may be composed with the transfer functions in the second trace to form a longer trace.

Composing transfer traces together

Lemma 2 (composition) *If*

$$l_1 : \Delta_1 \Longrightarrow \cdots \Longrightarrow l_n : \Delta_n \quad \textit{and} \quad l'_1 : \Delta'_1 \Longrightarrow \cdots \Longrightarrow l'_m : \Delta'_m$$

and

$$l_n = l'_1 \quad \textit{and} \quad \Delta'_1(s) = s \textit{ for all } s \in \mathsf{Rng}(\Delta_n),$$

then

$$\begin{array}{l} l_1 : \Delta_1 \Longrightarrow \cdots \Longrightarrow l_n : \Delta_n \Longrightarrow \\ \quad \Longrightarrow l'_2 : (\Delta'_2 \circ \Delta_n) \Longrightarrow \cdots \Longrightarrow l'_m : (\Delta'_m \circ \Delta_n). \end{array}$$

Composing two loop iterations in our example

The transfer trace

$$\mathsf{L}_2 : \lambda i.i \Longrightarrow \mathsf{L}_3 : \lambda i.(i+1) \Longrightarrow \mathsf{L}_2 : \lambda i.(i+1) \qquad [i \geq 0]$$

representing the loop body can be composed with itself

$$\mathsf{L}_2 : \lambda i.i \Longrightarrow \mathsf{L}_3 : \lambda i.(i+1) \Longrightarrow \mathsf{L}_2 : \lambda i.(i+1) \qquad [i \geq 0]$$

to yield a transfer trace representing two iterations

$$\begin{array}{l} \mathsf{L}_2 : \lambda i.i \Longrightarrow \mathsf{L}_3 : \lambda i.(i+1) \Longrightarrow \mathsf{L}_2 : \lambda i.(i+1) \Longrightarrow \\ \qquad\qquad \mathsf{L}_3 : \lambda i.(i+2) \Longrightarrow \mathsf{L}_2 : \lambda i.(i+2) \qquad [i \geq 0]. \end{array}$$

Intuitively, this correpsonds to *loop unrolling*.

All loop traces via iterated composition

Corollary 1 (loop) *If*

$$l_1 : \Delta_1 \Longrightarrow \cdots \Longrightarrow l_n : \Delta_n$$

and

$$l_1 = l_n \quad \textit{and} \quad \Delta_1(s) = s \textit{ for all } s \in \mathsf{Rng}(\Delta_n),$$

then, for all $k \geq 1$,

$$\begin{array}{l} l_1 : \Delta_1 \Longrightarrow l_2 : \Delta_2 \Longrightarrow \cdots \Longrightarrow l_n : \Delta_n \Longrightarrow \\ \qquad l_2 : (\Delta_2 \circ \Delta_n) \Longrightarrow \cdots \Longrightarrow l_n : (\Delta_n \circ \Delta_n) \Longrightarrow \\ \qquad l_2 : (\Delta_2 \circ {\Delta_n}^{(2)}) \Longrightarrow \cdots \Longrightarrow l_n : (\Delta_n \circ {\Delta_n}^{(2)}) \Longrightarrow \\ \qquad\qquad \vdots \\ \qquad l_2 : (\Delta_2 \circ {\Delta_n}^{(k)}) \Longrightarrow \cdots \Longrightarrow l_n : (\Delta_n \circ {\Delta_n}^{(k)}). \end{array}$$

The loop traces of our example

The loop-body transfer trace

$$\mathsf{L}_2 : \lambda i.i \Longrightarrow \mathsf{L}_3 : \lambda i.(i+1) \Longrightarrow \mathsf{L}_2 : \lambda i.(i+1) \qquad [i \geq 0]$$

can be composed with itself k times to yield

$$\begin{array}{l} \mathsf{L}_2 : \lambda i.i \Longrightarrow \mathsf{L}_3 : \lambda i.(i+1) \Longrightarrow \mathsf{L}_2 : \lambda i.(i+1) \Longrightarrow \\ \qquad \mathsf{L}_3 : \lambda i.(i+2) \Longrightarrow \mathsf{L}_2 : \lambda i.(i+2) \Longrightarrow \\ \qquad \mathsf{L}_3 : \lambda i.(i+3) \Longrightarrow \mathsf{L}_2 : \lambda i.(i+3) \Longrightarrow \\ \qquad\qquad \vdots \\ \qquad \mathsf{L}_3 : \lambda i.(i+k) \Longrightarrow \mathsf{L}_2 : \lambda i.(i+k) \qquad [i \geq 0]. \end{array}$$

Closed form of kth composition is not always computable, but for many applications this is not needed, as we are about to see.

Outline

- **introduction: why loops and compositionality are difficult**
- **limitations of state-based analyses and the intuition of trace-based analyses**
- **transfer transition systems: the vehicle of trace-based analysis**
- some example applications:
  - software pipelining (a code transformation on loops)
  - loop-invariant removal
  - a compositional alias analysis

Modeling loop code with $(\Delta_{\text{entry}}, \Delta_{\text{loop}}, \Delta_{\text{exit}})$

$$
\begin{array}{lll}
\text{L}_1 & : & \ldots \\
\vdots & & \vdots \\
\text{L}_2 & : & \textbf{if } e \textbf{ then} \\
\vdots & & \quad \vdots \\
\text{L}_3 & : & \quad \textbf{goto } \text{L}_2 \\
\vdots & & \vdots \\
\text{L}_4 & : & \ldots
\end{array}
$$

$$\underbrace{\text{L}_1 \cdots}_{\text{entry}} \underbrace{\text{L}_2 \cdots \text{L}_3}_{\text{loop}} \underbrace{\text{L}_2 \cdots \text{L}_3}_{\text{loop}} \cdots \underbrace{\text{L}_2 \cdots \text{L}_3}_{\text{loop}} \underbrace{\text{L}_2 \cdots \text{L}_4}_{\text{exit}}$$

$$
\begin{array}{l}
\text{L}_1 : \lambda s.s \Longrightarrow \cdots \Longrightarrow \text{L}_2 : \Delta_{\text{entry}} \\
\text{L}_2 : \lambda s.s \Longrightarrow \cdots \Longrightarrow \text{L}_2 : \Delta_{\text{loop}} \\
\text{L}_2 : \lambda s.s \Longrightarrow \cdots \Longrightarrow \text{L}_4 : \Delta_{\text{exit}}
\end{array}
$$

The observational behavior μ maps input states to output states:

$$
\begin{array}{rcll}
\mu(s) & = & \{s' \mid \text{L}_1 : s \longmapsto \cdots \longmapsto \text{L}_4 : s'\} & \\
& = & \{(\Delta_{\text{exit}} \circ \Delta_{\text{loop}}{}^{(k)} \circ \Delta_{\text{entry}})(s) \mid k \geq 0\} & \text{(from lemmas)}
\end{array}
$$

Two code fragments with the same μ can be interchanged.

Application: software pipelining

Software pipelining is a program transformation that maximizes instruction-level parallelism for VLIW and superscalar architectures.

$$
\begin{array}{lll}
\text{L}_1 & : & i \leftarrow i+1 \\
\text{L}_2 & : & j \leftarrow i+h \\
\text{L}_3 & : & k \leftarrow i+g \\
\text{L}_4 & : & l \leftarrow j+1 \\
\text{L}_5 & : & test \leftarrow i < n \\
\text{L}_6 & : & \textbf{if } test \textbf{ then} \\
\text{L}_7 & : & \quad \textbf{goto } \text{L}_1 \\
& & \textbf{endif} \\
\text{L}_8 & : & \ldots
\end{array}
\quad \Rightarrow \quad
\begin{array}{lll}
\text{L}_1 & : & i \leftarrow i+1 \\
\text{L}_2 & : & j;\ k;\ test \leftarrow i+h;\ i+g;\ i<n \\
\text{L}_3 & : & \textbf{if } test \textbf{ then} \\
\text{L}_4 & : & \quad l;\ i \leftarrow j+1;\ i+1 \\
\text{L}_5 & : & \quad \textbf{goto } \text{L}_2 \\
& & \textbf{endif} \\
\text{L}_6 & : & l \leftarrow j+1 \\
\text{L}_7 & : & \ldots
\end{array}
$$

Note that the increment $i \leftarrow i+1$ has "slid" one iteration.

Properties of function composition can reason about observational equivalence of loops. Here, Δ represents the computation that "slides" between iterations in software pipelining ($i \leftarrow i+1$ in the example).

Theorem 1 *If two code fragments are modeled by* $(\Delta_{\text{entry}}, \Delta_{\text{loop}}, \Delta_{\text{exit}})$ *and* $(\Delta'_{\text{entry}}, \Delta'_{\text{loop}}, \Delta'_{\text{exit}})$ *and there exists a* Δ *such that*

$$
\begin{array}{rcl}
\Delta \circ \Delta_{\text{entry}} & = & \Delta'_{\text{entry}} \\
\Delta \circ \Delta_{\text{loop}} & = & \Delta'_{\text{loop}} \circ \Delta \\
\Delta_{\text{exit}} & = & \Delta'_{\text{exit}} \circ \Delta
\end{array}
$$

(unprimed: original)
(primed: optimized)

then the two code fragments are observationally equivalent.

Proof: $\forall k.\ \Delta_{\text{entry}} \circ \Delta_{\text{loop}}{}^{(k)} \circ \Delta_{\text{exit}} = \Delta'_{\text{entry}} \circ \Delta'_{\text{loop}}{}^{(k)} \circ \Delta'_{\text{exit}}$. ■

Application: loop-invariant removal

Software pipelining becomes loop-invariant removal by changing the axioms. This time, Δ represents the computation factored out.

Theorem 2 *If two code fragments are modeled by* $(\Delta_{\mathrm{entry}}, \Delta_{\mathrm{loop}}, \Delta_{\mathrm{exit}})$ *and* $(\Delta'_{\mathrm{entry}}, \Delta'_{\mathrm{loop}}, \Delta'_{\mathrm{exit}})$ *and there exists a* Δ *such that*

$$\begin{aligned} \Delta \circ \Delta_{\mathrm{entry}} &= \Delta'_{\mathrm{entry}} \\ \Delta_{\mathrm{loop}} &= \Delta'_{\mathrm{loop}} \circ \Delta \\ \Delta \circ \Delta'_{\mathrm{loop}} &= \Delta'_{\mathrm{loop}} \\ \Delta_{\mathrm{exit}} &= \Delta'_{\mathrm{exit}} \circ \Delta \end{aligned}$$

(unprimed: original)
(primed: optimized)

then the two code fragments are observationally equivalent.

Proof: (Uses a different lemma about function composition.) ■

Taking stock

We have developed a framework for reasoning about traces and observational equivalence, particularly of loops, that is *independent* of the language or semantics.

To prove these code transformations *automatically*, for a particular language we need:

- A finite representation of transfer functions for a small-step semantics of the language.
- A computable composition operation on that representation.
- A way of approximating transfer function equality.

Application: compositional alias analysis

What does the following code fragment do?

$$\begin{aligned} \mathsf{L}_1 &: b.\mathsf{P}.\mathsf{P} \leftarrow a.\mathsf{P} \\ \mathsf{L}_2 &: a.\mathsf{P} \leftarrow b.\mathsf{P} \\ \mathsf{L}_3 &: b.\mathsf{P} \leftarrow a \\ \mathsf{L}_4 &: \ldots \end{aligned}$$

$$\mathsf{L}_1 : \lambda s.s \Longrightarrow \mathsf{L}_2 : \Delta^{\mathsf{L}_1}_{\mathsf{L}_2} \Longrightarrow \mathsf{L}_3 : \Delta^{\mathsf{L}_2}_{\mathsf{L}_3} \circ \Delta^{\mathsf{L}_1}_{\mathsf{L}_2} \Longrightarrow \mathsf{L}_4 : \Delta^{\mathsf{L}_3}_{\mathsf{L}_4} \circ \Delta^{\mathsf{L}_2}_{\mathsf{L}_3} \circ \Delta^{\mathsf{L}_1}_{\mathsf{L}_2}$$

The behavior is given by $\Delta^{\mathsf{L}_3}_{\mathsf{L}_4} \circ \Delta^{\mathsf{L}_2}_{\mathsf{L}_3} \circ \Delta^{\mathsf{L}_1}_{\mathsf{L}_2}$.

Represent transfer functions as *conditional parallel assignments*.

The composition operation determines all possible ways the two computations can "interfere" and generates the minimal sets of *sharing conditions* specifying each possibility (details in paper).

Toward "reusable" transfer functions

In many languages, the state can be coded with a *graph*.

Nodes are *values*, outgoing edges are *l-values*, and a state maps each outgoing edge to its target node.

$$\begin{aligned} s &::= \{lv_1 \mapsto v_1, \ldots, lv_n \mapsto v_n\} \\ v &::= k \mid \phi && \text{values (constants, records)} \\ lv &::= x \mid \phi.f && \text{l-values (variables, record fields)} \end{aligned}$$

Our loop increment example from earlier:

$$\mathsf{L}_1 : s \longmapsto \mathsf{L}_2 : \{n \mapsto 0\} \longmapsto \mathsf{L}_3 : \{n \mapsto 1\} \longmapsto \mathsf{L}_2 : \{n \mapsto 1\} \longmapsto \mathsf{L}_3 : \{n \mapsto 2\} \cdots$$

Expressions denote values in a state s (via $\mathcal{E}[\![]\!]s$).
L-expressions denote l-values in a state s (via $\mathcal{LE}[\![]\!]s$).

$$\begin{array}{rcll} e & ::= & k \mid le \mid e\ op\ e' & \text{expressions} \\ le & ::= & x \mid le.f & \text{l-expressions (paths in the graph)} \end{array}$$

$$\begin{array}{rcl} \mathcal{E}[\![k]\!]s = k & \quad & \mathcal{LE}[\![x]\!]s = x \\ \mathcal{E}[\![le]\!]s = s(\mathcal{LE}[\![le]\!]s) & & \mathcal{LE}[\![le.f]\!]s = (\mathcal{E}[\![le]\!]s).f \\ \mathcal{E}[\![e\ op\ e']\!]s = (\mathcal{E}[\![e]\!]s)\ op\ (\mathcal{E}[\![e']\!]s) & & \end{array}$$

A transfer function is a set of conditional parallel edge assignments:

$$\begin{array}{rcll} \Delta & ::= & \{[\sigma_1 \mid C_1], \ldots, [\sigma_n \mid C_n]\} & \text{transfer functions} \\ \sigma & ::= & le_1; \ldots; le_n \leftarrow e_1; \ldots; e_n & \text{parallel edge assignments} \\ C & ::= & e_1, \ldots, e_n & \text{condition sets} \end{array}$$

Examples of transfer functions

$$\Delta \quad = \{[]\}$$
$$\Delta(s) = s$$

$$\Delta \quad = \{[x;\ y \leftarrow y + z;\ x \mid x > y], [z \leftarrow x - 1 \mid x < y]\}$$
$$\Delta(s) = \begin{cases} s[x \mapsto s(y) + s(z)][y \mapsto s(x)] & \text{if } s(x) > s(y) \\ s[z \mapsto s(x) - 1] & \text{if } s(x) < s(y) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\Delta \quad = \{[x;\ \mathbf{1}.\mathrm{CAR};\ \mathbf{1}.\mathrm{CDR} \leftarrow \mathbf{1};\ 42;\ 24]\}$$
$$\Delta(s) = s[x \mapsto \phi][\phi.\mathrm{CAR} \mapsto 42][\phi.\mathrm{CDR} \mapsto 24]$$
$$\text{where } \phi = \mathit{new}(s, 1)$$

$$\Delta \quad = \{[x.\mathrm{CDR}.\mathrm{CDR} \leftarrow x]\}$$
$$\Delta(s) = s[s(x.\mathrm{CDR}).\mathrm{CDR} \mapsto x]$$

Composing transfer functions

Consider the following code fragment:

$$\begin{array}{lcl} \mathrm{L}_1 & : & a.\mathrm{P} \leftarrow b \\ \mathrm{L}_2 & : & c.\mathrm{P}.\mathrm{P} \leftarrow d \\ \mathrm{L}_3 & : & \ldots \end{array}$$

The semantics defines the following single-step transfer functions:

$$\Delta^{\mathrm{L}_1}_{\mathrm{L}_2} = \{[a.\mathrm{P} \leftarrow b]\} \qquad \Delta^{\mathrm{L}_2}_{\mathrm{L}_3} = \{[c.\mathrm{P}.\mathrm{P} \leftarrow d]\}$$

So the transfer transition trace of this fragment is:

$$\mathrm{L}_1 : \lambda s.s \Longrightarrow \mathrm{L}_2 : \Delta^{\mathrm{L}_1}_{\mathrm{L}_2} \Longrightarrow \mathrm{L}_3 : \Delta^{\mathrm{L}_3}_{\mathrm{L}_2} \circ \Delta^{\mathrm{L}_2}_{\mathrm{L}_1}$$

How do we represent the transfer function $\Delta^{\mathrm{L}_3}_{\mathrm{L}_2} \circ \Delta^{\mathrm{L}_2}_{\mathrm{L}_1}$?

One possibility is that there are no relevant initial data aliases:

$$[a.\mathrm{P};\ c.\mathrm{P}.\mathrm{P} \leftarrow b;\ d \mid a \neq c, a \neq c.\mathrm{P}]$$

But suppose that $s(a) = s(b) = s(c)$ for some initial state s:

$$[a.\mathrm{P} \leftarrow d \mid a = c, a = b]$$

The composition $\Delta^{\mathrm{L}_3}_{\mathrm{L}_2} \circ \Delta^{\mathrm{L}_2}_{\mathrm{L}_1}$ is a set of these two and two others.

Application: compositional alias analysis

What does the following code fragment do?

$$
\begin{array}{lll}
\mathsf{L}_1 & : & b.\mathsf{P}.\mathsf{P} \leftarrow a.\mathsf{P} \\
\mathsf{L}_2 & : & a.\mathsf{P} \leftarrow b.\mathsf{P} \\
\mathsf{L}_3 & : & b.\mathsf{P} \leftarrow a \\
\mathsf{L}_4 & : & \ldots
\end{array}
$$

Answer: compute the transfer trace

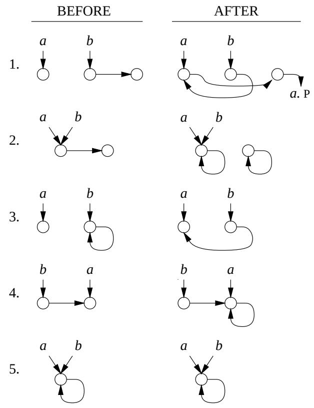
$$\mathsf{L}_1 : \lambda s.s \Longrightarrow \mathsf{L}_2 : \Delta^{\mathsf{L}_1}_{\mathsf{L}_2} \Longrightarrow \mathsf{L}_3 : \Delta^{\mathsf{L}_2}_{\mathsf{L}_3} \circ \Delta^{\mathsf{L}_1}_{\mathsf{L}_2} \Longrightarrow \mathsf{L}_4 : \Delta^{\mathsf{L}_3}_{\mathsf{L}_4} \circ \Delta^{\mathsf{L}_2}_{\mathsf{L}_3} \circ \Delta^{\mathsf{L}_1}_{\mathsf{L}_2}$$

and then examine the transfer function $\Delta^{\mathsf{L}_3}_{\mathsf{L}_4} \circ \Delta^{\mathsf{L}_2}_{\mathsf{L}_3} \circ \Delta^{\mathsf{L}_1}_{\mathsf{L}_2}$.

Automatic verification: first line gives minimal correctness conditions.

Static debugging: four types of errors, human-readable output.

$$
\begin{array}{l}
\{[b.\mathsf{P}.\mathsf{P};\ a.\mathsf{P};\ b.\mathsf{P} \leftarrow a.\mathsf{P};\ b.\mathsf{P};\ a \mid \\
\quad a \neq b.\mathsf{P}, b \neq b.\mathsf{P}, b \neq a], \\
\ [b.\mathsf{P}.\mathsf{P};\ a.\mathsf{P} \leftarrow a.\mathsf{P};\ a \mid \\
\quad a \neq b.\mathsf{P}, b \neq b.\mathsf{P}, b = a], \\
\ [b.\mathsf{P} \leftarrow a \mid a \neq b.\mathsf{P}, b = b.\mathsf{P}], \\
\ [a.\mathsf{P} \leftarrow a \mid a = b.\mathsf{P}, b \neq b.\mathsf{P}], \\
\ [b.\mathsf{P} \leftarrow a \mid a = b.\mathsf{P}, b = b.\mathsf{P}]\}
\end{array}
$$



Approximating equality of transfer functions

Recall the axioms in the software pipelining application:

$$
\begin{array}{rcl}
\Delta \circ \Delta_{\mathsf{entry}} & = & \Delta'_{\mathsf{entry}} \\
\Delta \circ \Delta_{\mathsf{loop}} & = & \Delta'_{\mathsf{loop}} \circ \Delta \\
\Delta_{\mathsf{exit}} & = & \Delta'_{\mathsf{exit}} \circ \Delta
\end{array}
$$

Structural equality on transfer function representations can be sufficient for many tasks, including this one and loop-invariant removal.

Various algebras may be possible, depending on the specific set of primitive operations.

Conclusion

Trace-based analysis is founded on *transfer transition systems*:

- Addresses code transformation and optimization problems involving loops that are currently done by *ad hoc* means.
- A compositional alias analysis that relates code behavior with initial aliases: needed by PLDI community (C compilers, *etc*).
- Symbolic representation and compositionality is useful for static debugging and verification.

Future work:

- Abstract interpretation of transfer transition systems.
- Syntactic discipline (natural operational semantics).
- Implement for a real language.