# Linear Logical Approximations *

Robert J. Simmons Frank Pfenning
Carnegie Mellon University
{rjsimmon,fp}@cs.cmu.edu

## Abstract

The abstract interpretation of programs relates the exact semantics of a programming language to an approximate semantics that can be effectively computed. We show that, by specifying operational semantics in a bottom-up, linear logic programming language – a technique we call *substructural operational semantics* (SSOS) – manifestly sound program approximations can be derived by simple and intuitive approximations of the logic program. As examples, we describe how to derive a simple alias analysis, 0CFA, and $k$CFA analysis from a substructural operational semantics of the relevant languages.

***Categories and Subject Descriptors*** D.1.6 [*Programming Techniques*]: Logic programming; F.3.2 [*Logics and Meanings of Program*]: Semantics of Programming Languages–Program analysis

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

***Keywords*** abstract interpretation, operational semantics, bottom-up linear logic programming

## 1. Introduction

A general recipe for constructing a sound program analysis is to (1) specify the operational semantics of the underlying programming language via an interpreter, and (2) specify a terminating approximation of the interpreter itself. This is the basic idea behind *abstract interpretation* (Cousout and Cousot 1977) which provides techniques for constructing approximations (for example, by exhibiting a Galois connection between concrete and abstract domains). The correctness proof establishes the appropriate relationship between the concrete and abstract computations, and shows termination. We need to vary both the specification of the operational semantics and the form of the approximation in order to obtain various kinds of program analyses, sometimes with considerable ingenuity.

In this paper we propose a new class of instances of the general schema of abstract interpretation. The interpreters are specified in logical form, using the recently proposed *substructural operational semantics* (SSOS) (Cervesato et al. 2002; Pfenning 2004). Briefly, we represent the state of the interpreter as a collection of linear propositions[1] and its computation steps as inference rules, to be applied in a forward-chaining style. This kind of specification creates the opportunity for describing, by logical means, approximations which are correct by construction. We explore several such logical approximations: (1) replacing linear predicates by persistent ones, which yields a form of collecting semantics, (2) eliminating existential quantification by Skolemization, and (3) introducing equations that collapse infinite domains into finite ones. The resulting approximations are now (non-linear) bottom-up logic programs which can be run to saturation, generalizing proposals by McAllester and Ganzinger (McAllester 2002; Ganzinger and McAllester 2002) with certain higher-order features.

We illustrate our approach by deriving alias analysis as presented by Aho et al. (2007), 0CFA (Shivers 1988) and $k$CFA in the form presented by Van Horn and Mairson (2008) from natural SSOS specifications. While defining these specific approximations requires insight, their correctness proofs do not, because they follow from a general metatheorem justifying the kinds of approximations we make, together with straightforward termination arguments.

## 2. A simple example

As simple illustration of our techniques, both for specification and approximation, we define a stateful system that generates infinitely many distinct names and relates them with a successor predicate. We show how to approximate this process to obtain the (a) the natural numbers, (b) the natural numbers modulo 2, and (c) the two-element abstract domain with zero and positive numbers.

Consider a linear proposition $\mathsf{at}(x)$, representing a piece of stateful information (that we are at position $x$), and a persistent proposition $!\mathsf{next}(x, y)$ which means that $x$ is followed by $y$.[2] Initially, we are at the origin, $\mathsf{at}(0)$, and there is no next place. If we are at position $x$ we can transition by creating a new place $y$, moving to it, and asserting that $y$ follows $x$. In linear logic, this process can be specified as

$$\forall x.\, \mathsf{at}(x) \multimap \exists y.\, !\mathsf{next}(x, y) \otimes \mathsf{at}(y).$$

We write this directly as an inference rule in the form

$$\frac{\mathsf{at}(x)}{\begin{array}{l}!\mathsf{next}(x, y)\\ \mathsf{at}(y)\end{array}}\ \exists y.$$

Applying this rule in a state will consume a linear proposition $\mathsf{at}(x)$ and add the linear proposition $\mathsf{at}(y)$ for a new $y$, as well

* This work was supported by the Fundação para a Ciência e a Tecnologia (FCT), Portugal, under a grant from the Information and Communications Technology Institute (ICTI) at Carnegie Mellon University, and by a National Science Foundation Graduate Resource Fellowship for the first author.

Copyright ACM, 2009. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *PEPM'09*, January 19–20, 2009, Savannah, Georgia, USA.
http://doi.acm.org/10.1145/1480945.1480949

[1] In the sense of linear logic (Girard 1987), not linear arithmetic.

[2] We always write linear propositions as plain names and persistent propositions preceded by an exclamation point, as one would in linear logic.

as the persistent proposition $!\mathsf{next}(x, y)$. Each state we can reach is characterized by a collection of available place names, persistent, and linear propositions. It is easy to see that the state can only evolve as follows:

| Names | Persistent Propositions | Linear Propositions |
|---|---|---|
| $0$ | | $\mathsf{at}(0)$ |
| $0, y_1$ | $!\mathsf{next}(0, y_1)$ | $\mathsf{at}(y_1)$ |
| $0, y_1, y_2$ | $!\mathsf{next}(0, y_1), !\mathsf{next}(y_1, y_2)$ | $\mathsf{at}(y_2)$ |
| | ... | |

The combination of linearity and existential quantification for name generation in this form is well-known, and has been used extensively for specification (e.g., in LO (Bozzano and Delzanno 2002), MSR (Cervesato and Scedrov 2006), or CLF (Cervesato et al. 2002; Watkins et al. 2008)). We can see that this system generates a structure isomorphic to the natural numbers, with each number receiving a distinct name and next as the successor predicate.

### 2.1 Approximating linearity by persistence

We can create a kind of collecting semantics by making all linear predicates persistent. The approximate rule reads

$$\frac{!\mathsf{at}(x)}{\begin{array}{l}!\mathsf{next}(x, y)\\ !\mathsf{at}(y)\end{array}}\ \exists y.$$

and the state might evolve as follows:

| Names | Persistent Propositions | |
|---|---|---|
| $0$ | | $!\mathsf{at}(0)$ |
| $0, y_1$ | $!\mathsf{next}(0, y_1),$ | $!\mathsf{at}(0), !\mathsf{at}(y_1)$ |
| $0, y_1, y_2$ | $!\mathsf{next}(0, y_1), !\mathsf{next}(y_1, y_2),$ | $!\mathsf{at}(0), !\mathsf{at}(y_1), !\mathsf{at}(y_2)$ |
| | ... | |

At first this might seem like a useful approximation, with the at predicate now being true for all generated names, representing precisely the natural number predicate, but this is not the case. In the second state above we can also apply our rule again with the same premise, $!\mathsf{at}(0)$, which is now still part of the state since we have made it persistent. This yields a *new* constant, say $z_1$, and the state

$$!\mathsf{next}(0, y_1), !\mathsf{next}(0, z_1), !\mathsf{at}(0), !\mathsf{at}(y_1), !\mathsf{at}(z_1)$$

with names $0, y_1, z_1$. Because of the generative nature of the existential quantifier and the persistent nature of the approximation, this can be repeated over and over again, creating a very poor approximation of the original system.

### 2.2 Eliminating existentials by Skolemization

If we inspect the logical form of the rule

$$\forall x.\, !\mathsf{at}(x) \multimap \exists y.\, !\mathsf{next}(x, y) \otimes !\mathsf{at}(y).$$

we notice a quantifier dependence that suggests Skolemization. Instead of generating a new name every time a rule is applied, we replace $y$ by a Skolem function $\mathsf{s}$ which depends on $x$.

$$\forall x.\, !\mathsf{at}(x) \multimap !\mathsf{next}(x, \mathsf{s}(x)) \otimes !\mathsf{at}(\mathsf{s}(x)),$$

or, in rule form

$$\frac{!\mathsf{at}(x)}{\begin{array}{l}!\mathsf{next}(x, \mathsf{s}(x))\\ !\mathsf{at}(\mathsf{s}(x))\end{array}}$$

Note that this does *not* preserve the meaning of the original rule (which is much more prolific), but approximates it by choosing a unique $y$ for any given $x$, calling it $\mathsf{s}(x)$.

Now the system can only evolve as follows, with 0 as the only name throughout:

$$\begin{array}{ll}
 & !\mathsf{at}(0)\\
!\mathsf{next}(0, \mathsf{s}(0)), & !\mathsf{at}(0), !\mathsf{at}(\mathsf{s}(0))\\
!\mathsf{next}(0, \mathsf{s}(0)), !\mathsf{next}(\mathsf{s}(0), \mathsf{s}(\mathsf{s}(0))), & !\mathsf{at}(0), !\mathsf{at}(\mathsf{s}(0)), !\mathsf{at}(\mathsf{s}(\mathsf{s}(0)))\\
\ldots &
\end{array}$$

Applying the rule again to $!\mathsf{at}(0)$ in the second or third state as before only infers persistent facts that are already known. By contraction on persistent propositions (each is recorded only once), this does not change the state.

The net results of the first two approximations is that we have generated a structure isomorphic to the natural numbers, including the successor (next) and natural number (at) predicates. However, this approximation does not yet terminate.

### 2.3 Approximation via equality

In order to generate a finite approximation we can now, for example, consider the natural numbers modulo 2, which corresponds to an abstract domain of even and odd numbers. We specify this with the equality $0 = \mathsf{s}(\mathsf{s}(0))$. After two steps we have (again, with 0 remaining the only name throughout):

$$\begin{array}{ll}
 & !\mathsf{at}(0)\\
!\mathsf{next}(0, \mathsf{s}(0)), & !\mathsf{at}(0), !\mathsf{at}(\mathsf{s}(0))\\
!\mathsf{next}(0, \mathsf{s}(0)), !\mathsf{next}(\mathsf{s}(0), 0)), & !\mathsf{at}(0), !\mathsf{at}(\mathsf{s}(0))
\end{array}$$

Applying our rule with premise $!\mathsf{at}(\mathsf{s}(0))$ to the second state yields the third state because the conclusion $!\mathsf{next}(\mathsf{s}(0), \mathsf{s}(\mathsf{s}(0))) = !\mathsf{next}(\mathsf{s}(0), 0)$ and $!\mathsf{at}(\mathsf{s}(\mathsf{s}(0))) = !\mathsf{at}(0)$ by the assumed equality. Any further rule applications from the last state will only generate facts already known, modulo our equality, so computation of the approximation terminates by *saturation*.

### 2.4 Equality and Skolemization

We can also exploit equality to streamline Skolemization and obtain a wider range of approximations. When Skolemizing, we leave the quantifier in place and just assert that the parameter be equal to a Skolem term.

$$\forall x.\, !\mathsf{at}(x) \multimap \exists y.\, y = \mathsf{s}(x) \otimes !\mathsf{next}(x, y) \otimes !\mathsf{at}(y)$$

Equalities are always persistent, once they are generated, so this means exactly the same as the earlier

$$\forall x.\, !\mathsf{at}(x) \multimap !\mathsf{next}(x, \mathsf{s}(x)) \otimes !\mathsf{at}(\mathsf{s}(x)).$$

The equality-based formulation of Skolemization suggests that it may not be necessary for the Skolem function to depend on *all* universally quantified variables. The approximation will be cruder if we make it depend on fewer, but it will still be sound. For example, if instead of the Skolem function $\mathsf{s}(x)$ we introduce a single Skolem constant $\mathsf{p}$ (for "positive") we obtain

$$\forall x.\, !\mathsf{at}(x) \multimap \exists y.\, y = \mathsf{p} \otimes !\mathsf{next}(x, y) \otimes !\mathsf{at}(y)$$

which is finitary and represents the two-element abstract domain consisting of 0 and positive numbers. The approximation again saturates in two steps, but with a different state recording that the successor of a positive number is positive.

$$\begin{array}{ll}
 & !\mathsf{at}(0)\\
!\mathsf{next}(0, \mathsf{p}), & !\mathsf{at}(0), !\mathsf{at}(\mathsf{p})\\
!\mathsf{next}(0, \mathsf{p}), !\mathsf{next}(\mathsf{p}, \mathsf{p}), & !\mathsf{at}(0), !\mathsf{at}(\mathsf{p})
\end{array}$$

## 3. Linear logical algorithms

In the example of the previous section we exploited linear logic to represent state change and name generation, which then served as the basis for our approximations. In this section we define a specific

fragment of linear logic and endow it with an operational semantics. This language is rich enough to support the specifications of concrete interpreters (presented in Sec. 4) which we can then approximate with the techniques sketched above to obtain program analyses (presented in Secs. 5, 6 and 7).

The connection between linear logic and logic programming dates back to Andreoli (1992) and has been explored in several other forms, including Hodas and Miller (1994), but with a few exceptions, most of this work relates linear logic to a backward-chaining (i.e., "top-down") style of logic programming.

Our language is a forward-chaining (i.e., "bottom-up") logic programming language where rules are exhaustively applied to derive new propositions until no more new propositions can be derived. It is effectively a restriction of Lollimon[3] (López et al. 2005), which integrates backward-chaining and forward-chaining linear logic programming, and a generalization of the language for specifying linear logical algorithms (Simmons and Pfenning 2008), which is purely forward-chaining. We deal with linear logical algorithms from an essentially operational perspective in this work; previous work describes the connection to linear logic in detail.

A program $\mathcal{P}$ is a set of rules $R$ with the following form:

$$\frac{\text{0 or more premises}}{\text{0 or more conclusions}}\ \exists x_1, \ldots, x_n$$

Each premise is either a *linear atomic proposition* or a *persistent atomic proposition*, and each conclusion is either a linear atomic proposition, persistent atomic proposition, or a *constraint*, and both conclusions and premises are interpreted as linear conjunctions. Atomic propositions are of the form $\mathsf{pred}(t_1, \ldots, t_n)$ where $\mathsf{pred}$ is some predicate and the $t_i$ are terms. A *constraint* is of the form $t = s$, where $t$ and $s$ are terms. Persistent atomic propositions are marked with an exclamation point.[4] Each rule $R$ has a set of schematic variables, which are implicitly universally quantified, and a set of existential variables $x_1, \ldots, x_n$. We require two properties of programs:

1. *Range restriction*: Every variable occurring in a conclusion must either appear strictly[5] in some premise or as one of the existentially bound parameters $x_1, \ldots, x_n$.
2. *Separation*: Every predicate must be used consistently, appearing either only in linear atomic propositions or only in persistent atomic propositions in a given program – therefore we speak of predicates, not just propositions, as being either linear or persistent.

The language that appears here is the language of Simmons and Pfenning (2008) extended with existential parameters, equational constraints, and a higher-order term language. The first two extensions – existential parameters and equational constraints – were introduced in Sec. 2; we will briefly discuss the implications of a higher-order term language, and then define the operational semantics of our linear logical algorithms language.

### 3.1 Skolemization and higher-order terms

We will be quite informal with the notation we use to describe the term languages used throughout this paper, but the underlying term language is a well-understood language based on the simply-typed $\lambda$-calculus as found in $\lambda$Prolog or Twelf. This means that, in addition to term constructors $\mathsf{a}(t_1, \ldots, t_n)$, the language of terms includes functions $\lambda x.t$. This allows us to express the binding structure of the languages we will discuss in the style of higher-order abstract syntax. We tacitly assume that all terms, substitutions, equations, and propositions are well-typed.

Therefore, when we write $\lambda\!\!\lambda x.e$ later in the paper, this should be understood as syntactic sugar for a term $\mathsf{lambda}(\lambda x.e(x))$, where $\lambda$ is actually a lambda in the term language and where $\mathsf{lambda}$ is a constructor with type $(\mathsf{tm} \to \mathsf{tm}) \to \mathsf{tm}$. Similarly, when we write substitution as $[y/x]e$, this corresponds to the term $e(y)$ – substitution in the object language (the language we are specifying) is expressed as application in the term language (the language we are using to write things down). For this reason, we require that each schematic variable have a *strict* occurrence in the premise (Pfenning and Schürmann 1998), which makes higher-order matching unitary and decidable (Schürmann 2000), even in the presence of definitional equations and dependent types.

A fairly common design pattern in saturating, forward-chaining logic programs is to enumerate the subterms of a term and then perform a computation on those subterms. Consider the problem of describing the subterms of untyped lambda-calculus terms provided by the grammar

$$e ::= x \mid e_1(e_2) \mid \lambda x.e$$

A purely persistent approach, where a new existential parameter is substituted into open terms, will not saturate for reasons already discussed: the rule on the right can be applied repeatedly, generating a new name each time.

$$\frac{!\mathsf{subterms}(e_1(e_2))}{\begin{array}{l}!\mathsf{subterms}(e_1)\\ !\mathsf{subterms}(e_2)\end{array}} \qquad \frac{!\mathsf{subterms}(\lambda x.e)}{!\mathsf{subterms}([y/x]e)}\ \exists y$$

However, if we Skolemize the second rule (calling the Skolem function $\mathsf{var}$) we obtain the following rule (and a saturating algorithm).

$$\frac{!\mathsf{subterms}(\lambda x.e)}{\begin{array}{l}!\mathsf{subterms}([y/x]e)\\ y = \mathsf{var}(\lambda x.e)\end{array}}\ \exists y$$

In this representation, a variable is effectively a *pointer* back to its binding site. In the rest of the paper, our default position when encountering a parameter substituted into a higher-order function $\lambda x.e$ will be to equate that parameter with $\mathsf{var}(\lambda x.e)$, even if the rule has other schematic variables.

### 3.2 Operational semantics

We begin with a few definitions that will allow us to specify the operational behavior of linear logical algorithms. First, we define *program states*, which describe single configurations of the parameters, equational constraints, linear, and persistent propositions, and *rule firings* which describe the ways in which a rule may be applied in a given state.

**Definition 1.** *A* program state $\mathcal{S}$ *is a tuple* $\langle \Sigma, \Lambda, \Gamma, \Delta \rangle$*, where* $\Sigma$ *is a set of* parameters *that have been introduced by existential quantification,* $\Lambda$ *is a system of equations,* $\Gamma$ *is a multiset of persistent propositions, and* $\Delta$ *is a multiset of linear propositions.*

We consider $\Lambda$ to be an unspecified set of equality constraints such that equational entailment, $\Lambda \vdash t = s$, is decidable. It is not necessary that $\Gamma$ be a multiset, since it represents a context of persistent propositions, but it simplifies the subsequent discussion and proofs to do so. We will write "$\Delta, \Gamma$" to represent the multiset union of $\Delta$ and $\Gamma$.

[3] Our language is not a precise fragment or Lollimon – the latter system treats the "polarity" of atomic propositions differently – but the difference is unimportant for the purposes of this paper.

[4] In (Simmons and Pfenning 2008) persistent atomic propositions were unmarked and linear atomic propositions were underlined – the difference is merely notational, though the exclamation point is intentionally suggestive of the exponential operator of linear logic.

[5] As explained in the next section.

**Definition 2.** *A* rule firing *is a tuple* $(\mathcal{S}, R, \Delta^*, \sigma)$*, where* $\mathcal{S} = \langle \Sigma, \Lambda, \Gamma, (\Delta, \Delta^*)\rangle$ *and* $\sigma$ *is a substitution of arbitrary terms (which may include the parameters in* $\Sigma$*) for the free variables of the rule* $R$*, if for every persistent premise* $A_i$ *in* $R$*, there is an* $A \in \Gamma$ *such that,* $\Lambda \vdash \sigma A_i = A$ *and if for every linear premise* $B_i$ *in* $R$*, there is a* distinct $B \in \Delta^*$ *s.t.* $\Lambda \vdash \sigma B_i = B$*.*

These two concepts give us the necessary machinery to define the *evolution* of a program state into another program state by firing a rule, which may add parameters, constraints, and propositions.

**Definition 3.** *An* evolution *of* $\mathcal{S} = \langle \Sigma, \Lambda, \Gamma, (\Delta, \Delta^*)\rangle$ *under a rule firing* $(\mathcal{S}, R, \Delta^*, \sigma)$ *is a new program state of the form* $\mathcal{S}' = \langle (\Sigma, \Sigma'), \Lambda', (\Gamma, \Gamma'), (\Delta, \Delta')\rangle$*, where there is one fresh parameter in* $\Sigma'$ *for each existential parameter* $x_i$ *in* $R$*, if, letting* $\delta$ *be the substitution* $\sigma$ *extended to substitute a unique parameter in* $\Sigma'$ *for each* $x_i$*,* $\Lambda'$ *is* $\Lambda$ *extended with the constraints in the conclusion of* $R$ *under* $\delta$ *and* $\Gamma'$ *and* $\Delta'$ *are the multisets of persistent and linear conclusions (respectively) of* $R$ *under* $\delta$*.*

An evolution as described above is *productive* if there are parameters in $\Sigma'$ that are not equal to terms that existed previously under $\Lambda'$, if $\Lambda'$ makes any terms equal that were not equal under $\Lambda$, if there are any propositions in $\Gamma'$ that are not equal to propositions in $\Gamma$ under $\Lambda'$, or if $\Delta'$ is nonempty. Then we define a *program trace* $\mathcal{S}_1, \ldots, \mathcal{S}_n$ to be a list of states, each of which is a productive evolution of the previous one under some rule firing. A *complete trace* is one where the final state $\mathcal{S}_n$ cannot take any productive transitions; borrowing terminology from Lollimon, such a program is said to have reached *quiescence*. This is an operational semantics without backtracking – at each step, one of possibly many rule firings is applied, and the trace continues without ever reconsidering this choice.

### 3.3 Approximate versions of programs

As described previously, a feature of linear logical algorithms is that turning all linear predicates into persistent predicates results in a new program that is a sound approximation of the original program; as described in the definition below, there are a number of other transformations that may be performed.

**Definition 4.** *A program* $\mathcal{P}_a$ *is an* approximate version *of a program* $\mathcal{P}$ *if all predicates in* $\mathcal{P}_a$ *are persistent and if, for each rule in* $\mathcal{P}_a$*, the existential parameters are identical to the existential parameters of the corresponding rule in* $\mathcal{P}$*, the premises are a subset of the premises of the corresponding rule in* $\mathcal{P}$*, and the conclusions are a superset of the conclusions of the corresponding rule in* $\mathcal{P}$*.*

We will formally define the correctness requirement in Sec. 8, where we also prove that any terminating approximate version of a program produces a valid abstraction of the original program; intuitively, however, correctness means that the approximate version of a program exhibits all the properties of the original program. Therefore, if the approximate version of a program reaches quiescence (or *saturation*, which is a more appropriate term in the absence of linear propositions), we can "read off" a conservative approximation of all the behaviors of the original program from the final state of the approximate program.

## 4. Substructural Operational Semantics

Substructural operational semantics (SSOS) was originally proposed by Pfenning (2004) as an alternative to structural operational semantics (SOS) for specifying the dynamic semantics of programming languages. While SOS is a powerful and successful method for specifying the dynamic semantics of programming languages, it is essentially non-modular. The transitions in the language are described by the relation $e_1 \mapsto e_2$, which is defined inductively. However, if we make the state of the system more complicated than just an expression, the relation must get similarly more complicated. In order for a store $\sigma$ to be part of the state, we must change the relation to $\langle \sigma_1, e_1\rangle \mapsto \langle \sigma_2, e_2\rangle$ and change every rule accordingly. If we then want to have a multiset of processes $\Delta$, as well as global state, we must change the relation to $\langle \sigma_1, (\Delta_1, e_1)\rangle \mapsto \langle \sigma_2, (\Delta_2, e_2)\rangle$ and again change every rule accordingly.

Proposed solutions to this problem, such as Modular Structural Operational Semantics, presented by Mosses (2004), identify a need for a notion of *ambient state*, but preserve the notion that the state transition relation $\mapsto$ is inductively defined. In the case of SSOS we define the transitions in the dynamic semantics of the programming language as transitions in a forward-chaining linear logic program; ambient state can be represented by the ambient context of linear and persistent propositions. In this section, we will introduce the concepts of SSOS style gradually, beginning by presenting a pure continuation-passing style language fragment where all intermediate values are named and all function calls are tail calls so that no stack is necessary and working our way towards an $A$-normal form language[6] with mutable pairs.

### 4.1 Pure continuation-passing style language

We begin by describing a pure continuation-passing style language. The expressions $e$ enforce that all computations immediately be named and allow only tail calls. The pure fragment of the language handles pairs and lambda abstractions:

$$\begin{array}{lcl} c & ::= & \mathsf{fst} \mid \mathsf{snd} \\ e & ::= & y \mid \mathsf{let}\ x = \langle\!\langle y_1, y_2\rangle\!\rangle\ \mathsf{in}\ e \mid \mathsf{let}\ x = c\ y\ \mathsf{in}\ e \\ & & \mid \mathsf{let}\ x = \text{Λ}x_0.e_0\ \mathsf{in}\ e \mid y_1(y_2) \\ v & ::= & () \mid \langle v_1, v_2\rangle \mid \lambda x.e \end{array}$$

### 4.2 Evaluation

Evaluation is handled by linear forward-chaining rules, and the crucial propositions are $\mathsf{eval}_d(e)$, which is a linear proposition declaring that the expression $e$ is evaluating with the *destination* $d$, and $\mathsf{bind}(x, v)$, which is a persistent proposition declaring that the parameter $x$ is associated with the value $v$. Taken as a whole, the persistent propositions of the form $\mathsf{bind}(x, v)$ make up an *environment* mapping parameters to variables. The rule for evaluating a $\mathsf{fst}$ redex is

$$\frac{\begin{array}{l}\mathsf{eval}_d(\mathsf{let}\ x = \mathsf{fst}\ y_1\ \mathsf{in}\ e) \\ !\mathsf{bind}(y_1, \langle v_1, v_2\rangle)\end{array}}{\begin{array}{l}!\mathsf{bind}(y, v_1) \\ \mathsf{eval}_d([y/x]e)\end{array}}\ \exists y$$

This rule should be read "if there is a linear proposition describing a computation with destination $d$ evaluating $\mathsf{let}\ x = \mathsf{fst}\ y_1\ \mathsf{in}\ e$, where $y_1$ is bound to the pair $\langle v_1, v_2\rangle$, the linear proposition may be consumed and replaced by a computation with destination $d$ evaluating $[y/x]e$, where $y$ is a newly created parameter bound to $v_1$."

It is important that the $y$ in the conclusion $!\mathsf{bind}(y, v_1)$ is *newly created* – over the course of a program's execution, the above rule might be fired multiple times with the same premises, and so substituting a newly created parameter into the conclusion ensures that we do not end up with an over-approximation in which a variable is bound to two different values. The program will maintain the invariant that each parameter $y$ is associated with exactly one persistent proposition $!\mathsf{bind}(y, v_1)$.

The evaluation predicate takes two arguments – the fact that $d$ is subscript is just a bit of notation, and we could equivalently have written $\mathsf{eval}(e, d)$. We will focus on destinations $d$ in greater

[6] As described by Flanagan et al. (1993).

detail, but for now we can observe that multiple linear propositions, each of which represent a process, can appear simultaneously; we *already* have a specification that can deal with either one process or a multiset of processes! Destinations can be seen as just tags that distinguish different processes from one another.

Destinations also allow us to mark the value that is ultimately returned from an expression. The rule for the final case where we are evaluating a bare parameter $y$ is

$$\frac{\mathsf{eval}_d(y) \quad !\mathsf{bind}(y, v)}{\mathsf{return}_d(v)}$$

This rule should be read "if there is a linear proposition of the form $\mathsf{eval}_d(y)$, where $y$ is bound to $v$, it may be consumed and replaced with a linear proposition of the form $\mathsf{return}_d(v)$."

The other two reduction rules for evaluating the pure fragment are as follows:

$$\frac{\mathsf{eval}_d(\mathsf{let}\ x = \mathsf{snd}\ y_1\ \mathsf{in}\ e) \quad !\mathsf{bind}(y_1, \langle v_1, v_2\rangle)}{!\mathsf{bind}(y, v_2) \quad \mathsf{eval}_d([y/x]e)}\ \exists y \qquad \frac{\mathsf{eval}_d(y_1(y_2)) \quad !\mathsf{bind}(y_1, \lambda x.e) \quad !\mathsf{bind}(y_2, v_2)}{!\mathsf{bind}(y, v_2) \quad \mathsf{eval}_d([y/x]e)}\ \exists y$$

Beyond that, we have the two further rules for creating pair values and function values from expressions.

$$\frac{\mathsf{eval}_d(\mathsf{let}\ x = \langle\!\langle y_1, y_2\rangle\!\rangle\ \mathsf{in}\ e) \quad !\mathsf{bind}(y_1, v_1) \quad !\mathsf{bind}(y_2, v_2)}{!\mathsf{bind}(y, \langle v_1, v_2\rangle) \quad \mathsf{eval}_d([y/x]e)}\ \exists y$$

$$\frac{\mathsf{eval}_d(\mathsf{let}\ x = \lambda x_0.e_0\ \mathsf{in}\ e)}{!\mathsf{bind}(y, \lambda x_0.e_0) \quad \mathsf{eval}_d([y/x]e)}\ \exists y$$

In each case that performs some computation, one of the values bound to a variable must take a certain form. A proposition such as $\mathsf{eval}_d(\mathsf{let}\ x' = \mathsf{fst}\ y\ \mathsf{in}\ e')$ would be in an undesirable state if $y$ was bound to $\lambda x.e$ – this would correspond to a "stuck states" in SOS formulations.

### 4.3 Extension to mutable state

If we want to assign to our pairs in the style of Lisp cons cells, we need to replace the rules that dealt with pairs before – pairs must be heap-allocated in order to be mutable – but we do not have to change any other rules. Our syntax is extended with a single construct for assignment, and we have a new value of heap locations:

$$\begin{array}{rcl} e & ::= & \ldots \mid y_1.c := y_2; e \\ v & ::= & \ldots \mid \mathsf{loc}(d) \end{array}$$

Locations are runtime artifacts that are introduced by evaluating expressions that create pairs, thus allocating space on a heap represented by that destination. We write location values as $\mathsf{loc}(d)$, where the argument $d$ is a destination that is now being used to mark the location of mutable data instead of marking the destination of a computation, which was up until now our only use of destinations. The rule for creating a new pair is now as follows:

$$\frac{\mathsf{eval}_d(\mathsf{let}\ x = \langle\!\langle y_1, y_2\rangle\!\rangle\ \mathsf{in}\ e) \quad !\mathsf{bind}(y_1, v_1) \quad !\mathsf{bind}(y_2, v_2)}{!\mathsf{bind}(y, \mathsf{loc}(d')) \quad \mathsf{store}_{d'}(\mathsf{fst}, v_1) \quad \mathsf{store}_{d'}(\mathsf{snd}, v_2) \quad \mathsf{eval}_d([y/x]e)}\ \exists y, d'$$

This rule should be read "if there is a proposition of the form $\mathsf{eval}_d(\mathsf{let}\ x = \langle\!\langle y_1, y_2\rangle\!\rangle\ \mathsf{in}\ e)$ with $y_1$ bound to $v_1$ and $y_2$ bound to $v_2$, then it may be consumed, which creates a new destination $d'$ where $v_1$ and $v_2$ are stored in the $\mathsf{fst}$ and the $\mathsf{snd}$ positions. Execution continues with $\mathsf{eval}_d([y/x]e)$, where $y$ is a new parameter bound to $\mathsf{loc}(d')$."

Just as we maintain the invariant that each parameter $y$ is associated with exactly one proposition $!\mathsf{bind}(y, v)$, we will maintain the invariant that each destination $d$ will be associated with a unique first projection $\mathsf{store}_d(\mathsf{fst}, v_1)$ and second projection $\mathsf{store}_d(\mathsf{snd}, v_2)$ in a given state. This invariant is more interesting because of mutability – we may consume a linear proposition $\mathsf{store}_d(c, v)$, but it will always be replaced by some other linear proposition $\mathsf{store}_d(c, v')$.

The two rules for assignment and dereference are as follows. Notice that both rules consume and create a linear proposition representing a memory cell – the dereference rule must "write back" the value in memory after it reads it, or else an attempt to read a location twice would result in a stuck state.

$$\frac{\mathsf{eval}_d(\mathsf{let}\ x = !\ y_1\ \mathsf{in}\ e) \quad !\mathsf{bind}(y_1, \mathsf{loc}(d')) \quad \mathsf{store}_{d'}(c, v)}{!\mathsf{bind}(y, v) \quad \mathsf{eval}_d([y/x]e) \quad \mathsf{store}_{d'}(c, v)}\ \exists y \qquad \frac{\mathsf{eval}_d(y_1.c := y_2; e) \quad !\mathsf{bind}(y_1, \mathsf{loc}(d')) \quad !\mathsf{bind}(y_2, v_2) \quad \mathsf{store}_{d'}(c, v)}{\mathsf{eval}_d(e) \quad \mathsf{store}_{d'}(c, v_2)}$$

### 4.4 Introducing stacks

Now we are ready to step away from a purely continuation-passing style by introducing non-tail calls, which requires introducing the notion of a stack of frames waiting for functions to return. We will extend the syntax of expressions as follows:

$$e \quad ::= \quad \ldots \mid \mathsf{let}\ x = y_1(y_2)\ \mathsf{in}\ e$$

We implement stacks using the same destinations that we used before for representing mutable state – while linear propositions of the form $\mathsf{store}_d(\ldots)$ held values, linear propositions of the form $\mathsf{comp}_d(\ldots)$ will hold *suspended computations*. A linear proposition of the form $\mathsf{comp}_d(\mathsf{let}\ x = d'\ \mathsf{in}\ e)$ means that the expression $e$ (with $x$ free) will not evaluate at destination $d$ until some value is returned to destination $d'$. A value will be returned to that destination precisely when the function call returns, and so a stack frame is a suspended computation waiting on a function to return. There are two rules for implementing non-tail calls:

$$\frac{\mathsf{eval}_d(\mathsf{let}\ x = y_1(y_2)\ \mathsf{in}\ e) \quad !\mathsf{bind}(y_1, \lambda x_0.e_0) \quad !\mathsf{bind}(y_2, v_2)}{!\mathsf{bind}(y_0, v_2) \quad \mathsf{eval}_{d'}([y_0/x_0]e_0) \quad \mathsf{comp}_d(\mathsf{let}\ x = d'\ \mathsf{in}\ e)}\ \exists y_0, d'$$

$$
\begin{array}{lcl}
c & ::= & \mathsf{fst} \mid \mathsf{snd} \\
e & ::= & y \mid \mathsf{let}\ x = \lambda\!\!\lambda x_0.{e_0}^{l_0}\ \mathsf{in}\ e^l \mid \mathsf{let}\ x = y_1(y_2)\ \mathsf{in}\ e^l \mid \mathsf{let}\ x = \langle\!\langle y_1, y_2 \rangle\!\rangle\ \mathsf{in}\ e^l \mid \mathsf{let}\ x = \ c\ y_1\ \mathsf{in}\ e^l \mid y_1.c := y_2; e^l \\
v & ::= & \lambda x.e^l \mid \mathsf{loc}(d)
\end{array}
$$

$$
\frac{\mathsf{eval}_d(y) \quad !\mathsf{bind}(y, v)}{\mathsf{return}_d(v)}
$$

$$
\frac{\mathsf{eval}_d(\mathsf{let}\ x = \lambda\!\!\lambda x_0.{e_0}^{l_0}\ \mathsf{in}\ e^l)}{!\mathsf{bind}(y, \lambda x_0.{e_0}^{l_0}) \quad \mathsf{eval}_d([y/x]e)}\ \exists y
$$

$$
\frac{\mathsf{eval}_d(\mathsf{let}\ x = y_1(y_2)\ \mathsf{in}\ e^l) \quad !\mathsf{bind}(y_1, \lambda x_0.{e_0}^{l_0}) \quad !\mathsf{bind}(y_2, v)}{!\mathsf{bind}(y_0, v) \quad \mathsf{eval}_{d'}([y_0/x_0]e_0) \quad \mathsf{comp}_d(\mathsf{let}\ x = d'\ \mathsf{in}\ e^l)}\ \exists y_0, d'
$$

$$
\frac{\mathsf{comp}_d(\mathsf{let}\ x = d'\ \mathsf{in}\ e^l) \quad \mathsf{return}_{d'}(v)}{!\mathsf{bind}(y, v) \quad \mathsf{eval}_d([y/x]e)}\ \exists y
$$

$$
\frac{\mathsf{eval}_d(\mathsf{let}\ x = \langle\!\langle y_1, y_2 \rangle\!\rangle\ \mathsf{in}\ e^l) \quad !\mathsf{bind}(y_1, v_1) \quad !\mathsf{bind}(y_2, v_2)}{!\mathsf{bind}(y, \mathsf{loc}(d')) \quad \mathsf{store}_{d'}(\mathsf{fst}, v_1) \quad \mathsf{store}_{d'}(\mathsf{snd}, v_2) \quad \mathsf{eval}_d([y/x]e)}\ \exists y, d'
$$

$$
\frac{\mathsf{eval}_d(\mathsf{let}\ x = \ c\ y_1\ \mathsf{in}\ e^l) \quad !\mathsf{bind}(y_1, \mathsf{loc}(d')) \quad \mathsf{store}_{d'}(c, v)}{!\mathsf{bind}(y, v) \quad \mathsf{eval}_d([y/x]e) \quad \mathsf{store}_{d'}(c, v)}\ \exists y
$$

$$
\frac{\mathsf{eval}_d(y_1.c := y_2; e^l) \quad !\mathsf{bind}(y_1, \mathsf{loc}(d')) \quad !\mathsf{bind}(y_2, v_2) \quad \mathsf{store}_{d'}(c, v)}{\mathsf{eval}_d(e) \quad \mathsf{store}_{d'}(c, v_2)}
$$

**Figure 1.** Summary of the rules for functions and mutable pairs in Sec. 4 with labels added as described in Sec. 5.

$$
\frac{\mathsf{comp}_d(\mathsf{let}\ x = d'\ \mathsf{in}\ e) \quad \mathsf{return}_{d'}(v)}{!\mathsf{bind}(y, v) \quad \mathsf{eval}_d([y/x]e)}\ \exists y
$$

This completes the introduction of a simple language in $A$-normal form with function types, product types, and mutable pairs. Much larger languages have been described in this style – the core of Concurrent ML was described in a similar style by Cervesato et al. (2002), and a number of language features such as futures and first-class continuations were explored in SSOS by Pfenning (2004). Our minimal language will form the basis of the alias analysis we present in the next section.

## 5. Alias analysis

Now that we have the elements of a substructural operational semantics in place, we are in a position to present an alias analysis for the language from the previous section, collected in Fig. 1.

First, however, we have to make a minor modification to the language that is motivated by the information we want to obtain from the alias analysis we perform. Alias analysis is described in terms of where variables were declared in a source program; therefore, it is important to have a way to uniquely identify a location in the source program. We can't use the trick of "naming all variables uniquely" that is used in many first-order presentations of syntax, because our use of higher-order abstract syntax means that our variables are subject to arbitrary renaming. Therefore, we will introduce labels $l$ into each expression that can be uniquely placed in the source program, attaching them to each continuation expression $e$ and to the body of each abstraction. In effect, this gives us a name for each line and each function. It has no effect on the operational semantics and will only come into play when we describe alias analysis. As an example, the syntax for declaring a new function is "$\mathsf{let}\ x = \lambda\!\!\lambda x_0.{e_0}^{l_0}\ \mathsf{in}\ e^l$," and the syntax for frames is "$\mathsf{let}\ x = d\ \mathsf{in}\ e^l$."

The full SSOS presentation of the resulting language is in Fig. 1.

### 5.1 Approximating to alias analysis

Now we will show how systematic approximations of the language in Fig. 1 produce a program for alias analysis. We want to be able to ask the following two questions about the relationships between pointers:

- Could a variable declared at label $l_1$ ever reference a pair created at label $l_2$?
- Could the first or second component of a pair created at label $l_1$ ever reference a pair created at label $l_2$?

Our approximation methodology makes the first two steps clear – first, we turn all linear predicates into persistent predicates, generating a *collecting semantics* as the program now "remembers" all intermediate values. However, as a linear logical algorithm this will not terminate; rules could be applied over and over again to create new parameters without ever reaching a point of saturation. Therefore, at any point where a variable $y$ is generated only to be substituted into an expression $e^l$ (with $x$ free), Skolemization (as described in Sec. 3.1) suggests substituting $\mathsf{var}(\lambda x.e^l)$ into $e$ instead, which we achieve by adding a new conclusion $y = \mathsf{var}(\lambda x.e^l)$. We treat this equality declaration as a *notational definition* as introduced by Pfenning and Schürmann (1998), which leads to decidable notion of equational entailment and higher-order matching (Schürmann 2000) if every schematic variable has a strict occurrence (which we enforce for our language).

After we have followed these two standard steps, we still must decide how to deal with the destination $d'$ introduced in the rule for function calls and the similar destination introduced in the rule for pair allocation. To get the results that we want, one solution is to equate the destination with the label $l_0$ naming the function in the case of function calls, and to equate the destination with the label $l$ naming the line in the pair allocation case. The result is Fig. 2. When we have performed exhaustive bottom-up reasoning using the rules written there, the result gives us answers to the questions we posed above:

- The variable declared at label $l_1$ might point to a pair created at label $l_2$ if $\mathsf{bind}(\mathsf{var}(\lambda x.e^{l_1}), \mathsf{loc}(l_2))$ appears in the saturated database.
- The first component of a pair created at label $l_1$ might reference a pair created at label $l_2$ if $\mathsf{store}_{l_1}(\mathsf{fst}, \mathsf{loc}(l_2))$ appears in the saturated database, and likewise for the second component.

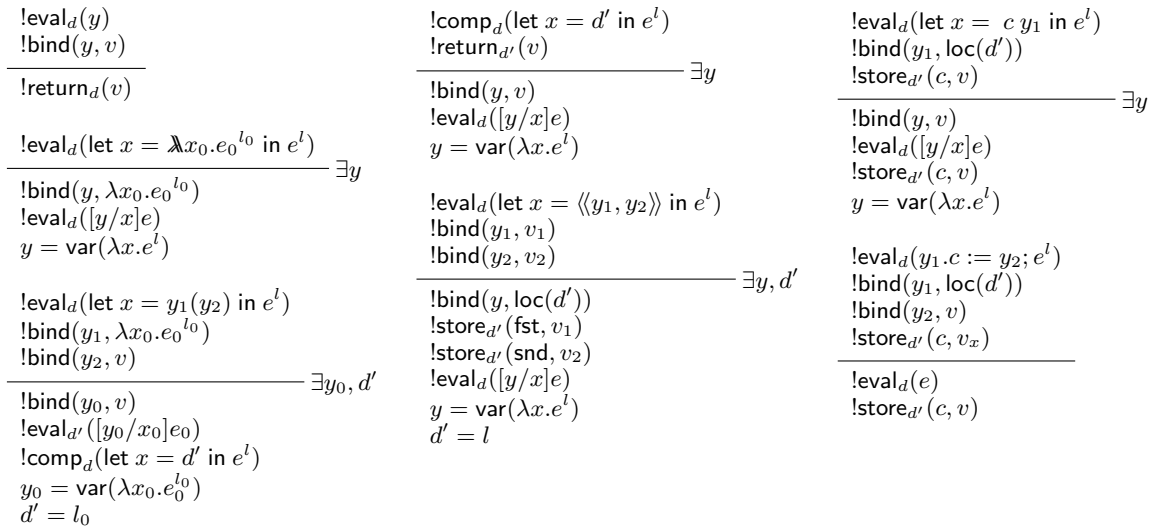


**Figure 2.** The language from Fig. 1 transformed into an alias analysis by Skolemizing all existential parameters and making all propositions persistent.

The termination of this analysis needs to be justified; this justification comes from the fact that there are only a finite number of values that can arise – one value $\mathsf{loc}(d)$ for each variable declaration, and one value $\lambda x_0.e_0^{l_0}$ for each function declaration. The evaluation propositions only deal with subterms of the original program, and the possible destinations $d$ are limited by the labels in the original program. Therefore, for any finite input program we can give a finite bound to the size of the saturated database of propositions, which is enough to ensure termination.

The resulting analysis can be seen as an adaptation of the pointer analysis in Chapter 12.4 of (Aho et al. 2007), where alias analysis is presented as a bottom-up logic program, to a functional language with first-class functions. The main difference is that this analysis takes control flow into account, only analyzing reachable parts of the program. Some extra machinery is also necessary to take into account our language's first-class functions that return values.

## 6. Control flow analysis

Programming languages are not generally specified in the $A$-normal form considered thus far in the paper. A more natural presentation (though one that maintains separate syntactic classes for expressions and values) looks like this:

$$\begin{array}{lcl} e & ::= & x \mid \lambda\!\!\lambda x.e \mid e_1(e_2) \\ v & ::= & \lambda x.e \end{array}$$

We can describe the substructural operational semantics of this language in two ways. We could add an expression to the previous language "let $x = e'$ in $e$," which allows the creation of arbitrary stack frames, and then define a translation into the previous language that names all subterms and makes evaluation order explicit.

However, we don't want to define such a translation for every language we define in SSOS, so we extend the language of *frames* to include "$d_1(e_2)$," an application waiting for its function position to be evaluated at $d_1$, "$v_1(d_2)$," an application waiting for its argument to be evaluated at $d_2$, and $\mathsf{call}(d_0)$, a frame waiting for a function call to return. The resulting SSOS definition is described by the six rules in Fig. 3.

The question addressed by 0CFA is, "for any given call site in the the source program, what are the functions that might be invoked at that location?" However, the question is often answered by answering the more general question, "what values do each subexpression and each function call evaluate to in the course of evaluating the program?" This is the insight that we use to derive the 0CFA analysis in Fig. 4.

As our methodology suggests, we make all predicates persistent, and then equate the parameter $y_0$ with the Skolemization of the function it is being substituted into, $\mathsf{var}(\lambda x.e)$. This leaves three destination parameters in three different rules that must be equated with pre-existing terms. In each case, we equate the destination with the relevant subexpression, so that instead of the linear proposition $\mathsf{return}_d(v)$ representing the return of a value to a destination, we have the persistent proposition !$\mathsf{return}_e(v)$ directly expressing that the subexpression $e$ might evaluate to $v$ during the course of evaluating the program. The answer to the relevant question of control flow analysis is then that a function $\lambda\!\!\lambda x.e$ might be invoked at a call site $e_1(e_2)$ if $\mathsf{return}_{e_1}(\lambda x.e)$ appears in the saturated database. The comment made about reachability in alias analysis is relevant here as well – if a function in the source program is only ever invoked with a non-terminating argument, then that function will not be analyzed, as it would be in other presentations of 0CFA.

There is one other caveat to this analysis. In a 0CFA analysis of the source program "let $x = \lambda\!\!\lambda y.y$ in $(x(x))(v)$" (for some value $v$) we would expect to learn that two arguments are passed to the function $\lambda\!\!\lambda y.y$: first, the function itself, and second, the value $v$. In contrast, if we take the program "$((\lambda\!\!\lambda x.x)(\lambda\!\!\lambda y.y))v$," we would expect 0CFA to notice that only $\lambda\!\!\lambda y.y$ is passed to the function $\lambda\!\!\lambda x.x$ and that only $v$ is passed to the function $\lambda\!\!\lambda y.y$. However, because of our use of higher-order abstract syntax, $\lambda\!\!\lambda x.x$ and $\lambda\!\!\lambda y.y$ are syntactically identical (names of bound variables don't matter), so in order to get the expected results from the second program, we would need to add distinct labels to terms as we did for alias analysis.

## 7. Instrumented interpreters and $k$CFA

In this section we develop a SSOS presentation of a language from a presentation of Van Horn and Mairson's lambda calculus interpreter, and then following our methodology to derive the same $k$CFA analysis discussed in (Van Horn and Mairson 2007, 2008).

$$\frac{\mathsf{eval}_d(x)\ \ !\mathsf{bind}(x,v)}{\mathsf{return}_d(v)} \qquad \frac{\mathsf{eval}_d(\lambda\!\!\lambda x.e)}{\mathsf{return}_d(\lambda x.e)} \qquad \frac{\mathsf{eval}_d(e_1(e_2))}{\mathsf{eval}_{d_1}(e_1)\ \ \mathsf{comp}_d(d_1(e_2))}\exists d_1 \qquad \frac{\mathsf{comp}_d(d_1(e_2))\ \ \mathsf{return}_{d_1}(v_1)}{\mathsf{eval}_{d_2}(e_2)\ \ \mathsf{comp}_d(v_1(d_2))}\exists d_2 \qquad \frac{\mathsf{comp}_d((\lambda x.e)(d_2))\ \ \mathsf{return}_{d_2}(v_2)}{!\mathsf{bind}(y_0,v_2)\ \ \mathsf{eval}_{d_0}([y_0/x]e)\ \ \mathsf{comp}_d(\mathsf{call}(d_0))}\exists y_0,d_0 \qquad \frac{\mathsf{comp}_d(\mathsf{call}(d_0))\ \ \mathsf{return}_{d_0}(v)}{\mathsf{return}_d(v)}$$

**Figure 3.** A SSOS interpreter for the lambda calculus.

$$\frac{!\mathsf{eval}_d(x)\ \ !\mathsf{bind}(x,v)}{!\mathsf{return}_d(v)} \qquad \frac{!\mathsf{eval}_d(\lambda\!\!\lambda x.e)}{!\mathsf{return}_d(\lambda x.e)} \qquad \frac{!\mathsf{eval}_d(e_1(e_2))}{!\mathsf{eval}_{d_1}(e_1)\ \ !\mathsf{comp}_d(d_1(e_2))\ \ d_1=e_1}\exists d_1 \qquad \frac{!\mathsf{comp}_d(d_1(e_2))\ \ !\mathsf{return}_{d_1}(v_1)}{!\mathsf{eval}_{d_2}(e_2)\ \ !\mathsf{comp}_d(v_1(d_2))\ \ d_2=e_2}\exists d_2 \qquad \frac{!\mathsf{comp}_d((\lambda x.e)(d_2))\ \ !\mathsf{return}_{d_2}(v_2)}{!\mathsf{bind}(y_0,v_2)\ \ !\mathsf{eval}_{d_0}([y_0/x]e)\ \ !\mathsf{comp}_d(\mathsf{call}(d_0))\ \ y_0=d_0=\mathsf{var}(\lambda x.e)}\exists y_0,d_0 \qquad \frac{!\mathsf{comp}_d(\mathsf{call}(d_0))\ \ !\mathsf{return}_{d_0}(v)}{!\mathsf{return}_d(v)}$$

**Figure 4.** A 0CFA analysis for the lambda calculus based on the SSOS interpreter in Fig. 3.

### 7.1 Van Horn and Mairson's interpreter

Van Horn and Mairson (2007, 2008) describe an interpreter for a $\lambda$-calculus that forms the basis for their discussion of $k$CFA. The abstract syntax is first-order – variables are just labels, and we adopt the traditional convention that every variable in the input program is distinct. Every term is additionally tagged with a label $l$. The syntax of the language is as follows:

$$\begin{array}{rrcl} \textit{Tagged expressions} & e & ::= & t^l \\ \textit{Untagged terms} & t & ::= & x \mid e_1(e_2) \mid \lambda\!\!\lambda(x,e) \\ \textit{Values} & v & ::= & \lambda(x,e,ce) \\ \textit{Frames} & f & ::= & l_1(l_2) \mid \mathsf{call}(l_0) \end{array}$$

The program on the left-hand-side of Fig. 5 is a literal translation of the "instrumented interpreter" by Van Horn and Mairson (2008), with one exception – rather than evaluating first the left-hand-side of an application and then evaluating the right hand side, the rule for application is as follows:

$$\frac{\mathsf{eval}(t_1^{l_1}(t_2^{l_2}),l,\delta,ce)}{\mathsf{eval}(t_1,l_1,\delta,ce)\ \ \mathsf{eval}(t_2,l_2,\delta,ce)\ \ \mathsf{comp}(l_1(l_2),l,\delta,ce)}$$

This rule means that both parts of the application are evaluated *simultaneously*. This change is completely non-essential, and is used here only to reduce the number of required rules to five from the six we would need otherwise.

This interpreter's evaluation predicate is $\mathsf{eval}(t,l,\delta,ce)$, where $t$ is an expression, $l$ is the label that used to be attached to the term $t$, and $\delta$ is a "contour," a stack of locations representing the list of function calls that have been made up to this point. Together $l$ and $\delta$ stand in for the destinations that we have used previously, as we will discuss in the next section. Finally, $ce$ is the "current environment," a finite map from variable names to contours. Finite maps do not fall within the linear logical algorithms language, but we can easily enough describe the required operations. We need lookup $ce(x)$, which returns the contour associated with $x$ in $ce$, extension $ce[x \mapsto l]$ which adds a new mapping to the finite map (or overrides the one that was already there), and restriction $ce|^t$, which returns the sub-map of $ce$ whose domain is the free variables of $t$.

A notable point about this presentation is that we can perform our approximation methodology of turning linear predicates into persistent predicates and we are left with *the same algorithm* – if we start with the single proposition $\mathsf{eval}(t,l,\mathsf{nil},\mathsf{emp})$, where $\mathsf{nil}$ is an empty list and $\mathsf{emp}$ is an empty map, the output in terms of what (if any) proposition $\mathsf{return}(v,l,\mathsf{nil})$ is eventually calculated is the same, regardless of whether $\mathsf{eval}$, $\mathsf{comp}$, and $\mathsf{return}$ are all linear or all persistent predicates.

### 7.2 SSOS-style interpreter

We can avoid finite maps and the first-order treatment of binding by rewriting the specification using higher-order abstract syntax, parameters, and an ambient notion of state as presented in the middle column in Fig. 5. The syntax is as follows:

$$\begin{array}{rcl} e & ::= & t^l \\ t & ::= & e_1(e_2) \mid \lambda\!\!\lambda x.e \\ v & ::= & \lambda x.e \\ f & ::= & l_1(l_2) \mid \mathsf{call}(l_0) \end{array}$$

What we have is essentially a substructural operational semantics definition of a language with a very different notion of destination-passing. Instead of using parameters to associate each portion of the computation with a unique destination, the combination of program labels and contours gives each computation a destination in a direct style.

### 7.3 Collecting semantics and $k$CFA

Consider the difference between the second premise of the first rule (the one that handles variables) in the first two columns of Fig. 5. In the Van Horn-style interpreter it is $\mathsf{bind}(x,ce(x),v)$, whereas in the SSOS-style interpreter it is $\mathsf{bind}(x,v)$. In the Van Horn-style interpreter, the current environment $ce$ is carried around precisely for this case – in order to locate the correct value, it is not enough to specify the variable, we must also know the variable's unique calling context, represented by the contour $ce(x) = \delta'$. In the SSOS-style presentation the dynamic creation of parameters that are uniquely associated with values ensures this property.

The (exact) collecting interpreter on the right-hand side of Fig. 5 therefore substitutes into a function, not just $\mathsf{var}(\lambda x.e)$ as before, but $\mathsf{var}(\lambda x.e,\delta)$, where $\delta$ is the calling contour. This ensures that each new binding is unique, and therefore that when we encounter a variable we will have all the information necessary to find the unique value associated with it.

This gives us the handle we need to specify $k$CFA. In Van Horn and Mairson's $k$CFA interpreter, at every point where a contour $\delta$ is extended, the list is truncated to contain $k$ elements at most. In our framework, we can just add the constraint that two lists where

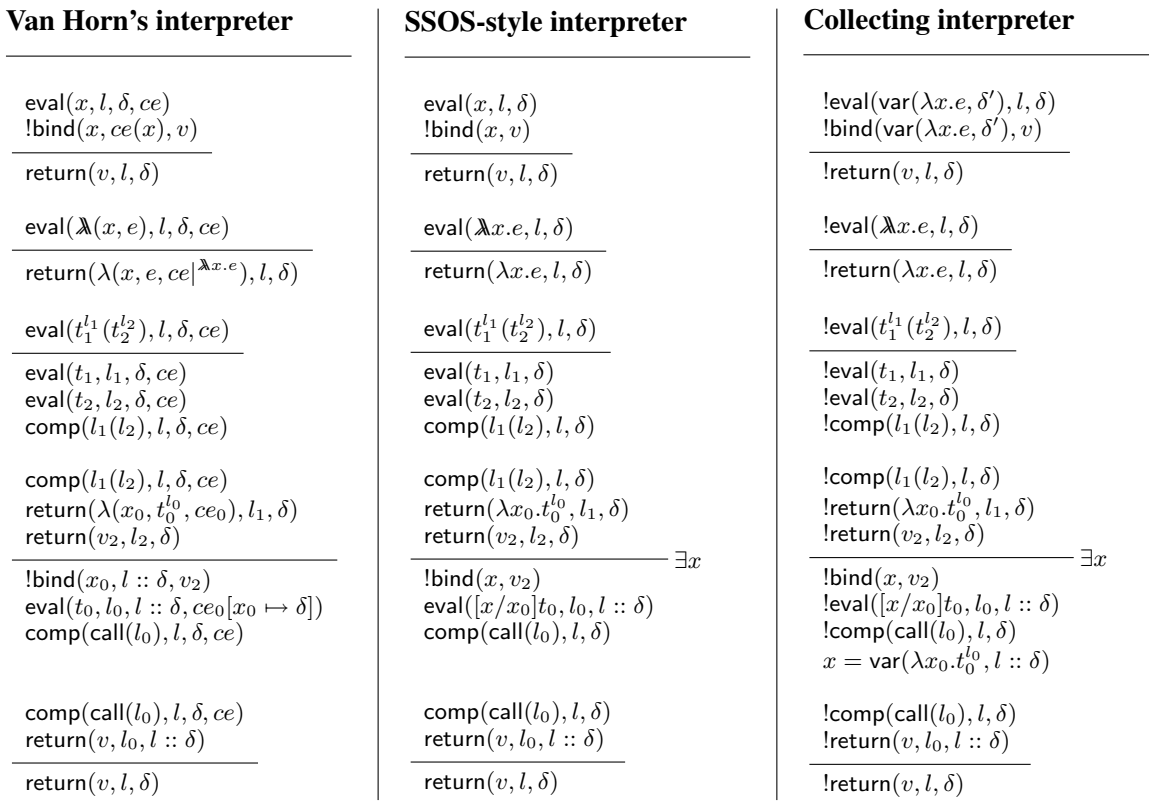


**Figure 5.** Comparing Van Horn and Mairson's instrumented interpreter (left) to an SSOS interpreter that does not use destination-passing (center) and a collecting interpreter that, with an additional equality constraint, captures $k$CFA as described in Sec. 7 (right).

the first $k$ elements are equal are, themselves, equal. Under such a constraint, our collecting semantics becomes a $k$CFA analysis as described by Van Horn and Mairson (2008). For instance, if we start the constraint system with the constraint that for all $a$, $l_1$, and $l_2$, we have $a :: l_1 = a :: l_2$, the result will be 1CFA. Of course, if if we treat all lists as equal ($k = 0$), we might as well just use the same Skolem function that gave us 0CFA in the previous section.

## 8. Correctness of approximation

We now prove that the approximation techniques we have used in this paper make sense generally. We will first define a notion of abstraction, and then state the theorem that any approximate version of a program, if terminating, produces an abstraction, thus capturing all the behaviors of the original program.

**Definition 5.** *A state $\langle \Sigma_g, \Lambda_g, \Gamma_g, \emptyset \rangle$ is a* generalization *of a state $\langle \Sigma, \Lambda, \Gamma, \Delta \rangle$ if there is a substitution function $\sigma$ from parameters in $\Sigma$ to parameters in $\Sigma_g$ such that, for all propositions $A \in \Gamma, \Delta$, there exists a proposition $A_g \in \Gamma_g$ such that $\Lambda_g \vdash \sigma A = A_g$, and if whenever $\Lambda \vdash t = s$, then $\Lambda_g \vdash \sigma t = \sigma s$.*

**Definition 6.** *A state $\mathcal{S}$ is an* abstraction *of a program $\mathcal{P}$ with an initial state $\mathcal{S}_0$ if, for any program trace $\mathcal{S}_0, \ldots, \mathcal{S}'$, $\mathcal{S}$ is a generalization of $\mathcal{S}'$.*

Now we can state the meta-approximation theorem (proved in Appendix A), that relates the definition of *abstraction* above to the concept of an *approximate version* of a program as specified by Definition 4.

**Theorem 1** (Meta-approximation)**.** *If $\mathcal{P}_a$ is an approximate version of $\mathcal{P}$, $\mathcal{S}_0 = \langle \Sigma_0, \Lambda_0, \Gamma_0, \Delta_0 \rangle$ is an initial state of $\mathcal{P}$ and $\langle \Sigma_0, \Lambda_0, (\Gamma_0, \Delta_0), \emptyset \rangle, \ldots, \mathcal{S}_a$ is a complete trace of the program $\mathcal{P}_a$, then $\mathcal{S}_a$ is an abstraction of $\mathcal{P}$ with initial state $\mathcal{S}_0$.*

## 9. Conclusion

We have defined simple and sound transformations to approximate the computation of a forward-chaining linear logic programming language with higher-order terms and persistent equality constraints. This language is suitable for writing interpreters for programming languages in the style of substructural operational semantics. We show that approximating such interpreters with our transformations yields static analyses of programs written in those languages. The relative ease of encoding two rather different analyses, alias analysis and $k$CFA, suggests that our technique can be used to derive other program analyses.

This work is similar to work by Bozzano et al. (2002, 2004) in both its goals and its methodology. They encode distributed systems and communication protocols in a style is significantly different from ours, but in a language that essentially includes all the features of our language except for equality constraints. Then abstractions of those programs are used to verify properties of concurrent protocols that were encoded in the logic (Bozzano and Delzanno 2002). Their approach differs from ours in a number of ways; in particular, a general purpose approximation is used, in contrast to our methodology of describing a whole class of approximations. Furthermore, Bozzano et al.'s methods are designed to consider properties of systems as a whole, not static analyses of individual inputs as is the case in our work. Clarifying the precise relationship between these two kinds of approximation more precisely is an interesting direction for future work.

Jagadeesan et al. (2005) have also noted the usefulness of constraints in the specification of concurrent systems, but that work was focused on $\lambda$Prolog, a higher-order backward-chaining logic programming language without linearity, and was concerned solely with specification, not approximation and analysis.

Another line of related work is the use of the $\nabla$ quantifier (Miller and Tiu 2005) for name generation, which we could have used here instead of the existential quantifier, although the difference does not become significant until we carry out formal meta-reasoning (Tiu 2007). As a point of future work we conjecture it may be possible to apply our use of Skolemization in the realm of reasoning with and about generic judgments.

Another item of future work is to extend and exploit the metacomplexity theorems about saturating bottom-up logic programs (McAllester 2002; Ganzinger and McAllester 2002; Simmons and Pfenning 2008) in order to obtain, perhaps nearly mechanically, a bound on the complexity of various program analyses that can be derived from a given interpreter. Prior theorems treat neither higher-order features nor the logical notion of equality underlying our approximations.

A fundamentally different kind of approximation of linear logic programs via predicate substitution has been described by Miller (To Appear). Miller's approximations remain linear, which we have ruled out so far in order to obtain a simple meta-approximation theorem. We plan to investigate if the approaches can be fruitfully combined.

***Acknowledgments*** We would like to acknowledge the three anonymous reviewers for their helpful comments.

## References

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools, Second Edition*. Pearson Education, Inc, 2007.

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992.

Marco Bozzano and Giorgio Delzanno. Automated protocol verification in linear logic. In *PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 38–49, New York, NY, USA, 2002. ACM.

Marco Bozzano, Giorgio Delzanno, and Maurizio Martelli. An effective fixpoint semantics for linear logic programs. *TPLP*, 2(1):85–122, 2002.

Marco Bozzano, Giorgio Delzanno, and Maurizio Martelli. Model checking linear logic specifications. *TPLP*, 4(5-6):573–619, 2004.

Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Electr. Notes Theor. Comput. Sci.*, 165:145–176, 2006.

Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, School of Computer Science, Carnegie Mellon University, March 2002. Revised May 2003.

Patrick Cousout and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, 1977. ACM Press.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations (with retrospective). In Kathryn S. McKinley, editor, *Best of PLDI*, pages 502–514. ACM, 1993.

Harald Ganzinger and David A. McAllester. Logical algorithms. In *ICLP '02: Proceedings of the 18th International Conference on Logic Programming*, pages 209–223, London, UK, 2002. Springer-Verlag.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Inf. Comput.*, 110(2):327–365, 1994.

Radha Jagadeesan, Gopalan Nadathur, and Vijay A. Saraswat. Testing concurrent systems: An interpretation of intuitionistic logic. In Ramaswamy Ramanujam and Sandeep Sen, editors, *FSTTCS*, volume 3821 of *Lecture Notes in Computer Science*, pages 517–528. Springer, 2005.

Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In Pedro Barahona and Amy P. Felty, editors, *PPDP*, pages 35–46. ACM, 2005.

David A. McAllester. On the complexity analysis of static analyses. *J. ACM*, 49(4):512–537, 2002.

Dale Miller. A proof-theoretic approach to the static analysis of logic programs. In *Festschrift in Honor of Peter Andrews*, Studies in Logic. Elsevier Science, To Appear.

Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, 6(4):749–783, 2005.

Peter D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

Frank Pfenning. Substructural operational semantics and linear destination-passing style (invited talk). In Wei-Ngan Chin, editor, *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, page 196. Springer, 2004.

Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In Thorsten Altenkirch, Wolfgang Naraschewski, and Bernhard Reus, editors, *TYPES*, volume 1657 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 1998.

Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2000. Available as Technical Report CMU-CS-00-146.

Olin Shivers. Control flow analysis in Scheme. In *Proceedings the Conference on Programming Language Design and Implementation (PLDI'88)*, pages 164–174. ACM Press, 1988.

Robert J. Simmons and Frank Pfenning. Linear logical algorithms. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 336–347. Springer, 2008.

Alwen Tiu. A logic for reasoning about generic judgments. In *Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'06)*, pages 3–18, 2007. Published in ENTCS 174(5), June 2007.

David Van Horn and Harry G. Mairson. Relating complexity and precision in control flow analysis. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 85–96, New York, NY, USA, 2007. ACM.

David Van Horn and Harry G. Mairson. Deciding $k$CFA is complete for EXPTIME. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 275–282, New York, NY, USA, 2008. ACM.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. Specifying properties of concurrent computations in clf. *Electr. Notes Theor. Comput. Sci.*, 199:67–87, 2008.

## A. Proof of Theorem 1

First, we need to define *general program traces*, which are the same as a program traces except that each state is required to be an evolution – not necessarily a productive evolution – of the previous state.

The first two lemmas show that an approximate version of a program can simulate the program it approximates, and next two formalize the notion that, in an approximate version of a program, the saturated database at the conclusion of a complete program trace captures all of the "behaviors" of that approximate program. These two facts in combination mean that all of the behaviors of a program are captured by the saturated database at the conclusion of a complete trace of its approximate version.

**Lemma 1** (One-step simulation)**.** *If program state $\mathcal{S}$ can evolve to program state $\mathcal{S}^+$ under the rule firing $(\mathcal{S}, R, \Delta^*, \sigma)$, with $R \in \mathcal{P}$, and if $\mathcal{P}_a$ is an approximate version of $\mathcal{P}$ and $\mathcal{S}_g$ is a generalization of $\mathcal{S}$, then there exists a rule firing $(\mathcal{S}_g, R_a, \emptyset, \sigma_g)$ with $R_a \in \mathcal{P}_a$ under which $\mathcal{S}_g$ evolves to $\mathcal{S}_g^+$ and $\mathcal{S}_g^+$ is a generalization of $\mathcal{S}^+$.*

*Proof.* Let the program state $\mathcal{S}$ be $\langle \Sigma, \Lambda, \Gamma, (\Delta, \Delta^*)\rangle$, the program state $\mathcal{S}^+$ be $\langle (\Sigma, \Sigma'), \Lambda', (\Gamma, \Gamma'), (\Delta, \Delta')\rangle$. We are given a program state $\mathcal{S}_g = \langle \Sigma_g, \Lambda_g, \Gamma_g, \emptyset\rangle$ that is a generalization of $\mathcal{S}$, so there is a substitution $\tau$ from $\Sigma$ to $\Sigma_g$ such that all the properties for generalization hold.

In the rule $R_a$ that corresponds to the rule $R$, if we take a premise $A_i$ of $R_a$, that premise appears in $R$, and so there is some proposition $A \in \Gamma, \Delta, \Delta^*$ such that $\Lambda \vdash \sigma A_i = A$. By the definition of generalization (second condition), $\Lambda_g \vdash \tau(\sigma A_i) = \tau A$. Also by the definition of generalization (first condition), we have an $A_g \in \Gamma_g$ such that $\Lambda_g \vdash \tau A = A_g$. By transitivity, $\Lambda_g \vdash (\tau \circ \sigma) A_i = A_g$, and we can construct a rule firing $(\mathcal{S}_g, \emptyset, R_a, \tau \circ \sigma)$ that evolves $\mathcal{S}_g$ to $\mathcal{S}_g^+$.

We have to show that $\mathcal{S}_g^+ = \langle (\Sigma_g, \Sigma_g'), \Lambda_g', (\Gamma_g, \Gamma_g'), \emptyset\rangle$ is a generalization of $\mathcal{S}^+$. We can extend $\tau$ to $\tau'$ so that it appropriately substitutes parameters in $\Sigma'$ with the parameter in $\Sigma_g'$ that is assigned to the same existential variable (the existential variables of $R$ and $R_a$ are identical). We need show that $\tau'$ satisfies the two conditions for generalization.

For the first condition, we are given an $A \in \Gamma, \Gamma', \Delta, \Delta'$ and must exhibit a $A_g \in \Gamma_g, \Gamma_g'$ such that $\Lambda_g' \vdash \tau' A_g = A$. If $A \in \Gamma, \Delta$, this follows immediately from the fact that we have an $A_g \in \Gamma_g$ such that $\Lambda_g \vdash \tau A_g = A$. If $A \in \Gamma', \Delta'$, then $A = (\delta \circ \sigma) B_i$ for some conclusion $B_i$ in $R$, where $\delta$ substitutes parameters in $\Sigma'$ for existential variables in $R$. The conclusion $B_i$ also appears in $R_a$, and so there is a $A_g \in \Gamma_g'$ such that $A_g = (\delta' \circ \tau \circ \sigma) B_i$, where $\delta'$ substitutes parameters in $\Sigma_g'$ for existential variables in $R_a$. Because of the way we constructed $\tau'$, we know $\delta' \circ \tau = \tau' \circ \delta$, so we have $A_g = (\tau' \circ \delta \circ \sigma) B_i = \tau'(\delta \circ \sigma) B_i = \tau' A$, and so $\Lambda_g' \vdash \tau' A = A_g$, which is what we needed to show.

For the second condition, we have $\Lambda' \vdash t = s$, and need to show $\Lambda_g' \vdash \tau' t = \tau' s$. Similar reasoning as above applies; old equalities continue to hold, and for every equality that is added to $\Lambda'$, an equivalent constraint is added to $\Lambda_g'$. □

**Lemma 2** (Simulation)**.** *If $\langle \Sigma_0, \Lambda_0, \Gamma_0, \Delta_0\rangle, \ldots, \mathcal{S}_n$ is a program trace of a program $\mathcal{P}$, and $\mathcal{P}_a$ is an approximate version of $\mathcal{P}$, then there exists a general program trace $\langle \Sigma_0, \Lambda_0, (\Gamma_0, \Delta_0), \emptyset\rangle, \ldots, \mathcal{S}_n'$ of the program $\mathcal{P}_a$ where $\mathcal{S}_n'$ is a generalization of $\mathcal{S}_n$.*

*Proof.* By induction on the length of the program trace.

In the base case, we need to know that $\langle \Sigma_0, \Lambda_0, (\Gamma_0, \Delta_0), \emptyset\rangle$ is a generalization of $\langle \Sigma_0, \Lambda_0, \Gamma_0, \Delta_0\rangle$. This is immediate from the definition of generalization.

In the inductive case, we are given a program trace $\mathcal{S}_0, \ldots, \mathcal{S}_n$ and a general program trace $\mathcal{S}_0', \ldots, \mathcal{S}_n'$ where $\mathcal{S}_n'$ is a generalization of $\mathcal{S}_n$. The state $\mathcal{S}_n$ has a productive evolution to state $\mathcal{S}_{n+1}$, and so by Lemma 1, $\mathcal{S}_n'$ has an evolution to state $\mathcal{S}_{n+1}'$ where $\mathcal{S}_{n+1}'$ is a generalization of $\mathcal{S}_{n+1}$, which is what we needed to show. □

**Lemma 3** (Monotonicity)**.** *If a program $\mathcal{P}$ and a state $\mathcal{S}$ contain no linear predicates, then if $\mathcal{S}$ evolves in zero or more steps to $\mathcal{S}'$, then $\mathcal{S}'$ is a generalization of $\mathcal{S}$.*

*Proof.* By induction on the length of the program trace. The base case is again immediate, and in the inductive case the relevant substitution is the identity, all of the propositions in the original $\mathcal{S}$ continue to be present in $\mathcal{S}'$, and the second condition appeals to the monotonicity of the equational constraints – extending $\Lambda$ to $\Lambda'$ can only cause it relate more terms. □

**Lemma 4** (Saturation)**.** *If the program $\mathcal{P}$ uses no linear predicates, and we have an initial state $\mathcal{S}_0 = \langle \Sigma, \Lambda, \Gamma, \emptyset\rangle$, then if $\mathcal{S}_0, \ldots, \mathcal{S}_a$ is a complete trace of $\mathcal{P}$ and $\mathcal{S}_0, \ldots, \mathcal{S}_n$ is a general trace of $\mathcal{P}$, $\mathcal{S}_a$ is a generalization of $\mathcal{S}_n$.*

*Proof.* By induction on the general trace $\mathcal{S}_0, \ldots, \mathcal{S}_n$. The base case follows from Lemma 3 – $\mathcal{S}_a$ generalizes $\mathcal{S}_0$ because the latter evolved from the former.

In the inductive case, $\mathcal{S}_a$ is a generalization of $\mathcal{S}_n$, and $\mathcal{S}_n$ that evolves to $\mathcal{S}_{n+1}$. We need to show $\mathcal{S}_a$ is a generalization of $\mathcal{S}_{n+1}$. By Lemma 1, $\mathcal{S}_a$ evolves to $\mathcal{S}_a^+$, which is a generalization of $\mathcal{S}_{n+1}$. Because generalization is transitive, it suffices to show that $\mathcal{S}_a$ is a generalization of $\mathcal{S}_a^+$, which follows from the fact that the evolution of $\mathcal{S}_a$ to $\mathcal{S}_a^+$ is not productive, so every new parameter and proposition in $\mathcal{S}_a^+$ is equal to a term or proposition in $\mathcal{S}_a$. □

Lemmas 2 and 4 give us the pieces needed to complete the proof of the meta-approximation theorem. We will first re-state the theorem from Sec. 8:

**Theorem 1** (Meta-approximation)**.** *If $\mathcal{P}_a$ is an approximate version of $\mathcal{P}$, $\mathcal{S}_0 = \langle \Sigma_0, \Lambda_0, \Gamma_0, \Delta_0\rangle$ is an initial state of $\mathcal{P}$ and $\langle \Sigma_0, \Lambda_0, (\Gamma_0, \Delta_0), \emptyset\rangle, \ldots, \mathcal{S}_a$ is a complete trace of the program $\mathcal{P}_a$, then $\mathcal{S}_a$ is an abstraction of $\mathcal{P}$ with initial state $\mathcal{S}_0$.*

*Proof of meta-approximation theorem.* In order to show that $\mathcal{S}_a$ is an abstraction of $\mathcal{P}$ with initial state $\mathcal{S}_0 = \langle \Sigma_0, \Lambda_0, \Gamma_0, \Delta_0\rangle$, we must show that for any trace $\mathcal{S}_0, \ldots, \mathcal{S}_n$ of the program $\mathcal{P}$, $\mathcal{S}_a$ is a generalization of $\mathcal{S}_n$. If we take some such arbitrary trace, we have a general trace $\langle \Sigma_0, \Lambda_0, (\Gamma_0, \Delta_0), \emptyset\rangle, \ldots, \mathcal{S}_n'$ of the program $\mathcal{P}_a$ by Lemma 2 such that $\mathcal{S}_n'$ is a generalization of $\mathcal{S}_n$.

Lemma 4 gives us that $\mathcal{S}_a$ is a generalization of $\mathcal{S}_n'$, and therefore, because generalization is transitive, we have what we needed to show, that $\mathcal{S}_a$ is a generalization of $\mathcal{S}_n$. □