Collection Analysis for Horn Clause Programs

[Extended Abstract]

Dale Miller

INRIA & LIX, Ècole Polytechnique, Rue de Saclay
91128 Palaiseau, France
dale.miller [at] inria.fr

Abstract

We consider approximating data structures with collections of the items that they contain. For examples, lists, binary trees, tuples, etc, can be approximated by sets or multisets of the items within them. Such approximations can be used to provide partial correctness properties of logic programs. For example, one might wish to specify than whenever the atom $sort(t, s)$ is proved then the two lists t and s contain the same multiset of items (that is, s is a permutation of t). If sorting removes duplicates, then one would like to infer that the sets of items underlying t and s are the same. Such results could be useful to have if they can be determined statically and automatically. We present a scheme by which such collection analysis can be structured and automated. Central to this scheme is the use of linear logic as a computational logic underlying the logic of Horn clauses.

Categories and Subject Descriptors F.4.1 [*Mathematical Logic*]: Computational logic; I.2.3 [*Deduction and Theorem Proving*]: Logic programming

General Terms Design, Theory, Verification

Keywords proof search, static analysis, Horn clauses, linear logic

1. Introduction

Static analysis of logic programs can provide useful information for programmers and compilers. Typing systems, such as in λProlog [23, 24], have proved valuable during the development of code: type errors often represent program errors that are caught at compile time when they are easier to find and fix than at runtime when they are much harder to repair. Static type information also provides valuable documentation of code since it provides a concise approximation to what the code does.

In this paper we describe a method by which it is possible to infer that certain relationships concerning collections underlying structured data hold. We shall focus on relations that are also decidable and can be done during compile time analysis of logic programs. We shall use *multisets* and *sets* to *approximate* more complicated structures as lists and binary trees. Consider, for example, a list sorting program that maintains duplicates of elements. 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 during compile time. 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 establish that if the atomic formula $sort(t, s)$ is provable, then the multiset associated to s is equal to the multiset associated to t. 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 done statically.

This paper considers exclusively the static analysis of first-order Horn clauses but it does so by making substitution instances of such Horn clauses that carry them into linear logic. Proofs for the resulting linear logic formulas are then attempted as part of static analysis.

2. The undercurrents

There are various themes that underlie our approach to inferring properties of Horn clause programs. We list them explicitly below. The rest of the paper can be seen as a particular example of how these themes can be developed.

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 can be useful for programmers: they can offer important invariants and document for code. Static analysis can also be used by compilers to uncover useful structures that allow compilers to make choices that can improve execution. While compilers might make use of multiple static analysis regimes, programmers do not usually have convenient access to multiple static analyzes for the code that they are composing. Sometimes, a programming language provides 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 (SML also statically determines if a given function defined over concrete data structures cover all possible input values). It seems clear, however, that such analysis of code, if it can be done quickly and incrementally, might have significant benefits for programmers during the process of writing code. For example, a programmer might find it valuable to know that a recursive program that she has just written has linear or quadratic runtime complexity, or that a relation she just specified actually defines a function. The Ciao system preprocessor [14] provides for such functionality by allowing a programmer to write various properties about code that the preprocessor attempts to verify.

PPDP'06 July 10–12, 2006, Venice, Italy.

Having an open set of properties and analysis tools is an interesting direction for the design of a programming language. The collection analysis we discuss here could be just one such analysis tool.

2.2 Logic programs as untyped λ-expressions

If we do not commit to just one typing discipline, then it seems sensible to use a completely untyped setting for encoding programs and declarations. Given that untyped λ-terms provide for arbitrary applications and arbitrary abstractions, such terms can provide an appealing setting for the encoding of program expressions, type expressions, assertions, invariants, etc. Via the well developed theory of λ-conversion, such abstractions can be instantiated with a variety of other objects. Abstractions can be used to encode quantifiers within formulas as well as binding declarations surrounding entire programs.

In logic programming, proofs can be viewed as computation traces and such proof objects can also be encoded as untyped λ-terms. Instantiations into proofs is also well understood since it is closely related to the elimination of cut in sequent calculus or to normalization in natural deduction proofs. 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 interesting problem of duality involving atomic formulas. The *initial rule* and the *cut rule* given as

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

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, all occurrence of the cut-rule can be eliminated (whether or not C is an atomic formula) 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 approaches that try to eliminate them entirely. For example, in the work on *definitions* and *fixed points* by Schroeder-Heister [26], Girard [11], and McDowell & Miller [17], atoms are defined to be other formulas. In this approach, the only primitive judgment involving terms is that of equality. In that setting, if definitions are *stratified* (no recursion through negations) and *noetherian* (no infinite descent in recursion), then all instances of cut and initial can be removed. The setting of *ludics* of Girard [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. 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 appropriate provisos if the proof of B depends on hypotheses). If we are in a first-order logic, then the free first-order variable x of B becomes bound in $\forall x.B$ by this inference rule.

Observe the following two things about this rule. First, if we are in an untyped setting, then we can, in principle, quantify over any variable in any expression, even those that play the role of predicates or functions. Mixing such rich abstractions with logic is well known to be inconsistent so when we propose such rich abstractions in logic, we must accompany it with some discipline (such as typing) that will yield consistency.

Second, we need to observe that differences 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 (via backchaining and unification) but the constructors for the empty and non-empty lists as well as the symbol denoting the append relation do not change 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 expression during static analysis.

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 replace that atom with a formula, which, in this paper, will be a linear logic formula.

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 multisets 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 $⅋$, $\perp$, $\otimes$, $\mathbf{1}$; the additives $\oplus$, $\mathbf{0}$, $\&$, $\top$; the exponentials $!$, $?$; the implications $\multimap$ (where $B \multimap C$ is defined as $B^\perp ⅋ C$) and $\Rightarrow$ (where $B \Rightarrow C$ is defined as $(!\,B)^\perp ⅋ C$); and the quantifiers $\forall$ and $\exists$ (higher-order quantification is allowed). The equivalence of formulas in linear logic, $B \multimapboth C$, is defined as the formula $(B \multimap C) \& (C \multimap B)$.

First-order Horn clauses can be described as 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. 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 conjunction $\&$ and 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.

4. A primer for proof theory

A sequent is a triple of the form $\Sigma\colon \Gamma \vdash \Delta$ were 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 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:

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

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

The premise $\Sigma \vdash t\colon \tau$ 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 instantiate things within the sequent calculus.

4.1 Substituting for types

Although we think of formulas and proofs as untyped expressions, we shall use simple typing within sequents to control the kind of formulas that are present. A signature is used to bind and declare typing for (eigen)variables and non-logical constants within a sequent. Simple types are, formally speaking, also a simple class of untyped λ-terms: the type o is used to denote formulas (following Church [7]). In a sequent calculus proof, simple type expressions are global and admit no bindings. As a result, it is an easy matter to show that if one takes a proof with a type constant σ and replaces everywhere σ with some type, say, τ, one gets another valid proof. We shall do this later when we replace a list by a multiset that approximates it: since we are using linear logic, we shall use formulas to encode multisets and so we shall replace the type constant `list` with `o`.

4.2 Substituting for non-logical constants

Consider the 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. Since this is a universal quantifier, 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 term 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 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 merge (substitution) the proof of a formula (here, B) with a use of that formula as an assumption. 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,

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

to a proof of

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

We shall see this style of reasoning about proofs several times below. This allows us to "split an atom" $p(t_1, \ldots, t_m)$ into a formula $S[t_1/x_1, \ldots, t_m/x_m]$ and to transform proofs of the atom into proofs of 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 help reasoning about them appear in [20].

5. Encoding multisets as formulas

We wish to encode multisets and sets and simple judgments about them (such as inclusion and equality) as linear logic formulas. We consider multisets first. Let token *item* be a linear logic predicate of one argument: the linear logic atomic formula *item* x will denote the multiset containing just the one 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 multiset 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 $\perp$ for the empty multiset and $⅋$ to combine two multisets. Proofs search using this style encoding places multisets on the right of the sequent arrow. 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.

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

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

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

Figure 2. The result of instantiating various non-logical constants in the above Horn clauses.

Let S and T be the two formulas $\mathit{item}\ s_1 \parr \cdots \parr \mathit{item}\ s_n$ and $\mathit{item}\ t_1 \parr \cdots \parr \mathit{item}\ t_m$, respectively ($n, m \geq 0$). 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 now, however, 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, $\vdash \mathbf{0} \multimapboth \forall p.p$ (linear logic absurdity) and

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

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 predicate variable (a variable of type o) to be used to denote a (necessarily open) multiset expression. An example of an open multiset expression is $\mathit{item}\ f(X) \parr \bot \parr Y$, 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 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 $S_0, T_0, \ldots, S_n, T_n$ 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

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

if for all 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 $\models_m S_0\theta\ \rho_0\ T_0\theta$.

The following Proposition is central to our use of linear logic to establish multiset statements for Horn clause programs.

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 $\exists q(s \parr q \multimap t)$ if ρ is $\sqsubseteq$ and $t \multimapboth s$ 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

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

This Proposition shows that linear logic can be used in a sound way to infer valid multiset statement. 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 the validity of a multiset statement, 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
leq    : int -> int -> o
gr     : int -> int -> o
```

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

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

Figure 3. A change in the specification of splitting lists to drop duplicates.

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

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

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 $(\mathtt{sort}\ s\ t)$ is provable then the multiset of items in the list denoted by s is equal to the multiset of items in the list denoted by t. If this property holds then t and s 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.

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

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

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

Under such a mapping, the list $(\mathtt{cons}\ 1\ (\mathtt{cons}\ 3\ (\mathtt{cons}\ 2\ \mathtt{nil})))$ 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 $(\mathtt{append}\ r\ s\ t)$ or $(\mathtt{split}\ u\ t\ r\ s)$ 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 $(\mathtt{sort}\ s\ t)$ 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:

$$\begin{aligned} \mathtt{append} &\mapsto \lambda x \lambda y \lambda z.\, (x \parr y) \multimapboth z \\ \mathtt{split} &\mapsto \lambda u \lambda x \lambda y \lambda z.\, (y \parr z) \multimapboth x \\ \mathtt{sort} &\mapsto \lambda x \lambda y.\, x \multimapboth y \end{aligned}$$

The predicates `leq` and `gr` (for the least-than-or-equal-to and greater-than relations) make no statement about collections of items, so that they can be mapped to a trivial tautology via the substitution

$$\mathtt{leq} \mapsto \lambda x \lambda y.\, \mathbf{1} \qquad \mathtt{gr} \mapsto \lambda x \lambda y.\, \mathbf{1}$$

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 presentation of linear logic.

Let Γ denote a set of formulas that contains those in Figure 1. Let θ denote the substitution described above for the type `list`, for the constructors `nil` and `cons`, and for the predicates in Figure 1. If Σ is the signature for Γ then split Σ into the two signatures Σ_1 and Σ_2 so that Σ_1 is the domain of the substitution θ and let Σ_3 be the signature of the range of θ (in this case, it just contains the constant *item*). Thus, $\Gamma\theta$ is the set of formula in Figure 2.

Assume now that $\Sigma_1, \Sigma_2; \Gamma \vdash sort(t, s)$ is provable. Given the discussion in Sections 4.1 and 4.2, we know that

$$\Sigma_1, \Sigma_3; \Gamma\theta \vdash t\theta \multimapboth s\theta$$

is provable. Since the formulas in $\Gamma\theta$ are provable, we can use substitution into proofs (Section 4.3) to conclude that $\Sigma_1, \Sigma_3; \vdash t\theta \multimapboth s\theta$. 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 $(\mathtt{append}\ r\ s\ t)$ and $(\mathtt{split}\ u\ t\ r\ s)$ 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 $(\mathtt{leq}\ t\ s)$ and $(\mathtt{gr}\ t\ s)$, and closed atoms $(\mathtt{sort}\ s\ t)$ 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 suggested here using linear logic and deduction remains syntactic and proof theoretic: in particular, showing that a model satisfies a Horn clause is replaced by a deduction within linear logic.

8. Sets approximations

It is rather easy to encode sets and the equality and subset judgments on sets into linear logic. In fact, the transition to set from multiset is provided by the use of the linear logic exponential: since we are using 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 $?\,\mathit{item}\ t$ 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 $\{x_1, \ldots, n_n\}$ can be encoded as $?\,\mathit{item}\ x_1 \parr \cdots \parr ?\,\mathit{item}\ x_n$. Using logical equivalences of linear logic, this formula is also equivalent to the formula $?(\textit{item}\ x_1 \oplus \cdots \oplus \textit{item}\ x_n)$. 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 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 predicate variable (a variable of type o) to be used to denote a (necessarily open) set expression. An example of an open multiset expression is $\textit{item}\ f(X) \oplus \mathbf{0} \oplus Y$, 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

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

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

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

if for all 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 $\models_s S_0\theta\ \rho_0\ T_0\theta$.

The following Proposition is central to our use of linear logic to establish set statements for Horn clause programs.

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 $(?\,s \multimap ?\,t) \mathbin{\&} (?\,t \multimap ?\,s)$ if ρ is $\stackrel{s}{=}$. If*

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

is provable in linear logic, then

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

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

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

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

For a simple example of using set approximates, 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" for the splitting `X` is equal to, less than (using the `lt` predicate), or greater than the first member of the list being split. Using the following substitutions for predicates

$$\begin{aligned}
\texttt{append} &\mapsto \lambda x \lambda y \lambda z.\ ?(x \oplus y) \multimapboth ?\,z \\
\texttt{split} &\mapsto \lambda u \lambda x \lambda y \lambda z.\ ?\,x \multimap ?(\textit{item}\ u \oplus y \oplus z) \\
\texttt{sort} &\mapsto \lambda x \lambda y.\ ?\,x \multimapboth ?\,y
\end{aligned}$$

(as well as the trivial substitution for `lt` and `ge`), we can show that sort relates two lists only if those lists are approximated by the same set.

$$\frac{}{\Gamma; A_i \vdash A_1 \oplus \cdots \oplus A_n} \oplus \mathrm{R}$$

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

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

Here, $n, m \geq 0$ and in the BC (backchaining) 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 $A \in \{A_1, \ldots, A_n\}$.

Figure 5. Specialized proof rules for proving set statements.

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].

9. Automation of deduction

We describe how automation of proof for the linear logic translations of set and multiset statements given in Propositions 1 and 2 can be performed.

In order to understand how to automatically prove the required formulas, we first provide a normal form theorem for the fragment of linear logic for which we are interested. 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.

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*

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

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 this result of focused proofs, 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) negative polarity. Second, we need to translation the two sided sequent $\Gamma; S \vdash T$ to $\Gamma^\perp; T \Uparrow S^\perp$ when S is not atomic (that is, its top-level logical connective is $\oplus$) 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 the resulting proofs are essentially bottom-up: one reasons from formulas on the left of the sequent arrow to formulas on the right.

We can now conclude that it is decidable to determine whether or not the linear logic translation of a set statement is provable. Notice that in a proof built using the inference rules in Figure 5, if

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

$$\frac{\Gamma; S \vdash T_1, T_2, \Delta}{\Gamma; S \vdash T_1 \parr T_2, \Delta} \parr \mathrm{R}$$

$$\frac{\Gamma; S \vdash A_1, \ldots, A_n, \Delta}{\Gamma; S \vdash B_1, \ldots, B_m, \Delta} \mathrm{BC}$$

Here, $n, m \geq 0$ and in the BC (backchaining) inference rule, it must be the case that the formula

$$(A_1 \parr \cdots \parr A_n) \multimap (B_1 \parr \cdots \parr B_m)$$

is a member of Γ.

Figure 6. Specialized proof rules for proving multiset statements.

the endsequent is $\Gamma; S \vdash T$ then all sequents in the proof have the form $\Gamma; S' \vdash T$, for some S'. 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 atomic formulas that can be S'.

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 the form

$$(A_1 \parr \cdots \parr A_n) \multimap (B_1 \parr \cdots \parr B_m),$$

where $A_1, \ldots, A_n, B_1, \ldots, B_m$ are atomic formulas. Such formulas can be called *multiset rewriting clauses* since backchaining on such clauses amounts to rewriting the right-hand-side multiset of a sequent (see rule BC in Figure 6). Such rewriting clauses are particularly simple since they do not involve quantification.

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 multiset rewriting rules. 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.*

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 positive. ∎

Notice that the proofs using the rules in Figure 6 are straight line proofs (no branching) and that they are top-down (or goal-directed). Given these observation, it follows that determining if $S_0 \multimap T_0$ is provable from a set of multiset rewriting clauses is decidable, since this problem is contained within the reachability problem of Petri Nets [9]. Proving a multiset inclusion judgment $\exists q(S_0 \parr q \multimap T_0)$ involves first instantiating this higher-order quantifier. In principle, this instantiation can be delayed until attempting to apply the sole instance of the $\parr$ L rule (Figure 6).

10. List approximations

We now consider using lists as approximations. Since lists have more structure than sets and multisets, it is more involved to encode and reason with them. We only illustrate their use and do not follow a full formal treatment for them.

Since the order of elements in a list is important, the encoding of lists into linear logic must involve a connective that is not commutative. (Notice that both $\parr$ and $\oplus$ are commutative.) Linear implication provides a good candidate for encoding the order used in lists. For example, consider proof search with the formula

$$\textit{item}\ a \multimapinv (p \multimapinv (\textit{item}\ b \multimapinv (p \multimapinv \bot)))$$

on the right. (This formula is equivalent to *item* $a \parr (p^{\perp} \otimes (\textit{item}\ b \parr p^{\perp}))$.) Such a formula can be seen as describing a process that is willing to output the item a then go into input mode waiting for the atomic formula p to appear. If that formula appears, then item b is output and again it goes into input waiting mode looking for p. If another occurrence of p appears, this process becomes the inactive process. Clearly, a is output prior to when b is output: this ordering is faithfully captured by proof search in linear logic. Such an encoding of asynchronous process calculi into linear logic has been explored in a number of papers: see, for example, [16, 21].

The example above suggests that lists and list equality can be captured directly in linear logic using the following encoding:

$$\texttt{nil} \mapsto \lambda l.\bot \qquad \texttt{cons} \mapsto \lambda x \lambda R \lambda l.\ \textit{item}\ x \multimapinv (l \multimapinv (R\ l))$$

The encoding of the list, say ($\texttt{cons}\ a\ (\texttt{cons}\ b\ \texttt{nil})$), is given by the λ-abstraction

$$\lambda l.\textit{item}\ a \multimapinv (l \multimapinv (\textit{item}\ b \multimapinv (l \multimapinv \bot))).$$

The following proposition can be proved by induction on the length of the list t.

PROPOSITION 5. *Let s and t be two lists (built using* `nil` *and* `cons`*) and let S and T be the translation of those lists into expressions of type $o \to o$ via the substitution above. Then $\forall l.(Sl) \multimapboth (Tl)$ is provable in linear logic if and only if s and t are the same list.*

This presentation of lists can be "degraded" to multisets simply by applying the translation of a list to the formula $\bot$. For example, applying the translation of ($\texttt{cons}\ a\ (\texttt{cons}\ b\ \texttt{nil})$) to $\bot$ yields the formulas

$$\textit{item}\ a \multimapinv (\bot \multimapinv (\textit{item}\ b \multimapinv (\bot \multimapinv \bot)))$$

which is linear logically equivalent to *item* $a \parr$ *item* b.

Given this presentation of lists, there appears to be no simple combinator for, say, list concatenation and, as a result, there is no direct way to express the judgments of prefix, suffix, sublist, etc. Thus, beyond equality of lists (by virtual of Proposition 5) there are few natural judgments that can be stated for list. More can be done, however, by considering difference lists.

11. Difference list approximations

Since our framework includes λ-abstractions, it is natural to represent difference lists as a particular kind of list abstraction over a list. For example, in λProlog a difference list is naturally represented as a λ-term of the form

$$\lambda L.\texttt{cons}\ x_1\ (\texttt{cons}\ x_2\ (\ldots (\texttt{cons}\ x_n\ L) \ldots)).$$

Such abstracted lists are appealing since the simple operation of composition encodes the concatenation of two lists. Given concatenation, it is then easy to encode the judgments of prefix and suffix. To see other example of computing on difference lists described in fashion, see [4].

Lists can be encoded using the difference list notion with the following mapping into linear logic formulas.

$$\begin{aligned} \texttt{nil} &\mapsto \lambda L \lambda l.\ L\ l \\ \texttt{cons} &\mapsto \lambda x \lambda R \lambda L \lambda l.\ \textit{item}\ x \multimapinv (l \multimapinv (R\ L\ l)) \end{aligned}$$

The encoding of the list, say ($\texttt{cons}\ a\ (\texttt{cons}\ b\ \texttt{nil})$), is given by the λ-abstraction

$$\lambda L \lambda l.\textit{item}\ a \multimapinv (l \multimapinv (\textit{item}\ b \multimapinv (l \multimapinv L\ l))).$$

$$\texttt{(traverse emp null)}$$
$$\forall \texttt{N}.\forall \texttt{R}.\forall \texttt{S}.\ (\texttt{traverse R S}) \Rightarrow\ (\texttt{traverse (bt N emp R) (cons N S)})$$
$$\forall \texttt{N}.\forall \texttt{M}.\forall \texttt{R}.\forall \texttt{S}.\forall \texttt{L1}.\forall \texttt{L2}.\ (\texttt{traverse (bt M L1 (bt N L2 R)) S}) \Rightarrow\ (\texttt{traverse (bt N (bt M L1 L2) R) S})$$

Figure 7. Traversing a binary tree to produce a list.

$$\forall W.\forall w.W w \circ\!\!-\!\!\circ W w$$
$$\forall N.\forall R.\forall S.\forall W.\forall w.\mathit{item}\ N \circ\!- (w \circ\!- R\ W\ w) \circ\!\!-\!\!\circ (\mathit{item}\ N \circ\!- (w \circ\!- S\ W\ w)) \circ\!- \forall W.\forall w.R\ W\ w \circ\!\!-\!\!\circ S\ W\ w$$
$$\forall N.\forall M.\forall L_1.\forall L_2.\forall R.\forall S.\forall W.\forall w.$$
$$L_1(\lambda k.itemM \circ\!- (k \circ\!- L_2(\lambda l.itemN \circ\!- (l \circ\!- R\ W\ l))k))w \circ\!\!-\!\!\circ S\ W\ w \circ\!-$$
$$\forall W.\forall w.L_1(\lambda k.itemM \circ\!- (k \circ\!- L_2(\lambda l.itemN \circ\!- (l \circ\!- R\ W\ l))k))w \circ\!\!-\!\!\circ S\ W\ w$$

Figure 8. Linear logic formulas arising from a difference list approximation.

In Figure 7, a predicate for traversing a binary tree is given. Binary trees are encoded using the type `btree` and are constructed using the constructors `emp`, for the empty tree, and `bt` of type `int` $\rightarrow$ `btree` $\rightarrow$ `btree` $\rightarrow$ `btree`, for building non-empty trees. A useful invariant of this program is that the list of items approximating the binary tree structure in the first argument of `traverse` is equal to the list of items in the second argument. Linear logic formulas for computing that approximation can be generated using the following approximating substitution.

$$\begin{aligned} \texttt{btree} &\mapsto \texttt{o} \\ \texttt{emp} &\mapsto \lambda L\lambda l.\ L\ l \\ \texttt{bt} &\mapsto \lambda x\lambda R\lambda S\lambda L\lambda l.(R\ (\lambda l.\mathit{item}\ x \circ\!- (l \circ\!- (S\ L\ l)))\ l)) \end{aligned}$$

The result of applying that substitution (as well as the one above for `nil` and `cons`) is displayed in Figure 8. While these formulas appear rather complex, they are all, rather simple theorems of higher-order linear logic: these theorems are essentially trivial since the λ-conversions used to build the formulas from the data structures has done all the essential work in organizing the items into a list. Establishing these formulas proves that the order and multiplicity of elements in the binary tree and in the list in a provable traverse computation are the same.

12. Future work

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 $\mathit{item}_\alpha(\cdot)$ 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 for binary tree (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

$$\begin{aligned} \texttt{btree} &\mapsto \lambda x.\texttt{o} \\ \texttt{emp} &\mapsto \bot \\ \texttt{bt} &\mapsto \lambda x\lambda y\lambda z.\mathit{item}_A(x) \parr x \parr y \end{aligned}$$

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

Abstract interpretation [8] can associate 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 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 very general methodology of abstract interpretation.

Acknowledgments

I am grateful to the anonymous reviewers for their helpful comments on an earlier draft of this paper. This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. This paper reflects only the author's views and the Community is not liable for any use that may be made of the information contained therein.

References

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

[2] J. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.

[3] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.

[4] P. Brisset and O. Ridoux. Naïve reverse can be linear. In *Eighth International Logic Programming Conference*, Paris, France, June 1991. MIT Press.

[5] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. 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] I. Cervesato and F. 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] A. Church. A formulation of the simple theory of types. *J.of Symbolic Logic*, 5:56–68, 1940.

[8] P. Cousot and R. 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] J. Esparza and M. Nielsen. Decidability issues for petri nets - a survey. *Bulletin of the EATCS*, 52:244–262, 1994.

[10] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[11] J.-Y. Girard. A fixpoint theorem in linear logic. An email posting to the mailing list linear@cs.stanford.edu, February 1992.

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

[13] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[14] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. 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] J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[16] N. Kobayashi and A. Yonezawa. Asynchronous communication model based on linear logic. *Formal Aspects of Computing*, 3:279–294, 1994.

[17] R. McDowell and D. Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.

[18] D. 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*, number 660 in LNCS, pages 242–265, Bologna, Italy, 1993. Springer-Verlag.

[19] D. Miller. Forum: A multiple-conclusion specification language. *Theoretical Computer Science*, 165(1):201–232, Sept. 1996.

[20] D. Miller. Higher-order quantification and proof search. In H. Kirchner and C. Ringeissen, editors, *Proceedings of AMAST 2002*, number 2422 in LNCS, pages 60–74, 2002.

[21] D. Miller. Encryption as an abstract data-type: An extended abstract. In I. Cervesato, editor, *Proceedings of FCS'03: Foundations of Computer Security*, pages 3–14, 2003.

[22] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[23] G. Nadathur and D. Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, August 1988. MIT Press.

[24] G. Nadathur and F. Pfenning. The type system of a higher-order logic programming language. In F. Pfenning, editor, *Types in Logic Programming*, pages 245–283. MIT Press, 1992.

[25] L. Pacholski and A. Podelski. Set constraints: A pearl in research on constraints. In *Principles and Practice of Constraint Programming - CP97*, number 1330 in LNCS, pages 549–562. Springer, 1997.

[26] P. 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.