Chapter 8

Linear logical approximation

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* [CC77], which provides techniques for constructing approximations (for example, by exhibiting a Galois connection between concrete and abstract domains). The correctness proof must establish the appropriate relationship between the concrete and abstract computations and show that the abstract computation terminates. 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 chapter, which is mostly derived from [SP11], we consider a new class of instances in the general schema of abstract interpretation that is based on the approximation of SSOS specifications in SLS. We apply logically justified techniques for manipulating and approximating SSOS specifications to yield approximations that are correct by construction. The resulting persistent logical specifications can be interpreted and executed as saturating logic programs, which means that derived specifications are executable program analyses.

The process described in this chapter does not claim to capture or derive all possible interesting program analyses. The methodology we describe only derives over-approximations (or *may*- analyses) that ensure all possible behaviors will be reported by the analysis. There is a whole separate class of under-approximations (or *must*- analyses) which ensure that if a behavior is reported by the analysis it is possible; we will not consider under-approximations here [GNRT10]. Instead, we argue for the utility of our methodology by deriving two fundamental and rather different over-approximation-based analyses: a context-insensitive control flow analysis (Section 8.4) and an alias analysis (Section 8.5). Might and Van Horn's closely related "abstracting abstract machines" methodology, described in Section 8.6 along with other related work, suggests many more examples.

8.1 Saturating logic programming

Concurrent SLS specifications where all positive atomic propositions are persistent (and where all inclusions of negative propositions in positive propositions – if there are any – have the form

$!A^-$, not $\downarrow A^-$ or $¡A^-$) have a distinct logical and operational character. Logically, by the discussion in Section 3.7 we are justified in reading such specifications as specifications in persistent intuitionistic logic or persistent lax logic. Operationally, while persistent specifications have an interpretation as transition systems, that interpretation is not very useful. This is because if we can take a transition once – for instance, using the rule $\mathsf{a} \rightarrowtail \{\mathsf{b}\}$ to derive the persistent atomic proposition b from the persistent atomic proposition a – none of the facts that enabled that transition can be consumed, as all facts are persistent. Therefore, we can continue to make the same transition indefinitely; in the above-mentioned example, such transitions will derive multiple redundant copies of b.

The way we will understand the meaning of persistent and concurrent SLS specifications is in terms of *saturation*. A process state $(\Psi; \Delta)$ is saturated relative to the signature Σ if, for any step $(\Psi; \Delta) \leadsto_\Sigma (\Psi'; \Delta')$, it is the case that Ψ and Ψ' are the same (the step unified no distinct variables and introduced no new variables), $x{:}\langle p^+_{pers}\rangle \in \Delta'$ implies $x{:}\langle p^+_{pers}\rangle \in \Delta$, and $x{:}A^-\ pers \in \Delta'$ implies $x{:}A^-\ pers \in \Delta$. This means that a signature with a rule that produces new variables by existential quantification, like $\mathsf{a} \rightarrowtail \{\exists x.\mathsf{b}(x)\}$ has no saturated process states where a is present. We will cope with rules of this form by turning them into rules of the form $\mathsf{a} \rightarrowtail \{\exists x.\mathsf{b}(x) \bullet x \doteq t\}$ for some t, neutralizing the free existential variable as a notational definition. Notions of saturation that can cope with free existentially generated variables in other ways are interesting, but are beyond the scope of this dissertation.

A *minimal* saturated process state is one with no duplicated propositions; we can compute a minimal process state from any saturated process state by removing duplicates. For purely persistent specifications and process states, minimal saturated process states are unique when they exist: if $(\Psi; \Delta) \leadsto^*_\Sigma (\Psi_1; \Delta_1)$ and $(\Psi; \Delta) \leadsto^*_\Sigma (\Psi_2; \Delta_2)$ and both $(\Psi_1; \Delta_1)$ and $(\Psi_2; \Delta_2)$ are saturated, then $(\Psi_1; \Delta_1)$ and $(\Psi_2; \Delta_2)$ have minimal process states that differ only in the names of variables.

Furthermore, if a saturated process state exists for a given initial process state, the minimal saturated process state can be computed by the usual forward-chaining semantics where only transitions that derive *new* persistent atomic propositions or equalities $t \doteq s$ are allowed. This forward-chaining logic programming interpretation of persistent logic is extremely common, usually associated with the logic programming language Datalog. A generalization of Datalog formed the basis of McAllester and Ganzinger's meta-complexity results: they gave a cost semantics to their logic programming language, and then they used that cost semantics to argue that many program analyses could be *efficiently* implemented as logic programs [McA02, GM02]. Persistent SLS specifications can be seen as an extension of McAllester and Ganzinger's language (and, transitively, as a generalization of Datalog). We will not deal with cost semantics or efficiency, however, as our use of higher-order abstract syntax appears to complicate McAllester and Ganzinger's cost semantics.

Just as the term *persistent logic* was introduced in Chapter 2 to distinguish what is traditionally referred to as intuitionistic logic from intuitionistic ordered and linear logic, we will use the term *saturating logic programming* to distinguish what is traditionally referred to as forward-chaining logic programming from the forward-chaining logic programming interpretation that makes sense for ordered and linear logical specifications. There is a useful variant of substructural forward chaining, forward chaining with *quiescence* [LPPW05], that acts like saturating logic programming on purely-persistent specifications and like simple committed-choice

```
hd: dest -> dest -> prop pers.
left: tok -> dest -> dest -> prop pers.
right: tok -> dest -> dest -> prop pers.
stack: tok -> dest -> dest -> prop pers.

push: hd L M * left X M R
       >-> {Exists m. stack X L m * hd m R * m == fm X L M R}.

pop:  stack X L M1 * hd M1 M2 * right X M2 R >-> {hd L R}.
```

Figure 8.1: Skolemized approximate version of the PDA specification from Figure 7.2

forward chaining on specifications with no persistent propositions. We refined this interpretation and gave it a cost semantics in [SP08], but this more sophisticated interpretation is not relevant to the examples in this dissertation.

8.2 Using approximation

The meta-approximation theorem that we present in the next section gives us a way of building abstractions from specifications and initial process states: we interpret the approximate version of the program as a saturating logic program over that initial state. If we can obtain a saturated process state using the logic programming interpretation, it is an abstraction of the initial process state. It is not always possible to obtain a saturated process state using the logic programming interpretation, however: rules like $\forall x.\, \mathsf{a}(x) \rightarrowtail \{\mathsf{a}(\mathsf{s}(x))\}$ and $\forall x.\, \mathsf{a}(x) \rightarrowtail \{\exists y.\mathsf{a}(y)\}$ lead to non-termination when interpreted as saturating logic programs. Important classes of programs are known to terminate in all cases, such as those in the Datalog fragment where the only terms in the program are variables and constants. Structured terms (like expressions encoded in the LF type exp) fall outside the Datalog fragment.

Consider the destination-passing PDA specification from Figure 7.2. If we simply turn all linear predicates persistent, the first step in the approximation methodology, then the push rule will lead to non-termination because the head $\exists m.\mathsf{stack}\, x\, l\, m \bullet \mathsf{hd}\, m\, r$ introduces a new existential parameter m. We can cope by adding a new conclusion $m \doteq t$; adding new conclusions is the second step in the approximation methodology. This, however, means we have to pick a t. The most general starting point for selecting a t is to apply Skolemization to the rule. By moving the existential quantifier for m in front of the implicitly quantified X, L, M, and R, we get a Skolem function $\mathsf{fm}\, X\, L\, M\, R$ that takes four arguments. Letting $t = \mathsf{fm}\, X\, L\, M\, R$ results in the SLS specification/logic program shown in Figure 8.1. (Remember that, because the specification in Figure 8.1 is purely persistent, we will omit the optional `!` annotation described in Section 4.5, writing `hd L M` instead of `!hd L M` and so on.)

Notice that we have effectively taken a specification that freely introduces existential quantification (and that therefore *definitely* will not terminate when interpreted as a saturating logic program) and produced a specification that uses structured terms $\mathsf{fm}\, X\, L\, R\, M$. But the introduction of structured terms takes us outside the Datalog fragment, which may also lead to non-

termination! This is not as bad as it may seem: when we want to treat a specification with structured terms as a saturating logic program, it is simply necessary to reason explicitly about termination. Giving any finite upper bound on the number of derivable facts is a simple and sufficient criteria for showing that a saturating logic program terminates.

Skolem functions provide a natural starting point for approximations, even though the Skolem constant that arises directly from Skolemization is usually more precise than we want. From the starting point in Figure 8.1, however, we can define approximations simply by instantiating the Skolem constant. For instance, we can equate the existentially generated destination in the conclusion with the one given in the premise (letting $\mathsf{fm} = \lambda X.\lambda L.\lambda M.\lambda R.\, M$). The result is equivalent to this specification:

```
push: hd L M * left X M R >-> {stack X L M * hd M R}.
pop:  stack X L M1 * hd M1 M2 * right X M2 R >-> {hd L R}.
```

This substitution yields a precise approximation that exactly captures the behavior of the original PDA as a saturating logic program.

To be concrete about what this means, let us recall how the PDA works and what it means for it to accept a string. To use the linear PDA specification in Figure 7.2, we encode a string as a sequence of linear atomic propositions $ptok_1 \ldots ptok_n$, where each $ptok_i$ either has the form $\mathsf{left}\ tok_i\ \mathsf{d_i}\ \mathsf{d_{i+1}}$ or the form $\mathsf{right}\ tok_i\ \mathsf{d_i}\ \mathsf{d_{i+1}}$. The term tok_i that indicates whether we're talking about a left/right parenthesis, curly brace, square brace, etc., and $\mathsf{d_0} \ldots \mathsf{d_{n+1}}$ are $n+2$ constants of type *dest*.[1] Let $\Delta = (h{:}\langle \mathsf{hd}\ \mathsf{d_0}\ \mathsf{d_1}\rangle\ eph, x_1{:}\langle ptok_1\rangle\ eph, \ldots, x_n{:}\langle ptok_n\rangle\ eph)$. The PDA accepts the string encoded as $ptok_1 \ldots ptok_n$ if and only if there is a trace under the signature in Figure 7.2 where $\Delta \rightsquigarrow^* (\Psi'; x{:}\langle \mathsf{hd}\ \mathsf{d_0}\ \mathsf{d_{n+1}}\rangle\ eph)$.

Now, say that we turn the predicates persistent and run the program described by the push and pop rules above as a saturating logic program, obtaining a saturated process state Δ_{sat} from the initial process state $(h{:}\langle \mathsf{hd}\ d_0\ d_1\rangle\ pers, x_1{:}\langle ptok_1\rangle\ pers, \ldots, x_n{:}\langle ptok_n\rangle\ pers)$. (We can see from the structure of the program that LF context will remain empty.) The meta-approximation theorem *ensures* that, if the original PDA accepted, then the proposition $\mathsf{hd}\ \mathsf{d_0}\ \mathsf{d_{n+1}}$ is in Δ_{sat}. It just so happens to be the case that the converse is also true – if $\mathsf{hd}\ \mathsf{d_0}\ \mathsf{d_{n+1}}$ is in Δ_{sat}, the original PDA specification accepts the string. That is why we say we have a precise approximation.

On the other hand, if we set m equal to l (letting $\mathsf{fm} = \lambda X.\lambda L.\lambda M.\lambda R.\, L$), the result is equivalent to this specification:

```
push: hd L M * left X M R >-> {stack X L L * hd L R}.
pop:  stack X L M1 * hd M1 M2 * right X M2 R >-> {hd L R}.
```

If the initial process state contains a single atomic proposition $\mathsf{hd}\ \mathsf{d_0}\ \mathsf{d_1}$ in addition to all the left and right facts, then the two rules above maintain the invariant that, as new facts are derived, the first argument of hd and the second and third arguments of stack will *always* be $\mathsf{d_0}$. These arguments are therefore vestigial, like the extra arguments to eval and retn discussed in Section 7.1.1, and we can remove them from the approximate specification, resulting in the specification in Figure 8.2.

[1] We previously saw destinations as only inhabited by parameters, but the guarantees given by the meta-approximation theorem are clearer when the initial state contains destinations that are constants.

```
hd: dest -> prop pers.
left: tok -> dest -> dest -> prop pers.
right: tok -> dest -> dest -> prop pers.
stack: tok -> prop pers.

push: hd M * left X M R >-> {stack X * hd R}.
pop:  stack X * hd M2 * right X M2 R >-> {hd R}.
```

Figure 8.2: Approximated PDA specification

This logical approximation of the original PDA accepts if we run saturating logic programming from the initial process state $(h{:}\langle \mathsf{hd}\ \mathsf{d}_1\rangle\ pers, x_1{:}\langle ptok_1\rangle\ pers, \ldots, x_n{:}\langle ptok_n\rangle\ pers)$ and $\mathsf{hd}\ \mathsf{d}_{n+1}$ appears in the saturated process state. Again, the meta-approximation theorem ensures that any string accepted by the original PDA will also be accepted by any approximation. This approximation will additionally accept every string where, for every form of bracket tok, at least one left tok appears before any of the right tok. The string `[ ] ] ] ( ( )` would be accepted by this approximated PDA, but the string `( ) ] [ [ ]` would not, as the first right square bracket appears before any left square bracket.

8.3 Logical transformation: approximation

The approximation strategy demonstrated in the previous section is quite simple: a signature in an ordered or linear logical specification can be approximated by making all atomic propositions persistent, and a flat rule $\forall \overline{x}.\ A^+ \rightarrowtail \{B^+\}$ containing only persistent atomic propositions can be further approximated by removing premises from A^+ and adding conclusions to B^+. Of particular practical importance are added conclusions that neutralize an existential quantification with a notational definition. The approximation procedure doesn't force us to neutralize all such variables in this way. However, as we explained above, failing to do so almost ensures that the specification cannot be run as a saturating logic program, and being able to interpret specifications as saturating logic programs is a prerequisite for applying the meta-approximation theorem (Theorem 8.4).

First, we define what it means for a specification to be an approximate version of another specification:

Definition 8.1. *A flat, concurrent, and persistent specification Σ_a is an* approximate version *of another specification Σ if every predicate* $\mathsf{a} : \Pi x_1{:}\tau_1 \ldots \Pi x_n{:}\tau_n.\ \mathsf{prop}\ \mathsf{lvl}$ *declared in Σ has a corresponding predicate* $\mathsf{a} : \Pi x_1{:}\tau_1 \ldots \Pi x_n{:}\tau_n.\ \mathsf{prop}\ \mathsf{pers}$ *in Σ_a and if for every rule* $\mathsf{r} : \forall \overline{x}.\ A_1^+ \rightarrowtail \{A_2^+\}$ *in Σ there is a corresponding rule* $\mathsf{r} : \forall \overline{x}.\ B_1^+ \rightarrowtail \{B_2^+\}$ *in Σ_a such that:*

- *The existential quantifiers in A_1^+ and A_2^+ are identical to the existential quantifiers in B_1^+ and B_2^+ (respectively),*
- *For each premise (p_{pers}^+ or $t \doteq s$) in B_1^+, the same premise appears in A_1^+, and*
- *For each conclusion (p_{lvl}^+ or $t \doteq s$) in A_2^+, the same premise appears in B_2^+.*

While approximation is a program transformation, it is not a deterministic one: Definition 8.1 describes a whole family of potential approximations. Even the nondeterministic operationalization transformation was just a bit nondeterministic, giving several options for operationalizing any given deductive rule. The approximation transformation, in contrast, needs explicit information from the user: which premises should be removed, and what new conclusions should be introduced? While there is value in actually implementing the operationalization, defunctionalization, and destination-adding transformations, applying approximation requires intelligence. Borrowing a phrase from Danvy, approximation is a candidate for "mechanization by graduate student" rather than mechanization by computer.

Next, we give a definition of what it means for a state to be an approximate version (we use the word "generalization") of another state or a family of states.

Definition 8.2. *The persistent process state* $(\Psi_g; \Delta_g)$ *is a* generalization *of the process state* $(\Psi; \Delta)$ *if there is a substitution* $\Psi_g \vdash \sigma : \Psi$ *such that, for all atomic propositions* $p^+_{lvl} = \mathsf{a}\, t_1 \ldots t_n$ *in* Δ*, there exists a persistent proposition* $p^+_{pers} = \mathsf{a}\, (\sigma t_1) \ldots (\sigma t_n)$ *in* Δ_g*.*

One thing we might prove about the relationship between process states and their generalizations is that generalizations can *simulate* the process states they generalize: that is, if $(\Psi_g; \Delta_g)$ is a generalization of $(\Psi; \Delta)$ and $(\Psi; \Delta) \rightsquigarrow_\Sigma (\Psi'; \Delta')$ then $(\Psi_g; \Delta_g) \rightsquigarrow_{\Sigma_a} (\Psi'_g; \Delta'_g)$ where $(\Psi'_g; \Delta'_g)$ is a generalization of $(\Psi'; \Delta')$. This property, *one-step simulation*, is true [SP11, Lemma 6], and we will prove it as a corollary on the way to the proof of Theorem 8.4. However, we are not interested in generalization per se; rather, we're interested in a stronger property, *abstraction*, that is defined in terms of generalization:

Definition 8.3. *A process state* $(\Psi_a; \Delta_a)$ *is an* abstraction *of* $(\Psi_0; \Delta_0)$ *under the signature* Σ *if, for any trace* $(\Psi_0; \Delta_0) \rightsquigarrow^*_\Sigma (\Psi_n; \Delta_n)$*,* $(\Psi_a; \Delta_a)$ *is a generalization of* $(\Psi_n; \Delta_n)$*.*

An abstraction of the process state $(\Psi_0; \Delta_0)$ is therefore a single process state that captures *all possible future behaviors* of the state $(\Psi_0; \Delta_0)$ because, for any atomic proposition $p^+_{lvl} = \mathsf{a}\, t_1 \ldots t_n$ that may be derived by evolving $(\Psi_0; \Delta_0)$, there is a substitution σ such that $\mathsf{a}\, (\sigma t_1) \ldots (\sigma t_n)$ is already present in the abstraction. The meta-approximation theorem relates this definition of abstraction to the concept of approximate versions of programs as specified by Definition 8.1.

Theorem 8.4 (Meta-approximation)**.** *If* Σ_a *is an approximate version of* Σ*, and if there is a state* $(\Psi_0; \Delta_0)$ *well-formed according to* Σ*, and if for some* $\Psi'_0 \vdash \sigma : \Psi_0$ *there is a trace* $(\Psi'_0; \sigma\Delta_0) \rightsquigarrow^*_{\Sigma_a} (\Psi_a; \Delta_a)$ *such that* $(\Psi_a; \Delta_a)$ *is a saturated process state, then* $(\Psi_a; \Delta_a)$ *is an abstraction of* $(\Psi_0; \Delta_0)$*.*

Proof. The central lemma is one-step simulation, mentioned above, which is established by induction on the structure of the step. A multi-step *simulation* lemma immediately follows by induction on traces: If Σ_a is an approximate version of Σ, $(\Psi_g; \Delta_g)$ is a generalization of $(\Psi; \Delta)$ and $(\Psi; \Delta) \rightsquigarrow^*_\Sigma (\Psi'; \Delta')$ then $(\Psi_g; \Delta_g) \rightsquigarrow^*_{\Sigma_a} (\Psi'_g; \Delta'_g)$ where $(\Psi'_g; \Delta'_g)$ is a generalization of $(\Psi'; \Delta')$ [SP11, Lemma 7].

The *monotonicity* lemma establishes that transitions in a purely-persistent specification only increase the generality of a process state: if $(\Psi; \Delta) \leadsto^*_\Sigma (\Psi'; \Delta')$ and Σ defines no ordered or mobile predicates, then $(\Psi'; \Delta')$ is a generalization of $(\Psi; \Delta)$ [SP11, Lemma 8].

We use the monotonicity lemma to prove the *saturation* lemma: if $(\Psi; \Delta) \leadsto^*_\Sigma (\Psi_s; \Delta_s)$, Σ defines no ordered or mobile predicates, and $(\Psi_s; \Delta_s)$ is saturated, then whenever $(\Psi; \Delta) \leadsto^*_\Sigma (\Psi'; \Delta')$ $(\Psi_s; \Delta_s)$ is a generalization of $(\Psi'; \Delta')$. The proof proceeds by induction on the last steps of the trace witnessing $(\Psi; \Delta) \leadsto^*_\Sigma (\Psi'; \Delta')$.

- In the base case, $(\Psi; \Delta) = (\Psi'; \Delta')$ and we appeal to monotonicity.
- In the inductive case, we have $(\Psi; \Delta) \leadsto^*_\Sigma (\Psi''; \Delta'') \leadsto_\Sigma (\Psi'; \Delta')$. By the induction hypothesis we have that $(\Psi_s; \Delta_s)$ is a generalization of $(\Psi''; \Delta'')$, and by one-step simulation $(\Psi_s; \Delta_s) \leadsto_\Sigma (\Psi'_s; \Delta'_s)$ such that $(\Psi'_s; \Delta'_s)$ is a generalization of $(\Psi'; \Delta')$. But saturation means that $\Psi_s = \Psi'_s$ and that all the propositions in Δ'_s already appear in Δ_s, so $(\Psi_s; \Delta_s)$ must be a generalization of $(\Psi'; \Delta')$ as well. [SP11, Lemma 9]

Finally, we prove meta-approximation. Consider a trace $(\Psi_0; \Delta_0) \leadsto^*_\Sigma (\Psi_n; \Delta_n)$ of the original program. By the simulation lemma, there is a trace $(\Psi_0; \sigma\Delta_0) \leadsto^*_{\Sigma_a} (\Psi'_n; \Delta'_n)$ where $(\Psi'_n; \Delta'_n)$ is a generalization of $(\Psi_n; \Delta_n)$. By the saturation lemma, $(\Psi_a; \Delta_a)$ is a generalization of $(\Psi'_n; \Delta'_n)$, and so because generalization is transitive, $(\Psi_a; \Delta_a)$ is a generalization of $(\Psi_0; \Delta_0)$, which is what we needed to show [SP11, Theorem 3]. □

The meaning of the meta-approximation theorem is that if (1) we can approximate a specification and an initial state and (2) we can obtain a saturated process state from that approximate specification and approximate initial state, then the saturated process state captures all possible future behaviors of the (non-approximate) initial state.

8.4 Control flow analysis

The initial process state for destination-passing SSOS specifications generally has the form $(d{:}\mathsf{dest}; x{:}\langle \mathsf{eval}\, t\, d\rangle)$ for some program represented by the LF term $t = \ulcorner e \urcorner$. This means that we can use the meta-approximation result to derive abstractions from initial expressions e using the saturating logic programming interpretation of approximated SSOS specifications.

A control flow analysis is a fundamental analysis on functional programs, attributed to Shivers [Shi88]. It is used for taking an expression and "determining for each subexpression a hopefully small number of functions that it may evaluate to; thereby it will determine where the flow of control may be transferred to in the case where the subexpression is the operator of a function application" [NNH05, p. 142]. That is, we want to take a program and find, for every subexpression e of that unevaluated program, all the values v that the subexpression may evaluate to over the course of evaluating the program to a value. Because we are talking about subexpressions of the unevaluated program, the answer might not be unique. Consider the evaluation of $(\lambda f \ldots (f\,(\lambda y \ldots)) \ldots (f\,(\lambda z \ldots)) \ldots)\,(\lambda x.x)$. The function $\lambda x.x$ gets bound to f and therefore may get called twice, once with the argument $(\lambda y \ldots)$ and once with the argument $(\lambda z \ldots)$. The subexpression x of $\lambda x.x$ can therefore evaluate to $(\lambda y \ldots)$ in the context of the call $f\,(\lambda y \ldots)$

and to $(\lambda z \ldots)$ in the context of the call $f\,(\lambda z \ldots)$. As a *may*-analysis, the output of a control flow analysis is required to report both of these possibilities.[2]

When we *use* a control flow analysis, it is relevant that the calculation of which subexpressions evaluate to which values is done in service of a different goal: namely, determining which functions may be called from which calling sites. However, the ultimate goal of control flow analysis is irrelevant to our discussion of deriving control flow analyses from SSOS specifications, so we will concentrate on the question of which subexpressions evaluate to which values. Before we begin, however, we will address the issue of what it even means to be a (closed) subterm of an expression e that has been encoded with higher-order abstract syntax into the canonical forms of LF.

8.4.1 Subexpressions in higher-order abstract syntax

When given a term $\mathsf{a}\,(\mathsf{b}\,\mathsf{c}\,\mathsf{c})$, it is clear that there are three distinct subterms: the entire term, $\mathsf{b}\,\mathsf{c}\,\mathsf{c}$, and c. Therefore, it is meaningful to bound the size of a saturated process state using some function that depends on the number of subterms of the original term. But what are the subterms of $\mathsf{lam}\,(\lambda x.\,\mathsf{app}\,x\,x)$, and how can we write a saturating logic program that derives all those subterms? The rule for application is easy:

$$\mathsf{sub/app}: \quad \forall e_1{:}\mathsf{exp}.\,\forall e_2{:}\mathsf{exp}.\,\mathsf{subterms}(\mathsf{app}\,e_1\,e_2) \rightarrowtail \{\mathsf{subterms}\,e_1 \bullet \mathsf{subterms}\,e_2\}$$

What about the rule for lambda abstractions? Experience with LF says that, when we open up a binder, we should substitute a fresh variable into that binder. This would correspond to the following rule:

$$\mathsf{sub/lam/ohno}: \quad \forall e{:}\mathsf{exp} \to \mathsf{exp}.\,\mathsf{subterms}(\mathsf{lam}(\lambda x.e\,x)) \rightarrowtail \{\exists x.\,\mathsf{subterms}(e\,x)\}$$

The rule $\mathsf{sub/lam/ohno}$ will, as we have discussed, lead to nontermination when we interpret the rules as a saturating logic program. The solution is to apply Skolemization as described in Section 8.2, which introduces a new constant we will call var. The rule $\mathsf{sub/lam/ohono}$ can then be approximated as a terminating rule:

$$\mathsf{sub/lam}: \quad \forall e{:}\mathsf{exp} \to \mathsf{exp}.\,\mathsf{subterms}(\mathsf{lam}(\lambda x.e\,x)) \rightarrowtail \{\mathsf{subterms}(e(\mathsf{var}(\lambda x.e\,x)))\}$$

The subterms of any closed term e of LF type exp can then be enumerated by running this saturating logic program starting with the fact $\mathsf{subterms}(e)$, where $\mathsf{subterms}$ is a persistent positive proposition. We start counting subterms from the outside, and stop when we reach a variable represented by a term $\mathsf{var}(\lambda x.e)$. The logic program and discussion above imply that there are three distinct subterms of $\mathsf{lam}\,(\lambda x.\,\mathsf{app}\,x\,x)$: the entire term, $\mathsf{app}\,(\mathsf{var}(\lambda x.\,\mathsf{app}\,x\,x))\,(\mathsf{var}(\lambda x.\,\mathsf{app}\,x\,x))$, and $\mathsf{var}(\lambda x.\,\mathsf{app}\,x\,x)$.

Another solution, discussed in the next section, is to uniquely tag the lambda expression with a label. This has the same effect of allowing us to associate the variable x with a different concrete term, the tag, that represents the site where x was bound.

[2]This statement assumes that both of the calling sites $f\,(\lambda y \ldots)$ and $f\,(\lambda z \ldots)$ are reachable: the control flow analysis we derive performs some dead-code analysis, and it may not report that x evaluates to $(\lambda y \ldots)$, for instance, if the call $f\,(\lambda y \ldots)$ is certain to never occur.

```
bind: exp -> exp -> prop pers.
eval: exp -> dest -> prop lin.
retn: exp -> dest -> prop lin.
cont: frame -> dest -> dest -> prop lin.

ev/bind: eval X D * !bind X V >-> {retn V D}.

ev/lam:  eval (lam \x. E x) D >-> {retn (lam \x. E x) D}.

ev/app:  eval (app E1 E2) D
          >-> {Exists d1. eval E1 d1 * cont (app1 E2) d1 D}.

ev/app1: retn (lam \x. E x) D1 * cont (app1 E2) D1 D
          >-> {Exists d2. eval E2 d2 * cont (app2 \x. E x) d2 D}.

ev/app2: retn V2 D2 * cont (app2 \x. E x) D2 D
          >-> {Exists x. !bind x V *
               Exists d3. eval (E x) d3 * cont app3 d3 D}.

ev/app3: retn V D3 * cont app3 D3 D >-> {retn V D}.
```

Figure 8.3: Alternative environment semantics for CBV evaluation

8.4.2 Environment semantics

The starting point for deriving a control flow analysis is the environment semantics for call-by-value shown in Figure 8.3. It differs from the environment semantics shown in Figure 6.19 in three ways. First and foremost, it is a destination-passing specification instead of an ordered abstract machine specification, but that difference is accounted for by the destination-adding transformation in Chapter 7. A second difference is that the existentially generated parameter x associated with the persistent proposition $\mathsf{bind}\, x\, v$ is introduced as late as possible in the multi-stage protocol for evaluating an application (rule ev/app2 in Figure 8.3), not as early as possible (rule ev/appenv1 in Figure 6.19). The third difference is that there is an extra frame app3 and an extra rule ev/app3 that consumes such frames. The app3 frame is an important part of the control flow analysis we derive, but in [SP11] the addition of these frames was otherwise unmotivated. Based on our discussion of the logical correspondence in Chapters 5 and 6, we now have a principled account for this extra frame and rule: it is precisely the pattern we get from operationalizing a natural semantics *without* tail-recursion optimization and then applying defunctionalization and destination-adding.

8.4.3 Approximation to 0CFA

In order for us to approximate Figure 8.3 to derive a finite control flow analysis, we turn all linear atomic propositions persistent and then must deal with the variables introduced by existential quantification. The variable x introduced in ev/app2 will be equated with $\mathsf{var}(\lambda x.E\, x)$, which is

```
bind: exp -> exp -> prop pers.
eval: exp -> exp -> prop pers.
retn: exp -> exp -> prop pers.
cont: frame -> exp -> exp -> prop pers.

ev/bind: eval X D * bind X V >-> {retn V D}.

ev/lam:  eval (lam \x. E x) D >-> {retn (lam \x. E x) D}.

ev/app:  eval (app E1 E2) D
          >-> {Exists d1. eval E1 d1 * cont (app1 E2) d1 D *
               d1 == E1}.

ev/app1: retn (lam \x. E x) D1 * cont (app1 E2) D1 D
          >-> {Exists d2. eval E2 d2 * cont (app2 \x. E x) d2 D *
               d2 == E2}.

ev/app2: retn V2 D2 * cont (app2 \x. E x) D2 D
          >-> {Exists x. bind x V *
               Exists d3. eval (E x) d3 * cont app3 d3 D *
               x == var (\x. E x) *
               d3 == E x}.

ev/app3: retn V D3 * cont app3 D3 D >-> {retn V D}.
```

Figure 8.4: A control-flow analysis derived from Figure 8.3

consistent with making $E\,x$ – which is now equal to $E(\mathsf{var}(\lambda x.E\,x))$ – a subterm of $\mathsf{lam}(\lambda x.\,E\,x)$. The new constructor var is also a simplified Skolem function for x that only mentions the LF term E; the most general Skolem function in this setting would have also been dependent on V, D, and D_2. The existentially generated variable x was also the first argument to bind, so bind, as a relation, will now associate binding sites and values instead of unique variables and values.

The discussion above pertains to the existentially generated variable x in rule ev/app2, but we still need some method for handling destinations d_1, d_2, and d_3 in ev/app, ev/app1, and ev/app2 (respectively). To this end, we need recall the question that we intend to answer with control flow analysis: what values may a given subexpression evaluate to? A destination passing specification attempts to return a value to a destination: we will instead return *to an expression* by equating destinations d with the expressions they represent. One way to do this would be to introduce a new constructor $\mathsf{d} : \mathsf{exp} \to \mathsf{dest}$, but we can equivalently conflate the two types exp and dest to get the specification in Figure 8.4.

The specification in Figure 8.4 has a point of redundancy along the lines of the redundancy in our second PDA approximation: the rules maintain the invariants that the two arguments to $\mathsf{eval}\,e\,d$ are always the same. Therefore, the second argument to eval can be treated as vestigial; by removing that argument, we get a specification equivalent to Figure 8.5. That figure includes another simplifications as well: instead of introducing expressions d_1, d_2, and d_3 by existential

```
bind: exp -> exp -> prop pers.
eval: exp -> prop pers.
retn: exp -> exp -> prop pers.
cont: frame -> exp -> exp -> prop pers.

ev/bind: eval X * bind X V >-> {retn V X}.

ev/lam:  eval (lam \x. E x) >-> {retn (lam \x. E x) (lam \x. E x)}.

ev/app:  eval (app E1 E2) * E == app E1 E2
          >-> {eval E1 * cont (app1 E2) E1 E}.

ev/app1: retn (lam \x. E0 x) E1 * cont (app1 E2) E1 E
          >-> {eval E2 * cont (app2 \x. E0 x) E2 E}.

ev/app2: retn V2 E2 * cont (app2 \x. E0 x) E2 E
          >-> {Exists x. bind x V *
               eval (E0 x) * cont app3 (E0 x) E *
               x == var (\x. E0 x)}.

ev/app3: retn V E3 * cont app3 E3 E >-> {retn V E}.
```

Figure 8.5: Simplification of Figure 8.4 that eliminates the vestigial argument to eval

quantification just to equate them with expressions e_1, e_2, and e, we substitute in the equated expressions where the respective destinations appeared in Figure 8.4; this modification does not change anything at the level of synthetic inference rules.

Let's consider the termination of specification in Figure 8.5 interpreted as a saturating logic program. Fundamentally, the terms in the heads of rules are all subterms (in the generalized sense of Section 8.4.1), which is a sufficient condition for the termination of a saturating logic program. More specifically, consider that we start the database with a single fact $\mathsf{eval}\ulcorner e\urcorner$, where $\ulcorner e\urcorner$ has n subterms by the analysis in Section 8.4.1. We can only ever derive n new facts $\mathsf{eval}\, e$ – one for every subterm. If we deduced that every subexpression was a value that could be returned at every subexpression, there would still be only n^2 facts $\mathsf{retn}\, e\, e'$, and the same analysis holds for facts of the form $\mathsf{cont}\, \mathsf{app3}\, e\, e'$. A fact of the form $\mathsf{cont}\, (\mathsf{app1}\, e_2)\, e_1\, e$ will only be derived when $e = \mathsf{app}\, e_1\, e_2$, so there are at most n of these facts. A fact of the form $\mathsf{cont}\, (\mathsf{app2}\, \lambda x.\, e_0\, x)\, e_2\, e$ will only be derived when $e = \mathsf{app}\, e_1\, e_2$ for some e_1 that is also a subterm, so there are most n^2 of these facts too. This means that we can derive no more than $2n + 3n^2$ facts starting from a database containing $\mathsf{eval}\ulcorner e\urcorner$, where e has n subterms. We could give a much more precise analysis than this, but this imprecise analysis certainly bounds the size of the database, ensuring termination, which was our goal.

There is one important caveat to the control flow analysis we have derived. If for some value v we consider the program $\ulcorner((\lambda x.x)\,(\lambda y.y))\,v\urcorner$, we might expect a reasonable control flow analysis to notice that only $\ulcorner\lambda y.y\urcorner$ is passed to the function $\ulcorner\lambda x.x\urcorner$ and that only v is passed to the function $\ulcorner\lambda y.y\urcorner$. Because of our use of higher-order abstract syntax, however, $\ulcorner\lambda y.y\urcorner$ and

$\ulcorner \lambda x.x \urcorner$ are α-equivalent and therefore equal in the eyes of the logic programming interpreter. This is not a problem with correctness, but it means that our analysis may be less precise than expected, because the analysis distinguishes only subterms, not *subterm occurrences*. One solution would be to add distinct labels to terms, marking the α-equivalent $\lambda x.x$ and $\lambda y.y$ with their distinct positions in the overall term. Adding a label on the inside of every lambda-abstraction would seem to suffice, and in any real example labels would already be present in the form of source-code positions or line numbers. The alias analysis presented in the next section demonstrates the use of such labels.

8.4.4 Correctness

The termination analysis for the derived specification in Figure 8.5, together with the meta-approximation theorem (Theorem 8.4), ensures that we have derived some sort of program analysis. How do we know that it is a control flow analysis?

The easy option is to simply inspect the analysis and compare it to the behavior of the SSOS semantics whose behavior the analysis is approximating. Note that the third argument e to $\mathsf{cont}\ f\ e'\ e$ is always a term $\mathsf{app}\ e_1\ e_2$ – that is, a call site. The rule ev/app2 starts evaluating the function $\mathsf{lam}(\lambda x.e_0\ x)$ and generates the fact $\mathsf{cont}\ \mathsf{app3}\ (e(\mathsf{var}(\lambda x.e_0\ x)))\ e$. This means that, in the course of evaluating some initial expression e_{init}, the function $\mathsf{lam}(\lambda x.e_0\ x)$ may be called from the call site e only if $\mathsf{cont}\ \mathsf{app3}\ (e_0(\mathsf{var}(\lambda x.e_0\ x)))\ e$ appears in a saturated process state that includes the persistent atomic proposition $\mathsf{eval}(e_{init})$.

The analysis above is a bit informal, however. Following Nielson et al., an *acceptable control flow analysis* takes the form of two functions. The first, $\widehat{\mathsf{C}}$, is a function from expressions e to sets of values $\{v_1, \ldots, v_n\}$, and the second, $\widehat{\rho}$, is a function from variables x to sets of values $\{v_1, \ldots, v_n\}$. $\widehat{\mathsf{C}}$ and $\widehat{\rho}$ are said to represent an acceptable control flow analysis for the expression e if a coinductively defined judgment $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e$ holds.

We would like to interpret a saturated program state Δ as a (potentially acceptable) control flow analysis as follows (keeping in mind that, given our current interpretation of subterms, $\ulcorner x \urcorner = \mathsf{var}(\lambda x.\ E\ x)$ for some E):

- $\widehat{\mathsf{C}}(e) = \{v \mid \mathsf{retn}\ulcorner v\urcorner\ulcorner e\urcorner\}$, and
- $\widehat{\rho}(x) = \{v \mid \mathsf{bind}\ulcorner x\urcorner\ulcorner v\urcorner\}$.

Directly adapting Nielson et al.'s definition of an acceptable control flow analysis from [NNH05, Table 3.1] turns out not to work. The control flow analysis we derived in Figure 8.5 is rather sensitive to non-termination: if we let $\omega = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$, then our derived control flow analysis will not analyze the argument e_2 in an expression $\omega\ e_2$, nor will it analyze the function body e in an expression $(\lambda x.e)\ \omega$. Nielson et al.'s definition, on the other hand, demands that both e_2 in $\omega\ e_2$ and e in $(\lambda x.e)\ \omega$ be analyzed. In Exercise 3.4, of their book, Nielson et al. point out that a modified analysis, which takes order of evaluation into account, is possible.

We can carry out Nielson et al.'s Exercise 3.4 to get the definition of an acceptable control flow analysis given in Figure 8.6. Relative to this definition, it is possible to prove that the abstractions computed by the derived SLS specification in Figure 8.5 are acceptable control flow analyses.

$$
\begin{array}{ll}
[var] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models x \text{ iff } \widehat{\rho}(x) \subseteq \widehat{\mathsf{C}}(x) \\
[lam] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models \lambda x.e \text{ iff } \{(\lambda x.e)\} \subseteq \widehat{\mathsf{C}}(\lambda x.e) \\
[app] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models e_1\, e_2 \text{ iff} \\
& \qquad (\widehat{\mathsf{C}}, \widehat{\rho}) \models e_1 \wedge \\
& \qquad (\forall (\lambda x.e_0) \in \widehat{\mathsf{C}}(e_1) : \\
& \qquad\quad (\widehat{\mathsf{C}}, \widehat{\rho}) \models e_2 \wedge \\
& \qquad\quad (\widehat{\mathsf{C}}(e_2) \subseteq \widehat{\rho}(x)) \wedge \\
& \qquad\quad (\forall (v) \in \widehat{\mathsf{C}}(e_2) : \\
& \qquad\qquad (\widehat{\mathsf{C}}, \widehat{\rho}) \models e_0 \wedge \\
& \qquad\qquad (\widehat{\mathsf{C}}(e_0) \subseteq \widehat{\mathsf{C}}(e_1\, e_2))))
\end{array}
$$

Figure 8.6: Coinductive definition of an acceptable control flow analysis

Theorem 8.5. *If Δ is a saturated process state that is well-formed according to the signature in Figure 8.5, and if $\widehat{\mathsf{C}}$ and $\widehat{\rho}$ are defined in terms of Δ as described above, then* $\mathsf{eval}\ulcorner e \urcorner \in \Delta$ *implies that* $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e$.

Proof. By coinduction on the definition of acceptability in Figure 8.6, and case analysis on the form of e.

- $e = x$, so $\ulcorner e \urcorner = \ulcorner x \urcorner = \mathsf{var}(\lambda x.\, E_0\, x)$
  We have to show $\widehat{\rho}(x) \subseteq \widehat{\mathsf{C}}(x)$. In other words, if $\mathsf{bind}\ulcorner x \urcorner \ulcorner v \urcorner \in \Delta$, then $\mathsf{retn}\ulcorner v \urcorner \ulcorner x \urcorner \in \Delta$. Because $\mathsf{eval}\ulcorner e \urcorner \in \Delta$, this follows by the presence of rule ev/bind – if $\mathsf{eval}\ulcorner e \urcorner \in \Delta$ and $\mathsf{bind}\ulcorner x \urcorner \ulcorner v \urcorner \in \Delta$, then $\mathsf{retn}\ulcorner v \urcorner \ulcorner x \urcorner \in \Delta$ as well; if it were not, the process state would not be saturated!

- $e = \lambda x.e$, so $\ulcorner e \urcorner = \ulcorner \lambda x.e_0 \urcorner = \mathsf{lam}(\lambda x.\, E_0\, x)$
  We have to show $\{(\lambda x.e)\} \subseteq \widehat{\mathsf{C}}(\lambda x.e)$. In other words, $\mathsf{retn}\ulcorner \lambda x.e \urcorner \ulcorner \lambda x.e \urcorner \in \Delta$. This follows by rule ev/lam by the same reasoning given above.

- $e = e_1\, e_2$, so $\ulcorner e \urcorner = \ulcorner e_1\, e_2 \urcorner = \mathsf{app}\, E_1\, E_2$
  We have to show several things. The first, that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_1$, follows from the coinduction hypothesis – by rule ev/app, $\mathsf{eval}\ulcorner e_1 \urcorner \in \Delta$. That rule also allows us to conclude that $\mathsf{cont}\ \mathsf{app1}\ulcorner e_2 \urcorner \ulcorner e_1 \urcorner \ulcorner e_1\, e_2 \urcorner \in \Delta$.

  Second, given a $(\lambda x.e_0) \in \widehat{\mathsf{C}}(e_1)$ (meaning $\mathsf{retn}\ulcorner \lambda x.e_0 \urcorner \ulcorner e_1 \urcorner \in \Delta$) we have to show that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_2$. This follows from the coinduction hypothesis: by rule ev/app1, because $\mathsf{retn}\ulcorner \lambda x.e_0 \urcorner \ulcorner e_1 \urcorner \in \Delta$ and $\mathsf{cont}\ (\mathsf{app1}\ulcorner e_2 \urcorner) \ulcorner e_1 \urcorner \ulcorner e_1\, e_2 \urcorner \in \Delta$, $\mathsf{eval}\ulcorner e_2 \urcorner \in \Delta$. This same reasoning allows us to conclude that $\mathsf{cont}\ (\mathsf{app2}\ (\lambda x.\ulcorner e_0 \urcorner)) \ulcorner e_2 \urcorner \ulcorner e_1\, e_2 \urcorner \in \Delta$ given that $(\lambda x.e_0) \in \widehat{\mathsf{C}}(e_1)$.

Third, given a $(\lambda x.e_0) \in \widehat{\mathsf{C}}(e_1)$, we have to show that $(\widehat{\mathsf{C}}(e_2) \subseteq \widehat{\rho}(x))$: in other words, that $\mathsf{retn}\ulcorner v_2 \urcorner\ulcorner e_2 \urcorner \in \Delta$ implies $\mathsf{bind}\,(\mathsf{var}(\lambda x.\ulcorner e_0 \urcorner))\ulcorner v_2 \urcorner \in \Delta$. Because we know by the reasoning above that $\mathsf{cont}\,(\mathsf{app2}\,(\lambda x.\ulcorner e_0 \urcorner))\ulcorner e_2 \urcorner\ulcorner e_1\, e_2 \urcorner \in \Delta$, this follows by rule ev/app2.

The same reasoning from ev/app2 allows us to conclude that both $(\lambda x.e_0) \in \widehat{\mathsf{C}}(e_1)$ and $\mathsf{retn}\ulcorner v_2 \urcorner\ulcorner e_2 \urcorner \in \Delta$ together imply $\mathsf{eval}\ulcorner e_0 \urcorner \in \Delta$ (and therefore that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_0$ by the coinduction hypothesis, the fourth thing we needed to prove) in addition to implying $\mathsf{cont}\,\mathsf{app3}\ulcorner e_0 \urcorner\ulcorner e_1\, e_2 \urcorner \in \Delta$ (which with ev/app3 implies $\widehat{\mathsf{C}}(e_0) \subseteq \widehat{\mathsf{C}}(e_1\, e_2)$, the last thing we needed to prove).

This completes the proof. □

We claim that, if we had started with an analysis that incorporated both parallel evaluation of functions and arguments (in the style of Figure 7.6 from Section 7.2.1) and the call-by-future functions discussed in Figure 7.9 from Section 7.2.3, then the derived analysis would have satisfied a faithful representation of Nielson et al.'s acceptability relation. The proof, in this case, should proceed along the same lines as the proof of Theorem 8.5.

8.5 Alias analysis

The control flow analysis above was derived from the SSOS specification of a language that looked much like the Mini-ML-like languages considered in Chapters 6 and 7, and we described how to justify such an analysis in terms of coinductive specifications of what comprises a well-designed control flow analysis.

In this section, we work in the other direction: the starting point for this specification was the interprocedural object-oriented alias analysis presented as a saturating logic program in [ALSU07, Chapter 12.4]. We then worked backwards to get a SSOS semantics that allowed us to derive Aho et al.'s logic program as closely as possible. The result is a monadic SSOS semantics. There should not be any obstacle to deriving an alias analysis from a semantics that looks more like the specifications elsewhere in this dissertation.

8.5.1 Monadic language

The language we consider differentiates atomic actions, which we will call *expressions* (and encode in the LF type exp) and procedures or *commands* (which we encode in the LF type cmd). There are only two commands m in our monadic language. The first command, $\mathsf{ret}\,x$, is a command that returns the value bound to the variable x (rule ev/ret in Figure 8.7). The second command, $\ulcorner \mathsf{bnd}^l\, x \leftarrow e\,\mathsf{in}\,m \urcorner = \mathsf{bnd}\,l\ulcorner e \urcorner\,\lambda x.\ulcorner m \urcorner$, evaluates e to a value, binds that value to the variable x, and then evaluates m. Note the presence of l in the bind syntax; we will call it a *label*, and we can think of it as a line number or source-code position from the original program.

In the previous languages we have considered, values v were a syntactic refinement of the expressions e. In contrast, our monadic language will differentiate the two: there are five expression forms and three values that we will consider. An expression $\ulcorner \lambda x.m_0 \urcorner = \mathsf{fun}\,\lambda x.\ulcorner m_0 \urcorner$

```
bind: variable -> value -> prop pers.
eval: cmd -> dest -> prop lin.
retn: value -> dest -> prop lin.
cont: frame -> dest -> dest -> prop lin.

ev/ret:   eval (ret X) D * !bind X V >-> {retn V D}.

ev/fun:   eval (bnd L (fun \x. M0 x) (\x. M x)) D
           >-> {Exists y. eval (M y) D * !bind y (lam L \x. M0 x)}.

ev/call:  eval (bnd L (call F X) (\x. M x)) D *
          !bind F (lam L0 (\x. M0 x)) *
          !bind X V
           >-> {Exists d0. Exists y.
                eval (M0 y) d0 * cont (call1 L (\x. M x)) d0 D *
                !bind y V}.

ev/call1: retn V D0 * cont (call1 L (\x. M x)) D0 D
           >-> {Exists y. eval (M y) D * !bind y V}.
```

Figure 8.7: Semantics of functions in the simple monadic language

evaluates to a value $\ulcorner \lambda^l x.m_0 \urcorner = \mathsf{lam}\, l\, \lambda x.\ulcorner m_0 \urcorner$, where the label l represents the source code position where the function was bound. (A function value is a command m_0 with one free variable.) When we evaluate the command $\ulcorner \mathsf{bnd}^l\, y \leftarrow \lambda x.m_0\, \mathsf{in}\, m \urcorner$, the value $\ulcorner \lambda^l x.m_0 \urcorner$ gets bound to y in the body of the command m (rule ev/fun in Figure 8.7).

The second expression form is a function call: $\ulcorner f\, x \urcorner = \mathsf{app}\, f\, x$. To evaluate a function call, we expect a function value to be bound to the variable f; we then store the rest of the current command on the stack and evaluate the command m_0 to a value. Note that the rule ev/call in Figure 8.7 also stores the call site's source-code location l on the stack frame. The reason for storing a label here is that we need it for the alias analysis. However, it is possible to independently motivate adding these source-code positions to the operational semantics: for instance, it would allow us to model the process of giving a stack trace when an exception is raised. When the function we have called returns (rule ev/call1 in Figure 8.7), we continue evaluating the command that was stored on the control stack.

The rules for mutable pairs are given in Figure 8.8. Evaluating the expression newpair allocates a tuple with two fields fst and snd and yields a value $\mathsf{loc}\, l$ referring to the tuple; both fields in the tuple are initialized to the value null, and each field is represented by a separate linear cell resource (rule ev/new). The expressions $\ulcorner x.\mathsf{fst} \urcorner = \mathsf{proj}\, x\, \mathsf{fst}$ and $\ulcorner x.\mathsf{snd} \urcorner = \mathsf{proj}\, x\, \mathsf{snd}$ expect a pair location to be bound to x, and yield the value stored in the appropriate field of the mutable pair (rule ev/proj). The expressions $\ulcorner x.\mathsf{fst} := y \urcorner = \mathsf{set}\, x\, \mathsf{fst}\, y$ and $\ulcorner x.\mathsf{snd} := y \urcorner = \mathsf{set}\, x\, \mathsf{snd}\, y$ work much the same way. The difference is that the former expressions do not change the accessed field's contents, whereas the latter expressions replace the accessed field's contents with the value bound to y (rule ev/set).

This language specification bears some similarity to Harper's Modernized Algol with free

```
cell: locvar -> field -> value -> prop lin.

ev/new:   eval (bnd L newpair (\x. M x)) D
           >-> {Exists y. Exists l'. eval (M y) D *
                cell l' fst null * cell l' snd null *
                !bind y (loc l')}.

ev/proj:  eval (bnd L (proj X Fld) (\x. M x)) D *
          !bind X (loc L') *
          cell L' Fld V
           >-> {Exists y. eval (M y) D * cell L' Fld V *
                !bind y V}.

ev/set:   eval (bnd L (set X Fld Y) (\x. M x)) D *
          !bind X (loc L') *
          !bind Y V *
          cell L' Fld V'
           >-> {Exists y. eval (M y) D *
                cell L' Fld V *
                !bind y V'}.
```

Figure 8.8: Semantics of mutable pairs in the simple monadic language

assignables [Har12, Chapter 36]. The *free assignables* addendum is critical: SSOS specifications do not have a mechanism for enforcing the stack discipline of Algol-like languages.[3]

8.5.2 Approximation and alias analysis

To approximate the semantics of our monadic language, we can follow the methodology from before and turn the specification persistent. A further approximation is to remove the last premise from ev/set, as the meta-approximation theorem allows – the only purpose of this premise in Figure 8.8 was to consume the ephemeral proposition $\mathsf{cell}\, l'\, \mathit{fld}\, v$, and this is unnecessary if cell is not an ephemeral predicate. Having made these two moves (turning all propositions persistent, and removing a premise from ev/set), we are left with three types of existentially-generated variables that must be equated with concrete terms in order for our semantics to be interpreted as a saturating logic program:

- Variables y, introduced by every rule except for ev/ret,
- Mutable locations l, introduced by rule ev/new, and
- Destinations d; the only place where a destination is created by the destination-adding transformation is in rule ev/call.

[3] It is, however, possible to represent Algol-like languages that maintain a stack discipline even though the machinery of SLS does not enforce that stack discipline. This is analogous to the situation with pointer equality discussed in Section 6.5.1, as a stack discipline is an invariant that can be maintained in SLS even though the framework's proof theory does not enforce the invariant.

```
bind: label -> value -> prop pers.
eval: cmd -> label -> prop pers.
retn: value -> label -> prop pers.
cont: frame -> label -> label -> prop pers.
cell: label -> field -> value -> prop pers.

ev/ret:   eval (ret X) D * bind X V >-> {retn V D}.

ev/fun:   eval (bnd L (fun \x. M0 x) (\x. M x)) D
           >-> {eval (M L) D * bind L (lam L \x. M0 x)}.

ev/call:  eval (bnd L (call F X) (\x. M x)) D *
          bind F (lam L0 \x. M0 x) *
          bind X V
           >-> {eval (M0 L0) L0 * cont (call1 L (\x. M x)) L0 D}.

ev/call1: retn V D0 * cont (call1 L (\x. M x)) D0 D
           >-> {eval (M L) D * bind L V}.

ev/new:   eval (bnd L newpair (\x. M x)) D
           >-> {eval (M L) D *
                cell L fst null * cell L snd null *
                bind L (loc L)}.

ev/proj:  eval (bnd L (proj X Fld) (\x. M x)) D *
          bind X (loc L') *
          cell L' Fld V
           >-> {eval (M L) D * cell L' Fld V *
                bind L V}.

ev/set:   eval (bnd L (set X Fld Y) (\x. M x)) D *
          bind X (loc L') *
          bind Y V
           >-> {eval (M L) D * cell L' Fld V *
                bind L V'}.
```

Figure 8.9: Alias analysis for the simple monadic language

Variables y are generated to be substituted into the body of some command, so we could equate them with the Skolemized function body as we did when deriving a control flow analysis example. Another option comes from noting that, for any initial source program, every command is associated with a particular source code location, so a simpler alternative is just to equate the variable with that source code location. This is why we stored labels on the stack: if we had not done so, then the label l associated with m in the command $\ulcorner \mathsf{bnd}^l\ x \leftarrow \lambda x.m_0\ \mathsf{in}\ m \urcorner$ would no longer be available when we needed it in rule ev/call1.

We deal with mutable locations l in a similar manner: we equate them with the label l representing the line where that cell was generated.

There are multiple ways to deal with the destination d_0 generated in rule ev/call. We want our analysis, like Aho et al.'s, to be insensitive to control flow, so we will equate d_0 with the label l_0 associated with the function we are calling. If we instead equated d_0 with the label l associated with the call-site or with the pair of the call site and the called function, the result would be an analysis that is more sensitive to control flow.

The choices described above are reflected in Figure 8.9, which takes the additional step of inlining uses of equality in the conclusions of rules. We can invoke this specification as a program analysis by packaging a program as a single command m and deriving a saturated process state from the initial process state $(l_{init}{:}\mathsf{loc}; x{:}\langle \mathsf{eval} \ulcorner m \urcorner\ l_{init}\rangle)$. The use of source-code position labels makes the answers to some of the primary questions asked of an alias analysis quite concise. For instance:

- *Might the first component of a pair created at label l_1 ever reference a pair created at label l_2?* Only if $\mathsf{cell}\ l_1\ \mathsf{fst}\ (\mathsf{loc}\ l_2)$ appears in the saturated process state (and likewise for the second component).
- *Might the first component of a pair created at label l_1 ever reference the same object as the first component of a pair created at label l_2?* Only if there is some l' such that $\mathsf{cell}\ l_1\ \mathsf{fst}\ (\mathsf{loc}\ l')$ and $\mathsf{cell}\ l_1\ \mathsf{fst}\ (\mathsf{loc}\ l')$ both appear in the saturated process state.

8.6 Related work

The technical aspects of linear logical approximation are similar to work done by Bozzano et al. [BDM02, BDM04], which was also based on the abstract interpretation of a logical specification in linear logic. They encode distributed systems and communication protocols in a framework that is similar to the linear fragment of SLS without equality. Abstractions of those programs are then used to verify properties of concurrent protocols that were encoded in the logic [BD02].

There are a number of significant difference between our work and Bozzano et al.'s, however. The style they use to encode protocols is significantly different from any of the SSOS specification styles presented in this dissertation. They used a general purpose approximation, which could therefore potentially be mechanized in the same way we mechanized transformations like operationalization; in contrast, the meta-approximation result described here captures a whole class of approximations. Furthermore, Bozzano et al.'s methods are designed to consider properties of a system as a whole, not static analyses of individual inputs as is the case in our

work.

Work by Might and Van Horn on abstracting abstract machines can be seen as a parallel approach to our methodology in a very different setting [MSV10, Mig10, VM10]. Their emphasis is on deriving a program approximation by approximating a *functional* abstract interpreter for a programming language's operational semantics. Their methodology is similar to ours in large part because we are doing the same thing in a different setting, deriving a program approximation by approximating a destination-passing SSOS specification (which we could, in turn, have derived from an ordered abstract machine by destination-adding).

Many of the steps that they suggest for approximating programs have close analogues in our setting. For instance, their *store-allocated bindings* are analogous to the SSOS environment semantics, and their *store-allocated continuations* – which they motivate by analogy to implementation techniques for functional languages like SML/NJ – are precisely the destinations that arise naturally from the destination-adding transformation. The first approximation step we take is forgetting about linearity in order to obtain a (non-terminating) persistent logical specification. This step is comparable to Might's first approximation step of "throwing hats on everything" (named after the convention in abstract interpretation of denoting the abstract version of a state space Σ as $\hat{\Sigma}$. The "mysterious" introduction of power domains that this entails is, in our setting, a perfectly natural result of relaxing the requirement that there be at most one persistent proposition $\mathsf{bind}\, x\, v$ for every x. As a final point of comparison, the "abstract allocation strategy" discussed in [VM10] is quite similar to our strategy of introducing and then approximating Skolem functions as a means of deriving a finite approximation. Our current discussion of Skolem functions in Section 8.4 is partially inspired by the relationship between our use of Skolemization and the discussion of abstract allocation in [VM10].

The independent discovery of a similar set of techniques for achieving similar goals in such different settings (though both approaches were to some degree inspired by Van Horn and Mairson's investigations of the complexity of k-CFA [VM07]) is another indication of the generality of both techniques, and the similarity also suggests that the wide variety of approximations considered in [VM10], as well as the approximations of object-oriented programming languages in [Mig10], can be adapted to this setting.

Bibliography

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools, Second Edition*. Pearson Education, Inc, 2007. 8.5

[BD02] Marco Bozzano and Giorgio Delzanno. Automated protocol verification in linear logic. In *Principles and Practice of Declarative Programming (PPDP'02)*, pages 38–49. ACM, 2002. 8.6

[BDM02] Marco Bozzano, Giorgio Delzanno, and Maurizio Martelli. An effective fixpoint semantics for linear logic programs. *Theory and Practice of Logic Programming*, 2(1):85–122, 2002. 8.6

[BDM04] Marco Bozzano, Giorgio Delzanno, and Maurizio Martelli. Model checking linear logic specifications. *Theory and Practice of Logic Programming*, 4(5–6):573–619, 2004. 8.6

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, 1977. 8

[GM02] Harald Ganzinger and David A. McAllester. Logical algorithms. In *International Conference on Logic Programming (ICLP'02)*, pages 209–223. Springer LNCS 2401, 2002. 8.1

[GNRT10] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Principles of Programming Languages (POPL'10)*, pages 43–56. ACM, 2010. 8

[Har12] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012. 3.3.4, 6, 6.4.1, 6.5.1, 6.5.2, 6.5.4, 8, 6.5.5, 8.5.1, 7, 9.3.2, 9.4

[LPPW05] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *Principles and Practice of Declarative Programming (PPDP'05)*, pages 35–46. ACM, 2005. 4.6, 4.6.2, 4.7.3, 8.1

[McA02] David A. McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002. 8.1

[Mig10] Matthew Might. Abstract interpreters for free. In *Static Analysis Symposium (SAS'10)*, pages 407–421. Springer LNCS 6337, 2010. 8.6

[MSV10] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploit-

ing the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Programming Language Design and Implementation (PLDI'10)*, pages 305–315. ACM, 2010. 8.6

[NNH05] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2005. 8.4, 8.4.4

[Shi88] Olin Shivers. Control flow analysis in scheme. In *Programming Language Design and Implementation (PLDI'88)*, pages 164–174. ACM, 1988. 8.4

[SP08] Robert J. Simmons and Frank Pfenning. Linear logical algorithms. In *Proceedings of the International Colloquium on Automata, Languages and Programming, Track B (ICALP'08)*, pages 336–347. Springer LNCS 5126, 2008. 3.6.1, 8.1, 10.1

[SP11] Robert J. Simmons and Frank Pfenning. Logical approximation for program analysis. *Higher-Order and Symbolic Computation*, 24(1–2):41–80, 2011. 1.2, 2.5.3, 5.1, 6.5.3, 7, 7, 7.1, 1, 7.1, 7.2, 8, 8.3, 8.3, 8.4.2

[VM07] David Van Horn and Harry G. Mairson. Relating complexity and precision in control flow analysis. In *International Conference on Functional Programming (ICFP'07)*, pages 85–96. ACM, 2007. 8.6

[VM10] David Van Horn and Matthew Might. Abstracting abstract machines. In *International Conference on Functional Programming (ICFP'10)*, pages 51–62. ACM, 2010. 8.6