A Proof-Theoretic Approach to the Static Analysis of Logic Programs

Dale Miller

Dedicated to Peter Andrews
on the occasion of his 70^{th} birthday.

1 Introduction

Static analysis of programs can provide useful information for programmers and compilers. Type checking, a common form of static analysis, can help identify errors during program compilation that might otherwise be found only when the program is executed, possibly by someone other than the programmer. The concise invariants that comes from static analysis can also provide valuable documentation about the meaning of code.

We describe a method that approximates a data structure by a collection of the elements it contains and statically determines whether or not the relations computed by a logic program satisfy certain relations over those approximations. More specifically, we shall use *multisets* and *sets* to *approximate* more structured data such as lists and binary trees. Consider, for example, a list sorting program that maintains duplicates of elements during sorting. Part of the correctness of a sort program includes the fact that if the atomic formula $(sort\ t\ s)$ is provable then s is a permutation of t that is in-order. The proof of such a property is likely to involve inductive arguments requiring the invention of invariants: in other words, this is not likely to be a property that can be inferred statically. On the other hand, if the lists t and s are approximated by multisets (that is, if we forget the order of items in lists), then it might be possible to prove automatically half of this property about the sorting program: namely, if the atomic formula $(sort\ t\ s)$ is provable then the multiset associated to t is equal to the multiset associated to s. If that is so, then it is immediate that the lists t and s are, in fact, permutations of one another (in other words, no elements were dropped, duplicated, or created during sorting). As we shall see, such properties based on using multisets to approximate lists can often be proved statically.

This paper, which is based on [21], exploits three aspects of proof theory to present a scheme for static analysis. First, logical formulas, even

those comprising just first-order Horn clauses, are considered as part of a higher-order logic, such as the Simple Theory of Types [4, 7]. In such a setting, all constants, including predicate and function constants, can be abstracted and instantiated by other logical expressions: such abstractions and instantiations can be completely explained following the usual rules for the λ-calculus. Second, traces of logic program executions can be seen as cut-free sequent calculus proofs [22] and since sequent calculus proofs also support rich notions of abstraction and instantiation, it is possible to reason directly on logic program computations via standard proof-theoretic notions. Third, linear logic can be seen as the computational logic behind logic and via the instantiation mechanisms available for both formulas and proofs, linear logic can be put behind-the-scenes of Horn clause computation. In this background world, linear logic is used to perform basic computations with sets and multisets.

2 The undercurrents

There are various themes that underlie this approach to inferring properties of Horn clause programs. This section enumerates several such themes. The rest of this paper can be seen as a particular manifestation of these themes.

2.1 If typing is important, why use only one?

Types and other static properties of programming languages have proved important on a number of levels. Typing is useful to programmers since it offers important invariants and documentation for code. Static analysis can also be used by compilers to uncover useful structures that allow compilers to improve program execution. While compilers might make use of multiple static analyses, programmers do not have convenient access to multiple static analyzes. Sometimes a programming language definition provides for no static analysis, as is the case with Lisp and Prolog. Other programming languages offer exactly one typing discipline, such as the polymorphic typing disciplines of Standard ML and λProlog. Simple and fixed static checks are occasionally also part of a language definition, as is the case in SML where static checking is done to determine if a given function definition over concrete data structures covers all possible input values. It seems clear, however, that static analysis of code, if it can be done quickly and incrementally, should have significant benefits for programmers during the process of writing code. For example, a programmer might find it valuable to know that the recursive program that she has just written has linear or quadratic run-time complexity, or that a relation she just specified actually defines a function. The Ciao system pre-processor [14] provides for such functionality by allowing a programmer to write various properties about

code that the pre-processor attempts to verify. Providing a flexible framework for the integration of static analysis into programming languages is an interesting direction of research in the design of programming languages. The paper provides one possible scheme for such integration.

2.2 Abstracting over programs and computation traces

A computational system can be seen as encoding symbolic systems on a number of levels: types, program expressions, static analysis expressions, and computation traces. All of these can benefit from representations that allow for natural notions of abstractions and instantiations. For example, we shall consider first-order Horn clauses as part of the Church's Simple Theory of Types [4, 7]. As is well understood in that setting, quantifier instantiation is completely described using the theory's underlying rules for λ-conversion. Similarly, traces of logic program computations can be seen as cut-free proofs and such proof objects also have rich notions of abstraction and application, given by the cut-elimination theorem for sequent calculus. The fact that proofs and programs can be related simply in a setting where substitution into both has well understood properties is certainly one of the strengths of the proof-theoretic foundations of logic programming (see, for example, [22]).

2.3 What good are atomic formulas?

In proof theory, there is an interesting problem of duality involving atomic formulas. The *initial rule* and the *cut rule*, given as

$$\frac{}{C \vdash C}\ \text{Initial} \quad \text{and} \quad \frac{\Gamma_1 \vdash C, \Delta_1 \quad \Gamma_2, C \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \text{Cut},$$

can be seen as being dual to each other [13]. In particular, the initial rule states that an occurrence of a formula on the left is stronger than the same occurrence on the right, whereas the cut rule states the dual: an occurrence of a formula on the right is strong enough to remove the same occurrence from the left. In most well designed proof systems, atomic and non-atomic occurrences of the cut-rule can be eliminated whereas only non-atomic initial rules (where C is non-atomic) can be eliminated. Atoms seem to spoil the elegant duality of the meta-theory of these inference rules.

While the logic programming world is most comfortable with the existence of atomic formulas, there have been a couple of recent proof-theoretic developments that try to eliminate them entirely. For example, in the work on *definitions* and *fixed points* described in [11, 17, 26], atoms are defined to be other formulas and the only primitive judgment involving terms is that of equality. Furthermore, if fixed points are *stratified* (no recursion through negations) and *noetherian* (no infinite descent in recursion), then

all instances of cut and initial can be removed. Girard's *ludics* [12] is a more radical presentation of logic in which atomic formulas do not exist: formulas can be probed to arbitrary depth to uncover "subformulas". Another approach to atoms is to consider *all* constants as being variables: such an approach is possible in a higher-order logic by, say, replacing constants with universally quantified variables. On one hand this is a trivial position: if there are no constants (thus, no predicate constants) there are no atomic formulas (which are defined as formulas with non-logical constants at their head). On the other hand, adopting a point-of-view that constants can vary has some appeal. We describe this next.

2.4 Viewing constants and variables as one

The inference rule of $\forall$-generalization states that if B is provable then $\forall x.B$ is provable (with the appropriate proviso if the proof of B depends on hypotheses). In a first-order setting, only a free first-order variable, say x of B, can become bound in $\forall x.B$ by this inference rule. In a higher-order setting, any variable in any expression, even those that play the role of predicates or functions, can be quantified.

The difference between constants and variables can be seen as one of "scope," at least from a syntactic, proof-theoretic, and computational point of view. For example, variables are intended as syntactic objects that can "vary." During the computation of, say, the relation of appending lists, universal quantified variables surrounding Horn clauses change via substitution (during back-chaining and unification) but the constructors for lists as well as the symbol denoting the append relation do not change (are not instantiated) and, hence, can be seen as constants. But from a compiling and linking point-of-view, the append predicate might be considered something that varies: if append is in a module of Prolog that is separately compiled, the append symbol might denote a particular object in the compiled code that is later changed when the code is loaded and linked. In a similar fashion, we shall allow ourselves to instantiate constants with expressions during static analysis: that is, constants can be seen as varying over different approximations of their intended meaning.

Substituting for constants allows us to "split the atom": that is, by substituting for the predicate p in the atom $p(t_1, \ldots, t_n)$, we can replace that atom with a formula, which, in this paper, will be a linear logic formula that accounts for some resources related to the arguments $t_1, \ldots, t_n$.

2.5 Linear logic underlies computational logic

Linear logic [10] is able to explain the proof theory of usual Horn clause logic programming (and even richer logic programming languages [15]). It is also able to provide means to reason about resources, such as items in multi-

sets and sets. Thus, linear logic will allow us to sit within one declarative framework to describe both usual logic programming as well as "sub-atomic" reasoning about the resources implicit in the arguments of predicates.

3 A primer for linear logic

Linear logic connectives can be divided into the following groups: the multiplicatives $\parr$, $\bot$, $\otimes$, $\mathbf{1}$; the additives $\oplus$, $\mathbf{0}$, $\&$, $\top$; the exponentials $!$, $?$; the implications $\multimap$ (where $B \multimap C$ can be defined as $B^{\bot} \parr C$) and $\Rightarrow$ (where $B \Rightarrow C$ can be defined as $(!\, B)^{\bot} \parr C$); and the quantifiers $\forall$ and $\exists$ (higher-order quantification is allowed). The equivalence of formulas in linear logic, $B \circ\!\!-\!\!\circ C$, is defined as the formula $(B \multimap C) \,\&\, (C \multimap B)$. The quantifiers should be typed, say as $\forall_\tau$ and $\exists_\tau$, where τ is a simple type: in general, however, we will not write these type subscripts and assume that the reader can reconstruct them from context when their value is important.

First-order Horn clauses are formulas of the form

$$\forall x_1 \ldots \forall x_m [A_1 \wedge \ldots \wedge A_n \supset A_0] \qquad (n, m \geq 0)$$

where $\wedge$ and $\supset$ are intuitionistic or classical logic conjunction and implication and $x_1, \ldots, x_m$ are variables of primitive types. There are at least two natural mappings of Horn clauses into linear logic. The "multiplicative" mapping uses the $\otimes$ and $\multimap$ for the conjunction and implication: this encoding is used in, say, the linear logic programming settings, such as Lolli [15], where Horn clause programming can interact with the surrounding linear aspects of the full programming language. Here, we are not interested in linear logic programming *per se* but with using linear logic to help establish invariants about Horn clauses when these are interpreted in the usual, classical setting. As a result, we shall encode Horn clauses into linear logic using the "additive" conjunction $\&$ and the implication $\Rightarrow$: that is, we take Horn clauses to be formulas of the form

$$\forall x_1 \ldots \forall x_m [A_1 \,\&\, \ldots \,\&\, A_n \Rightarrow A_0]. \qquad (n, m \geq 0)$$

The usual proof search behavior of first-order Horn clauses in classical (and intuitionistic) logic is captured precisely when this style of linear logic encoding is used. An example of a Horn clause logic program is given in Figure 1.

4 A primer for proof theory

A *sequent* is a triple of the form $\Sigma; \Gamma \vdash \Delta$ where Σ, the signature, is a list of non-logical constants and eigenvariables paired with a simple type, and where both Γ and Δ are multisets of Σ-formulas (i.e., formulas all of whose

$$\forall \mathtt{K}.\ [\mathtt{append\ nil\ K\ K}]$$
$$\forall \mathtt{X, L, K, M}.\ [\mathtt{append\ L\ K\ M} \Rightarrow \mathtt{append\ (cons\ X\ L)\ K\ (cons\ X\ M)}]$$
$$\forall \mathtt{X}.\ [\mathtt{split\ X\ nil\ nil\ nil}]$$
$$\forall \mathtt{X, A, B, R, S}.[\mathtt{le\ A\ X} \,\&\, \mathtt{split\ X\ R\ S\ B} \Rightarrow \mathtt{split\ X\ (cons\ A\ R)\ (cons\ A\ S)\ B}]$$
$$\forall \mathtt{X, A, B, R, S}.[\mathtt{gr\ A\ X} \,\&\, \mathtt{split\ X\ R\ S\ B} \Rightarrow \mathtt{split\ X\ (cons\ A\ R)\ S\ (cons\ A\ B)}]$$
$$\mathtt{sort\ nil\ nil}$$
$$\forall \mathtt{F, R, S, Sm, B, SS, BS}.\ [$$
$$\mathtt{split\ F\ R\ Sm\ B} \,\&\, \mathtt{sort\ Sm\ SS} \,\&\, \mathtt{sort\ B\ BS} \,\&\, \mathtt{append\ SS\ (cons\ F\ BS)\ S}$$
$$\Rightarrow \mathtt{sort\ (cons\ F\ R)\ S}]$$

Figure 1. Some Horn clauses for specifying a sorting relation.

non-logical symbols are in Σ). The rules for linear logic are the standard ones [10], except here signatures have been added to sequents. The rules for quantifier introduction are the only rules that require the signature and they are reproduced here.

$$\frac{\Sigma, y{:}\,\tau; B[y/x], \Gamma \vdash \Delta}{\Sigma; \exists_\tau x.B, \Gamma \vdash \Delta}\exists L \quad \frac{\Sigma \vdash t{:}\,\tau \quad \Sigma; \Gamma \vdash B[t/x], \Delta}{\Sigma; \Gamma \vdash \exists_\tau x.B, \Delta}\exists R$$

$$\frac{\Sigma \vdash t{:}\,\tau \quad \Sigma; B[t/x], \Gamma \vdash \Delta}{\Sigma; \forall_\tau x.B, \Gamma \vdash \Delta}\forall L \quad \frac{\Sigma, y : \tau; \Gamma \vdash B[y/x], \Delta}{\Sigma; \Gamma \vdash \forall_\tau x.B, \Delta}\forall R$$

The premise $\Sigma \vdash t{:}\,\tau$ is the judgment that the term t has the (simple) type τ given the typing declaration contained in Σ.

We now outline three ways to do instantiation within the sequent calculus.

4.1 Substituting for types

Following Church [7], we shall assume that formulas are given the simple type o (omicron). Simple type expressions appear within sequent calculus proofs (in particular, within signatures and subscripts to quantifiers) without abstractions: that is, they are global and (in this setting) admit no bindings. As a result, it is an easy matter to show that if one replaces every occurrence of a type constant (different from o) in a proof with another type expressions, the result is another valid proof. We shall do this kind of substitution for type constants later when we replace a list by a multiset that approximates it: since we use linear logic formulas to encode multisets, we shall replace the type constant `list` with `o`.

4.2 Substituting for non-logical constants

Consider the linear logic sequent

$$\Sigma, p\colon \tau; !\, D_1, !\, D_2, !\, \Gamma \vdash p(t_1, \ldots, t_m),$$

where the type τ is a predicate type (that is, it is of the form $\tau_1 \to \cdots \to \tau_m \to o$) and where p appears in, say, D_1 and D_2 and in no formula of Γ. The linear logic exponential ! is used here to encode the fact that the formulas D_1 and D_2 are available for arbitrary reuse within a proof (the usual case for program clauses). Using the right introduction rules for implication and the universal quantifier, it follows that the sequent

$$\Sigma; !\, \Gamma \vdash \forall p[D_1 \Rightarrow D_2 \Rightarrow p(t_1, \ldots, t_m)]$$

is also provable. Using the rules for universal quantifiers, there must be proofs for all instances of this quantifier. Let θ be the substitution $[p \mapsto \lambda x_1 \ldots \lambda x_m.S]$, where S is a formula over the signature $\Sigma \cup \{x_1, \ldots, x_m\}$ of type o. A consequence of the proof theory of linear logic is that there is a proof also of

$$\Sigma; !\, \Gamma \vdash D_1\theta \Rightarrow D_2\theta \Rightarrow S[t_1/x_1, \ldots, t_m/x_m]$$

and of the sequent

$$\Sigma; !\, D_1\theta, !\, D_2\theta, !\, \Gamma \vdash S[t_1/x_1, \ldots, t_m/x_m].$$

As this example illustrates, it is possible to instantiate a predicate (here p) with an abstraction of a formula (here, $\lambda x_1 \ldots \lambda x_m.\ S$): such an instantiation carries a provable sequent to a provable sequent.

4.3 Substituting for assumptions

An instance of the cut-rule (mentioned earlier) is the following:

$$\frac{\Sigma; \Gamma_1 \vdash B \qquad \Sigma; B, \Gamma_2 \vdash C}{\Sigma; \Gamma_1, \Gamma_2 \vdash C}$$

This inference rule (especially when associated with the cut-elimination procedure) provides a way to instantiate a hypothetical use of a formula (here, B) with a proof of that formula. For example, consider the following situation. Given the example in the Section 4.2, assume that we can prove

$$\Sigma; !\, \Gamma \vdash !\, D_1\theta \quad \text{and} \quad \Sigma; !\, \Gamma \vdash !\, D_2\theta.$$

Using two instances of the cut rule and the proofs of these sequent, it is possible to obtain a proof of the sequent

$$\Sigma; !\, \Gamma \vdash S[t_1/x_1, \ldots, t_m/x_m]$$

(contraction on the left for !'ed formulas must be applied).

Thus, by a series of instantiations of proofs, it is possible to move from a proof of, say,

$$\Sigma, p{:}\,\tau; !\,D_1, !\,D_2, !\,\Gamma \vdash p(t_1, \ldots, t_m)$$

to a proof of

$$\Sigma; !\,\Gamma \vdash S[t_1/x_1, \ldots, t_m/x_m].$$

Such reasoning about proofs allows us to "split the atom" $p(t_1, \ldots, t_m)$ into a formula $S[t_1/x_1, \ldots, t_m/x_m]$ and to transform proofs involving that atom into proofs involving that formula. In what follows, the formula S will be a linear logic formula that provides an encoding of some judgment about the data structures encoded in the terms $t_1, \ldots, t_m$.

A few simple examples of using higher-order instantiations of logic programs in order to support reasoning about them appear in [20].

5 Encoding multisets as formulas

Linear logic can encode multisets and sets as well as simple judgments about them (such as inclusion and equality). We consider multisets first and tackle sets in Section 8. Let the token *item* be a linear logic predicate of one argument: the linear logic atomic formula *item* x will denote the multiset containing the element x occurring once. There are two natural encoding of multisets into formulas using this predicate. The *conjunctive* encoding uses $\mathbf{1}$ for the empty multiset and $\otimes$ to combine two multisets. For example, the multiset $\{1, 2, 2\}$ is encoded by the linear logic formula *item* $1 \otimes$ *item* $2 \otimes$ *item* 2. Proofs search using this style encoding places multisets on the left of the sequent arrow. This approach is favored when an intuitionistic subset of linear logic is used, such as in Lolli [15], LinearLF [6], and MSR [5]. The dual encoding, the *disjunctive* encoding, uses $\bot$ for the empty multiset and $\parr$ to combine two multisets. Proofs search using this style encoding places multisets on the right of the sequent arrow and, hence, multiple conclusion sequents are now required. Systems such as LO [2] and Forum [19] use this style of encoding. If negation is available, then the choice of which encoding one chooses is mostly a matter of style. We pick the disjunctive encoding for the rather shallow reason that the inclusion judgment for multisets and sets is encoded as an implication instead of a reverse implication, as we shall now see.

Let S and T be the two formulas

$$\mathit{item}\ s_1 \parr \cdots \parr \mathit{item}\ s_n \text{ and } \mathit{item}\ t_1 \parr \cdots \parr \mathit{item}\ t_m, \quad (n, m \geq 0)$$

respectively. Notice that $\vdash S \multimap T$ if and only if $\vdash T \multimap S$ if and only if the two multisets $\{s_1, \ldots, s_n\}$ and $\{t_1, \ldots, t_m\}$ are equal. Consider the following two ways for encoding the multiset inclusion $S \sqsubseteq T$.

- $S \parr 0 \multimap T$. This formula mixes multiplicative connectives with the additive connective 0: the latter allows items that are not matched between S and T to be deleted.

- $\exists q(S \parr q \multimap T)$. This formula mixes multiplicative connectives with a higher-order quantifier. While we can consider the instantiation for q to be the multiset difference of S from T, there is no easy way in the logic to enforce that interpretation of the quantifier.

As it turns out, these two approaches are equivalent in linear logic: in particular, $\vdash \mathbf{0} \circ\!\!-\!\!\circ \forall p.p$ (linear logic absurdity) and

$$\vdash \forall S \forall T[(S \parr 0 \multimap T) \circ\!\!-\!\!\circ \exists q(S \parr q \multimap T)].$$

Thus, below we can choose either one of these encodings for multiset inclusion.

6 Multisets approximations

A *multiset expression* is a formula in linear logic built from the predicate symbol *item* (denoting the singleton multiset), the linear logic multiplicative disjunction $\parr$ (for multiset union), and the unit $\bot$ for $\parr$ (used to denote the empty multiset). We shall also allow a propositional variable (a variable of type o) to be used to denote a (necessarily open) multiset expression. An example of an open multiset expression is $item\ f(X) \parr \bot \parr Y$, where Y is a variable of type o, X is a first-order variable, and f is some first-order term constructor.

Let S and T be two multiset expressions. The two *multiset judgments* that we wish to capture are multiset inclusion, written as $S \sqsubseteq T$, and equality, written as $S \stackrel{m}{=} T$. We shall use the syntactic variable ρ to range over these two judgments, which are formally binary relations of type $o \to o \to o$. A *multiset statement* is a closed formula of the form

$$\forall \bar{x}[S_1\ \rho_1\ T_1 \mathbin{\&} \cdots \mathbin{\&} S_n\ \rho_n\ T_n \Rightarrow S_0\ \rho_0\ T_0],$$

where the quantified variables $\bar{x}$ are either first-order or of type o and formulas $S_0, T_0, \ldots, S_n, T_n$ are possibly open multiset expressions.

If S and T are closed multiset expressions, then we write $\models_m S \sqsubseteq T$ whenever the multiset (of closed first-order terms) denoted by S is contained in the multiset denoted by T, and we write $\models_m S \stackrel{m}{=} T$ whenever the multisets denoted by S and T are equal. Similarly, we write

$$\models_m \forall \bar{x}[S_1\ \rho_1\ T_1 \mathbin{\&} \cdots \mathbin{\&} S_n\ \rho_n\ T_n \Rightarrow S_0\ \rho_0\ T_0]$$

if for all *multiset-valued* closed substitutions θ such that $\models_m S_i\theta\ \rho_i\ T_i\theta$ for all $i = 1, \ldots, n$, it is the case that $\models_m S_0\theta\ \rho_0\ T_0\theta$. A *multiset-valued* substitution is one where variables of propositional type (type o) are mapped to multiset expressions.

The following proposition is central to our use of linear logic to establish multiset statements for Horn clause programs. The proof of this proposition is given in Section 9.2.

Proposition 1. Let $S_0, T_0, \ldots, S_n, T_n$ $(n \geq 0)$ be multiset expressions all of whose free variables are in the list of variables $\bar{x}$. For each judgment $s\ \rho\ t$ we write $s\ \hat{\rho}\ t$ to denote $s \otimes \mathbf{0} \multimap t$ if ρ is $\sqsubseteq$ and $t \multimap\!\circ\!\!- s$ if ρ is $\stackrel{m}{=}$. If

$$\forall\bar{x}[S_1\ \hat{\rho}_1\ T_1\ \&\ \ldots\ \&\ S_n\ \hat{\rho}_n\ T_n \Rightarrow S_0\ \hat{\rho}_0\ T_0]$$

is provable in linear logic, then

$$\models_m \forall\bar{x}[S_1\ \rho_1\ T_1\ \&\ \cdots\ \&\ S_n\ \rho_n\ T_n \Rightarrow S_0\ \rho_0\ T_0]$$

This proposition shows that linear logic can be used in a sound way to infer valid multiset statements. On the other hand, the converse (completeness) does not hold: the statement

$$\forall x \forall y.(x \sqsubseteq y)\ \&\ (y \sqsubseteq x) \Rightarrow (x \stackrel{m}{=} y)$$

is valid but its translation into linear logic is not provable.

To illustrate how deduction in linear logic can be used to establish a property of a logic program, consider the first-order Horn clause program in Figure 1. The signature for this collection of clauses can be given as follows:

```
nil     : list
cons    : int  -> list -> list
append  : list -> list -> list -> o
split   : int  -> list -> list -> list -> o
sort    : list -> list -> o
le, gr  : int  -> int  -> o
```

The first two declarations provide constructors for empty and non-empty lists; the next three are predicates defined by Horn clause in Figure 1; and the last two are order relations that are apparently defined elsewhere.

If we think of lists as collections of items, then we might want to check that the sort program as written does not drop, duplicate, or create any elements. That is, if the atom (`sort` t s) is provable then the multiset of items in the list denoted by t is equal to the multiset of items in the list denoted by s. If this property holds then s and t are lists that are

permutations of each other: of course, this does not say that it is the correct permutation but this more simple fact is one that, as we show, can be inferred automatically.

Checking this property of our example logic programming follows the following three steps.

First, we provide an approximation of lists as being, in fact, multisets: more precisely, as linear logic *formulas* denoting multisets. The first step, therefore, must be to substitute `o` for `list` in the signature above. Now we can interpret the constructors for lists using the substitution

$$\texttt{nil} \mapsto \bot \qquad \texttt{cons} \mapsto \lambda x \lambda y.\ \mathit{item}\ x \parr y.$$

Under such a mapping, the list (`cons 1` (`cons 3` (`cons 2 nil`))) is mapped to the multiset expression $\mathit{item}\ 1 \parr \mathit{item}\ 3 \parr \mathit{item}\ 2 \parr \bot$.

Second, we associate with each predicate in Figure 1 a multiset judgment that encodes an invariant concerning the multisets denoted by the predicate's arguments. For example, if (`append` $r\ s\ t$) or (`split` $u\ t\ r\ s$) is provable then the multiset union of the items in r with those in s is equal to the multiset of items in t, and if (`sort` $t\ s$) is provable then the multisets of items in lists s and t are equal. This association of multiset judgments to atomic formulas can be achieved formally using the following substitutions for constants:

$$\texttt{append} \mapsto \lambda x \lambda y \lambda z.\ (x \parr y) \multimap z \qquad \texttt{split} \mapsto \lambda u \lambda x \lambda y \lambda z.\ (y \parr z) \multimap x$$
$$\texttt{sort} \mapsto \lambda x \lambda y.\ x \multimap y$$

The predicates `le` and `gr` (for the less-than-or-equal-to and greater-than relations) make no statement about collections of items: thus they can be mapped to the trivial tautology $\mathbf{1}$ (the multiplicative truth) via the substitution

$$\texttt{le} \mapsto \lambda x \lambda y.\ \mathbf{1} \qquad \texttt{gr} \mapsto \lambda x \lambda y.\ \mathbf{1}.$$

Figure 2 presents the result of applying these mappings to Figure 1.

Third, we must now attempt to prove each of the resulting formulas. In the case of Figure 2, all the displayed formulas are trivial theorems of linear logic.

Having taken these three steps, we now claim that we have proved the intended collection judgments associate to each of the logic programming predicates above: in particular, we have now shown that our particular sort program computes a permutation.

7 Formalizing the method

The formal correctness of this three stage approach is easily justified given the substitution properties we presented in Section 4 for the sequent calculus

$$\forall \mathtt{K}.\ [\bot \parr \mathtt{K} \multimap \mathtt{K}]$$
$$\forall \mathtt{X}, \mathtt{L}, \mathtt{K}, \mathtt{M}.\ [\mathtt{L} \parr \mathtt{K} \multimap \mathtt{M} \Rightarrow \mathit{item}\ \mathtt{X} \parr \mathtt{L} \parr \mathtt{K} \multimap \mathit{item}\ \mathtt{X} \parr \mathtt{M}]$$
$$\forall \mathtt{X}.\ [\bot \parr \bot \multimap \bot]$$
$$\forall \mathtt{X}, \mathtt{A}, \mathtt{B}, \mathtt{R}, \mathtt{S}.\ [(\mathtt{S} \parr \mathtt{B}) \multimap \mathtt{R} \Rightarrow \mathbf{1} \Rightarrow \mathit{item}\ \mathtt{A} \parr \mathtt{S} \parr \mathtt{B} \multimap \mathit{item}\ \mathtt{A} \parr \mathtt{R}$$
$$\forall \mathtt{X}, \mathtt{A}, \mathtt{B}, \mathtt{R}, \mathtt{S}.\ [(\mathtt{S} \parr \mathtt{B}) \multimap \mathtt{R} \Rightarrow \mathbf{1} \Rightarrow \mathtt{S} \parr \mathit{item}\ \mathtt{A} \parr \mathtt{B} \multimap \mathit{item}\ \mathtt{A} \parr \mathtt{R}$$
$$[\bot \multimap \bot]$$
$$\forall \mathtt{F}, \mathtt{R}, \mathtt{S}, \mathtt{Sm}, \mathtt{Bg}, \mathtt{SS}, \mathtt{BS}.\ [$$
$$\mathtt{Sm} \parr \mathtt{B} \multimap \mathtt{R} \,\&\, \mathtt{Sm} \multimap \mathtt{SS} \,\&\, \mathtt{B} \multimap \mathtt{BS} \,\&\, \mathtt{SS} \parr \mathit{item}\ \mathtt{F} \parr \mathtt{BS} \multimap \mathtt{S} \Rightarrow$$
$$\mathit{item}\ \mathtt{F} \parr \mathtt{R} \multimap \mathtt{S}]$$

Figure 2. The linear logic formulas that result from instantiating the non-logical constants in the Horn clauses in Figure 1.

presentation of linear logic.

Let Γ denote a set of formulas displayed in Figure 1 and let Σ be the signature for Γ. Let θ denote the substitution described above for the type `list`, for the constructors `nil` and `cons`, and for the predicates in Figure 1. Finally, let Σ' be the signature of the range of θ (in this case, it just contains the constant *item*). Then, $\Gamma\theta$ is the set of formula in Figure 2.

Assume now that $\Sigma; \Gamma \vdash (\mathit{sort}\ t\ s)$ is provable. Given the discussion in Sections 4.1 and 4.2, we know that

$$\Sigma'; \Gamma\theta \vdash t\theta \multimap s\theta$$

is provable. Since the formulas in $\Gamma\theta$ are provable, we can use substitution into proofs (Section 4.3) to conclude that $\Sigma'; \vdash t\theta \multimap s\theta$ is provable. Given Proposition 1, we can conclude that $\models_m t\theta \stackrel{m}{=} s\theta$: that is, that $t\theta$ and $s\theta$ encode the same multiset.

Consider the following model theoretic argument for establishing similar properties of Horn clauses. Let $\mathcal{M}$ be the Herbrand model that captures the invariants that we have in mind. In particular, $\mathcal{M}$ contains the atoms (`append` $r\ s\ t$) and (`split` $u\ t\ r\ s$) if the items in the list r added to the items in list s are the same as the items in t. Furthermore, $\mathcal{M}$ contains all closed atoms of the form (`le` $t\ s$) and (`gr` $t\ s$), and closed atoms (`sort` $t\ s$) where s and t are lists that are permutations of one another. One can now show that $\mathcal{M}$ satisfies all the Horn clauses in Figure 1. As a consequence of the soundness of first-order classical logic, any atom provable from the clauses in Figure 1, must be true in $\mathcal{M}$. By construction of $\mathcal{M}$, this means that the desired invariant holds for all atoms proved from the program.

The approach of this paper essentially replaces the construction of a

model and the determination of truth in that model with deduction in linear logic.

8 Sets approximations

Linear logic can also be used to encode sets and the judgments of set equality and inclusion. In fact, the transition to sets from multisets is provided by the use of the linear logic exponential: since we are using the disjunctive encoding of collections (see the discussion in Section 5), we use the ? exponential (if we were using the conjunctive encoding, we would use the ! exponential).

The expression ? *item* t can be seen as describing the presence of an item for which the exact multiplicity does not matter: this formula represents the capacity to be used any number of times. Thus, the set $\{x_1, \ldots, x_n\}$ can be encoded as ? *item* $x_1 \parr \cdots \parr$? *item* x_n. Using logical equivalences of linear logic, this formula is also equivalent to the formula ?(*item* $x_1 \oplus \cdots \oplus$ *item* x_n). This latter encoding is the one that we shall use for building our encoding of sets.

A *set expression* is a formula in linear logic built from the predicate symbol *item* (denoting the singleton set), the linear logic additive disjunction $\oplus$ (for set union), and the unit $\mathbf{0}$ for $\oplus$ (used to denote the empty set). We shall also allow a propositional variable (a variable of type o) to be used to denote a (necessarily open) set expression. An example of an open set expression is *item* $f(X) \oplus \mathbf{0} \oplus Y$, where Y is a variable of type o, X is a first-order variable, and f is some first-order term constructor.

Let S and T be two set expressions. The two *set judgments* that we wish to capture are set inclusion, written as $S \subseteq T$, and equality, written as $S \stackrel{s}{=} T$. We shall use the syntactic variable ρ to range over these two judgments, which are formally binary relations of type $o \to o \to o$. A *set statement* is a formula of the form

$$\forall \bar{x}[S_1\ \rho_1\ T_1 \,\&\, \cdots \,\&\, S_n\ \rho_n\ T_n \Rightarrow S_0\ \rho_0\ T_0]$$

where the quantified variables $\bar{x}$ are either first-order or of type o and formulas $T_0, S_0, \ldots, T_n, S_n$ are possibly open set expressions.

If S and T are closed set expressions, then we write $\models_s S \subseteq T$ whenever the set (of closed first-order terms) denoted by S is contained in the set denoted by T, and we write $\models_s S \stackrel{s}{=} T$ whenever the sets denoted by S and T are equal. Similarly, we write

$$\models_s \forall \bar{x}[S_1\ \rho_1\ T_1 \,\&\, \cdots \,\&\, S_n\ \rho_n\ T_n \Rightarrow S_0\ \rho_0\ T_0]$$

if for all *set-valued* closed substitutions θ such that $\models_s S_i\theta\ \rho_i\ T_i\theta$ for all $i = 1, \ldots, n$, it is the case that $\models_s S_0\theta\ \rho_0\ T_0\theta$. A *set-valued* substitution

is one where variables of propositional type (type o) are mapped to set expressions.

The following proposition is central to our use of linear logic to establish set statements for Horn clause programs. The proof of this proposition in Section 9.1.

Proposition 2. Let $S_0, T_0, \ldots, S_n, T_n$ $(n \geq 0)$ be set expressions all of whose free variables are in the list of variables $\bar{x}$. For each judgment $s\ \rho\ t$ we write $s\ \hat{\rho}\ t$ to denote $?\,s \multimap ?\,t$ if ρ is $\subseteq$ and $(?\,s \multimap ?\,t)\ \&\ (?\,t \multimap ?\,s)$ if ρ is $\stackrel{s}{=}$. If

$$\forall \bar{x}[S_1\ \hat{\rho}_1\ T_1\ \&\ \ldots\ \&\ S_n\ \hat{\rho}_n\ T_n \Rightarrow S_0\ \hat{\rho}_0\ T_0]$$

is provable in linear logic, then

$$\models_s \forall \bar{x}[S_1\ \rho_1\ T_1\ \&\ \cdots\ \&\ S_n\ \rho_n\ T_n \Rightarrow S_0\ \rho_0\ T_0]$$

Lists can be approximated by sets by using the following substitution:

$$\texttt{nil} \mapsto \mathbf{0} \qquad \texttt{cons} \mapsto \lambda x \lambda y.\ \mathit{item}\ x \oplus y.$$

Under such a mapping, the list (`cons 1 (cons 2 (cons 2 nil))`) is mapped to the set expression $\mathit{item}\ 1 \oplus \mathit{item}\ 2 \oplus \mathit{item}\ 2 \oplus \mathbf{0}$. This expression is equivalent $(\circ\!\!-\!\!\circ)$ to the set expression $\mathit{item}\ 1 \oplus \mathit{item}\ 2$.

For a simple example of using set approximations, consider modifying the sorting program provided before so that duplicates are not kept in the sorted list. Do this modification by replacing the previous definition for splitting a list with the clauses in Figure 3. That figure contains a new definition of splitting that contains three clauses for deciding whether or not the "pivot" `X` for the splitting is equal to, less than, or greater than the first member of the list being split. Using the following substitutions for predicates

$$\begin{aligned}
\texttt{append} &\mapsto \lambda x \lambda y \lambda z.\ ?(x \oplus y) \circ\!\!-\!\!\circ ?\,z \\
\texttt{split} &\mapsto \lambda u \lambda x \lambda y \lambda z.\ ?(\mathit{item}\ u \oplus x) \circ\!\!-\!\!\circ ?(\mathit{item}\ u \oplus y \oplus z) \\
\texttt{sort} &\mapsto \lambda x \lambda y.\ ?\,x \circ\!\!-\!\!\circ ?\,y
\end{aligned}$$

(as well as the trivial substitution for `gr`), we can show that sort relates two lists only if those lists are approximated by the same set. Figure 4 contains the result of instantiating the `split` specification in Figure 3.

In the case of determining the validity of a set statement, the use of linear logic here appears to be rather weak when compared to the large body of results for solving set-based constraint systems [1, 25].

$$\forall \mathtt{X}.\ [\mathtt{split\ X\ nil\ nil\ nil}]$$
$$\forall \mathtt{X},\mathtt{B},\mathtt{R},\mathtt{S}.\ [\mathtt{split\ X\ R\ S\ B} \Rightarrow \mathtt{split\ X\ (cons\ X\ R)\ S\ B}]$$
$$\forall \mathtt{X},\mathtt{A},\mathtt{B},\mathtt{R},\mathtt{S}.[\mathtt{gr\ X\ A} \,\&\, \mathtt{split\ X\ R\ S\ B} \Rightarrow \mathtt{split\ X\ (cons\ A\ R)\ (cons\ A\ S)\ B}]$$
$$\forall \mathtt{X},\mathtt{A},\mathtt{B},\mathtt{R},\mathtt{S}.[\mathtt{gr\ A\ X} \,\&\, \mathtt{split\ X\ R\ S\ B} \Rightarrow \mathtt{split\ X\ (cons\ A\ R)\ S\ (cons\ A\ B)}]$$

Figure 3. A specification of splitting lists that drops duplicates.

$$\forall \mathtt{X}.\ [(?(\mathit{item}\ \mathtt{X} \oplus \mathbf{0}) \circ\!\!-\!\!\circ ?(\mathit{item}\ \mathtt{X} \oplus \mathbf{0} \oplus \mathbf{0}))]$$
$$\forall \mathtt{X},\mathtt{B},\mathtt{R},\mathtt{S}.\ [(?(\mathit{item}\ \mathtt{X} \oplus \mathtt{R}) \circ\!\!-\!\!\circ ?(\mathit{item}\ \mathtt{X} \oplus \mathtt{S} \oplus \mathtt{B})) \Rightarrow (?(\mathit{item}\ \mathtt{X} \oplus \mathit{item}\ \mathtt{X} \oplus \mathtt{R}) \circ\!\!-\!\!\circ ?(\mathit{item}\ \mathtt{X} \oplus \mathtt{S} \oplus \mathtt{B}))]$$
$$\forall \mathtt{X},\mathtt{A},\mathtt{B},\mathtt{R},\mathtt{S}.\ [\mathbf{1}\&(?(\mathit{item}\ \mathtt{X} \oplus \mathtt{R}) \circ\!\!-\!\!\circ ?(\mathit{item}\ \mathtt{X} \oplus \mathtt{S} \oplus \mathtt{B})) \Rightarrow (?(\mathit{item}\ \mathtt{X} \oplus \mathit{item}\ \mathtt{A} \oplus \mathtt{R}) \circ\!\!-\!\!\circ ?(\mathit{item}\ \mathtt{X} \oplus \mathit{item}\ \mathtt{A} \oplus \mathtt{S} \oplus \mathtt{B}))]$$
$$\forall \mathtt{X},\mathtt{A},\mathtt{B},\mathtt{R},\mathtt{S}.\ [\mathbf{1}\&(?(\mathit{item}\ \mathtt{X} \oplus \mathtt{R}) \circ\!\!-\!\!\circ ?(\mathit{item}\ \mathtt{X} \oplus \mathtt{S} \oplus \mathtt{B})) \Rightarrow (?(\mathit{item}\ \mathtt{X} \oplus \mathit{item}\ \mathtt{A} \oplus \mathtt{R}) \circ\!\!-\!\!\circ ?(\mathit{item}\ \mathtt{X} \oplus \mathtt{S} \oplus \mathit{item}\ \mathtt{A} \oplus \mathtt{B}))]$$

Figure 4. The result of substituting set approximations into the `split` program.

9 Automation of deduction

We describe some proof theory results that can be used to automate deduction for the linear logic formulas that occur in Propositions 1 and 2. The key result of linear logic surrounding the search for cut-free proofs is given by the completeness of *focused proofs* [3]. Focused proofs are a normal form that significantly generalizes standard completeness results in logic programming, including the completeness of SLD-resolution and uniform proofs as well as various forms of bottom-up and top-down reasoning [16].

9.1 An proof system for additive connectives

We first analyze the nature of proof search for the linear logic translation of set statements. Note that when considering provability of set statements, there is no loss of generality if the only set judgment it contains is the subset judgment since set equality can be expressed as two inclusions. We now prove that the proof system in Figure 5 is sound and complete for proving set statements.

Proposition 3. Let $S_0, T_0, \ldots, S_n, T_n$ $(n \geq 0)$ be set expressions all of whose free variables are in the list of variables $\bar{x}$. The formula

$$\forall \bar{x}[(?\,S_1 \multimap ?\,T_1) \,\&\, \ldots \,\&\, (?\,S_n \multimap ?\,T_n) \Rightarrow (?\,S_0 \multimap ?\,T_0)]$$

$$\frac{}{\Gamma; A_i \vdash A_1 \oplus \cdots \oplus A_n} \oplus \mathrm{R} \qquad \frac{\Gamma; A_1 \vdash C \quad \ldots \quad \Gamma; A_n \vdash C}{\Gamma; A_1 \oplus \cdots \oplus A_n \vdash C} \oplus \mathrm{L}$$

$$\frac{\Gamma; B_1 \oplus \cdots \oplus B_m \vdash C}{\Gamma; A \vdash C} \mathrm{FC}$$

Figure 5. Specialized proof rules for proving set statements. Here, $n, m \geq 0$, $1 \leq i \leq n$, and in the FC (forward-chaining) inference rule, the formula $?(A_1 \oplus \cdots \oplus A_n) \multimap ?(B_1 \oplus \cdots \oplus B_m)$ must be a member of Γ and $A \in \{A_1, \ldots, A_n\}$.

is provable in linear logic if and only if the sequent

$$(?\,S_1 \multimap ?\,T_1), \ldots, (?\,S_n \multimap ?\,T_n); S_0 \vdash T_0$$

is provable using the proof system in Figure 5.

Proof. The soundness part of this proposition ("if") is easy to show. For completeness ("only if"), we use the completeness of focused proofs in [3]. In order to use that result, we need to give a polarity to all atomic formulas. We do this by assigning all atomic formulas (those of the form *item* $(\cdot)$ and those symbols in $\bar{x}$ of type o) positive polarity. Second, we need to translate the two-sided sequent $\Gamma; S \vdash T$ to $\Gamma^{\perp}, T; \Uparrow S^{\perp}$ when S is not atomic and to $\Gamma^{\perp}, T; S^{\perp} \Uparrow \cdot$ when S is a atom. Completeness then follows directly from the structure of focused proofs. ∎

Notice that when building proofs in a bottom-to-top fashion using the inference rules in Figure 5, the left-hand-side of sequents change until one reaches the top inference rule.

We can now provide a proof of Proposition 2. Assume that

$$\forall \bar{x}[S_1\ \rho_1\ T_1\ \& \cdots \&\ S_n\ \rho_n\ T_n \Rightarrow S_0\ \rho_0\ T_0]$$

is provable in linear logic and let θ be a set-valued substitution for $\bar{x}$. Thus, the formula

$$S_1\theta\ \rho_1\ T_1\theta\ \& \cdots \&\ S_n\theta\ \rho_n\ T_n\theta \Rightarrow S_0\theta\ \rho_0\ T_0\theta$$

is provable. By Proposition 3, it follows by a simple induction on the structure of proofs in Figure 5 that

$$\models_s S_1\theta\ \rho_1\ T_1\theta\ \& \cdots \&\ S_n\theta\ \rho_n\ T_n\theta \Rightarrow S_0\theta\ \rho_0\ T_0\theta.$$

$$\frac{}{\Gamma; A_1 \parr \cdots \parr A_n \vdash A_1, \ldots, A_n} \parr \mathrm{L}$$

$$\frac{}{\Gamma; A_1 \parr \cdots \parr A_n \parr \mathbf{0} \vdash A_1, \ldots, A_n, \Delta} \parr \mathbf{0}\mathrm{L}$$

$$\frac{\Gamma; S \vdash T_1, T_2, \Delta}{\Gamma; S \vdash T_1 \parr T_2, \Delta} \parr \mathrm{R} \qquad \frac{\Gamma; S \vdash T, \Delta}{\Gamma; S \vdash A_1, \ldots, A_n, \Delta} \mathrm{BC}$$

Figure 6. Specialized proof rules for proving multiset statements. Here, $n \geq 0$ and $A_1, \ldots, A_n$ are atomic (in particular, they are not $\mathbf{0}$). In the BC inference rule, $T \multimap (A_1 \parr \cdots \parr A_m)$ must be a member of Γ.

9.2 A proof system for multiplicative connectives

The proof system in Figure 6 can be used to characterize the structure of proofs of the linear logic encoding of multiset statements. Let

$$\forall \bar{x}[S_1 \ \hat{\rho}_1 \ T_1 \ \& \ \ldots \ \& \ S_n \ \hat{\rho}_n \ T_n \Rightarrow S_0 \ \hat{\rho}_0 \ T_0]$$

be the translation of a multiset statement into linear logic. Provability of this formula can be reduced to attempting to prove $S_0 \ \hat{\rho}_0 \ T_0$ from assumptions of one of the following two forms:

$$(B_1 \parr \cdots \parr B_n) \multimap (A_1 \parr \cdots \parr A_m)$$

$$(B_1 \parr \cdots \parr B_n \parr \mathbf{0}) \multimap (A_1 \parr \cdots \parr A_m)$$

Here, $A_1, \ldots, A_m, B_1, \ldots, B_n$ are atomic formulas.

Proposition 4. Let S_0 and T_0 be multiset expressions all of whose free variables are in the list of variables $\bar{x}$ and let Γ be a set of (linear logic encodings of) multiset judgments. The formula $S_0 \multimap T_0$ is a linear logic consequence of Γ if and only if the sequent $\Gamma; S_0 \vdash T_0$ is provable using the inference rules in Figure 6. Similarly, the formula $S_0 \parr \mathbf{0} \multimap T_0$ is a linear logic consequence of Γ if and only if the sequent $\Gamma; S_0 \parr \mathbf{0} \vdash T_0$ is provable using the inference rules in Figure 6.

Proof. The soundness part of this proposition ("if") is easy to show. Completeness ("only if") is proved elsewhere, for example, in [18, Proposition 2]. It is also an easy consequence of the the completeness of focused proofs in [3]: fix the polarity to all atomic formulas to be negative. ∎

Notice that proofs using the rules in Figure 6 are straight-line proofs (no branching) and that they are goal-directed in the sense that the right-hand side (the "goal") changes during the bottom-to-top construction of a proof.

A proof of Proposition 1 follows from the Proposition 4 by a simple induction on the structure of proofs using the proof system in Figure 6.

9.3 Decidability and Practical Implementation

Determining whether or not the (additive) linear logic translation of a set statement is provable is decidable. In particular, notice that in a proof of the endsequent $\Gamma; S \vdash T$ using the inference rules in Figure 5, all sequents in the proof have the form $\Gamma; S' \vdash T$, where S is either an atomic formula or the conclusion of some implication in Γ. Thus, the search for a proof either succeeds (proof search ends by placing $\oplus$ R on top), or fails to find a proof, or it cycles, a case we can always detect since there is only a finite number of different formulas that can be S'.

Decidability for the proof system of Figure 6 is currently open. If all judgments in a multiset statement are equivalences ($\stackrel{m}{=}$) then deduction in the multiplicative proof system is an example of multiset rewriting and, as such, is a subset of the Petri net reachability problem, which is know to be decidable [9].

A simple prototype implementation of the proof systems in this section within the λProlog programming language [23] illustrates that a naive implementation of provability can be effective in finding proofs of provable linear logic statements generated by the examples in this paper. Also, when proofs existed, they existed under the assumption that any given assumed implication is used at most once. Exploiting such an observation allows one to search for short proofs first will a high chance of success.

10 Extensions

Various extensions of the basic scheme described here are natural to consider. In particular, it should be easy to consider approximating data structures that contain items of differing types: each of these types could be mapped into different $item_\alpha(\cdot)$ predicates, one for each type α.

It should also be simple to construct approximating mappings given the *polymorphic* typing of a given constructor's type. For example, if we are given the following declaration of binary trees (written here in λProlog syntax)

```
kind btree   type -> type.
type emp     btree A.
type bt      A -> btree A -> btree A -> btree A.
```

it should be possible to automatically construct the mapping

$$\begin{array}{l}\texttt{btree} \mapsto \lambda x.\texttt{o} \\ \quad \texttt{emp} \mapsto \bot \\ \quad\quad \texttt{bt} \mapsto \lambda x \lambda y \lambda z.\ item_A(x) \parr y \parr z\end{array}$$

that can, for example, approximate a binary tree with the multiset of the labels for internal nodes.

Extending this work to do static analysis for higher-order Horn clauses [24] also seems most natural to consider. In the paper [21], collections based on lists and (functional) difference lists are also considered.

Abstract interpretation [8] associates to a program an approximation to its semantics. Such approximations can help to determine various kinds of properties of programs. It will be interesting to see how well the particular notions of collection analysis described here can be related to abstract interpretation. More challenging would be to see to what extent the general methodology described here – the substitution into proofs (computation traces) and use of linear logic – can be related to the general methodology of abstract interpretation.

BIBLIOGRAPHY

[1] Alexander Aiken. Set constraints: results, applications, and future directions. In *PPCP94: Principles and Practice of Constraint Programming*, LNCS 874, pages 171–179, 1994.

[2] J. M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.

[3] Jean-Marc Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *J. of Logic and Computation*, 2(3):297–347, 1992.

[4] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory.* Academic Press, 1986.

[5] Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A Meta-Notation for Protocol Analysis. In R. Gorrieri, editor, *Proceedings of the 12th IEEE Computer Security Foundations Workshop — CSFW'99*, pages 55–69, Mordano, Italy, 28–30 June 1999. IEEE Computer Society Press.

[6] Iliano Cervesato and Frank Pfenning. A Linear Logic Framework. In *Proceedings, Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[7] Alonzo Church. A Formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5:56–68, 1940.

[8] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.

[9] Javier Esparza and Mogens Nielsen. Decidability Issues for Petri Nets - a survey. *Bulletin of the EATCS*, 52:244–262, 1994.

[10] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[11] Jean-Yves Girard. A Fixpoint Theorem in Linear Logic. An email posting to the mailing list linear@cs.stanford.edu, February 1992.

[12] Jean-Yves Girard. Locus solum. *Mathematical Structures in Computer Science*, 11(3):301–506, June 2001.

[13] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge University Press, 1989.

[14] Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Sci. Comput. Program.*, 58(1-2):115–140, 2005.

[15] Joshua Hodas and Dale Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation*, 110(2):327–365, 1994.
[16] Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. In J. Duparc and T. A. Henzinger, editors, *CSL 2007: Computer Science Logic*, LNCS 4646, pages 451–465. Springer-Verlag, 2007.
[17] Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.
[18] Dale Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *3rd Workshop on Extensions to Logic Programming*, LNCS 660, pages 242–265, Bologna, Italy, 1993. Springer-Verlag.
[19] Dale Miller. Forum: A Multiple-Conclusion Specification Logic. *Theoretical Computer Science*, 165(1):201–232, September 1996.
[20] Dale Miller. Higher-order quantification and proof search. In Hélène Kirchner and Christophe Ringeissen, editors, *Proceedings of AMAST 2002*, LNCS 2422, pages 60–74, 2002.
[21] Dale Miller. Collection analysis for Horn clause programs. In *Proceedings of PPDP 2006: 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 179–188, July 2006.
[22] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
[23] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, August 1988. MIT Press.
[24] Gopalan Nadathur and Dale Miller. Higher-order Horn Clauses. *Journal of the ACM*, 37(4):777–814, October 1990.
[25] Leszek Pacholski and Andreas Podelski. Set Constraints: A Pearl in Research on Constraints. In *Principles and Practice of Constraint Programming - CP97*, LNCS 1330, pages 549–562. Springer, 1997.
[26] Peter Schroeder-Heister. Rules of Definitional Reflection. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 222–232. IEEE Computer Society Press, IEEE, June 1993.