# Trace-based Program Analysis

Christopher Colby[†] Peter Lee

July 1995
CMU-CS-95-179

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU-CS-FOX-95-04

**Abstract**

We present *trace-based program analysis*, a semantics-based framework for statically analyzing and transforming programs with loops, assignments, and nested record structures. Trace-based analyses are based on *transfer transition systems*, which define the small-step operational semantics of programming languages. Intuitively, transfer transition systems provide direct support for reasoning about the possible execution traces of a program, instead of just individual program states. The traces in a transfer transition system have many uses, including the finite representation of all possible terminating executions of a loop. Also, traces may be systematically "pieced together", thus allowing the composition of separately analyzed program fragments. The utility of the approach is demonstrated by showing three applications: software pipelining, loop-invariant removal, and data alias detection.

[†]Work performed while on leave at École Polytechnique, France.

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050, and in part by the National Science Foundation under PYI grant #CCR-9057567.

**Keywords:** semantics-based program analysis, abstract interpretation, operational semantics, program transformation

# 1 Introduction

In this paper, we consider semantics-based program analysis for the purposes of static optimization, transformation, program verification, and debugging. We have two goals. First, to capture precise information about the run-time behavior of programs that contain loops, either from the use of looping constructs such as **goto** or from recursive functions. And second, to compose the analyses of separate code fragments to achieve the analysis of a larger piece of code. We propose a general framework of *trace-based analysis* to address these problems, and we present a worked case study along with some sample applications that illustrate the solution of our goals.

Loops have long been the bane of semantics-based program analysis. In a programming language with a looping construct, the semantic meaning is usually modeled by infinite structures. For example, denotational semantics [28] uses limits of infinite chains to model the meaning of recursive functions. In standard formulations of structural operational semantics [26] and small-step operational semantics [15], the behavior of loops is modeled by unbounded sequences of transitions. An analysis based on any of these semantic models is thus forced to develop some method for reasoning finitely about these (potentially) infinite objects. Usually, some form of *approximation* or *abstraction* is used and, as we shall argue in the next section, the typical methods often incur a substantial loss of precision. Determining precise information about loops, however, is often of critical importance for static optimization of code. One of the most striking examples is software pipelining [2, 22], a strategy for statically transforming the structure of a loop in order to take advantage of the potential instruction-level parallelism both within a single iteration and between adjacent iterations. Software pipelining algorithms are complex and specialized, but we demonstrate with a worked-through example that trace-based analysis provides a general semantics-based formulation for a form of software pipelining. In fact, with nothing more than the alteration of a three very simple equations, the same technique becomes a strategy for factoring out loop invariants.

Another common difficulty for semantics-based program analyses is that they are rarely compositional. A well-known tradeoff between denotational semantics and small-step operational semantics is that denotational models are compositional, but usually abstract away from *how* the result of the program was computed, while small-step operational models expose useful details of the computation process itself, but usually lack the compositionality properties provided by denotational models. Some special cases of program analyses, such as strictness analysis [24, 29], can be formalized quite elegantly and compositionally using a denotational model, and a general framework for denotational-semantics-based analysis has been developed in [25]. However, many program analyses are concerned with details of the actual computation process itself and thus *must* be based on an operational model, making it difficult to analyze a program compositionally. Again, this is a serious problem in practice. Consider for example the analysis of data aliases in a language with assignment. A fragment of code (basic block, procedure, *etc.*) that performs data structure manipulation and assignment may have several distinctly different behaviors depending on the data aliases occurring at the beginning of its execution. Furthermore, each of these behaviors may produce different aliases when execution leaves that fragment and enters another. For this purpose, it is important to have a compositional analysis. We give an example of how the same trace-based analysis that is used for software pipelining and loop optimizations also provides a compositional alias analysis. In fact, the output of the analysis can uncover surprising possibilities for the behavior of even simple imperative code and has clear applications for debugging purposes and program verification as well as the traditional uses of alias information in optimizing compilers.

Trace-based analysis is based on a framework for describing the operational semantics of programs that we call *transfer transition systems.* This approach to describing the semantics of programming languages can be derived from, and maintains all the computational detail of, the original small-step semantics of the language, and yet provides some of the compositionality properties enjoyed by denotational semantics. The underlying motivation for transfer transition systems is to aid reasoning about the possible execution traces of the program instead of simply the possible states at each program point. Intuitively, a *trace* is a sequence of steps in the small-step operational semantics of the program. But a single trace in the transfer transition system can represent precisely a large or even infinite set of traces in the small-step operational semantics, thereby making it easier to reason about traces in the transfer transition system. In fact, a transfer transition trace of a loop body can represent all possible terminating executions of the loop, which is ideal information for reasoning about its behavior. Also, transfer transition traces may be systematically "pieced together," thus forming the basis for compositional analyses.

We begin by discussing our original motivations and explain why previous ideas such as "polyvariant" analyses fall short of solving the problems. Then, we present our trace-based analysis, starting with an informal explanation designed to provide the right intuitions, followed by a formal presentation of the analysis framework. As a worked case study, we take a small language with (parallel) assignment, loops (*via* **goto**), and data structures (nested records). Though simple, the language is expressive enough to serve as a core language for many realistic programming languages. Finally, we show applications of the trace-based analysis for this language, including a software pipelining transformation as well as analyses for removal of loop invariants and detection of aliases.

## 2 Limitations of State-based Program Analyses

It is perhaps surprising that loops are not handled entirely satisfactorily by current semantics-based methods for analyzing programs. The basic problem can be illustrated informally by some simple examples. Consider first the following program fragment, written in a kind of structured assembly language:

$$
(1) \qquad
\begin{array}{rl}
 & \vdots \\
 & b \leftarrow \text{TRUE} \\
\text{L}_X: & \textbf{goto}\ \text{L}_C \\
 & \vdots \\
 & b \leftarrow \text{FALSE} \\
\text{L}_Y: & \textbf{goto}\ \text{L}_C \\
 & \vdots \\
\text{L}_C: & \textbf{if}\ b\ \textbf{then}\ \ldots\ \textbf{endif} \\
 & \vdots
\end{array}
$$

Suppose we would like to predict the set of possible values that the variable $b$ can have at each point in the program. In the customary formulation, a program analyzer would produce a function $\mathcal{S}$ that maps program points to the properties of interest[1]:

$$\mathcal{S} \in \mathit{Label} \rightarrow \mathit{Property}$$

[1]Although this is a rather naïve formulation, an examination of the current literature on semantics-based program analysis reveals that a large number (perhaps the majority) of existing analyses take essentially this form.

Here, *Label* is the set of program labels and *Property* is the set of properties. Often, a considerable amount of effort goes into the design of the property set, with the requirement that its elements have finite descriptions so that the analyzer can terminate. Examples of the kinds of properties that are useful in practice include:

- The pairs of data objects that may or must be aliases at a given label [3, 14].
- Grammars giving all of the possible shapes of structured data at a given label [12, 20].
- Linear relationships amongst integer-valued variables (*e.g.*, $x = y - z$) at a given label [16, 21].

Typically, the construction of an analyzer begins with a so-called "concrete" or "collecting" semantics [11, 18] in which programs are assigned meanings of the form

$$\mathcal{C} \in \mathit{Label} \rightarrow \wp(\mathit{State})$$

where $\wp(\mathit{State})$ is the powerset of program states. Then, from the collecting semantics, a program analyzer (that is, an algorithm for computing $\mathcal{S}$ from the program text) is derived in such a way that each element of *Property* corresponds to a set of states, in particular, the set of states for which the property is true. Because of this relationship between properties and sets of states, we refer to program analyzers like $\mathcal{S}$ as *state-based analyzers* and their properties as *state properties.* Notationally, we will usually refer to the set of state properties as $\mathit{Property}[\mathit{State}]$ and thus write the functionality of $\mathcal{S}$ as

$$\mathcal{S} \in \mathit{Label} \rightarrow \mathit{Property}[\mathit{State}]$$

In order to prove that an analysis is sound, it must be shown that, for any program, $\mathcal{C}$ and $\mathcal{S}$ will obey a certain relationship. Also, the analysis (*i.e.*, the computation of $\mathcal{S}$) and the $\mathcal{S}$ function itself must be computable and, for practical reasons, efficient. *Abstract interpretation* [8] is a comprehensive mathematical theory for devising and reasoning about such relationships and decidability problems, and in addition it provides a semantics-based framework for devising the analysis algorithms.

Returning now to Program 1, it is well known that with this simple formulation the two possible values of $b$ at label $\mathrm{L}_C$ are unavoidably "folded" together, hence resulting in a loss of information. In particular, we lose the fact that the value of $b$ at label $\mathrm{L}_C$ is TRUE if the flow of control entered $\mathrm{L}_C$ from $\mathrm{L}_X$, and FALSE if from $\mathrm{L}_Y$. In programs with functions, a similar loss of information can occur when a function is called from several different points in the program. This is particularly problematic for recursive functions.

A similar situation arises with loops. For example, consider the following (rather contrived) program:

$$\begin{array}{ll} & \mathrm{L}_1 : n \leftarrow 1 \\ (2) & \mathrm{L}_2 : n \leftarrow n+1 \\ & \mathrm{L}_3 : \textbf{goto}\ \mathrm{L}_2 \end{array}$$

Semantically, we can view the program execution as stepping through a transition system, with each step moving from some label-state pair, $l\!:\!s$, to a new label-state pair, $l'\!:\!s'$. (We will formalize this notion of language semantics in the next section.) We use the following notation for such transitions:

$$l\!:\!s \longmapsto l'\!:\!s'$$

Then, the execution of the program goes as follows:

$$\mathrm{L}_1\!:\!s_1 \longmapsto \mathrm{L}_2\!:\!s_2 \longmapsto \mathrm{L}_3\!:\!s_3 \longmapsto \mathrm{L}_2\!:\!s_4 \longmapsto \mathrm{L}_3\!:\!s_5 \longmapsto \cdots$$

This program has initial state $\text{L}_1 : s_1$ and immediately steps into a loop that alternates between labels $\text{L}_2$ and $\text{L}_3$. Note that the states are continually changing throughout the loop.

The state properties are essentially the set of objects for which the property is true. Any state-based analysis will thus determine properties

$$\begin{aligned}\mathcal{S}(\text{L}_1) &\supseteq \{s_1\}\\ \mathcal{S}(\text{L}_2) &\supseteq \{s_2, s_4, \ldots\}\\ \mathcal{S}(\text{L}_3) &\supseteq \{s_3, s_5, \ldots\}\end{aligned}$$

Of course, the practical question is what, exactly, are $\mathcal{S}(\text{L}_1)$, $\mathcal{S}(\text{L}_2)$, and $\mathcal{S}(\text{L}_3)$. The smaller they are, the more precise the properties, but in general the occurrences of $\supseteq$ cannot be replaced with $=$ (up to isomorphism) because those exact sets may not be computable.[2]

In order to address this problem, there has been much recent work on refining the notion of "program point" to something more descriptive of a run-time point of execution than just the simple textual label at that point. The basic idea is to determine properties of the form

$$\mathcal{S} \in \mathit{Point} \to \mathit{Property}[\mathit{State}]$$

where

$$\mathit{Point} = \mathit{Label} \times \mathit{Occurrence}$$

In this improved formulation, we are given the possibility of partitioning the states that occur at a given label into a finite number of *occurrences* and giving each such class its own property. In the literature, this idea is sometimes informally referred to as *polyvariance* [7, 23] and is also closely related to the notion of *cloning* in dataflow analysis. A common form of polyvariance involves partitioning certain blocks of code, such as function bodies or loop bodies, by the label that preceded their entry.

For Program 1, we might take $\mathit{Occurrence} = \mathit{Label}$ and use occurrences to keep track of the label that preceded the current one. This would allow the analysis to produce a result that maps $(\text{L}_C, \text{L}_X)$ to a property that indicates that $b$ must be TRUE, and $(\text{L}_C, \text{L}_Y)$ to a property indicating that $b$ must be FALSE.

For Program 2, we might use $\mathit{Occurrence} = \{\text{EVEN}, \text{ODD}\}$ to differentiate between the odd and even iterations of a loop. This would allow an analysis to infer that the value of the variable $n$ in the program is odd during odd iterations of the loop, and even during even iterations.

Such "polyvariant" analyses can be quite effective, especially for analyzing programs with non-recursive functions. In these cases, functions might be called from several syntactic points in the program, each of which might cause very different run-time behaviors worth describing separately. Also appealing is the fact that it is straightforward to extend this idea and keep track of, say, the last *two* calling points instead of just the last one, or in general the last $k$ for some finite $k$ [19, 27]. Indeed, there is quite a bit of room for creativity here, as the only real requirement on the set *Point* is that its elements must have finite descriptions. Some recent proposals have used fairly complex mechanisms such as *procedure strings* [17].

Unfortunately, the idea of polyvariance, and indeed any formulation of program analysis that involves only a refinement of the notion of "program point," is fundamentally limited. Consider

[2] In general, we may be satisfied with easier questions than membership (*i.e.*, the question of whether a given state has the property in question or not), but any interesting questions are uncomputable in general.

the following program that uses Euclid's algorithm to compute the greatest common divisor:

$$
\begin{array}{ll}
 & \mathrm{L}_1 : \textbf{if } b = 0 \textbf{ then } \mathrm{L}_{\mathrm{exit}} \textbf{ else } \mathrm{L}_2 \\
(3) & \mathrm{L}_2 : a, b \leftarrow b, a \textbf{ mod } b \\
 & \mathrm{L}_3 : \textbf{if } b = 0 \textbf{ then } \mathrm{L}_{\mathrm{exit}} \textbf{ else } \mathrm{L}_2
\end{array}
$$

Note that we use a parallel assignment operator in order to simplify some of the examples later on in this paper.

At first glance it seems that we should be able to express the fact that the $b$ of one iteration is always equal to the $a$ of the next iteration (perhaps using a variation of the even/odd occurrences from above). This property might enable a compiler optimization that unrolls the loop once and compiles the second copy with the opposite register order, thereby cutting the number of assignments in half:

$$
\begin{array}{l}
\mathrm{L}_1 : \textbf{if } b = 0 \textbf{ then } \mathrm{L}_{\mathrm{exit}} \textbf{ else } \mathrm{L}_2 \\
\mathrm{L}_2 : a \leftarrow a \textbf{ mod } b \\
\mathrm{L}_3 : \textbf{if } a = 0 \textbf{ then } \mathrm{L}_{\mathrm{exit}} \textbf{ else } \mathrm{L}_4 \\
\mathrm{L}_4 : b \leftarrow b \textbf{ mod } a \\
\mathrm{L}_5 : \textbf{if } b = 0 \textbf{ then } \mathrm{L}_{\mathrm{exit}} \textbf{ else } \mathrm{L}_2
\end{array}
$$

However, we cannot find a simple or elegant expression of this fact, no matter how precise our notion of program point, because we need somehow to associate $s_2$ with $s_4$, $s_3$ with $s_5$, $s_4$ with $s_6$, *ad infinitum*, but the set *Point* of descriptions of program points must be finite. To attempt to use polyvariance, or any other technique that involves a more precise notion of run-time points, is extremely difficult (perhaps even impossible) because we need to distinguish among *infinite* sets of states with only finite descriptions of program points. Fundamentally, all of the states that occur at the same label are described by the same property. Thus, refining (or enlarging) the set of program points does little more than delay the inevitable folding of states.

In fact, this is a problem that comes up in almost any analyzer for a program with loops (or recursive functions) and is a common source of both complexity and imprecision in semantics-based program analyzers.

## 3 Transfer Transition Systems

In order to solve this problem, we abandon the approach of refining the notion of "program point" and instead refine the notion of "property." Specifically, we generalize from state properties to *trace properties*:

$$\mathcal{T} \in \mathit{Point} \to \mathit{Property}[\mathit{Trace}]$$

Traces will be formalized below, so we will begin with an informal explanation. One can think of a trace as a record of a possible execution history of a program, starting at some label and state. In constrast to a state-based analysis, which infers properties as sets of states, a trace-based analysis infers properties as sets of entire evaluation traces. So, for example, for the greatest-common divisor program of Example 3 we have the following properties:

$$
\begin{array}{rl}
\mathcal{T}(\mathrm{L}_1) \supseteq & \{\mathrm{L}_1 : s_1 \mapsto \mathrm{L}_2 : s_2 \mapsto \mathrm{L}_3 : s_3 \mapsto \cdots\} \\
\mathcal{T}(\mathrm{L}_2) \supseteq & \{\mathrm{L}_2 : s_2 \mapsto \mathrm{L}_3 : s_3 \mapsto \mathrm{L}_2 : s_4 \mapsto \cdots, \\
 & \quad \mathrm{L}_2 : s_4 \mapsto \mathrm{L}_3 : s_5 \mapsto \mathrm{L}_2 : s_6 \mapsto \cdots, \ldots\} \\
\mathcal{T}(\mathrm{L}_3) \supseteq & \{\mathrm{L}_3 : s_3 \mapsto \mathrm{L}_2 : s_4 \mapsto \mathrm{L}_3 : s_5 \mapsto \cdots, \\
 & \quad \mathrm{L}_3 : s_5 \mapsto \mathrm{L}_2 : s_6 \mapsto \mathrm{L}_3 : s_7 \mapsto \cdots, \ldots\}
\end{array}
$$

For a label $l$, the trace property $\mathcal{T}(l)$ is a superset of the set of traces that can begin at $l$. Assuming that states are pair of integers $[a, b]$ giving the values of variables $a$ and $b$, the variable-swapping property of the program at label $\mathrm{L}_2$ can now be expressed (finitely) as follows:

$$\{\mathrm{L}_2 : [a, b] \mapsto \mathrm{L}_3 : [b, a \bmod b] \mapsto \mathrm{L}_2 : [b, a \bmod b] \mid a, b \in \mathbb{Z}\}$$

In effect, this says that if we are at label $\mathrm{L}_2$, then the next time we reach label $\mathrm{L}_2$ the value of $a$ will be equal to the current value of $b$. (For the sake of simplicity we have avoided writing out the full trace property for $\mathcal{T}(\mathrm{L}_2)$, which would also specify the behavior of the loop exit.) Note that this is exactly the information we desire! Furthermore, this property gives us *exact* information about $a$, rather than an upper approximation of its value, so in effect we have been able to model the program's loop by a *finite* transition sequence, without any loss of information.

As we shall see, this notion of trace properties will allow us to construct very precise analyzers for programs with loops and complex data structures. To do this, we shall introduce the semantic framework of *transfer transition systems*, which in addition to formalizing the notion of execution traces will allow us to define program transformation rules for complex optimizations such as software pipelining and classical loop optimizations.

### 3.1 Notation

We now present a formal framework for trace-based analyses. The following notation is used. If $f$ is a function, $\mathrm{Rng}(f)$ denotes the range (co-domain) of $f$. If $f$ is partial, $\mathrm{Dom}(f)$ denotes the domain of $f$. The set of partial functions from $A$ to $B$ is written as $A \rightharpoonup B$. Pairs (*i.e.*, elements of a set $A \times B$) are written either as $(a, b)$ or, when denoting a configuration of a transition system, $a : b$.

### 3.2 Single-step transition systems

Our framework starts with a small-step operational semantics. In order to have a standard basis for this, we use transition systems.

**Definition 1** *A* transition system *is a pair $(Config, \delta)$ where Config is the set of configurations of the system and $\delta \subseteq Config \times Config$ is the transition relation. We write $c\delta c'$ to mean that $(c, c') \in \delta$, and we write $c_1\delta \cdots \delta c_n$ to mean that $c_i \delta c_{i+1}$ for all $1 \leq i < n$. We write $\delta^*$ for the reflexive transitive closure of $\delta$.*

Specifically, we are interested in transition systems of the form $(Label \times State, \longmapsto)$ that satisfy the following condition:

**Condition 1** *For all $l, l' \in Label$ and $s \in State$, $l : s \longmapsto l' : s'$ and $l : s \longmapsto l' : s''$ implies $s' = s''$.*

Note that this condition is much weaker than requiring that the system be deterministic. Any transition system will satisfy this condition after some simple local transformations at each point in the system that fails the condition. For example, if $l : s \longmapsto l' : s'$ and $l : s \longmapsto l' : s''$ but $s' \neq s''$, we can simply create a new label $l''$ and change configuration $l' : s''$ to $l'' : s''$. Essentially, Condition 1 states that from a configuration $l : s$ there can be many possible single steps, but all of the target configurations must have different labels.

### 3.3 Transfer functions and transfer transition systems

A *transfer function* $\Delta \in State \rightharpoonup State$ is a partial function that describes how a state evolves during program execution. In essence, a transfer function represents part of the computation of a program. So, if $\Delta(s) = s'$, then at some point the program will compute state $s'$ when starting from state $s$.

Given a transition system $(Label \times State, \longmapsto)$ that satisfies Condition 1, we can define a *single-step* transfer function for each pair of labels that captures exactly the possible single-step transitions between those labels:

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

Then,

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

if and only if

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

This inspires the definition of a new transition system from the original one in which states are compound transfer functions built up from compositions of single-step transfer functions.

**Definition 2** *Given a transition system $(Label \times State, \longmapsto)$ that satisfies Condition 1, we define the* transfer transition system $(Label \times (State \rightharpoonup State), \Longrightarrow)$ *as*

$$l\!:\!\Delta \Longrightarrow l'\!:\!(\Delta^{l}_{l'} \circ \Delta) \textit{ for all } \Delta \in State \rightharpoonup State.$$

Intuitively, if $\Delta(s) = s'$, then $l\!:\!\Delta$ means that it is possible to reach a configuration $l\!:\!s'$ when starting from state $s$. The definition then shows how single-step transfer functions may be composed to specify additional steps of a program's computation. Thus, there is a direct correspondence between traces in a transfer transition system and traces in the standard single-step semantics. This is stated formally in the following lemma:

**Lemma 1** *If $l_1\!:\!\Delta_1 \Longrightarrow \cdots \Longrightarrow l_n\!:\!\Delta_n$ and $\Delta_1(s) = s_1$, then $\Delta_i(s) = s_i$ iff $l_1\!:\!s_1 \longmapsto \cdots \longmapsto l_n\!:\!s_n$, where $1 < i \leq n$.*

**Corollary 1** *$l\!:\!\lambda s''.s \Longrightarrow l'\!:\!\lambda s''.s'$ iff $l\!:\!s \longmapsto l'\!:\!s'$.*

The corollary states that the transfer transitions between the constant transfer functions are isomorphic to the transitions in the original system. But more interesting is the case of non-constant transfer functions. For instance, suppose that

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

and

$$l_1\!:\!s'_1 \longmapsto \cdots \longmapsto l_n\!:\!s'_n$$

where $s_1 \neq s'_1$. By the above lemma, both of these sequences can be represented by the single transfer sequence

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

where there exist two states $s$ and $s'$ such that $\Delta_1(s) = s_1$ and $\Delta_1(s') = s'_1$. The original two transition sequences are then exactly

$$l_1\!:\!\Delta_1(s) \longmapsto \cdots \longmapsto l_n\!:\!\Delta_n(s)$$

and

$$l_1 : \Delta_1(s') \longmapsto \cdots \longmapsto l_n : \Delta_n(s').$$

It does not matter exactly what $s$ and $s'$ are, only that they "select" the two starting states $s_1$ and $s'_1$. In fact, this finite sequence of transfer functions might specify an arbitrary number of possible program behaviors, depending on the size of $\mathrm{Dom}(\Delta_1)$. This is the key point about transfer transition systems: the transfer functions in a sequence carry information about how the state is altered *relative to the first transfer function in the sequence.*

The relative nature of transfer functions allows two separate sequences of transfer transitions to be "composed" to create a third long sequence. This is accomplished by changing, in a systematic way, the transfer functions of the second sequence so that they describe how the state is changed relative to the beginning of the *first* sequence. Intuitively, given a transfer function $\Delta$, we can view its range $\mathrm{Rng}(\Delta) \subseteq \mathit{State}$ to be a boolean property satisfied by all states $s$ such that $l : \Delta$ might correspond to $l : s$ in some application of Lemma 1. If $\Delta(s) = s$, then $s$ already satisfies this property. Suppose $\Delta'(s) = s$ for all $s \in \mathrm{Rng}(\Delta)$. This intuitively means that the "output property" of a sequence ending with $\Delta$ is stronger than the "input property" of a sequence beginning with $\Delta'$, because $\Delta'$ may also be defined for some $s' \notin \mathrm{Rng}(\Delta)$. In this case, the two sequences may be composed to form a larger sequence. The following lemma specifies how the transfer functions of the second sequence must be modified in order to carry out the composition.

**Lemma 2** *If $l_1 : \Delta_1 \Longrightarrow \cdots \Longrightarrow l_n : \Delta_n$ and $l'_1 : \Delta'_1 \Longrightarrow \cdots \Longrightarrow l'_m : \Delta'_m$ and $l_n = l'_1$ and $\Delta'_1(s) = s$ for all $s \in \mathrm{Rng}(\Delta_n)$, then*

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

*Proof:* By the definition, for all $2 \leq j \leq m$, $\Delta'_j = \Delta^{l'_{j-1}}_{l'_j} \circ \cdots \circ \Delta^{l'_1}_{l'_2} \circ \Delta'_1$. If $\Delta'_1(s) = s$ for all $s \in \mathrm{Rng}(\Delta_n)$, then $\Delta'_1 \circ \Delta_n = \Delta_n$, and thus $\Delta'_j \circ \Delta_n = \Delta^{l'_{j-1}}_{l'_j} \circ \cdots \circ \Delta^{l'_1}_{l'_2} \circ \Delta_n$. By definition,

$$l'_1 : \Delta_n \Longrightarrow l'_2 : (\Delta^{l'_1}_{l'_2} \circ \Delta_n) \Longrightarrow \cdots \Longrightarrow l'_m : (\Delta^{l'_{m-1}}_{l'_m} \circ \cdots \circ \Delta^{l'_1}_{l'_2} \circ \Delta_n),$$

and so, since $l_n = l'_1$,

$$l_n : \Delta_n \Longrightarrow l'_2 : (\Delta'_2 \circ \Delta_n) \Longrightarrow \cdots \Longrightarrow l'_m : (\Delta'_m \circ \Delta_n).$$

□

This lemma provides the theoretical foundation for the compositionality of trace-based analyses. Particularly useful is the following corollary, which describes how a sequence that begins and ends at the same label may be composed with itself any number of times.

**Corollary 2** *If $l_1 : \Delta_1 \Longrightarrow \cdots \Longrightarrow l_n : \Delta_n$ and $l_1 = l_n$ and $\Delta_1(s) = s$ for all $s \in \mathrm{Rng}(\Delta_n)$, then, for all $k \geq 0$,*

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

*where* $f^{(i)} = \overbrace{f \circ \cdots \circ f}^{i \text{ times}}$.

This means that some transfer transition sequences of length $n$ contain enough information to describe sequences in the original transition system of $k(n-1)+1$ for any $k$. Later, we will use transfer transition systems to define programming languages, and this corollary will allow us to model and systematically examine all possible finite behaviors of loops with a transfer sequence that corresponds to just one iteration. In essence, one can think of $\mathrm{Rng}(\Delta_i)$ as encoding the "loop invariant" at label $l_i$.

# 4 A Case Study

To illustrate how transfer transition systems can be used to analyze programs, we will outline the development of the system for a simple assembly-like language with data structures (records), assignment, and loops, three language features that typically cause difficulties for analyzers. The development is organized as follows:

1. Define the semantics of the language as a small-step transition system.
2. Design a finite representation of transfer functions of this system.
3. Redefine the semantics in terms of single-step transfer functions and show equivalence to the first formulation.
4. Define an effectively-computable composition operation on the transfer function representation for the purposes of the transfer transition system.
5. Choose a strategy for exploring the transfer transition system.

Although the language we have chosen is rather low-level, this choice is mainly for simplicity of presentation. What is most important is that the set *State* that we will choose for the transition semantics of the language is a very general choice—essentially a store graph—and can be used for many different languages, including languages with functions. So the development in this section can be reused for any language whose state of evaluation can be defined with this *State*.

## 4.1 The language and a transition system semantics

A program in this language is a command, where a command is either a set of parallel assignments, a record creation, a conditional command, an unconditional jump, or a compound command. On the left side of each assignment is an *l-expression* denoting an *l-value*, which is either a variable or a field of a record. For instance, the l-expression $x.f.f'$ denotes field $f'$ of the record in field $f$ of the record in variable $x$. Fields can be updated without restriction; for instance, the assignment $x.f \leftarrow x$ sets field $f$ of the record in $x$ to point to that record itself. Binary operations range over a set of basic constants.

$$
\begin{array}{rcll}
\mathit{comm} & ::= & l\!:\!le_1,\ldots,le_n \leftarrow e_1,\ldots,e_n & \text{parallel assignment} \\
& | & l\!:\!le \leftarrow \langle f_1 = e_1,\ldots,f_n = e_n\rangle & \text{record creation} \\
& | & l\!:\!\mathbf{if}\ e\ \mathbf{then}\ \mathit{comm}\ \mathbf{endif} & \text{conditional} \\
& | & l\!:\!\mathbf{goto}\ l' & \text{jump} \\
& | & \mathit{comm};\ \mathit{comm}' & \text{sequencing}
\end{array}
$$

$$
\begin{array}{rcll}
e & ::= & k \mid le \mid e \; op \; e' & \text{expressions denote values} \\
le & ::= & x \mid le.f & \text{l-expressions denote l-values} \\
op & ::= & + \mid - \mid = \mid \cdots & \text{primitive operations on values} \\
\\
v & ::= & k \mid \phi & \text{values (constants, pointers to records)} \\
lv & ::= & x \mid \phi.f & \text{l-values (variables, record fields)} \\
\\
l & \in & \mathit{Label} & \text{labels} \\
k & \in & \mathit{Const} & \text{constants} \\
x & \in & \mathit{Var} & \text{variables} \\
f & \in & \mathit{Field} & \text{record field names} \\
\phi & \in & \mathit{Ptr} & \text{record pointers}
\end{array}
$$

We define the small-step semantics of this language by a transition system $(\mathit{Label} \times \mathit{State}, \longmapsto)$, where a configuration $l\!:\!s$ comprises a label representing the current syntactic point of evaluation and a state representing the current store. States are finite partial maps from l-values to values:

$$
s ::= \{lv_1 \mapsto v_1, \ldots, lv_n \mapsto v_n\}
$$

Each command in the program induces a family of transitions given by the following rules. Here, $\mathit{next}(l)$ denotes $l'$ when $l$ is the label of a sequence of commands $l\!:\!s;\; l'\!:\!s'$. We also use the notation $s[lv \mapsto v]$ to denote the state that maps $lv$ to $v$ and is everywhere else equivalent to $s$, and $\mathit{new}(s, i)$ returns the $i$th next pointer not occurring in $s$.[3]

$$
\frac{\begin{array}{c} l\!:\!le_1, \ldots, le_n \leftarrow e_1, \ldots, e_n \quad l' = \mathit{next}(l) \\ lv_i = \mathcal{LE}[\![le_i]\!]s \quad v_i = \mathcal{E}[\![e_i]\!]s \end{array}}{l\!:\!s \longmapsto l'\!:\!s[lv_1 \mapsto v_1] \cdots [lv_n \mapsto v_n]}
$$

$$
\frac{\begin{array}{c} l\!:\!le \leftarrow \langle f_1 = e_1, \ldots, f_n = e_n \rangle \quad l' = \mathit{next}(l) \\ lv = \mathcal{LE}[\![le]\!]s \quad v_i = \mathcal{E}[\![e_i]\!]s \quad \phi = \mathit{new}(s, 1) \end{array}}{l\!:\!s \longmapsto l'\!:\!s[lv \mapsto \phi][\phi.f_1 \mapsto v_1] \cdots [\phi.f_n \mapsto v_n]}
$$

$$
\frac{\begin{array}{c} l\!:\!\textbf{if } e \textbf{ then } l_{\text{TRUE}}\!:\!\ldots \textbf{ endif};\; l_{\text{FALSE}}\!:\!\ldots \\ \mathcal{E}[\![e]\!]s = v \in \{\text{TRUE}, \text{FALSE}\} \end{array}}{l\!:\!s \longmapsto l_v\!:\!s}
\qquad
\frac{l\!:\!\textbf{goto } l'}{l\!:\!s \longmapsto l'\!:\!s}
$$

The tasks of looking up values in the store and applying primitive operations are done by the basic partial functions $\mathcal{LE}[\![]\!]s$ and $\mathcal{E}[\![]\!]s$ that evaluate l-expressions to l-values and expressions to values, respectively, in a given state. These functions are simple and do not modify the state, so they are performed within a single transition.

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

Note that the semantics of the parallel assignment command performs the bindings from left-to-right, and so if multiple l-expressions evaluate to the same l-value it is the rightmost corresponding

[3]For this purpose, it is assumed that the set $\mathit{Ptr}$ is equipped with an enumeration.

expression that gets bound. However, it will simplify the presentation of the development to follow if we know that all the l-values in a parallel assignment are different, thus rendering the order of binding irrelevant. Therefore, we make a syntactic restriction on valid assignment commands: No two l-expressions on the left side of a parallel assignment may be the same variable, nor may they terminate with the same field name.

## 4.2 Representing transfer functions

We know from Section 3.3 that we can define a single-step transfer function $\Delta^l_{l'} \in State \rightharpoonup State$ to describe the transitions between configurations at label $l$ and those at $l'$. We review the definition here:

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

Furthermore, we can compose these functions to define the transfer transition system $(Label \times (State \rightharpoonup State), \Longrightarrow)$. Later, in Section 4.5, we will be generating traces of the transfer system. But to output these traces, we first need a way of *representing* transfer functions. As a technicality, it will be convenient to augment the set of expressions and l-expressions with an integer representation of pointers:

$$\begin{array}{rcll} le & ::= & \cdots \mid i.f & \text{augmented l-expressions} \\ e & ::= & \cdots \mid i & \text{augmented expressions} \end{array}$$

where $i \in \mathbb{Z}$.

Then, to represent a (potentially infinite) transition function $\Delta \in State \rightharpoonup State$, we use the following grammar.[4]

$$\begin{array}{rcll} \Delta & ::= & \{(\sigma_1, C_1), \ldots, (\sigma_n, C_n)\} & \text{transfer functions} \\ \sigma & ::= & \{le_1 \mapsto e_1, \ldots, le_n \mapsto e_n\} & \text{store modifications} \\ C & ::= & \{e_1, \ldots, e_m\} & \text{condition sets} \end{array}$$

In this representation, a transfer function consists of a finite set of pairs $(\sigma, C)$. Each pair handles a disjoint subset of the domain of $\Delta$, and together they handle the entire domain of $\Delta$. Suppose that the pair $(\sigma, C)$ handles the subset $S \subseteq \mathrm{Dom}(\Delta)$. Then $\sigma$ is a finite map from l-expressions to expressions representing the modifications that $\Delta$ makes to each state $s \in S$, and where $C$ is a finite *condition set* of expressions that must all evaluate to true in any state $s \in S$.

The conditions maintain two kinds of information: *control constraints* (for modeling conditional expressions) and *sharing constraints*. To give a feel for how these objects represent transfer

[4]In an abuse of notation, we denote by $\Delta$ both the transfer function *representation* and the actual transfer *function* that it represents.

functions, we now present some simple examples.

$$\begin{aligned}
\Delta \quad &= \mathbf{I} = \{(\emptyset, \emptyset)\} \\
\Delta(s) &= s
\end{aligned}$$

$$\begin{aligned}
\Delta \quad &= \{(\{x \mapsto y + z\}, \emptyset)\} \\
\Delta(s) &= s[x \mapsto s(y) + s(z)]
\end{aligned}$$

$$\begin{aligned}
\Delta \quad &= \{(\{x \mapsto 1, 1.\text{CAR} \mapsto a.\text{CAR}, 1.\text{CDR} \mapsto b\}, \emptyset)\} \\
\Delta(s) &= s[x \mapsto \phi][\phi.\text{CAR} \mapsto s((s(a)).\text{CAR})][\phi.\text{CDR} \mapsto s(b)] \\
&\qquad \text{where } \phi = \mathit{new}(s, 1)
\end{aligned}$$

$$\begin{aligned}
\Delta \quad &= \{(\{x \mapsto z\}, \{x > y\})\} \\
\Delta(s) &= \begin{cases} s[x \mapsto s(z)] & \text{if } s(x) > s(y) \\ \text{undefined} & \text{otherwise} \end{cases}
\end{aligned}$$

Note that we denote the identity function $\{(\emptyset, \emptyset)\}$ by $\mathbf{I}$.

All of these examples have just a single pair $(\sigma, C)$, but in general an exact representation of a transfer function requires more than one pair. This is because a fragment of code might behave completely differently when evaluated in contexts with different sharing; an example of this will be given in Section 4.4.

In general, the meaning of a transfer function representation $\Delta$ is

$$\Delta(s) = \begin{cases} s' \quad \text{if} \begin{cases} (\{le_1 \mapsto e_1, \ldots, le_n \mapsto e_n\}, C) \in \Delta \\ \mathcal{LE}[\![le_i]\!]s = lv_i,\ \mathcal{E}[\![e_i]\!]s = v_i,\ 1 \leq i \leq n. \\ \forall e \in C.\ \mathcal{E}[\![e]\!]s = \text{TRUE} \\ s' = s[lv_1 \mapsto v_1] \cdots [lv_n \mapsto v_n] \end{cases} \\ \text{undefined} \quad \text{otherwise} \end{cases}$$

where $\mathcal{LE}[\![]\!]s$ and $\mathcal{E}[\![]\!]s$ are extended to handle the "augmented" expressions and l-expressions as follows:

$$\mathcal{LE}[\![i.f]\!]s = (\mathit{new}(s, i)).f \qquad \mathcal{E}[\![i]\!]s = \mathit{new}(s, i)$$

At first glance it may appear that there are two potential sources of ambiguity in the definition of $\Delta(s)$. There may be more than one pair $(\sigma, C)$ in $\Delta$ that may be applicable to a given state, and also the order of the bindings is unspecified. These potential ambiguities are avoided by maintaining the following representation invariants:

1. For any state $s$, there is at most one pair $(\sigma, C) \in \Delta$ for which all $e \in C$, $\mathcal{E}[\![e]\!]s = \text{TRUE}$. Intuitively, each element of $\Delta$ imposes a disjoint set of sharing constraints on $s$.

2. For the unique pair $(\{le_1 \mapsto e_1, \ldots, le_n \mapsto e_n\}, C)$ that does satisfy this condition, if that pair exists, the sharing constraints in $C$ will ensure that $\mathcal{LE}[\![le_i]\!]s = \mathcal{LE}[\![le_j]\!]s$ implies $i = j$, and thus the order of binding is irrelevant.

### 4.3 The semantics as single-step transfer functions

Now that we have a useful finite representation for transfer functions, we will redefine our small-step semantics in terms of single-step transfer functions $\Delta_{l'}^{l}$. This is quite straightforward, and in fact the rules become even simpler because all applications of $\mathcal{LE}[\![]\!]s$ and $\mathcal{E}[\![]\!]s$ are handled implicitly by the transfer functions themselves. Recall that $\mathbf{I}$ denotes the identity transfer function $\{(\emptyset, \emptyset)\}$.

$$\frac{l\!:\!le_1, \ldots, le_n \leftarrow e_1, \ldots, e_n \quad l' = next(l)}{\Delta^l_{l'} = \{(\{le_1 \mapsto e_1, \ldots, le_n \mapsto e_n\}, \emptyset)\}}$$

$$\frac{l\!:\!le \leftarrow \langle f_1 = e_1, \ldots, f_n = e_n\rangle \quad l' = next(l)}{\Delta^l_{l'} = \{(\{le \mapsto 1, 1.f_1 \mapsto e_1, \ldots, 1.f_n \mapsto e_n\}, \emptyset)\}}$$

$$\frac{\begin{array}{c} l\!:\!\textbf{if}\ e\ \textbf{then}\ l_{\text{TRUE}}\!:\!\ldots\ \textbf{endif};\ l_{\text{FALSE}}\!:\!\ldots \\ v \in \{\text{TRUE}, \text{FALSE}\}\end{array}}{\Delta^l_{l_v} = \{(\emptyset, \{e = v\})\}} \qquad \frac{l\!:\!\textbf{goto}\ l'}{\Delta^l_{l'} = \mathbf{I}}$$

The following theorem states that the above rules do indeed define the correct single-step transfer functions of the semantics.

**Theorem 1**

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

*Proof:* Straightforward for each of the four rules. □

## 4.4 The composition operation

We now have a representation of transfer functions and we have defined all the single-step transfer functions $\Delta^l_{l'}$. These are the building blocks of a trace-based analysis, but to use Definition 2 to build traces of the transfer system, and to use Lemma 2 and Corollary 2 to compose the traces, we must also have an effectively computable *composition operation* on transfer function representations. The composition operation is the universal tool of a trace-based analysis framework; it captures the essence of many seemingly unrelated analysis problems (with several examples are given in Section 5).

What makes a composition operation so subtle is the interaction with the possible sharing in the two transfer functions being composed. As an example, consider the following code fragment that performs two record field assignments:

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

The semantics defines the following single-step transfer functions:

$$\Delta^{\text{L}_1}_{\text{L}_2} = \{(\{a.\text{F} \mapsto b\}, \emptyset)\}$$

$$\Delta^{\text{L}_2}_{\text{L}_3} = \{(\{c.\text{F}.\text{F} \mapsto d\}, \emptyset)\}$$

By the definition of transfer transition systems, we can consider the following transfer trace from the identity transfer function $\mathbf{I}$:

$$\text{L}_1\!:\!\mathbf{I} \Longrightarrow \text{L}_2\!:\!\Delta^{\text{L}_1}_{\text{L}_2} \circ \mathbf{I} \Longrightarrow \text{L}_3\!:\!\Delta^{\text{L}_3}_{\text{L}_2} \circ \Delta^{\text{L}_2}_{\text{L}_1} \circ \mathbf{I}$$

We should expect $\Delta^{\mathrm{L}_1}_{\mathrm{L}_2} \circ \mathbf{I} = \Delta^{\mathrm{L}_1}_{\mathrm{L}_2}$, but what then is $\Delta^{\mathrm{L}_2}_{\mathrm{L}_3} \circ \Delta^{\mathrm{L}_1}_{\mathrm{L}_2}$? It is a function that, given a state $s$, returns the state resulting from the two assignment operations. But to represent this function, we need to take into consideration the different types of sharing that might occur in $s$. If there is no relevant sharing, then the composition might be simply

$$\Delta^{\mathrm{L}_2}_{\mathrm{L}_3} \circ \Delta^{\mathrm{L}_1}_{\mathrm{L}_2} = \{(\{a.\mathrm{F} \mapsto b, c.\mathrm{F}.\mathrm{F} \mapsto d\}, \{a \neq c, a \neq c.\mathrm{F}\})\}$$

where the second component of the pair is the condition set that imposes the required sharing constraints. But suppose that $s(a) = s(b) = s(c)$ for some initial state $s$. Then after these two assignment operations take place, $a.\mathrm{F}$, $b.\mathrm{F}$, and $c.\mathrm{F}$ are all equal to $d$, and the composition might then be represented by

$$\Delta^{\mathrm{L}_2}_{\mathrm{L}_3} \circ \Delta^{\mathrm{L}_1}_{\mathrm{L}_2} = \{(\{a.\mathrm{F} \mapsto d\}, \{a = c, a = b\})\}.$$

Even though this is a very simple code fragment, this result is rather surprising. It says that if the fragment is evaluated from any state in which $a = b = c$, then the *entire* behavior of the code is to assign $d$ to $a.\mathrm{F}$; that is, this sequence of assignment commands is equivalent to the single assignment command

$$a.\mathrm{F} \leftarrow d$$

In particular, $c.\mathrm{F}.\mathrm{F}$ might not equal $d$ even though that was the final assignment that occurred. A more detailed example of alias detection is given in Section 5.3.

Both of the possibilities above must be included in the transfer function $\Delta^{\mathrm{L}_2}_{\mathrm{L}_3} \circ \Delta^{\mathrm{L}_1}_{\mathrm{L}_2}$, and this is why a transfer function representation can have more than one pair $(\sigma, C)$. For this particular example, the composition operation actually finds two additional cases. The result of the composition is:

$$\begin{aligned}\Delta^{\mathrm{L}_2}_{\mathrm{L}_3} \circ \Delta^{\mathrm{L}_1}_{\mathrm{L}_2} = \{&(\{a.\mathrm{F} \mapsto d\}, \{a = c, a = b),\\ &(\{a.\mathrm{F} \mapsto b, b.\mathrm{F} \mapsto d\}, \{a = c, a \neq b\}),\\ &(\{a.\mathrm{F} \mapsto d\}, \{a \neq c, a = c.\mathrm{F}\}),\\ &(\{a.\mathrm{F} \mapsto b, c.\mathrm{F}.\mathrm{F} \mapsto d\}, \{a \neq c, a \neq c.\mathrm{F}\})\}\end{aligned}$$

The composition operation is defined structurally as follows:

$$\begin{aligned}\Delta' \circ \Delta = \{&(s[le'_1 \mapsto e'_1] \cdots [le'_n \mapsto e'_n], C') \mid\\ &(\{le_1 \mapsto e_1, \ldots, le_n \mapsto e_n\}, \{e''_1, \ldots, e''_m\}) \in \Delta',\\ &(\sigma, C) \in \Delta,\\ &(le'_i, C_i) \in \widetilde{\mathcal{LE}}[\![le_i]\!]\sigma \text{ and } (e'_i, C'_i) \in \widetilde{\mathcal{E}}[\![e_i]\!]\sigma, \text{for } 1 \leq i \leq n,\\ &(e'''_i, C''_i) \in \widetilde{\mathcal{E}}[\![e''_i]\!]\sigma, \text{for } 1 \leq i \leq m,\\ &C' = C \cup C_1 \cup \ldots \cup C_n \cup C'_1 \cup \ldots \cup C'_n\\ &\qquad \cup C''_1 \cup \ldots \cup C''_m \cup \{e'''_1, \ldots, e'''_m\},\\ &C' \text{ is consistent}\}\end{aligned}$$

where:

$$\begin{aligned}
\widetilde{\mathcal{LE}}[\![x]\!]\sigma &= \{(x, \emptyset)\} \\
\widetilde{\mathcal{LE}}[\![i.f]\!]\sigma &= \{((\widetilde{new}(s,i)).f, \emptyset)\} \\
\widetilde{\mathcal{LE}}[\![le.f]\!]\sigma &= \{(i.f, C) \mid (i, C) \in \widetilde{\mathcal{E}}[\![le]\!]\sigma\} \\
&\cup \{(le''.f, C \cup \{le' = le''\}) \mid \\
&\qquad (le', C) \in \widetilde{\mathcal{E}}[\![le]\!]\sigma,\ le''.f \in \mathrm{Dom}(\sigma)\} \\
&\cup \{(le'.f, C \cup \{le' \neq le'' \mid le''.f \in \mathrm{Dom}(\sigma)\}) \mid \\
&\qquad (le', C) \in \widetilde{\mathcal{E}}[\![le]\!]\sigma\}
\end{aligned}$$

$$\begin{aligned}
\widetilde{\mathcal{E}}[\![k]\!]\sigma &= \{(k, \emptyset)\} \\
\widetilde{\mathcal{E}}[\![e_1\ op\ e_2]\!]\sigma &= \{(e_1'\ \widetilde{op}\ e_2', C_1 \cup C_2) \mid \\
&\qquad (e_1, C_1) \in \widetilde{\mathcal{E}}[\![e_1]\!]\sigma,\ (e_2, C_2) \in \widetilde{\mathcal{E}}[\![e_2]\!]\sigma\} \\
\widetilde{\mathcal{E}}[\![i]\!]\sigma &= \{(\widetilde{new}(\sigma, i), \emptyset)\} \\
\widetilde{\mathcal{E}}[\![le]\!]\sigma &= \{(\sigma[le], C) \mid (le, C) \in \widetilde{\mathcal{LE}}[\![le]\!]\sigma\}
\end{aligned}$$

Here, $\sigma[le]$ equals $\sigma(le)$ if it is defined, and $le$ otherwise; $\widetilde{new}(\sigma, i)$ returns the $i$th first integer not appearing in $\sigma$; and $e_1\ \widetilde{op}\ e_1$ returns an expression $e$ such that $\mathcal{E}[\![e_1\ op\ e_2]\!]s = \mathcal{E}[\![e]\!]s$ for all states $s$. It is always correct to choose "$e_1\ op\ e_2$" for $e_1\ \widetilde{op}\ e_2$, but it might be desirable to apply a system of simplification equations on the primitive operations, if such a system exists.

Conceptually, given $(\sigma, C) \in \Delta$ and $(\sigma', C') \in \Delta'$, we need to find all possible ways that sharing interference could affect the bindings in $\sigma'$. The functions $\widetilde{\mathcal{LE}}[\![]\!]\sigma$ and $\widetilde{\mathcal{E}}[\![]\!]\sigma$ determine all such possibilities and build up the associated condition sets describing the sharing constraints for each case. When that is done for all augmented l-expressions and augmented expressions in $(\sigma', C')$, then the bindings take place, possibly overwriting some bindings already existing in $\sigma$, and the constraints are updated for that particular set of bindings. The "consistent" condition is a structural requirement of the condition set that ensures that there is no $e$ and $e'$ such that $e = e'$ and $e \neq e'$ in the transitive closure of the equalities and inequalities in the condition set.

**Lemma 3** *For all transfer function representations $\Delta$ and $\Delta'$ and states $s$, $(\Delta' \circ \Delta)(s) = \Delta'(\Delta(s))$ (where $\circ$ is the composition operation defined above).*

## 4.5 The transfer transition system

Definition 2 defines the transfer transition system $(\mathit{Label} \times (\mathit{State} \rightharpoonup \mathit{State}), \Longrightarrow)$ from the semantics.

$$l : \Delta \Longrightarrow l' : (\Delta^l_{l'} \circ \Delta) \text{ for all } \Delta \in \mathit{State} \rightharpoonup \mathit{State}.$$

We now have a finite representation of these transfer functions $\Delta$, the single-step transfer functions $\Delta^l_{l'}$, and an effectively computable composition operation $\circ$ on these representations. Therefore, we have all the required tools for exploring the transfer transition system. Of course, this system is of infinite size and cannot be explored completely, but we can use our theoretical development in Section 3 to selectively explore the system. For instance, we can use Lemma 2 to piece individual traces together, and we can use Corollary 2 to reason about loops by exploring only one iteration. Many strategies are possible, and different ones will be appropriate for different problems. Below we describe a strategy that will be useful in our example applications in Section 5.

In general, given an initial transfer configuration $l_1 : \Delta_1$, we can start generating the possible transfer traces $l_1 : \Delta_1 \Longrightarrow l_2 : \Delta_2 \Longrightarrow \cdots$. Such traces fall into two classes:

1. Eventually, a configuration $l_i : \Delta_i$ is reached such that there exists a $j < i$ such that $l_i = l_j$. These traces correspond to evaluations of the program from $l_1$ that eventually take some form of loop.

2. Alternatively, these traces correspond to evaluations of the program from $l_1$ that reach a configuration from which no further transitions are possible before any form of loop is taken.

In both cases, the traces have length at most $n+1$, where $n$ is the number of labels in the program, and are thus computable. It is particularly useful to compute these traces from the identity transfer function $\mathbf{I}$, because then we are sure to be able to use Lemma 2 and Corollary 2 to "piece together" these traces in order to compute longer traces.

Given a label $l_1$, we can compute the set $loops(l_1)$ of traces of the first kind:

$$\begin{aligned} loops(l_1) = \{ & l_1 : \mathbf{I} \Longrightarrow \cdots \Longrightarrow l_n : \Delta \mid \\ & \quad \exists i < n.\ l_i = l_n,\ \ \neg\exists i < j < n.\ l_i = l_j) \} \end{aligned}$$

and the set $terminals(l_1)$ of traces of the second kind:

$$\begin{aligned} terminals(l_1) = \{ & l_1 : \mathbf{I} \Longrightarrow \cdots \Longrightarrow l_n : \Delta \mid \\ & \quad \neg\exists l, \Delta'.\ l_n : \Delta \Longrightarrow l : \Delta',\ \ \neg\exists i < j \leq n.\ l_i = l_j \} \end{aligned}$$

By repeated applications of Lemma 2, we can inductively define (but not necessarily compute) the sets $traces(l)$ for all labels $l$ of all traces from configuration $l : \mathbf{I}$. (The occurrences of $\Delta_1$ and $\Delta'_1$ in these rules are always equal to $\mathbf{I}$.)

$$\frac{l_1 : \Delta_1 \Longrightarrow \cdots \Longrightarrow l_n : \Delta_n \in terminals(l_1)}{l_1 : \Delta_1 \Longrightarrow \cdots \Longrightarrow l_n : \Delta_n \in traces(l_1)}$$

$$\frac{\begin{array}{c} l_1 : \Delta_1 \Longrightarrow \cdots \Longrightarrow l_n : \Delta_n \in loops(l_1) \\ l'_1 : \Delta'_1 \Longrightarrow \cdots \Longrightarrow l'_m : \Delta'_m \in traces(l'_1) \quad l_n = l'_1 \end{array}}{\begin{array}{l} l_1 : \Delta_1 \Longrightarrow \cdots \Longrightarrow l_n : \Delta_n \Longrightarrow \\ \quad l'_2 : \Delta'_2 \circ \Delta_n \Longrightarrow \cdots \Longrightarrow l'_m : \Delta'_m \circ \Delta_n \in traces(l_1) \end{array}}$$

# 5 Applications

In this section we present some concrete examples of how trace-based analysis can be applied to solve problems in program analysis and transformation.

## 5.1 Software pipelining

One of the advantages of trace-based analysis is that it provides a direct way to reason about program equivalences, and thus it can serve as a basis for specifying and proving the correctness of program transformations. In this section, we demonstrate this by showing how a form of software pipelining can be developed using a transfer transition system.

*Software pipelining* is a program transformation on loops that attempts to exploit instruction-level parallelism in superscalar and VLIW architectures [2, 22]. As an example, consider the following code fragment taken from [2] (but with a conditional statement added so that the loop exit can be expressed):

$$
\begin{array}{lll}
\mathrm{L}_{\mathrm{entry}} & : & i \leftarrow i + 1; \\
\mathrm{L}_1 & : & j \leftarrow i + h; \\
\mathrm{L}_2 & : & k \leftarrow i + g; \\
\mathrm{L}_3 & : & l \leftarrow j + 1; \\
\mathrm{L}_4 & : & \mathit{test} \leftarrow i < n; \\
\mathrm{L}_5 & : & \textbf{if } \mathit{test} \textbf{ then} \\
\mathrm{L}_6 & : & \qquad \textbf{goto } \mathrm{L}_{\mathrm{entry}} \\
 & & \textbf{endif}; \\
\mathrm{L}_{\mathrm{exit}} & : & \ldots
\end{array}
\tag{4}
$$

The classical approach to optimizing this loop involves unrolling it once, analyzing the data dependencies, and then optimizing for maximum parallelism within the loop body. This yields the following:

$$
\begin{array}{lll}
\mathrm{L}_{\mathrm{entry}} & : & i \leftarrow i + 1; \\
\mathrm{L}_1 & : & j, k, \mathit{test} \leftarrow i + h, i + g, i < n; \\
\mathrm{L}_2 & : & \textbf{if } \mathit{test} \textbf{ then} \\
\mathrm{L}_3 & : & \qquad l, i \leftarrow j + 1, i + 1; \\
\mathrm{L}_4 & : & \qquad j, k, \mathit{test} \leftarrow i + h, i + g, i < n; \\
\mathrm{L}_5 & : & \qquad \textbf{if } \mathit{test} \textbf{ then} \\
\mathrm{L}_6 & : & \qquad\qquad l \leftarrow j + 1; \\
\mathrm{L}_7 & : & \qquad\qquad \textbf{goto } \mathrm{L}_{\mathrm{entry}} \\
\mathrm{L}_8 & : & \qquad \textbf{endif}; \\
\mathrm{L}_9 & & \textbf{endif}; \\
\mathrm{L}_1 0 & : & l \leftarrow j + 1 \\
\mathrm{L}_{\mathrm{exit}} & : & \ldots
\end{array}
$$

While this is definitely an improvement, this simple approach fails to extract all of the parallelism available between adjacent loop iterations. In particular, the assignments to $l$ and $i$ at labels $\mathrm{L}_6$ and $\mathrm{L}_{\mathrm{entry}}$, respectively, can be performed in parallel, but this is not detected by the classical approach. Software pipelining achieves this additional parallelism by first determining the patterns of potential parallelism across loop iterations and then using this information to transform the loop. The effectiveness of this technique can be seen in the following code, which is the result of applying software pipelining to our original loop:

$$
\begin{array}{lll}
\mathrm{L}_{\mathrm{entry}} & : & i \leftarrow i + 1; \\
\mathrm{L}_1 & : & j, k, \mathit{test} \leftarrow i + h, i + g, i < n; \\
\mathrm{L}_2 & : & \textbf{if } \mathit{test} \textbf{ then} \\
\mathrm{L}_3 & : & \qquad l, i \leftarrow j + 1, i + 1; \\
\mathrm{L}_4 & : & \qquad \textbf{goto } \mathrm{L}_1 \\
 & & \textbf{endif}; \\
\mathrm{L}_5 & : & l \leftarrow j + 1 \\
\mathrm{L}_{\mathrm{exit}} & : & \ldots
\end{array}
\tag{5}
$$

The state-of-the-art in software pipelining is quite powerful, but rather *ad hoc*. The transformations rely on much *a priori* knowledge (such as the lack of aliases), as well as knowledge of the loop structure and data dependencies. The correctness proofs are thus long and tedious and are not based on satisfactory mathematical underpinnings. Using the transfer transition system semantics, we can formalize what it means for two looping program fragments to be equivalent and thus interchangable without affecting the observational behavior of the whole program.

Recall that a loop in our language is syntactically a command, *comm*. We assume that there is a distinguished entry label denoted by *entry*(*comm*) and a distinguished exit label denoted by *exit*(*comm*). In our current example, the entry and exit labels are $\mathrm{L}_{\text{entry}}$ and $\mathrm{L}_{\text{exit}}$. Given the transition system semantics, $(\mathit{Label} \times \mathit{State}, \longmapsto)$, the *meaning* of *comm*, $\mu(\mathit{comm})$, is defined as follows:

$$\mu(\mathit{comm}) = \lambda s.\ \{s' \mid (\mathit{entry}(\mathit{comm})){:}s \longmapsto^* (\mathit{exit}(\mathit{comm})){:}s'\}$$

Since our language is deterministic, the set $\mu(\mathit{comm})(s)$ is either $\emptyset$ if evaluation from state $s$ never reaches the exit label, or else $\{s'\}$, where $s'$ is the unique resulting state at the exit label. We note, however, that the following methodology is applicable to any transition system that satisfies Condition 1, including nondeterministic systems.

If $\mu(\mathit{comm}) = \mu(\mathit{comm}')$, it is safe to replace *comm* by *comm'* in any context. Of course, this question is undecidable in general, and indeed one of the most important motivations for the formal semantics of programming languages is to reason about such program equivalences. In many difficult cases, an analysis of the transfer transition system can automatically prove such an equality and thus support optimizing program transformations such as software pipelining. In general, we have the following theorem that proves semantic equivalence of two looping commands.

**Theorem 2** *Given two commands* $\mathit{comm}_1$ *and* $\mathit{comm}_2$, *for* $i \in \{1,2\}$ *let* $\mathit{terminals}_i$ *and* $\mathit{loops}_i$ *be computed from* $\mathit{comm}_i$ *as in Section 4.5, where* $\stackrel{i}{\Longrightarrow}$ *denotes the transfer transition relation of* $\mathit{comm}_i$, *and let* $l_{\text{entry}_i} = \mathit{entry}(\mathit{comm}_i)$ *and* $l_{\text{exit}i} = \mathit{exit}(\mathit{comm}_i)$. *If there exists a* $\Delta_{\text{shift}}$ *such that the following equations hold for* $i \in \{1,2\}$, *then* $\mu(\mathit{comm}_1) = \mu(\mathit{comm}_2)$.

$$\begin{aligned}
|\mathit{terminals}_i(l_{\text{entry}_i})| &= 1\\
\mathit{loops}_i(l_{\text{entry}_i}) &= \{\cdots \stackrel{i}{\Longrightarrow} l_{\text{loop}_i}{:}\Delta_{\text{entry}_i} \stackrel{i}{\Longrightarrow} \cdots \stackrel{i}{\Longrightarrow} l_{\text{loop}_i}{:}\Delta_i\}\\
\mathit{loops}_i(l_{\text{loop}_i}) &= \{\cdots \stackrel{i}{\Longrightarrow} l_{\text{loop}_i}{:}\Delta_{\text{loop}_i}\}\\
\mathit{terminals}_i(l_{\text{loop}_i}) &= \{\cdots \stackrel{i}{\Longrightarrow} l_{\text{exit}i}{:}\Delta_{\text{exit}i}\}
\end{aligned}$$

$$\begin{aligned}
\Delta_{\text{entry}_2} &= \Delta_{\text{shift}} \circ \Delta_{\text{entry}_1}\\
\Delta_{\text{loop}_2} \circ \Delta_{\text{shift}} &= \Delta_{\text{shift}} \circ \Delta_{\text{loop}_1}\\
\Delta_{\text{exit}2} \circ \Delta_{\text{shift}} &= \Delta_{\text{exit}1}
\end{aligned}$$

*Proof:* First we need an auxilliary lemma about function composition. If $f$, $g$, $h$, $f'$, $g'$, $h'$, and $\delta$ are functions in $X \rightharpoonup X$ and if (1): $f = \delta \circ f'$, (2): $g \circ \delta = \delta \circ g'$, and (3): $h \circ \delta = h'$, then $h \circ g^{(n)} \circ f = h' \circ g'^{(n)} \circ f'$ for all $n \geq 0$. To prove this, we apply axiom (1) to the statement that $h \circ g^{(n)} \circ \delta = h' \circ g'^{(n)}$ for all $n \geq 0$. The latter has a straightforward proof by induction on $n$; axiom (3) is the base case, and axiom (2) proves the inductive case.

Let $\stackrel{i}{\longmapsto}$ be the transition relation of the semantics of $\mathit{comm}_i$. Then $s' \in \mu(\mathit{comm}_1)(s)$ iff $(\mathit{entry}(\mathit{comm})){:}s \stackrel{1}{\longmapsto}^* (\mathit{exit}(\mathit{comm})){:}s'$, which in turn holds iff there exists a trace

$$l_{\text{entry}_1}{:}s \stackrel{1}{\longmapsto} l_1{:}s_1 \stackrel{1}{\longmapsto} \ldots \stackrel{1}{\longmapsto} l_n{:}s_n \stackrel{1}{\longmapsto} l_{\text{exit}1}{:}s',$$

And, by Lemma 1, this trace exists iff there exists a transfer trace

$$l_{\text{entry}_1}{:}\mathbf{I} \stackrel{1}{\Longrightarrow} l_1{:}\Delta_1 \stackrel{1}{\Longrightarrow} \ldots \stackrel{1}{\Longrightarrow} l_n{:}\Delta_n \stackrel{1}{\Longrightarrow} l_{\text{exit}1}{:}\Delta$$

such that $\Delta(s) = s'$. By Lemma 2 and Corollary 2 ($\mathit{terminals}_1$ and $\mathit{loops}_1$), such a transfer trace exists iff there exists a $k \geq 0$ such that $(\Delta_{\text{exit}1} \circ {\Delta_{\text{loop}_1}}^{(k)} \circ \Delta_{\text{entry}_1})(s) = s'$. Then, by the above lemma about function concatenation, this holds iff $(\Delta_{\text{exit}2} \circ \Delta_{\text{loop}_2}{}^{(k)} \circ \Delta_{\text{entry}_2})(s) = s'$, which in turn holds iff $s' \in \mu(comm_2)(s)$ (by a reversal of the above).

Therefore, we can conclude that $\mu(comm_1) = \mu(comm_2)$. □

The intuition behind this theorem is that $comm_1$ is a code fragment that contains a single loop; $\Delta_{\text{entry}_1}$ represents the computation from the entry of the command up to the first loop entry point, $\Delta_{\text{loop}_1}$ represents the computation from one loop entry point to the next (*i.e.*, one loop iteration), and $\Delta_{\text{exit}1}$ represents the computation from the loop entry point to the exit of the command. Any evaluation of $comm_1$ that exits will thus be represented by $\Delta_{\text{exit}1} \circ \Delta_{\text{loop}_1}{}^{(k)} \circ \Delta_{\text{entry}_1}$, where $k$ is the number of times the loop was taken. Everything is similar for $comm_2$, except that its loop is "shifted" by $\Delta_{\text{shift}}$.

This shifting essentially captures the notion of software pipelining. To illustrate this theorem, take $comm_1$ to be the original looping program (Program 4) and $comm_2$ to be the optimized version (Program 5). First, the computation of *terminals* and *loops* for each command generates:

$$\begin{aligned}
\Delta_{\text{entry}_1} &= \mathbf{I} \\
\Delta_{\text{loop}_1} &= \{(\{i \mapsto i+1,\ j \mapsto i+h+1,\ k \mapsto i+g+1, \\
&\qquad\quad l \mapsto i+h+2,\ test \mapsto \mathit{i{+}1} < n\}, \\
&\quad\ \{(i+1 < n) = \text{TRUE}\})\} \\
\Delta_{\text{exit}1} &= \{(\{i \mapsto i+1,\ j \mapsto i+h+1,\ k \mapsto i+g+1, \\
&\qquad\quad l \mapsto i+h+2,\ test \mapsto \mathit{i{+}1} < n\}, \\
&\quad\ \{(i+1 < n) = \text{FALSE}\})\} \\
\Delta_{\text{entry}_2} &= \{(\{i \mapsto i+1\}, \emptyset)\} \\
\Delta_{\text{loop}_2} &= \{(\{i \mapsto i+1,\ j \mapsto i+h,\ k \mapsto i+g, \\
&\qquad\quad l \mapsto i+h+1,\ test \mapsto i < n\}, \\
&\quad\ \{(i < n) = \text{TRUE}\})\} \\
\Delta_{\text{exit}2} &= \{(\{j \mapsto i+h,\ k \mapsto i+g, \\
&\qquad\quad l \mapsto i+h+1,\ test \mapsto i < n\}, \\
&\quad\ \{(i < n) = \text{FALSE}\})\}
\end{aligned}$$

Here, we have written bindings in lexicographic order and have assumed a canonical form for expressions, where for clarity we have replaced $1 + 1$ with 2. Note the differences between the bindings of $j$, $k$, and $l$ in $\Delta_{\text{loop}_1}$ and those of $\Delta_{\text{loop}_2}$. This is because in $comm_1$ these assignments take place after the increment of $i$, while in $comm_2$ they take place before.

Next, we find an appropriate $\Delta_{\text{shift}}$. Since $\Delta_{\text{entry}_1} = \mathbf{I}$, $\Delta_{\text{shift}}$ must be the same as $\Delta_{\text{entry}_2}$:

$$\Delta_{\text{shift}} = \Delta_{\text{entry}_2} = \{(\{i \mapsto i+1\}, \emptyset)\}$$

This corresponds to the fact that the loop of $comm_1$ has been "shifted" in $comm_2$. In each increment of $i$ in $comm_2$ conceptually corresponds to that of the *next* iteration.

The final step is the proof that the following hold:

$$\begin{aligned}
\Delta_{\text{entry}_2} &= \Delta_{\text{shift}} \circ \Delta_{\text{entry}_1} \\
\Delta_{\text{loop}_2} \circ \Delta_{\text{shift}} &= \Delta_{\text{shift}} \circ \Delta_{\text{loop}_1} \\
\Delta_{\text{exit}2} \circ \Delta_{\text{shift}} &= \Delta_{\text{exit}1}
\end{aligned}$$

First we compute the compositions and then approximate the equality check by checking for structural equality. If two transfer function representations are structurally equivalent, then they are guaranteed to represent the same function. Much more sophisticated strategies could be developed, for instance using algebras for expression equality, but the important point is that while structural equality is usually of very little use in standard transition systems, it is quite powerful for transfer transition systems. It is sufficient for the above example, and we believe that it will be sufficient for many uses.

In short, the entire proof is completely automated and with a tractable complexity.[5] The algorithm used the fact that $\Delta_{\text{entry}_1} = \mathbf{I}$ for the derivation of $\Delta_{\text{shift}}$ as $\Delta_{\text{entry}_2}$, but in general we need an algorithm to solve $\Delta = \Delta_{\text{shift}} \circ \Delta'$ given any $\Delta$ and $\Delta'$. One can imagine a unification algorithm for this purpose, but space does not permit further development.

The automation of the proof suggests that the procedure could be used directly to *derive* correct program transformations, not just for software pipelining, but for loop optimizations in general. In the next section we see another example.

### 5.2 Loop-invariant removal

We can use a technique similar to that of software pipelining to reason automatically about a transformation that factors out calculations that remain invariant in each iteration of a loop. The only difference between software pipelining and loop-invariant removal is the set of axioms.

**Theorem 3** *Given two commands $comm_1$ and $comm_2$, if there exists a $\Delta_{\text{inv}}$ such that the following equations hold, where $\Delta_{\text{entry}_i}$, $\Delta_{\text{loop}_i}$, and $\Delta_{\text{exit}\,i}$ are defined as in Theorem 2, then $\mu(comm_1) = \mu(comm_2)$.*

$$\begin{aligned}
\Delta_{\text{entry}_2} &= \Delta_{\text{inv}} \circ \Delta_{\text{entry}_1} \\
\Delta_{\text{loop}_2} \circ \Delta_{\text{inv}} &= \Delta_{\text{loop}_1} \\
\Delta_{\text{inv}} \circ \Delta_{\text{loop}_2} &= \Delta_{\text{loop}_2} \\
\Delta_{\text{exit}\,2} \circ \Delta_{\text{inv}} &= \Delta_{\text{exit}\,1}
\end{aligned}$$

*Proof:* The proof mirrors that of Theorem 2, but with a different lemma about function composition. If $f$, $g$, $h$, $f'$, $g'$, $h'$, and $\delta$ are functions in $X \rightharpoonup X$ and if (1): $f = \delta \circ f'$, (2): $g \circ \delta = g'$, (3): $\delta \circ g = g$, and (4) $h \circ \delta = h'$, then $h \circ g^{(n)} \circ f = h' \circ g'^{(n)} \circ f'$ for all $n \geq 0$. To prove this, we apply axiom (4) to the statement that $g^{(n)} \circ f = \delta \circ g'^{(n)} \circ f'$ for all $n \geq 0$. The latter has a straightforward proof by induction on $n$; axiom (1) is the base case, and axioms (2) and (3) prove the inductive case as follows: $g^{(n)} \circ f = \delta \circ g \circ g^{(n-1)} \circ f = \delta \circ g \circ \delta \circ g'^{(n-1)} \circ f' = \delta \circ g'^{(n)} \circ f'$. □

The transfer function $\Delta_{\text{inv}}$ ($\delta$ in the proof) represents some computation that $comm_1$ does inside every loop iteration, but which $comm_2$ does once and for all before the loop is first entered.

### 5.3 Alias analysis

The following program is intended to evaluate in a context in which $a$ is bound to a linked list of at least one element and $b$ is bound to a linked list of at least two elements. The intended result of the program is that $b$ should be bound to a list whose first element is the same, whose second element is $a$'s original first element, whose third element is $b$'s original second element, and which is thereafter equal to the rest of $a$. The code does this by destructive assignment rather than by

[5]Construction of eight traces of no more than $n+1$ in length each (every step of which is a composition operation), four additional composition operations, and three $O(n)$ structural equality tests. All composition operations are $O(n)$ because no sharing constraints are generated.

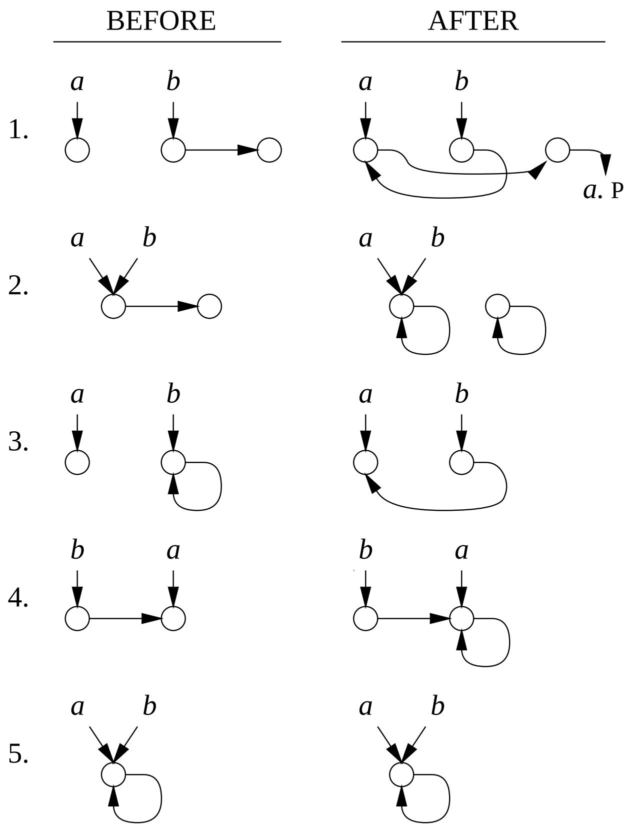


Figure 1: Output of the alias analysis.

copying nodes.

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

Here, the P field contains the link to the next list element.

Examining the transfer transitions from the initial configuration $\mathrm{L}_1 : \mathbf{I}$, representing all possible entry configurations, terminates after three steps at configuration $\mathrm{L}_4 : \Delta$, where $\Delta$ is a set of five pairs that describe all possible results:

$$
\begin{array}{l}
(\{b.\mathrm{P}.\mathrm{P} \mapsto a.\mathrm{P}, a.\mathrm{P} \mapsto b.\mathrm{P}, b.\mathrm{P} \mapsto a\}, \{a \neq b.\mathrm{P}, b \neq b.\mathrm{P}, b \neq a\}) \\
(\{b.\mathrm{P}.\mathrm{P} \mapsto a.\mathrm{P}, a.\mathrm{P} \mapsto a\}, \{a \neq b.\mathrm{P}, b \neq b.\mathrm{P}, b = a\}) \\
(\{b.\mathrm{P} \mapsto a\}, \{a \neq b.\mathrm{P}, b = b.\mathrm{P}\}) \\
(\{a.\mathrm{P} \mapsto a\}, \{a = b.\mathrm{P}, b \neq b.\mathrm{P}\}) \\
(\{b.\mathrm{P} \mapsto a\}, \{a = b.\mathrm{P}, b = b.\mathrm{P}\})
\end{array}
$$

The second component of each pair describes a possible set of sharing constraints on the input set, and the first element describes the updates to the store that take place under those sharing constraints. The updates should be interpreted as one big parallel assignment on the initial state. The first pair is the intended result, and the other four pairs represent different types of undesirable behavior. The analysis output can be equivalently described by the diagram shown in Figure 1.

One advantage to this analysis is that the composition operation on transfer functions automatically maintains only the *relevant* sharing constraints. For instance, in this example the five different pairs correspond to five truly different behaviors, and the sharing constraints in those pairs are the minimum constraints necessary to identify each case.

This analysis has several potential uses:

- A sophisticated alias analysis that is *exact* on straight-line code, that can *relate* the output aliases with the input aliases, and that can *compose* such relations together to analyze blocks of code separately or describe the aliasing of looping code. Currently, there is little understood about relational alias analyses or compositional alias analyses.
- A symbolic debugging tool. One can look at all possible ways a fragment of code might go wrong, and in what contexts.
- A modular program verification tool. In the example above, the analysis infers an exact minimal set of input preconditions $\{a \neq b.\mathrm{P}, b \neq b.\mathrm{P}, b \neq a\}$ that guarantees the correct output. More sophisticated code with loops might use Corollary 2 to determine a sufficient, but perhaps not necessary, set of preconditions.

# 6 Conclusions

In this paper we have presented *transfer transition systems*, a formal framework for describing the operational semantics of programs, and demonstrated its utility on a language with loops, assignments, and nested record structures. This framework allows us capture the notion of *trace properties* and *trace-based program analysis*, thereby realizing significant advantages in expressive power and elegance in solving complex static-analysis problems, particularly for programs with loops. Furthermore, we have shown how transfer transition systems can be used to derive and formalize program transformations for carrying out optimizations as complex as software pipelining, classical loop optimizations, and detection of aliases.

The idea of reasoning about traces instead of states is not new. The theoretical foundations go back to Cousot and Cousot [9]. Their work on $\mathrm{G}^{\infty}\mathrm{SOS}$ [10] is also related, but whereas the main focus of their system is to achieve a unified compositional operational model of non-termination, our motivation for transfer transition systems is ultimately a practical one: to provide a new framework not only for static analysis of programs, but also for developing complex program transformations and optimizations that depend on precise understanding of complex control flow and data structures.

There are a handful of "trace-based" analyses in the literature, such as Colby's analyses of concurrency [4, 5, 6] and Deutsch's online alias analysis [13, Sect. 4.4]. It is also common to use some notion of execution traces when reasoning about concurrent computations. But in the areas of static program analysis and program transformation the technique is little known. It is our belief that this is due to the lack of a presentation of the method that is both general enough for wide applicability and specific enough to be easily instantiable.

One might reasonably ask why trace-based analysis is necessary, since *ad hoc* techniques, many based on dataflow analysis, often work in practice, at least in simple cases [1]. Indeed, for the gcd example shown in Section 2, it is a simple matter to unroll the loop once and then perform standard optimizations such as constant propagation. One basic reason is that semantics-based approaches provide a way to reason about correctness and safety of analysis-based program transformations. But there is a less obvious reason of great practical importance: the formalism of semantics-based program analysis is extremely general, and thus yields insight into ways to solve much more complex analysis problems. Indeed, it is easy to see that the technique of "unroll once and then do constant propagation" is not very general, and in fact does not work for the purpose of software pipelining. Abstract interpretation, on the other hand, clearly aided the solution to control-flow analysis of higher-order functions [20, 27]. Even further, it is difficult to imagine many of the more advanced alias and storage analyses (*e.g.*, [14, 6]) without abstract interpretation, and nor is it likely that the analyses of concurrency in [5, 4] would have been found. The problem is that *ad hoc* techniques rarely generalize or shed any light on techniques that might be adapted for other problems, while the methodology of semantics-based approaches root analyses in the most general soil—a semantics of the language—from which other analyses may spring.

### Acknowledgements

The authors wish to thank Mark Leone, Chris Okasaki, and Frank Pfenning for their helpful comments and suggestions on earlier drafts of this paper.

## References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. In *Proceedings of the 1988 European Symposium on Programming*, LNCS. Springer-Verlag, March 1988.

[3] D.R. Chase, M. Wegman, and F.K. Zadeck. Analysis of pointers and structures. In *Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[4] Christopher Colby. Analysis of synchronization and aliasing with abstract interpretation. Unpublished.

[5] Christopher Colby. Analyzing the communication topology of concurrent programs. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manupulation*, pages 202–214, June 1995.

[6] Christopher Colby. Determining storage properties of sequential and concurrent programs with assignment and structured data. In *International Static Analysis Symposium*, 1995. To appear.

[7] Charles Consel. Polyvariant binding-time analysis for applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (SIGPLAN Notices, vol. 26, no. 9, September 1991)*, pages 66–77, 1993.

[8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Fourth Annual ACM Symposium on Principles of Programming Languages*, 1977.

[9] P. Cousot and R. Cousot. Semantic design of program analysis frameworks. In *Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, pages 269–282, 1979.

[10] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the 19th ACM Symposium on Principles of Programming Languages*, pages 83–94, 1992.

[11] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94), Toulouse, France*, pages 95–112, May 1994.

[12] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Conference on Functional Programming and Computer Architecture*, 1995.

[13] Alain Deutsch. *Operational Models of Programming Languages and Representations of Relations on Regular Languages with Application to the Static Determination of Dynamic Aliasing Properties of Data*. PhD thesis, LIX, Ecole Polytechnique, Palaiseau, France, 1992.

[14] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the IEEE 1992 International Conference on Computer Languages*, San Fransisco, California, pages 2–13, April 1992.

[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*, August 1986.

[16] P. Granger. Static analysis on linear congruence equalities among variables of a program. In *TAPSOFT'91*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192. Springer Verlag, 1991.

[17] Williams Ludwell Harrison. The interprocedural analysis and automatic parallelisation of scheme programs. *Lisp and Symbolic Computation*, 2(3):176–396, October 1989.

[18] Paul Hudak and Jonathan Young. Collecting interpretations of expressions. *ACM Transactions on Programming Languages and Systems*, 13(2):269–190, April 1991.

[19] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Proceedings of the* $22^{\text{nd}}$ *ACM Symposium on Principles of Programming Languages*, 1995.

[20] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, pages 244–256, January 1979.

[21] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[22] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[23] Torben Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Diaz and F. Orejas, editors, *TAPSOFT '89. Proc. Int. Conf. Theory and Practice of Software Development, Barcelona, Spain, March 1989 (Lecture Notes in Computer Science, vol. 352)*, pages 298–312. Springer-Verlag, 1989.

[24] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, Department of Computer Science, University of Edinburgh, Scotland, 1981.

[25] Flemming Nielson. Strictness analysis and denotational abstract interpretation. *Information and Computation*, 76(1):29–92, January 1988.

[26] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI–FN–19, Computer Science Department, Aarhus University, 1981.

[27] Olin Shivers. *Control-flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991.

[28] Joseph E. Stoy. *Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[29] Philip Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Third International Conference on Functional Programming and Computer Architecture*, 1987.