URL: http://www.elsevier.nl/locate/entcs/volume30.html 17 pages

A Static Analysis Method for a Classical Linear Logic Programming Language

Kyoung-Sun Kang Naoyuki Tamura

Department of Computer and Systems Engineering
Kobe University
Kobe, Japan

Abstract

In this paper, we propose a new static analysis method which is applicable for a classical linear logic programming language.

Andreoli et al. proposed a static analysis method for the classical linear logic programming language LO, but their method did not cover multiplicative connectives which are important for a resource-sensitive feature of linear logic.

Our method, in contrast, covers multiplicative conjunction in addition to multiplicative disjunction and linear implication. An abstract proof graph, an AND-OR graph representing all possible sequent proofs, is constructed from a given program and goal sequent. The graph can be repeatedly refined by propagating information to eliminate unprovable nodes from the graph.

We applied our prototype analyzer for a sorting program written in Forum. The sorting program was improved about 1000 times faster than the ordinary program without analysis, for sorting 6 elements by using the analysis result.

1 Introduction

In logic programming, the execution of a program can be viewed as a proof search in sequent calculus of logic. This principle can be also applied for linear logic which was proposed by J.-Y. Girard [6]. Linear logic views logical assumptions as consumable resources. Therefore, in linear logic programming languages, a resource can be represented as a formula rather than a term. Several logic programming languages based on linear logic have been proposed: Lygon [7], LO [2], LinLog [1], Lolli [9], Forum [14], etc.

In particular, Forum is complete for classical linear logic and its execution can be viewed as a goal-directed proof search called uniform proof [15]. However, the proof search of Forum programs is highly non-deterministic. For example, in the sorting program shown in section 6, execution time rises hyper-exponentially with list length [10]. This is the reason most of the exe-

cution time is spent trying to prove unprovable sequents. This problem is also common to the other linear logic programming languages.

Therefore, it is very important to find unprovable sequents before the execution of programs, and such a hope is realizable by a static analysis performing an abstract proof search on an abstract propositional proof. This is possible as an abstract interpretation of logic programs can be viewed as an abstract proof search in a sequent calculus [4][5]. Of course, it is impossible to find all unprovable sequents because the provability is undecidable in linear logic.

Andreoli et al. [3] proposed a static analysis method for the classical linear logic programming language LO, but their method did not cover multiplicative connections which are important for a resource-sensitive feature of linear logic. This follows from the fact that their system does not implement the lazy splitting of resources [1].

In this paper, we propose a new static analysis method which is applicable for a classical linear logic programming language including multiplicative conjunction in addition to multiplicative disjunction and linear implication. To cover multiplicative conjunction, we introduce the I/O model into our system. The I/O model was proposed by Hodas and Miller as a solution to reduce the non-determinism in the splitting of resources [9] [8].

An abstract proof graph, an AND-OR graph representing all possible sequent proofs, is constructed from a given program and goal sequent. The graph can be repeatedly refined by propagating information to remove unprovable nodes from the graph.

2 Classical Linear Logic Programming Language

In this section, we define a sequent calculus system ***LL*** of linear logic and its I/O model ***IO*** discussed in this paper.

2.1 The Language

In this paper, we choose a relatively small language. This is why we want to make the discussion in this paper as simple as possible. However, it is not difficult to extend our language widely. A *program* on our language is a set of clauses which is defined as follows.

Definition 2.1.1 (Clause) *A clause is a closed formula in the form*$(n \geq 1)$*:*

$$\forall \overrightarrow{x}(A_1 \wp \cdots \wp A_n :- G)$$

where $G = \bot$ *or* $\mathbf{1}$ *or* A *or* $B \wp C$ *or* $B \multimap C$ *or* $B \otimes C$*, and* A*,* B*,* C*,* A_i *are atomic formulas.*

[1] They describe adopting a lazy splitting technique as a future plan.

$$\frac{}{\Psi; A \longrightarrow A}\ (identity) \qquad \frac{\Psi; \Delta_1 \longrightarrow B, \mathcal{A}_1 \quad \Psi; \Delta_2 \longrightarrow C, \mathcal{A}_2}{\Psi; \Delta_1, \Delta_2 \longrightarrow A_1, \cdots, A_n, \mathcal{A}_1, \mathcal{A}_2}\ (\otimes)$$
$$(provided\ A_1 \wp \cdots \wp A_n :- B \otimes C \in \Psi)$$

$$\frac{}{\Psi; \longrightarrow A_1, \cdots, A_n}\ (\mathbf{1}) \qquad \frac{\Psi; B, \Delta \longrightarrow C, \mathcal{A}}{\Psi; \Delta \longrightarrow A_1, \cdots, A_n, \mathcal{A}}\ (\multimap)$$
$$(provided\ A_1 \wp \cdots \wp A_n :- \mathbf{1} \in \Psi) \qquad (provided\ A_1 \wp \cdots \wp A_n :- B \multimap C \in \Psi)$$

$$\frac{\Psi; \Delta \longrightarrow \mathcal{A}}{\Psi; \Delta \longrightarrow A_1, \cdots, A_n, \mathcal{A}}\ (\bot) \qquad \frac{\Psi; \Delta \longrightarrow B, C, \mathcal{A}}{\Psi; \Delta \longrightarrow A_1, \cdots, A_n, \mathcal{A}}\ (\wp)$$
$$(provided\ A_1 \wp \cdots \wp A_n :- \bot \in \Psi) \qquad (provided\ A_1 \wp \cdots \wp A_n :- B \wp C \in \Psi)$$

Fig. 1. The rules for a propositional proof system ***LL***

In the definition 2.1.1, the connective :− means the reverse linear implication, that is, $A :- G$ is equivalent to $G \multimap A$. Actually, it is easy to extend the body of clauses in the definition 2.1.1 to the following syntax:

$$G \ ::= \ \bot \mid \mathbf{1} \mid A \mid G_1 \wp G_2 \mid A \multimap G \mid G_1 \otimes G_2$$

There is also a technique to translate a clause of the extended form into clauses of the non-extended form. For example, the clause $A_1 \wp, \cdots, \wp A_n :- (B_1 \wp B_2) \otimes (C_1 \multimap C_2)$ can be translated into three clauses, $A_1 \wp, \cdots, \wp A_n :- B \otimes C$, $B :- B_1 \wp B_2$, and $C :- C_1 \multimap C_2$ by introducing two new atomic formulas B and C.

Definition 2.1.2 (Sequent) *A sequent is an expression $\Psi; \Delta \longrightarrow \mathcal{A}$, where Ψ is a set of clauses and Δ, $\mathcal{A}$ are multisets of atomic formulas.*

In the definition 2.1.2, the Ψ, Δ are called the *intuitionistic* context and *linear* context. These can be viewed as the *program* and *resources* respectively, and the formulas in $\mathcal{A}$ are called the *goals*. Clauses in Ψ are considered to be prefixed by the modality operator !. That is, the sequent $\Psi; \Delta \longrightarrow \mathcal{A}$ is equivalent to $!\Psi, \Delta \longrightarrow \mathcal{A}$ of standard linear logic.

Figure 1 presents the propositional sequent calculus system LL.

*2.2 An I/O model for the proof system **LL***

In searching a proof of a sequent including a multiplicative connective from the bottom up, it is necessary to split the linear context into two parts, and therefore becomes highly non-deterministic.

On the other hand, the multiplicative conjunction $\otimes$ is one of the most useful connectives in linear logic programming languages. Hodas and Miller proposed a solution to reduce the non-determinism in searching a proof of goal $G_1 \otimes G_2$, which is called the I/O model for an intuitionistic linear logic

$$\frac{}{\Psi; A, \Delta\backslash\Delta \longrightarrow A, \mathcal{A}\backslash\mathcal{A}}\ (identity)$$

$$\frac{\Psi; \Delta_I\backslash\Delta' \longrightarrow B, \mathcal{A}_I\backslash\mathcal{A}' \qquad \Psi; \Delta'\backslash\Delta_O \longrightarrow C, \mathcal{A}'\backslash\mathcal{A}_O}{\Psi; \Delta_I\backslash\Delta_O \longrightarrow A_1, \cdots A_n, \mathcal{A}_I\backslash\mathcal{A}_O}\ (\otimes)$$
$$(provided\ A_1 \wp \cdots \wp A_n :- B \otimes C \in \Psi\ and\ \mathcal{A}_O \subseteq \mathcal{A}' \subseteq \mathcal{A}_I)$$

$$\frac{}{\Psi; \Delta\backslash\Delta \longrightarrow A_1, \cdots A_n, \mathcal{A}\backslash\mathcal{A}}\ (\mathbf{1})$$
$$(provided\ A_1 \wp \cdots \wp A_n :- \mathbf{1} \in \Psi)$$

$$\frac{\Psi; B, \Delta_I\backslash\Delta_O \longrightarrow C, \mathcal{A}_I\backslash\mathcal{A}_O}{\Psi; \Delta_I\backslash\Delta_O \longrightarrow A_1, \cdots A_n, \mathcal{A}_I\backslash\mathcal{A}_O}\ (\multimap)$$
$$(provided\ A_1 \wp \cdots \wp A_n :- B \multimap C \in \Psi)$$

$$\frac{\Psi; \Delta_I\backslash\Delta_O \longrightarrow \mathcal{A}_I\backslash\mathcal{A}_O}{\Psi; \Delta_I\backslash\Delta_O \longrightarrow A_1, \cdots A_n, \mathcal{A}_I\backslash\mathcal{A}_O}\ (\bot)$$
$$(provided\ A_1 \wp \cdots \wp A_n :- \bot \in \Psi)$$

$$\frac{\Psi; \Delta_I\backslash\Delta_O \longrightarrow B, C, \mathcal{A}_I\backslash\mathcal{A}_O}{\Psi; \Delta_I\backslash\Delta_O \longrightarrow A_1, \cdots A_n, \mathcal{A}_I\backslash\mathcal{A}_O}\ (\wp)$$
$$(provided\ A_1 \wp \cdots \wp A_n :- B \wp C \in \Psi)$$

Fig. 2. A I/O model ***IO*** for a propositional proof system ***LL***

programming language Lolli [9] [8].

$$\frac{\Gamma; \Delta_1 \longrightarrow G_1 \quad \Gamma; \Delta_2 \longrightarrow G_2}{\Gamma; \Delta_1, \Delta_2 \longrightarrow G_1 \otimes G_2}\ \otimes_R \qquad \frac{\Gamma; \Delta_I\backslash\Delta' \longrightarrow G_1 \quad \Gamma; \Delta'\backslash\Delta_O \longrightarrow G_2}{\Gamma; \Delta_I\backslash\Delta_O \longrightarrow G_1 \otimes G_2}\ \otimes_R$$

In order to prove of the goal $G_1 \otimes G_2$ in the I/O model for Lolli, the first input context, Δ_1, Δ_2, are given as usable assumptions to prove G_1. The output context Δ_2 of G_1, which is not used by G_1, is given as input context of G_2 to prove G_2. The above $\otimes_R$ inference rule of the I/O model, the context on the left of the "\" is the input linear context of a goal G_i, and the one on the right is the output linear context of a goal G_i. By applying this idea to right-hand goal context, Hodas proposed the I/O model for classical linear logic programming language Forum [10]. We shall call our adaptation of this I/O model for the system ***LL*** using the same method as Hodas, the system ***IO***.

Definition 2.2.1 *A sequent of* ***IO*** *is an expression* $\Psi; \Delta_I\backslash\Delta_O \longrightarrow \mathcal{A}_I\backslash\mathcal{A}_O$, *where* Ψ *is a set of clauses and* Δ_I, Δ_O, $\mathcal{A}_I$, $\mathcal{A}_O$ *are multisets of atomic formulas.*

Figure 2 presents the I/O model ***IO*** for the proof system ***LL***. We now show that the I/O model ***IO*** is sound and complete relative to the system ***LL***.

Proposition 2.1

$$\boldsymbol{LL} \vdash \Psi; \Delta \longrightarrow \mathcal{A} \quad \Longleftrightarrow \quad \boldsymbol{IO} \vdash \Psi; \Delta, \Delta_O\backslash\Delta_O \longrightarrow \mathcal{A}, \mathcal{A}_O\backslash\mathcal{A}_O$$

Proof. This can be similarly proved as in [11]. ∎

3 Abstraction of the Sequents

As described in the introduction section, abstract interpretation of logic programs can be viewed as an abstract proof search in sequent calculus. In propositional logic, when the goal sequent is given, there are only finite number of sequents to be considered due to the subformula property and the idempotent laws of conjunction ($\wedge$) and disjunction ($\vee$). Therefore, proof search of propositional logic is decidable.

However, in linear logic, the number of formulas is finite but the number of sequents is infinite, because the idempotent laws do not hold for multiplicative conjunction ($\otimes$) and multiplicative disjunction ($\wp$), that is, the sequent $A, A \longrightarrow B, B$ is not equivalent to $A \longrightarrow B$. Therefore, proof search of linear logic is undecidable in general, even for propositional fragment [13].

Therefore, in order to develop abstract proof search in linear logic, we need to map the sequents into finite sets. Of course, the provability is not equivalent between the original sequent and the mapped sequent. However, our concern is to statically detect the unprovability of the sequents, so we aim to define the mapping to keep the provability property.

Let $\Psi; \Delta_I \backslash \Delta_O \longrightarrow \mathcal{A}_I \backslash \mathcal{A}_O$ be the given goal sequent to be proved. We first map the sequent into propositional fragment by removing quantifiers and the arguments of all atomic formulas over $\boldsymbol{F}$.

Let $\boldsymbol{F}$ be a set of all propositional atomic formulas occurring in the program and the goal sequent. A multiset over $\boldsymbol{F}$ can be defined as a mapping $\boldsymbol{F} \longrightarrow \boldsymbol{N}$ where $\boldsymbol{N}$ is a set of non-negative integers. Now, we consider a finite representation of multisets.

Definition 3.1 (M-representation) *Let $\langle M, +, 0\rangle$ be a finite monoid homomorphic to $\boldsymbol{N}$, that is, there exists a mapping $\phi : \boldsymbol{N} \longrightarrow M$ satisfying $\phi(0) = 0$, $\phi(x + y) = \phi(x) + \phi(y)$ for all $x,\ y \in \boldsymbol{N}$. Let X be a multiset over $\boldsymbol{F}$. An M-representation of X is a composite mapping $\phi \circ X$.*

Example 3.1 *Let $M_n = \{0, 1, 2, \cdots, n\}$. $\langle M_n, +, 0\rangle$ is a finite monoid homomorphic to $\boldsymbol{N}$ by the following mapping.*

$$\phi(x) = \begin{cases} x \ if & x < n \\ n \ if & x \geq n \end{cases}$$

By using the M-representation, we can finitely represent the multiset by finitely counting the number of formulas occurring in the multiset. Now, we define a sequent calculus system on the basis of M-representations.

Definition 3.2 (M-multiset, M-sequent) *An M-multiset (over $\boldsymbol{F}$) is a mapping $\boldsymbol{F} \longrightarrow M$. An M-sequent is an expression $\Psi; \Delta_I \backslash \Delta_O \longrightarrow \mathcal{A}_I \backslash \mathcal{A}_O$ where Ψ is a set of propositional clauses and Δ_I, Δ_O, $\mathcal{A}_I$, A_O are M-multisets over $\boldsymbol{F}$.*

We denote the M-multiset over $\boldsymbol{F}$ by superscripting the M-element over

each formula of $\boldsymbol{F}$, where the formulas with superscript of 0 are omitted. For example, M_2-multiset $\{a^2, (a \otimes b)^1\}$ is an M_2-representation of the multiset $\{a, a, a, a \otimes b\}$.

Definition 3.3 *The proof system $\boldsymbol{IO}_M$ is a sequent calculus system obtained by regarding the sequence of formulas $B_1, \cdots, B_n$ as an M-representation of a multiset $\{B_1, \cdots, B_n\}$ in the rules of $\boldsymbol{IO}$.*

Example 3.2 *Let the program Ψ be $a \wp a :- a \otimes b$ and the goal sequent S_1 be $\Psi; a, b\backslash \ \longrightarrow a, a\backslash$, then the following are an $\boldsymbol{IO}$ proof and an $\boldsymbol{IO}_{M_2}$ proof respectively.*

$$\frac{\overline{\Psi; a, b\backslash b \longrightarrow a\backslash b} \quad \overline{\Psi; b\backslash \ \longrightarrow b\backslash}}{\Psi; a, b\backslash \ \longrightarrow a, a\backslash} \ (\otimes),$$

$$\frac{\overline{\Psi; a^1, b^1\backslash b^1 \longrightarrow a^1\backslash b^1} \quad \overline{\Psi; b^1\backslash \ \longrightarrow b^1\backslash}}{\Psi; a^1, b^1\backslash \ \longrightarrow a^1\backslash} \ (\otimes)$$

Many $\boldsymbol{IO}$ proofs can be mapped into the same $\boldsymbol{IO}_M$ proof.

Proposition 3.1

$$\boldsymbol{IO} \vdash \Psi; \Delta_I\backslash\Delta_O \longrightarrow \mathcal{A}_I\backslash\mathcal{A}_O \implies \boldsymbol{IO}_M \vdash \Psi; \Delta'_I\backslash\Delta'_O \longrightarrow \mathcal{A}'_I\backslash\mathcal{A}'_O$$

where Δ'_I, Δ'_O, $\mathcal{A}'_I$, $\mathcal{A}'_O$ are M-representations of Δ_I, Δ_O, $\mathcal{A}_I$, $\mathcal{A}_O$ respectively.
Proof. By the hypothesis, there exists a proof Π of the system $\boldsymbol{IO}$. Then we can obtain a proof of the system $\boldsymbol{IO}_M$ by replacing all sequent in Π by its M-representation. ∎

As the contraposition of this Proposition 3.1, we know the sequent is unprovable in the system $\boldsymbol{IO}$ if its M-representation is unprovable in the system $\boldsymbol{IO}_M$.

Proposition 3.2 *The provability in the proof system $\boldsymbol{IO}_M$ is decidable.*
Proof. It is obvious that the proof search of in the system $\boldsymbol{IO}_M$ is decidable. This is because the number of M-sequents is finite. ∎

4 Abstract Proof Graph

As described in the previous section, $\boldsymbol{IO}_M$ provability is theoretically decidable. However, naïve proof search on $\boldsymbol{IO}_M$ is inefficient and practically inapplicable for large programs. Especially, when executing the multiplicative conjunction $\otimes$, there are a large number of possible sequents as the upper sequent due to the large possibility of the output context. For example, the following $\boldsymbol{IO}_{M_2}$ proof :

$$\frac{a :- b \otimes c, \Psi; \ \backslash \ \longrightarrow b^1, d^2, e^2\backslash\Delta \qquad a :- b \otimes c, \Psi; \ \backslash \ \longrightarrow c^1, \Delta\backslash}{a :- b \otimes c, \Psi; \ \backslash \ \longrightarrow a^1, d^2, e^2\backslash} \ (\otimes)$$

the M_2-multiset Δ have nine cases of $\emptyset$, $\{e^1\}$, $\{e^2\}$, $\{d^1\}$, $\{d^1, e^1\}$, $\{d^1, e^2\}$, $\{d^2\}$, $\{d^2, e^1\}$, $\{d^2, e^2\}$.

In order to improve the efficiency, we use a set of M-multisets to express the alternatives by one representation. Let $\mathcal{P}(M)$ be the power set of M. Then, $\langle \mathcal{P}(M), +, \{0\} \rangle$ is a finite monoid by defining the operation $+$ as follows:

$$X + Y = \{x + y \mid x \in X, y \in Y\}$$

We also define the operation $-$ as follows:

$$X - Y = \{z \in M \mid x \in X, y \in Y, x = y + z\}$$

Please note $\langle \mathcal{P}(M), \cap, \cup \rangle$ forms a complete lattice.

Definition 4.1 ($\mathcal{P}(M)$-representation, $\mathcal{P}(M)$-multiset) *Let X be a multiset over $\boldsymbol{F}$ and ϕ be its M-representation. A $\mathcal{P}(M)$-representation of X denoted by $[X]$ is a mapping $\boldsymbol{F} \longrightarrow \mathcal{P}(M)$ such that $[X](B) = \{\phi(B)\}$ for all $B \in \boldsymbol{F}$. A $\mathcal{P}(M)$-multiset over $\boldsymbol{F}$ is a mapping $\boldsymbol{F} \longrightarrow \mathcal{P}(M)$.*

We also denote the $\mathcal{P}(M)$-multiset over $\boldsymbol{F}$ by superscripting the $\mathcal{P}(M)$ element over each formula of $\boldsymbol{F}$, where the formulas with superscript of $\{0\}$ are omitted. Especially for $\mathcal{P}(M_2)$, which our prototype analyzer base on, we use the following notation for simplicity:

Notation	Stands For	Meaning
B	$B^{\{1\}}$	one B
B^2	$B^{\{2\}}$	two or more B's
B'	$B^{\{0,1\}}$	at most one B
B''	$B^{\{0,2\}}$	zero or more than one B's
B^+	$B^{\{1,2\}}$	one or more B's
B^*	$B^{\{0,1,2\}}$	any number of B's

When the monoid M becomes larger, the static analysis becomes more precisely, but it costs more.

A $\mathcal{P}(M)$-multiset represents a set of possible M values for each formula in $\boldsymbol{F}$. In other words, a $\mathcal{P}(M)$-multiset denotes a set of M-multisets.

Definition 4.2 *Let X be a $\mathcal{P}(M)$-multiset. $\|X\|$ is a set of M-multisets defined as follows:*

$$\|X\| = \{Y \mid Y \text{ is an } M\text{-multiset } Y(B) \in X(B) \text{ for all } B \in \boldsymbol{F}\}$$

Example 4.1 *Let $X = \{a', b^*\} (= \{a^{\{0,1\}}, b^{\{0,1,2\}}\})$.*
Then $\|X\| = \{\{b^1\}, \{b^2\}, \{a^1, b^1\}, \{a^1, b^2\}\}$.

Definition 4.3 *Let X and Y be $\mathcal{P}(M)$-multisets over $\boldsymbol{F}$. $X + Y$, $X - Y$, $X \cap Y$, $X \cup Y$ are $\mathcal{P}(M)$-multisets satisfying the following for all $B \in \boldsymbol{F}$ respectively.*

$$(X+Y)(B) = X(B) + Y(B)$$
$$(X-Y)(B) = X(B) - Y(B)$$
$$(X \cap Y)(B) = X(B) \cap Y(B)$$
$$(X \cup Y)(B) = X(B) \cup Y(B)$$

Example 4.2 *For $\mathcal{P}(M_2)$-multisets,*

$$\{a^{\{0,1\}}, b^{\{0,1,2\}}\} + \{a^{\{1\}}, b^{\{2\}}\} = \{a^{\{0,1\}+\{1\}}, b^{\{0,1,2\}+\{2\}}\} = \{a^{\{1,2\}}, b^{\{2\}}\},$$
$$\{a^{\{0,1\}}, b^{\{0,1,2\}}\} - \{a^{\{1\}}, b^{\{2\}}\} = \{a^{\{0,1\}-\{1\}}, b^{\{0,1,2\}-\{2\}}\} = \{a^{\{0\}}, b^{\{0,1,2\}}\},$$
$$\{a^{\{0,1\}}, b^{\{0,1,2\}}\} \cap \{a^{\{1\}}, b^{\{2\}}\} = \{a^{\{0,1\}\cap\{1\}}, b^{\{0,1,2\}\cap\{2\}}\} = \{a^{\{1\}}, b^{\{2\}}\},$$
$$\{a^{\{0,1\}}, b^{\{0,1,2\}}\} \cup \{a^{\{1\}}, b^{\{2\}}\} = \{a^{\{0,1\}\cup\{1\}}, b^{\{0,1,2\}\cup\{2\}}\} = \{a^{\{0,1\}}, b^{\{0,1,2\}}\}.$$

In other words,

$$\{a', b^*\} + \{a, b^2\} = \{a^+, b^2\}, \qquad \{a', b^*\} - \{a, b^2\} = \{b^*\},$$
$$\{a', b^*\} \cap \{a, b^2\} = \{a, b^2\}, \qquad \{a', b^*\} \cup \{a, b^2\} = \{a', b^*\}.$$

Definition 4.4 ($\mathcal{P}(M)$-sequent) *A $\mathcal{P}(M)$-sequent over $\boldsymbol{F}$ is an expression $\Psi; \Delta_I \backslash \Delta_O \longrightarrow \mathcal{A}_I \backslash \mathcal{A}_O$ where Ψ is a set of propositional clauses and Δ_I, Δ_O, $\mathcal{A}_I$, $\mathcal{A}_O$ are $\mathcal{P}(M)$-multisets over $\boldsymbol{F}$.*

Definition 4.5 *Let S be a $\mathcal{P}(M)$-sequent $\Psi; \Delta_I \backslash \Delta_O \longrightarrow \mathcal{A}_I \backslash \mathcal{A}_O$. The $\|S\|$ is a set of M-sequents defined by :*

$$\|S\| = \{\Psi; \Delta'_I \backslash \Delta'_O \longrightarrow \mathcal{A}'_I \backslash \mathcal{A}'_O \mid \Delta'_I \in \|\Delta_I\|, \Delta'_O \in \|\Delta_O\|, \mathcal{A}'_I \in \|\mathcal{A}_I\|, \mathcal{A}'_O \in \|\mathcal{A}_O\|\}$$

Definition 4.6 *An AND-OR graph [2] is a tuple $\langle V, E, v_0 \rangle$ where V is a set of nodes, E is a set of AND arcs ($E \subseteq V \cup V^2 \cup V^3$) and $v_0 \in V$ is the root node.*

A tuple $\langle v, v_1, v_2 \rangle \in E$ means an AND arc from v to v_1 and v_2. A pair $\langle v, v_1 \rangle \in E$ means an arc from v to v_1. A singleton $\langle v \rangle \in E$ means an arc from v but no destination nodes. A bundle of AND arcs from the same node v represents OR selection.

Definition 4.7 *Let $G \equiv \langle V, E, v_0 \rangle$ be an AND-OR graph. Let T be a tree in which the set of node is U and the root node is $u_0 \in U$. The tree T is contained in G if there exists a mapping $f : U \longrightarrow V$ satisfying:*

(i) *$f(u_0) = v_0$,*

(ii) *for all node $u \in U$ with child nodes $u_1, \cdots, u_n$, $\langle f(u), f(u_1), \cdots, f(u_n) \rangle \in E$ ($n = 1$ or 2) and*

(iii) *for all leaf node $u \in U$, $\langle f(u) \rangle \in E$.*

Even if the AND-OR graph is finite, infinitely many trees can be contained in it, due to the possibility of cycles in the graph.

Definition 4.8 *An AND-OR graph $\langle V, E, v_0 \rangle$ is called a proof graph if V is a set of $\mathcal{P}(M)$-sequents.*

[2] The definition of And-OR graph used here is different from that in standard textbooks.

Definition 4.9 *Let G be a proof graph and Π be an $\boldsymbol{IO}_M$ proof. An $\boldsymbol{IO}_M$ proof is contained in G if*

(i) *Π is contained in G as a tree by mapping f,*

(ii) *$S \in \|f(S)\|$ for any M-sequent S in Π.*

Definition 4.10 *Let G be a proof graph. G is called an abstract proof graph if all $\boldsymbol{IO}_M$ proofs are contained in G.*

Example 4.3 *Let Ψ be the following program*

$$s \wp s \ :- \ s \otimes m$$
$$m \ :- \ \mathbf{1}.$$

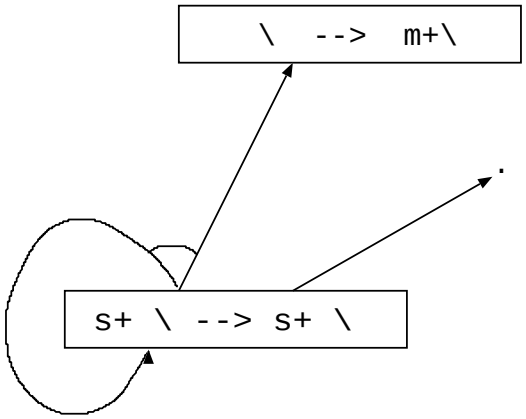
Let $\Psi; s^1\backslash \ \longrightarrow s^1\backslash$ be the goal M_1-sequent. The followings are its three $\boldsymbol{IO}_{M_1}$proofs.

$$\frac{}{\Psi; s^1\backslash \ \longrightarrow s^1\backslash} \ ,$$

$$\frac{\dfrac{}{\Psi; s^1\backslash \ \longrightarrow s^1\backslash} \quad \dfrac{}{\Psi; \backslash \ \longrightarrow m^1\backslash}}{\Psi; s^1\backslash \ \longrightarrow s^1\backslash} \ ,$$

$$\frac{\dfrac{\dfrac{}{\Psi; s^1\backslash \ \longrightarrow s^1\backslash} \quad \dfrac{}{\Psi; \backslash \ \longrightarrow m^1\backslash}}{\Psi; s^1\backslash \ \longrightarrow s^1\backslash} \quad \dfrac{}{\Psi; \backslash \ \longrightarrow m^1\backslash}}{\Psi; s^1\backslash \ \longrightarrow s^1\backslash}$$

Figure 3 shows an abstract proof graph containing all above three proofs.



where the program Ψ is omitted

Fig. 3. An abstract proof graph for the goal M_1-sequent, $\Psi; s^1\backslash \ \longrightarrow s^1\backslash$

Now, we describe an algorithm to create an abstract proof graph from a given goal $\mathcal{P}(M)$-sequent.

Definition 4.11 (Initial Proof Graph Algorithm) *Let S be the given goal $\mathcal{P}(M)$-sequent to be proved. The initial proof graph is a proof graph obtained by repeatedly adding nodes and arcs by applying the following steps*

until no more nodes or arcs can be added.
Let S be a root node v_0, and for any node v, $\Psi; \Delta_I \backslash \Delta_O \longrightarrow \mathcal{A}_I \backslash \mathcal{A}_O$,

(i) *if $\Delta_I \cap [\{A\}] \neq \emptyset$, $\mathcal{A}_I \cap [\{A\}] \neq \emptyset$ for some atomic formula A, then add an arc $\langle v \rangle$,*

(ii) *if $(A_1 \wp \cdots \wp A_n :- \mathbf{1}) \in \Psi$ and $\mathcal{A}_I \cap [\{A_1, \cdots, A_n\}] \neq \emptyset$, then add an arc $\langle v \rangle$,*

(iii) *if $(A_1 \wp \cdots \wp A_n :- \bot) \in \Psi$ and $\mathcal{A}_I \cap [\{A_1, \cdots, A_n\}] \neq \emptyset$, then add a node v_1, $\Psi; \Delta_I \backslash \Delta_O \longrightarrow \mathcal{A}_I - [\{A_1, \cdots, A_n\}] \backslash \mathcal{A}_O$, and add an arc $\langle v, v_1 \rangle$,*

(iv) *if $(A_1 \wp \cdots \wp A_n :- B \multimap C) \in \Psi$ and $\mathcal{A}_I \cap [\{A_1, \cdots, A_n\}] \neq \emptyset$, then add a node*
v_1, $\Psi; \Delta_I + [\{B\}] \backslash \Delta_O \longrightarrow \mathcal{A}_I - [\{A_1, \cdots, A_n\}] + [\{C\}] \backslash \mathcal{A}_O$, and add an arc $\langle v, v_1 \rangle$,

(v) *if $(A_1 \wp \cdots \wp A_n :- B \wp C) \in \Psi$ and $\mathcal{A}_I \cap [\{A_1, \cdots, A_n\}] \neq \emptyset$, then add a node*
v_1, $\Psi; \Delta_I \backslash \Delta_O \longrightarrow \mathcal{A}_I - [\{A_1, \cdots, A_n\}] + [\{B, C\}] \backslash \mathcal{A}_O$, and add an arc $\langle v, v_1 \rangle$,

(vi) *if $(A_1 \wp \cdots \wp A_n :- B \otimes C) \in \Psi$ and $\mathcal{A}_I \cap [\{A_1, \cdots, A_n\}] \neq \emptyset$, then add two nodes*
v_1,
$\Psi; \Delta_I \backslash Cl(\Delta_I) \longrightarrow \mathcal{A}_I - [\{A_1, \cdots, A_n\}] + [\{B\}] \backslash Cl(\mathcal{A}_I - [\{A_1, \cdots, A_n\}])$,
v_2, $\Psi; Cl(\Delta_I) \backslash \Delta_O \longrightarrow Cl(\mathcal{A}_I - [\{A_1, \cdots, A_n\}]) + [\{C\}] \backslash \mathcal{A}_O$, and add an arc $\langle v, v_1, v_2 \rangle$.

Where, the $Cl(X)$ used above is defined as follows.

$$Cl(X) = \{z \in M \mid x \in X, y \in M, x = y + z\}$$

This algorithm always stops because there are only finite number of nodes and arcs.

Note that $X \subset Cl(X)$ for any $\mathcal{P}(M)$-multiset X because $\{z \in M \mid x \in X, z = 0 + z\} \subset Cl(X)$.

Proposition 4.1 *The initial proof graph is an abstract proof graph.*
Proof. Let S be an M-sequent and

$$\frac{S_1 \cdots S_n}{S}$$

be a correct $\boldsymbol{IO}_M$ inference step ($n = 0, 1, 2$). It is easy to verify that when $S \in \|v\|$ for some node v in the initial proof graph, there exists an arc $\langle v, v_1, \cdots v_n \rangle$ and $S_i \in \|v_i\|$ ($i = 1, \cdots, n$). The case of $\otimes$ is easily proved by using $\mathcal{A} \subset Cl(\mathcal{A})$. ∎

5 Refinement of Abstract Proof Graph

In the previous section, we describe the algorithm to generate an initial abstract proof graph, but the graph might contain a lot of unprovable nodes.

In this section, we discuss how to eliminate those unprovable nodes. This can be seen as an optimization procedure on proof graphs, that is, the elimination of nodes and arcs means the elimination of redundant inference steps.

We borrow the idea of data-flow analysis for the optimization. Each formula $B \in \boldsymbol{F}$ is considered as a variable, and the $\mathcal{P}(M)$ value for B is considered as a set of possible values for the variable. By the data-flow analysis, the possible set will be reduced to be smaller, and when it becomes the empty set, the node will be removed.

The reader may ask "What is the data-flow in proof graphs". In the I/O model, the input context usually comes up from the lower sequent, and the output context usually goes down from the upper sequent. In addition, in the case of $(\otimes)$-rule, the output context of the upper left sequent goes to the input context of the upper right sequent.

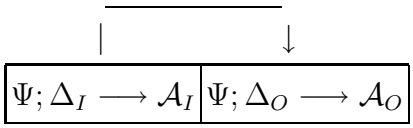
Therefore, we first split each node of proof graph into an input part and an output part, that is, the node $\Psi; \Delta_I \backslash \Delta_O \longrightarrow \mathcal{A}_I \backslash \mathcal{A}_O$ is split into $\Psi; \Delta_I \longrightarrow \mathcal{A}_I$ as the input part and $\Psi; \Delta_O \longrightarrow \mathcal{A}_O$ as the output part. So, the node can be figured as follows.

$\Psi; \Delta_I \longrightarrow \mathcal{A}_I$	$\Psi; \Delta_O \longrightarrow \mathcal{A}_O$

Now, we add data-flow links for each AND arc (that is, for each inference step) in the initial abstract proof graph. Each link has a constraint specifying the condition of the inference step. The following describes what links are added for each AND arc and their constraints.

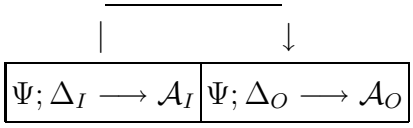
Definition 5.1 *Let G be a proof graph. The following defines the data-flow links for each AND arc in G.*

- *If the arc is for (identity)-rule of the atomic formula A, the following link is added.*



  Constraint: $(\Delta_I - [\{A\}]) \cap \Delta_O \neq \emptyset$, $(\mathcal{A}_I - [\{A\}]) \cap \mathcal{A}_O \neq \emptyset$

- *If the arc is for $(A_1 \wp \cdots \wp A_n :- \mathbf{1})$, the following link is added.*



  Constraint: $\Delta_I \cap \Delta_O \neq \emptyset$, $(\mathcal{A}_I - [\{A_1, \cdots, A_n\}]) \cap \mathcal{A}_O \neq \emptyset$

- *If the arc is for* $(A_1 \wp \cdots \wp A_n :- \bot)$*, the following two links are added.*

$$\begin{array}{|c|c|}\hline \Psi; \Delta'_I \longrightarrow \mathcal{A}'_I & \Psi; \Delta'_O \longrightarrow \mathcal{A}'_O \\ \hline \end{array}$$
$$\uparrow \qquad\qquad \downarrow$$
$$\begin{array}{|c|c|}\hline \Psi; \Delta_I \longrightarrow \mathcal{A}_I & \Psi; \Delta_O \longrightarrow \mathcal{A}_O \\ \hline \end{array}$$

  Constraint for the left link: $\Delta_I \cap \Delta'_I \neq \emptyset$, $(\mathcal{A}_I - [\{A_1, \cdots, A_n\}]) \cap \mathcal{A}'_I \neq \emptyset$
  Constraint for the right link: $\Delta'_O \cap \Delta_O \neq \emptyset$, $\mathcal{A}'_O \cap \mathcal{A}_O \neq \emptyset$

- *If the arc is for* $(A_1 \wp \cdots \wp A_n :- B \multimap C)$*, the following two links are added.*

$$\begin{array}{|c|c|}\hline \Psi; \Delta'_I \longrightarrow \mathcal{A}'_I & \Psi; \Delta'_O \longrightarrow \mathcal{A}'_O \\ \hline \end{array}$$
$$\uparrow \qquad\qquad \downarrow$$
$$\begin{array}{|c|c|}\hline \Psi; \Delta_I \longrightarrow \mathcal{A}_I & \Psi; \Delta_O \longrightarrow \mathcal{A}_O \\ \hline \end{array}$$

  Constraint for the left link: $(\Delta_I + [\{B\}]) \cap \Delta'_I \neq \emptyset$, $((\mathcal{A}_I - [\{A_1, \cdots, A_n\}]) + [\{C\}]) \cap \mathcal{A}'_I \neq \emptyset$
  Constraint for the right link: $\Delta'_O \cap \Delta_O \neq \emptyset$, $\mathcal{A}'_O \cap \mathcal{A}_O \neq \emptyset$

- *If the arc is for* $(A_1 \wp \cdots \wp A_n :- B \wp C)$*, the following two links are added.*

$$\begin{array}{|c|c|}\hline \Psi; \Delta'_I \longrightarrow \mathcal{A}'_I & \Psi; \Delta'_O \longrightarrow \mathcal{A}'_O \\ \hline \end{array}$$
$$\uparrow \qquad\qquad \downarrow$$
$$\begin{array}{|c|c|}\hline \Psi; \Delta_I \longrightarrow \mathcal{A}_I & \Psi; \Delta_O \longrightarrow \mathcal{A}_O \\ \hline \end{array}$$

  Constraint for the left link: $\Delta_I \cap \Delta'_I \neq \emptyset$, $((\mathcal{A}_I - [\{A_1, \cdots, A_n\}]) + [\{B, C\}]) \cap \mathcal{A}'_I \neq \emptyset$
  Constraint for the right link: $\Delta'_O \cap \Delta_O \neq \emptyset$, $\mathcal{A}'_O \cap \mathcal{A}_O \neq \emptyset$

- *If the arc is for* $(A_1 \wp \cdots \wp A_n :- B \otimes C)$*, the following three links are added.*

$$\begin{array}{|c|c|}\hline \Psi; \Delta'_I \longrightarrow \mathcal{A}'_I & \Psi; \Delta'_O \longrightarrow \mathcal{A}'_O \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|}\hline \Psi; \Delta''_I \longrightarrow \mathcal{A}''_I & \Psi; \Delta''_O \longrightarrow \mathcal{A}''_O \\ \hline \end{array}$$
$$\nwarrow \qquad\qquad \swarrow$$
$$\begin{array}{|c|c|}\hline \Psi; \Delta_I \longrightarrow \mathcal{A}_I & \Psi; \Delta_O \longrightarrow \mathcal{A}_O \\ \hline \end{array}$$

  Constraint for the left link: $\Delta_I \cap \Delta'_I \neq \emptyset$, $((\mathcal{A}_I - [\{A_1, \cdots, A_n\}]) + [\{B\}]) \cap \mathcal{A}'_I \neq \emptyset$
  Constraint for the top link: $\Delta'_O \cap \Delta''_I \neq \emptyset$, $(\mathcal{A}'_O + [\{C\}]) \cap \mathcal{A}''_I \neq \emptyset$
  Constraint for the right link: $\Delta''_O \cap \Delta_O \neq \emptyset$, $\mathcal{A}''_O \cap \mathcal{A}_O \neq \emptyset$

Proposition 5.1 *Let G be a proof graph with the root node v_0. If G contains an $\boldsymbol{IO}_M$ proof Π, there exists a path of data-flow links starting from the input part of v_0 and ending at the output part of v_0, and each data-flow link in the path satisfies its constraint condition.*

Proof. It is easy to verify that any correct inference step of the system $\boldsymbol{IO}_M$ satisfies the corresponding constraint conditions. ∎

Therefore, we can use an iterative algorithm widely used in optimizing

compilers by propagating information through all possible paths. The outline of the information propagation algorithm is as follows.

Definition 5.2 (A Forward Propagation Algorithm) *Let G be a proof graph with the root node v_0.*

(i) *Set all input and output parts except v_0's to be $\emptyset \longrightarrow \emptyset$.*

(ii) *Do the followings for all paths of data-flow links starting from the input part of v_0 and ending at the output part of v_0 until no informations are changed.*

- *If $\Delta \longrightarrow \mathcal{A}$ and $\Delta' \longrightarrow \mathcal{A}'$ are connected by a link with constraint $C(\Delta, \Delta', \mathcal{A}, \mathcal{A}')$, calculate the minimum Δ'' and $\mathcal{A}''$ satisfying $C(\Delta, \Delta'', \mathcal{A}, \mathcal{A}'')$ and update Δ' and $\mathcal{A}'$ by $\Delta' \cup \Delta''$ and $\mathcal{A}' \cup \mathcal{A}''$ respectively.*

A Backward Propagation Algorithm can be similarly defined.

Definition 5.3 (A Refinement Algorithm)

(i) *Do the forward propagation.*

(ii) *Do the elimination.*

- ***A Elimination Algorithm***
  Eliminate the nodes which the input or output parts including an empty set and the links having unsatisfiable constraints.

(iii) *Do the backward propagation.*

(iv) *Do the elimination.*

This refinement can be repeated any times. In other words, the repeating execution of the refinement can be stopped at anytime. However, of course, there is no meaning to repeat the refinement when the information propagation does not change anything anymore.

6 Performance Measurements

We developed a prototype system doing the static analysis described (based on $\mathcal{P}(M_2)$) in this paper. The list sorting program written in Forum is used for the analysis [10].

The following is the program given to the analyzer.

$$g \ :\!- \ s \multimap p$$

$$p \ :\!- \ s$$

$$p \ :\!- \ s \wp p$$

$$s \wp s \ :\!- \ s \otimes m$$

$$m \ :\!- \ \mathbf{1}$$

$$m \ :\!- \ m$$

First, the analyzer creates the initial abstract proof graph from the above program and the goal sequent $\longrightarrow g$. The initial abstract proof graph consists of the 57 nodes and 220 arcs. After the refinement for the abstract proof graph, the number of nodes and arcs are reduced as shown below.

	No. of nodes	No. of arcs
Initial graph	57	220
After 1 refinement	25	53
After 2 refinements	15	29

Because the above proof graph is too large to describe in this paper, we will show the initial abstract proof graph and refined graph for the goal sequent $s \longrightarrow s^2$ in Figure 4 and Figure 5 respectively. The program Ψ is omitted in Figure 4 and Figure 5 because it is invariant in the proof graph on this case. The initial abstract proof graph consists of 10 nodes, and the refined graph has only 4 nodes.

We translated the sorting program into our LLP language program by hands. LLP is a compiler system of an intuitionistic linear logic programming language [3] [16][12].

The translated program without using the analysis result checks whether it can use all rules at each inference step or not. The optimized translated program using the analysis result does not check the rules eliminated by the analyzer.

The former version spent about 9.5 seconds for sorting 6 elements, and the later only spend 10 milliseconds. The more elements are sorted, the larger the difference becomes.

7 Conclusion and Future work

In this paper, we proposed a new static analysis method which is applicable for a fragment of a classical linear logic programming language.

Our method, covers multiplicative conjunction in addition to multiplicative disjunction and linear implication.

We introduced an abstract proof graph representing all possible sequent proofs. The graph is constructed from a given program and goal sequent. Furthermore, the abstract proof graph can be repeatedly refined by propagating information to eliminate unprovable nodes from the graph. The method of refinement is based on the idea of data-flow analysis for the optimization.

Finally, we applied our prototype analyzer for a sorting program written in Forum. The sorting program was improved about 1000 times faster than the ordinary program without analysis, for sorting 6 elements by using the analysis result.

[3] `http://bach.seg.kobe-u.ac.jp/llp/`

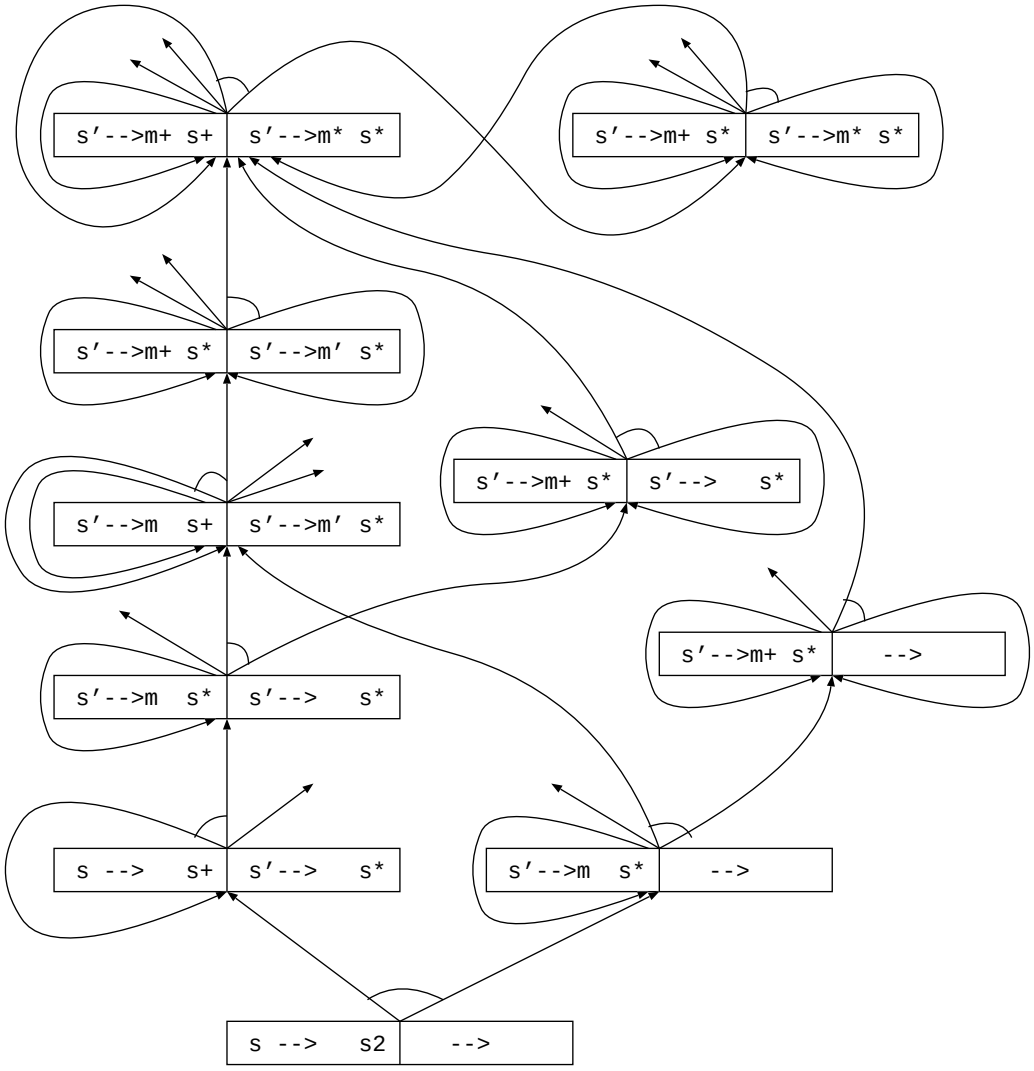


Fig. 4. Initial Abstract Proof Graph

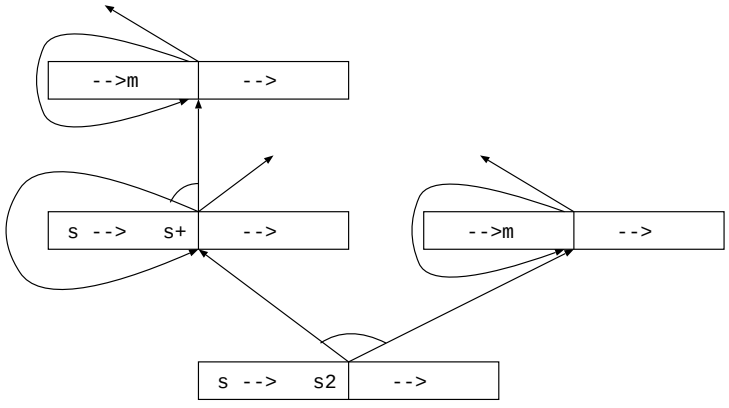


Fig. 5. Refined Abstract Proof Graph

As a future work, we are planning to extend of our static analysis method to include more connectives, such as the additive conjunction & and the intuitionistic implication $\Rightarrow$ (where $A \Rightarrow B \equiv !A \multimap B$).

References

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

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

[3] J.-M. Andreoli, R. Pareschi, and T. Castagnetti. Static analysis of linear logic programming. In *New Generation Computing,15*, pages 449–481, 1997.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoint. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[5] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. In *Journal of Logic Programming,13*, pages 103–179, 1992.

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

[7] J. Harland and D. Pym. The uniform proof-theoretic foundation of linear logic programming. In V. Saraswat and K. Ueda, editors, *Proceedings of the International Logic Programming Symposium*, pages 304–318, San Diego, California, Oct. 1991.

[8] J. S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design and Implementation*. PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1994.

[9] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. Extended abstract in the Proceedings of the Sixth Annual Symposium on Logic in Computer Science, Amsterdam, July 15–18, 1991.

[10] J. S. Hodas and J. Polakow. Forum as a logic programming language: Preliminary results and observations. In M. Okada, editor, *Proceedings of the Linear Logic '96 Meeting*, volume 3, Tokyo, Japan, 1996. Elsevier Electronic Notes in Theoretical Computer Science.

[11] J. S. Hodas and J. Polakow. Early observation on forum as a logic programming language. Unpublised, 1997.

[12] J. S. Hodas, K. Watkins, N. Tamura, and K.-S. Kang. Efficient implementation of a linear logic programming. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 145–159, 1998.

[13] P. Lincoln. Linear logic. *ACM SIGACT Notices*, 23(2):29–37, Spring 1992.

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

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

[16] N. Tamura and Y. Kaneda. Extension of WAM for a linear logic programming language. In T. Ida, A. Ohori, and M. Takeichi, editors, *Second Fuji International Workshop on Functional and Logic Programming*, pages 33–50. World Scientific, Nov. 1996.