TR-0765

Less Abstract Sematics for Abstract Interpretation of FGHC Programs

by

K. Horiuchi

April, 1992

To appear in the proceedings of the International Conference on Fifth Generation Computer Systems 1992, Tokyo, June 1992.

Less Abstract Semantics for Abstract Interpretation of FGHC Programs

Kenji Horiuchi

Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108, JAPAN
horiuchi@icot.or.jp

Abstract

In this paper we present a denotational semantics for Flat GHC. In the semantics, the reactive behavior of a goal is represented by a sequence of substitutions, which are annotated with + or − depending on whether the bindings are given from, or posted to the environment of the goal. Our objective in investigating the semantics is to develop a framework for abstract interpretation. So, the semantics is *less abstract* enough to allow an analysis of various properties closely related to program sources. We also demonstrate moded type inference of FGHC programs using abstract interpretation based on the semantics.

1 Introduction

Various work on the semantics for concurrent logic languages has been investigated by many researchers [Gerth *et al.* 1988][Murakami 1988][Gaifman *et al.* 1989] [Gabbrielli and Levi 1990][de Boer and Palamidessi 1990]. One of their main purposes is to identify one program with another syntactically different program, or distinguish between syntactically similar programs. And, since some researchers are interested in properties like *fully abstractness*, they may want to hide *internal communications* from the semantics or want to abstract even observable behaviors much further.

Since our main objective is to analyze a program unlike the above researchers, we want to have a fixpoint semantics suitable to the collecting semantics, on which our framework of abstract interpretation is based. But once we try to introduce one of their semantics to a framework of abstract interpretation, the semantics may be too abstract to obtain some of the properties we require.

In this paper we present a denotational fixpoint semantics for Flat GHC. In the semantics the reactive behavior of a goal is represented by a sequence of substitutions which are annotated with + or − depending on whether the bindings are given from, or posted to the environment of the goal. The semantics presented here is *less abstract* enough to allow an analysis of various properties closely related to program sources, e.g., on occurrences of symbols in programs or internal communications. We also demonstrate moded type inference of FGHC programs using abstract interpretation.

We briefly explain the concurrent logic programming language Flat GHC and its operational semantics in Section 3 after we introduce the preliminary notions in the next section. Next we present the fixpoint approach to the semantics of Flat GHC in Section 4, and then in Section 5 we show the relationship between the fixpoint semantics and the operational semantics. After reviewing a general framework for abstract interpretation, we show examples of analyzing FGHC programs.

2 Preliminaries

In this section, we introduce the following basic notions used in this paper, many of which are defined as usual [Lloyd 1987][Palamidessi 1990].

Definition 2.1 (Functor, Term, Atom, Predicate and Expression)
Let *Var* be a non empty set of *variables*, *Func* be a set of *functors*, *Term* be a set of all *terms* defined on *Var* and on *Func*, *Pred* be a set of all *predicates* and *Atom* be a set of all *atoms* defined on *Term* and *Pred*.

An *expression* is a term, an atom, a tuple of expressions or a (multi)set of expressions, and we denote a set of all expressions by *Exp*. We also denote the set of all variables appearing in an expression E by $var(E)$.

Definition 2.2 (Substitution)
A *substitution* θ is a mapping from *Var* to *Term* such that the *domain* of θ is finite, where the domain of θ, denoted by $dom(\theta)$, is defined by $\{V \in Var \mid \theta(V) \not\equiv V\}$. The substitution θ is also represented by a set of *assignments* such that $\{V \leftarrow t \mid V \in dom(\theta) \wedge \theta(V) \equiv t\}$. The identity mapping on *Var*, called an *identity substitution*, is denoted by $\emptyset$. The *range* of θ, denoted by $ran(\theta)$, is a set of all variables appearing in terms at the right hand side of each assignment of θ, i.e., $\bigcup_{V \in dom(\theta)} var(\theta(V))$. $var(\theta)$ also denotes the set of variables $dom(\theta) \cup ran(\theta)$

When E is an expression, $E\theta$ (or $(E)\theta$) denotes an expression obtained by replacing each variable V in E with $\theta(V)$. The *composition* of two substitution θ and σ, denoted by $\theta\sigma$, is defined as usual [Lassez *et al.* 1987][Palamidessi 1990]. A substitution θ is assumed to be always *idempotent* [Lassez *et al.* 1987]. (i.e.,

$dom(\theta) \cap ran(\theta) = \emptyset$, where $\emptyset$ denotes an empty set.) And the result of composing substitutions is also assumed to be idempotent. The set of all idempotent substitutions is denoted by *Subst* and the set of all renamings is denoted by *Ren*. A *restriction* of θ *onto* $var(E)$ is denoted by $\theta_{|E}$.

Definition 2.3 (Equivalence Class and Partial Ordering)
A pre-ordering $\preceq$ on *Subst*, called an *instantiation ordering*, is defined as follows: $\theta_1 \preceq \theta_2$ iff $\exists\sigma(\theta_1\sigma = \theta_2)$, where $\theta_1, \theta_2, \sigma \in Subst$. The equivalence relation w.r.t. an instantiation ordering $\preceq$, denoted by $\sim$, is defined as follows: $\theta_1 \sim \theta_2$ iff $\exists\eta(\theta_1\eta = \theta_2)$, where $\eta \in Ren$.

And substitutions θ_1 and θ_2 are said to *be in an equivalence class* when $\theta_1 \sim \theta_2$. A set of the equivalence classes of *Subst* is denoted by $Subst_{/\sim}$. A partial ordering on $Subst_{/\sim}$, also denoted by $\preceq$, is naturally induced from a pre-ordering $\preceq$ on *Exp*. We denote the equivalence class of a substitution θ by $\theta_\sim$, or simply by θ. Given $\top$ as the greatest element on $\preceq$ of $Subst_{/\sim}$, $Subst_{/\sim}$ can be naturally extended to $Subst^\top_{/\sim}$. Then $(Subst^\top_{/\sim}, \preceq)$ forms a complete lattice.

Definition 2.4 (Most General Unifier)
A *most general unifier (mgu)* θ of expressions E_1, E_2, denoted by $mgu(E_1, E_2)$, iff $E_1\theta \equiv E_2\theta$ and $E_1\theta' \equiv E_2\theta' \supset \theta \preceq \theta'$ for all θ'. Let U be a set of equations $\{s_1{=}t_1, \ldots, s_n{=}t_n\}$. Then $mgu([s_1, \ldots, s_n], [t_1, \ldots, t_n])$ is also denoted by $mgu(U)$. A substitution θ can also be represented by a set of equations, denoted by $Eq(\theta)$, such that $Eq(\theta) = \{X = t \mid (X \leftarrow t) \in \theta\}$.

Definition 2.5 (Directed)
Let θ_1, θ_2 be substitutions. Then θ_1 and θ_2 are said to *be directed*, denoted by $\theta_1 \bowtie \theta_2$, iff $var(\theta_1) \cap var(\theta_2) = dom(\theta_1) \cap dom(\theta_2)$.

Example 2.1 Consider two substitutions $\theta_1 = \{X \leftarrow U, Y \leftarrow U\}$, $\theta_2 = \{X \leftarrow Y\}$ in an equivalence class and a substitution $\sigma = \{X \leftarrow f(V), Y \leftarrow f(a)\}$. Then θ_1 and σ are directed, but θ_2 and σ are not directed.

As $(Subst^\top_{/\sim}, \preceq)$ forms a complete lattice, every subset of $Subst^\top_{/\sim}$ has the *lub (least upper bound)* and the *glb (greatest lower bound)* w.r.t. $\preceq$. Several algorithms for computing the *lub* and the *glb* have already been presented [Lassez *et al.* 1987][Palamidessi 1990]. In [Palamidessi 1990], two operations: $Subst_{/\sim} \times Subst_{/\sim} \rightarrow Subst_{/\sim}$ are provided, which are called a *parallel composition* $\uparrow$ and a *parallel factorization* $\downarrow$. This has shown $\theta_1 \uparrow \theta_2 = lub(\theta_1, \theta_2)$ and $\theta_1 \downarrow \theta_2 = glb(\theta_1, \theta_2)$.

We now review the two operations in [Palamidessi 1990] briefly. Let θ_1, θ_2 be (equivalence classes) of idempotent substitutions. $\theta_1 \uparrow \theta_2$ is defined $mgu(Eq(\theta_1) \cup Eq(\theta_2))$. And $\theta_1 \downarrow \theta_2$ is defined by using the *factorization algorithm* which repeatedly replaces the different symbol at the same position in the bindings by a variable and finally generates (an equivalence class of) a substitution η as the $glb(\theta_1, \theta_2)$ with two addenda σ_1, σ_2. Then, the following property is also shown between these substitutions: $\eta\sigma_1 = \theta_1$ and $\eta\sigma_2 = \theta_2$, where σ_1, σ_2 are called *side substitutions*. Here we call σ_1 (or σ_2) a *most general difference (mgd) of* θ_2 (or θ_1) *from* θ_1 (or θ_2), and denote it by $mgd(\theta_1, \theta_2)$ (or $mgd(\theta_2, \theta_1)$.)

Definition 2.6 (Compatibility and Complement)
θ_1 and θ_2 are said to *be compatible*, denoted by $\theta_1 \approx \theta_2$, iff $lub(\theta_1, \theta_2) \neq \top$. And they are said to *be incompatible*, denoted by $\theta_1 \not\approx \theta_2$, iff $lub(\theta_1, \theta_2) = \top$. A *complement* of a substitution $\theta_{/\sim} \in Subst_{/\sim}$, denoted by $\overline{\theta}_\sim$ or $\overline{\theta}$, is a set of all (equivalence classes of) substitutions incompatible with θ, which is defined by $\{\theta'_\sim \in Subst_{/\sim} \mid \theta_\sim \not\approx \theta'_\sim\}$

Example 2.2 Consider substitutions $\theta_1 = \{X \leftarrow f(Z), Y \leftarrow f(Z)\}$ and $\theta_2 = \{X \leftarrow f(a), Y \leftarrow f(b)\}$. Since the parallel composition $\theta_1 \uparrow \theta_2$ is $\top$, they are *incompatible*. And the parallel factorization $\theta_1 \downarrow \theta_2$ is the substitution $\{X \leftarrow f(U), Y \leftarrow f(V)\}$, and the most general difference $mgd(\theta_1, \theta_2)$ is $\{U \leftarrow V\}$ and $mgd(\theta_2, \theta_1)$ is $\{U \leftarrow a, V \leftarrow b\}$.

3 Flat Guarded Horn Clauses

Now, we briefly recall a concurrent programming language *Flat Guarded Horn Clauses (FGHC)*, and then define the operational semantics of Flat GHC in terms of a transition system [Ueda 1990b].

3.1 Syntax of FGHC

An *FGHC program* is a set of *flat guarded clauses*. A flat guarded clause (simply, *clause*) is of the form:

$$p(t_1, \ldots, t_k) \text{ :- } G_1, \ldots, G_m \mid B_1, \ldots, B_n. \qquad (k, m, n \geq 0),$$

where p is a k-ary predicate symbol, $t_1, \ldots, t_k$ are terms, and $G_1, \ldots, G_m$, $B_1, \ldots, B_n$ are atoms. The atom $p(t_1, \ldots, t_k)$ is called a *head*, the head and $G_1, \ldots, G_m$ are called a *guard* and $B_1, \ldots, B_n$ is called a *body*. One binary predicate "=" for unifying two terms is predefined by the language, a goal of which is called a *unification goal*. Each guard goal G_i must be a unification goal.

3.2 Operational Semantics of Flat GHC

In [Ueda 1990b], Ueda has defined the operational semantics of FGHC in the style of Plotkin. Here we present it by following his definition.

Definition 3.1 (Transition System of FGHC)
A *transition system of* an FGHC program P is defined by using a *configuration* and a *transition relation*. A configuration is a pair of the form $\langle B, E\rangle$ where B is a multiset of goals and E is a binding environment of B. A binding environment E is a multiset of equations C with a set of variables V such that $var(B) \cup var(C) \subseteq V$, denoted by $C{:}V$.

A transition relation under P, denoted by $Trans_P$, is the smallest set of binary relations on configurations, denoted by $\cdot \rightarrow \cdot$, such that:

$$\frac{\langle B_1,\, C_1\!:V_1\rangle \rightarrow \langle B_1',\, C_1'\!:V_1'\rangle}{\langle B_1 \cup B_2,\, C_1\!:V_1\rangle \rightarrow \langle B_1' \cup B_2,\, C_1'\!:V_1'\rangle} \qquad (1)$$

$$\frac{\langle \{A{=}H\} \cup G,\, C\!:V \cup var((H,G))\rangle \xrightarrow{*} \langle \emptyset,\, C \cup C_g\!:V'\rangle}{\langle \{A\},\, C\!:V\rangle \rightarrow \langle B,\, C \cup C_g\!:V' \cup var(B)\rangle}$$
$$\text{if } \exists c \in P\, \exists \eta \in Ren((H :\!-\, G \,|\, B) \equiv c\eta \wedge V \cap var(c\eta) \equiv \emptyset)$$
$$\text{and } \models \forall.(C \supset \exists(var(C_g) \setminus var(A)).C_g) \qquad (2)$$

$$\frac{}{\langle s{=}t,\, C\!:V\rangle \rightarrow \langle \emptyset,\, C \cup \{s{=}t\}\!:V\rangle} \qquad (3)$$

When $c_1 \rightarrow c_2 \subset Trans_P$, $c_1 \rightarrow c_2$ is said to *be in the transition system of* P or c_1 is said to *be reduced to* c_2 *under a program* P. Then, a computation of a program P with an initial goal B is represented by a (possibly infinite) sequence of transitions in $Trans_P$; $c_1 \rightarrow c_2 \rightarrow \ldots \rightarrow c_i \rightarrow \ldots$ such that c_1 is $\langle B, \emptyset\!: var(B)\rangle$. Each configuration $c_i (i \geq 1)$ is called a *possible configuration* from B.

We may use alternative denotations $\xrightarrow{\cup}$, $\xrightarrow{c}$ and $\xrightarrow{s=t}$ corresponding to transition rules (1),(2) and (3) respectively if it is necessary to identify them. The reflexive and transitive closure of $\rightarrow$, by applying $\xrightarrow{c}$ (or $\xrightarrow{s=t}$) once only, is denoted by $\overset{c}{\Rightarrow}$ (or $\overset{s=t}{\Rightarrow}$, respectively,) or by $\Rightarrow$ simply.

4 Fixpoint Approach to the Semantics

In this section, we show that a computation of a multiset of goals G is modeled as interleaving computations of each goal in G and the model can be computed as the fixpoint of the semantic function defined here.

4.1 Atom Reaction

We are interested in reactive behaviors between a given initial goal and a (possibly altered) environment which may be implemented by other goals, rather than the fixed behavior and the final result induced from the initial goal and the initial environment. In such a case the environment (i.e., the other goals) may also be (monotonically) altered by reacting against the initial goal and/or its subgoals during the computation of the initial goal.

Here possible reactive behaviors of a initial goal corresponding to various environments are denotationally modeled by using sequences of substitutions.

Definition 4.1 (Unit Reaction)
A *unit reaction* is a substitution with an annotation '+' or '−', denoted by θ^+ or θ^- where θ is a substitution. θ^+ is called an *input unit reaction* and θ^- is called an *output unit reaction*. We may denote a unit reaction without an annotation when we do not need to distinguish *input* or *output* from each other. A substitution θ obtained from a unit reaction $\delta \equiv \theta^a$ by removing an annotation a is denoted by $|\delta|$. A set of all input unit reactions $\{\theta^+ \mid \theta \in Subst\}$ is denoted by $Ureact^+$, a set of all output unit reactions $\{\theta^- \mid \theta \in Subst\}$ is denoted by $Ureact^-$, and a set of all unit reaction $Ureact^+ \cup Ureact^-$ is denoted by $Ureact$.

Next we introduce special symbols, called *termination symbols*, which represent special states in reactive behaviors.

Definition 4.2 (Terminal Symbol)
A *termination symbol* is $\perp_{suc}, \perp_{rf}, \perp_{uf}$ or $\perp_{dl}$, (or simply by $\perp$), which represent *finite success*, *reduction failure*, *unification failure* and *deadlock* respectively. Then $Ureact^{\perp}$ denotes $Ureact \cup \{\perp_{suc}, \perp_{rf}, \perp_{uf}, \perp_{dl}\}$. $|\perp_{suc}| = |\perp_{rf}| = |\perp_{dl}| = \emptyset$, and $|\perp_{uf}| = \top$.

Now we define various operations on unit reactions by extending operations on the substitutions defined above.

Definition 4.3 (Operations on Unit Reactions)
Let σ be a substitution, δ be a unit reaction and a be an annotation of δ. Then *domain* and *range* of unit reaction are defined by $dom(\delta) \equiv dom(|\delta|)$ and $ran(\delta) \equiv ran(|\delta|)$. mgu and mgd of a substitution σ and a unit reaction δ are defined by $mgu(\sigma, \delta) \equiv mgu(\delta, \sigma) \equiv mgu(|\delta|, \sigma)^a$, $mgd(\delta, \sigma) \equiv mgd(|\delta|, \sigma)^a$ and $mgd(\sigma, \delta) \equiv mgd(\sigma, |\delta|)$. $\sigma \bowtie \delta$ and $\delta \bowtie \sigma$ iff $\sigma \bowtie |\delta|$ or $|\delta| \bowtie \sigma$.

For a unit reaction δ_i and a sequence of unit reactions Δ, δ_i is said to *be in* Δ iff $\exists \delta_i (1 \leq i \leq n)(\Delta = \delta_1 \delta_2 \ldots \delta_n)$, and denoted by $\delta_i \in \Delta$. An empty sequence of unit reactions is denoted by $\Box$.

Definition 4.4 (Reaction Sequence)
A *reaction sequence* is an empty sequence $\Box$, a sequence of one unit reaction δ, or a sequence of more than two unit reactions Δ such that $\forall \delta_i, \delta_j \in \Delta\ (1 \leq i < j \leq n \wedge dom(\delta_i) \cap dom(\delta_j) \equiv \emptyset \wedge dom(\delta_i) \cap ran(\delta_j) \equiv \emptyset)$. A set of all reaction sequences is denoted by $Rseq$.

A *domain* of $\Delta \in Rseq$ is a set of variables such that $\{\, V \mid \exists \delta \in \Delta(V \in dom(\delta)) \wedge \forall \delta' \in \Delta(V \notin ran(\delta'))\,\}$. $var(\Delta)$ also denotes the set of variables $\bigcup_{\delta \in \Delta} var(\delta)$. A substitution σ and a reaction sequence Δ are said to *be directed*, denoted by $\sigma \bowtie \Delta$; iff $var(\sigma) \cap var(\Delta) = dom(\sigma) \cap dom(\Delta)$. Reaction sequences Δ_1 and Δ_2 are said to *be directed*, also denoted by $\Delta_1 \bowtie \Delta_2$, iff $var(\Delta_1) \cap var(\Delta_2) = dom(\Delta_1) \cap dom(\Delta_2)$.

When $\Delta = \delta_1 \ldots \delta_n \in Rseq$ and $\delta \in Ureact$, a concatenation of Δ and δ is denoted by $\Delta \cdot \delta$ or $\delta \cdot \Delta$, defined by $\Delta \cdot \delta = \delta_1 \ldots \delta_n \delta$ or $\delta \cdot \Delta = \delta \delta_1 \ldots \delta_n$. A sequence of unit reaction $\Delta \cdot \delta$ such that $\Delta \in Rseq$ and $\delta \in Ureact^{\perp}$ is also a reaction sequence. A set of all such reaction sequences is denoted by $Rseq^{\perp}$.

Definition 4.7 (Atom Reaction)
An *atom reaction* is a pair of an atom $A \in Atom$ and a reaction sequence $\Delta \in Rseq^{\perp}$, denoted by (A, Δ), such that $dom(\Delta) \subseteq var(A)$. Here a set of all atom

reactions is denoted by $Areact$, i.e., $Areact = \{(A, \Delta) \mid A \in Atom \wedge \Delta \in Rseq^{\perp}\}$.

A substitution θ and an atom reaction (A, Δ) are also said to *be directed*, denoted by $\theta \bowtie (A, \Delta)$, iff $\theta \bowtie \Delta$. Atom reactions (A_1, Δ_1) and (A_2, Δ_2) are said to *be directed*, denoted by $(A_1, \Delta_1) \bowtie (A_2, \Delta_2)$ iff $\Delta_1 \bowtie \Delta_2$.

An equivalence class of E, i.e., $E_{\sim}$, may be represented by E, as mentioned in Section 2. When we say "E_1 and E_2 such that $E_1 \bowtie E_2$" where E_1, E_2 are substitutions, reaction sequences or atom reactions, we mean that each E_1 or E_2 is restricted to a subset of $E_{1\sim}$ or $E_{2\sim}$ such that $E_1 \in E_{1\sim}$, $E_2 \in E_{2\sim}$ and $E_1 \bowtie E_2$.

Let $(A, \delta_1 \dots \delta_n), (A, \delta'_1 \dots \delta'_n)$ be atom reactions that are directed. Then $(A, \delta_1 \dots \delta_n)$ is said to be *more general than* $(A, \delta'_1 \dots \delta'_n)$ when the following condition hold:

(1) if $\delta_n \in \perp$ or $\delta'_n \in \perp$, then $\delta_n = \delta'_n$, and
(2) for all i $(1 \leq i \leq n)$,
  (a) $\delta_i \in Ureact^+$ iff $\delta'_i \in Ureact^+$,
  (b) $\delta_i \in Ureact^-$ iff $\delta'_i \in Ureact^-$,
  (c) if $\delta_i, \delta'_i \in Ureact^+$, then $\Pi\delta_i \preceq \Pi\delta'_i$, and
  (d) if $\delta_i, \delta'_i \in Ureact^-$, then $\Pi\delta'_i \sim lub(\Pi\delta'_{i-1}, \Pi\delta_i)$,

where $\Pi\delta_i$ is a composition of substitutions $|\delta_1| \dots |\delta_i|$.

Here we want to explain intuitively what is the notion that (A, Δ) is more general than (A, Δ'). (A, Δ') represents a reactive behavior such that a goal A gets more instantiated bindings from, and posts not more instantiated bindings to the environment of the goal A, than the reactive behavior represented by (A, Δ).

Definition 4.6 (Atom on a Program)
Given a FGHC program P, an *atom on a program* P is an atom A such that the predicate symbol of A appears in P (not necessarily at head parts.) A set of all atoms on P is denoted by $Atom_P$, and a set of all atom reactions (A, Δ) such that $A \in Atom_P$ and $\Delta \in Rseq$ is denoted by $Areact_P$.

Now we define the relation between atom reactions and operational behaviors more formally.

Definition 4.7 (Correct Atom Reaction)
When a program P and an atom $G_0 \in Atom_P$ are given, an atom reaction $(G_0, \delta_1 \dots \delta_n)$ is called a *correct atom reaction w.r.t.* a program P, when the following conditions hold, where $B_0 = \{G_0\}$, $C_0 = \emptyset$ and $V_0 = var(G_0)$, and, for all $i (1 \leq i \leq n)$, let C_{δ_i} be a set of equations such that $mgu(C_{\delta_i}) \sim |\delta_i|$.

(1) If $n = 0$, i.e., $\delta_1\delta_2 \dots \delta_n = \square$, then $(G_0, \delta_1 \dots \delta_n)$ is always correct,
(2) if $\delta_i \in Ureact^+$, there exists a transition $\langle B_{i-1}, C_{i-1} \cup C_{\delta_i} : V_{i-1}\rangle \overset{c}{\Rightarrow} \langle B_i, C_i : V_{i-1} \cup var(c\eta)\rangle$ such that $var(C_{\delta_i}) \cap V_i \subseteq var(G_0\Theta)$ and $B_i = (B_{i-1} \setminus \{H\}) \cup B$, where $(H \text{ :- } G \mid B) \equiv c\eta$, $\eta \in Ren$ and $\Theta = mgu(C_{i-1})$,
(3) if $\delta_i \in Ureact^-$, there exists a transition $\langle B_{i-1}, C_{i-1} : V_{i-1}\rangle \overset{t=s}{\Rightarrow} \langle B_i, C_i : V_{i-1} \cup var(s=t)\rangle$ such that $C_i = C_{i-1} \cup C_{\delta_i}$ and $B_i = B_{i-1} \setminus \{t=s\}$, or
(4) if $\delta_n \in \{\perp\}$, at least one of the conditions (1)–(3) holds for all i $(1 \leq i \leq n-1)$, and
  (a) if δ_n is $\perp_{suc}$, then B_{n-1} is $\emptyset$,
  (b) if δ_n is $\perp_{rf}$, then there exists $A \in B_{n-1}$ such that $mgu(C_{n-1}) \not\preceq mgu(\{A = H\} \cup G)$ for all clauses such that $(H \text{ :- } G \mid B) \equiv c\eta, \eta \in Ren$ and $c \in P$,
  (c) if δ_n is $\perp_{uf}$, then there exists $(t=s) \in B_{n-1}$ such that $mgu(C_{n-1}) \not\preceq mgu(t, s))$, or
  (d) if δ_n is $\perp_{dl}$, then there exists the same transition as in case (2) unless the condition $var(C_{\delta_{n-1}}) \cap V_i \subseteq var(G_0\Theta)$ exists, where $\Theta = mgu(C_{i-1})$.

In the following we define the most important atom reaction in correct atom reactions.

Definition 4.8 (Most General Correct Atom Reaction)
Let (A, Δ) be a correct atom reaction w.r.t. a program P. Then, (A, Δ) is called a *most general correct atom reaction w.r.t.* a program P, denoted by $(A, \Delta) \rightsquigarrow_P$, when (A, Δ) is more general than any other correct atom reactions (A, Δ') w.r.t. P.

Example 4.1 Suppose that $(A, \delta_1\delta_2\delta_3) \rightsquigarrow_P$ such that $\delta_1, \delta_2 \in Ureact^+$ and $\delta_3 \in Ureact^-$. Intuitively we can explain the notion of a *correct atom reaction* by considering a chain of the following transitions:

$$\begin{array}{lll}
\langle G, \emptyset\rangle & & \\
\langle G, C_{\delta_1}\rangle \overset{c}{\Rightarrow} \langle B_1, C_1\rangle & & \\
& \langle B_1, C_1 \cup C_{\delta_2}\rangle \overset{c}{\Rightarrow} \langle B_2, C_2\rangle & \\
& & \langle B_2, C_2\rangle \overset{t=s}{\Rightarrow} \langle B_3, C_2 \cup C_{\delta_3}\rangle
\end{array}$$

where $G = \{A\}$ and $mgu(C_{\delta_i}) \sim |\delta_i|$ $(1 \leq i \leq 3)$.

Example 4.2 Let P be a program
$\{p(A,B,C) \text{:-} A = f(D), C = g(a,E) \mid B = f(a)\}$,
Δ_1 be $\{X \leftarrow f(U), Z \leftarrow g(a,V)\}^+\{Y \leftarrow f(a)\}^-$,
Δ_2 be $\{X \leftarrow f(U), Z \leftarrow g(a,V), Y \leftarrow f(a)\}^+$, and
Δ_3 be $\{X \leftarrow f(U)\}^+\{Z \leftarrow g(a,V), Y \leftarrow f(a)\}^-$.
Then the following two atom reactions $(p(X,Y,Z), \Delta_1)$ and $(p(X,Y,Z), \Delta_2)$ are correct atom reaction w.r.t. P. $(p(X,Y,Z), \Delta_1)$ is a most general correct reaction, i.e., $(p(X,Y,Z), \Delta_1) \rightsquigarrow_P$. But $(p(X,Y,Z), \Delta_3)$ is *not correct* because the configuration $\langle\{p(X,Y,Z)\}, \{X = f(U)\}\rangle$ can not be reduced to any configuration under the program P.

4.2 Fixpoint Semantics

In this section, we present the semantic function after defining some operations on reaction sequences and the semantic domain. Next we show the least fixpoint of the function gives the semantics of the programs in the same way as used in an ordinary fixpoint semantics theory.

Firstly, we define an application of a substitution to an atom reaction when they are directed. Let $\theta \in Subst_{/\sim}$ and $\Delta = \delta_1\delta_2 \dots \delta_n \in Rseq^{\perp}$ such that $\theta \bowtie \Delta$. Then *an application of* a substitution θ *to* a reaction sequence Δ, denoted by $\Delta\theta$, is a sequence of unit reactions $\delta'_1\delta'_2 \dots \delta'_n$ such that $\delta'_i = mgd(\sigma_i, \sigma_{i-1})$ for all i $(1 \leq i \leq n)$, where $\sigma_0 = \theta$ and $\sigma_i = lub(\sigma_{i-1}, \delta_i)$.

Example 4.3 Let θ be a substitution
$\{X \leftarrow M, Y \leftarrow M, Z \leftarrow g(N,b)\}$,
and $\delta_1\delta_2$ be the reactive sequence Δ_1 same as in Example 4.2. Then, θ and Δ_1 are directed because
$var(\theta) \cap var(\Delta_1) = dom(\theta) \cap dom(\Delta_1) = \{X, Y, Z\}$.
Let σ_1 be $lub(\theta, \delta_1)$, i.e.,
$\{ X \leftarrow f(U), Y \leftarrow f(U), Z \leftarrow g(a,b), V \leftarrow b,$
$M \leftarrow f(U), N \leftarrow a \}$,
and let σ_2 be $lub(\sigma_1, \delta_2)$, i.e.,
$\{ X \leftarrow f(a), Y \leftarrow f(a), Z \leftarrow g(a,b), V \leftarrow b, U \leftarrow a$
$M \leftarrow f(a), N \leftarrow a \}$,
Therefore, the application of θ to Δ_1 is
$\{M \leftarrow f(U), N \leftarrow a\}^+\{U \leftarrow a\}^-(= \delta_1'\delta_2')$, where $\delta_1' = mgd(\sigma_1, \theta)$ and $\delta_2' = mgd(\sigma_2, \sigma_1)$.

If $\delta_i \in Ureact^+$ and $\delta_i \not\preceq \sigma_{i-1}$ for some i, then such an application is not defined, that is, we can ignore the result and remove it from our system. Because, although such a reduction can *not* be done by *the* clause (corresponding to the input unit reaction δ_i), it may be done by *another alternative* clause. That is, it is not necessary that a *reduction failure* is immediately induced by this application. On the other hand, in the case that $\delta_i \in Ureact^-$ and $\delta_i \not\preceq \sigma_{i-1}$ for some i, $\Delta\theta$ is $\delta_1'\delta_2' \ldots \delta_{i-1}\perp_{uf}$. This is because such an application induces a *unification failure* immediately.

Definition 4.9 (Application to Atom Reaction)
Let (A, Δ) be an atom reaction. Then *an application of* a substitution θ *to* an atom reaction (A, Δ) is $(A\theta, \Delta\theta)$, which is also an atom reaction.

Example 4.4 Let P be the same program and $(p(X,Y,Z), \Delta_1)$ be the same atom reaction as in Example 4.2, and let θ be the same substitution as in Example 4.3. Then, the application of θ to $(p(X,Y,Z), \Delta_1)$, i.e., $(p(X,Y,Z)\theta,\ \Delta_1\theta)$, is
$(p(M,M,g(N,b)),\ \{M \leftarrow f(U), N \leftarrow a\}^+\{U \leftarrow a\}^-)$.
Now the application of θ to it, $(p(X,Y,Z)\theta, \Delta_1\theta)$, intuitively represents a reactive behavior of a goal $p(M,M,g(N,b))$ under the program P. In fact, the atom reactions $(p(X,Y,Z), \Delta_1)$ and $(p(X,Y,Z)\theta, \Delta_1\theta)$ is both correct w.r.t. P.

Next we define possible interleavings of reaction sequences.

Definition 4.10 (Interleaving)
Possible interleavings of a set of reaction sequences $\{\Delta_1, \ldots, \Delta_n\}$ *on* a set of variables V, denoted by $int(\Delta_1, \ldots, \Delta_n)_{|V}$, is a set of all reaction sequences $\delta \cdot \Delta$ defined inductively as follows, where Δ_i is $\delta_i \cdot \Delta_i'$ for each i $(1 \leq i \leq n)$ such that Δ_i is not $\Box$:

(1) if $\delta_i \in Ureact^+$ and $V \subset dom(\delta_i)$ for all i $(1 \leq i \leq n)$, then $\Delta = \Box$ and $\delta = \perp_{dl}$, or
(2) otherwise, for some i $(1 \leq i \leq n)$,
  (a) if $\Delta_i = \Box$,
    then $\delta \cdot \Delta = int(\Delta_1, \ldots, \Delta_{i-1}, \Delta_{i+1}, \ldots, \Delta_n)_{|V}$,
  (b) if $\delta_i \in \{\perp_{rf}, \perp_{uf}, \perp_{dl}\}$, then $\Delta = \Box$ and $\delta = \delta_i$,
  (c) if $\Delta_j = \Delta_j'' \cdot \perp_{suc}$ for all j $(1 \leq j \leq n)$, then
    $\Delta = int(\Delta_1''\delta_i, \ldots, \Delta_i', \ldots, \Delta_n''\delta_i)_{|var(V|\delta_i|)} \cdot \perp_{suc}$
    and $\delta = \delta_i$, or
  (d) otherwise,
    $\Delta = int(\Delta_1\delta_i, \ldots, \Delta_i', \ldots, \Delta_n\delta_i)_{|var((V)|\delta_i|)}$ and
    $\delta = \delta_i$.

Definition 4.11 (Semantic Function)
Given a program P, we denote a power set of $Areact_P$ by Den_P, and let it be a *domain of* the following semantic function. Given a program P and a goal G_0, we define a semantic function $T_{P,G_0} : Den_P \rightarrow Den_P$ as follows:

$$
\begin{aligned}
&T_{P,G_0}(I) \ = \\
&\{(G_0, \Box)\} \cup \\
&\{(s{=}t, \theta^-) \mid (s{=}t, \Box) \in I \wedge \theta = mgu(s,t)\} \cup \\
&\{(B_i\theta_g, \Box) \mid \exists (A, \Box) \in I \wedge \exists (H :- G \mid B) \in P\ (B_i \in B)\} \cup \\
&\{(A, \theta_g^+\Delta) \mid (A, \Box) \in I \wedge \\
&\qquad \exists (H :- G \mid B) \in P\ \forall B_i, B_j \in B \\
&\qquad \exists (B_i\theta_g, \Delta_i), (B_j\theta_g, \Delta_j) \in I \\
&\qquad\quad ((B_i\theta_g, \Delta_i) \bowtie (B_j\theta_g, \Delta_j) \wedge \\
&\qquad\quad \Delta \in int(\Delta_1, \Delta_2, \ldots, \Delta_n)_{|var(A)})\} \cup \\
&\{(A, \theta^! \perp_{rf}) \mid (A, \Box) \in I \wedge \forall (H :- G \mid B) \in P\ (\theta \in \overline{\theta}_g)\}
\end{aligned}
$$

where $\theta_g = mgu(\{A{=}H\} \cup G)$ and $\overline{\theta}_g = \{\theta \mid \theta \not\preceq \theta_g\}$

The set Den_P forms a complete lattice under the ordering of set inclusion $\subseteq$ with a bottom element $\emptyset$ and a top element $Areact_P$.

The $T_{P,G_0}(I)$ is recursively defined by using I as the union of four sets of atom reactions each of which represents the following situation:

(1) when a unification goal $s{=}t$ is called, the binding $mgu(s,t)$ is posted to the environment,
(2) when a goal A exists, each goal $B_i\theta_g$ is generated as a sub-goal of A and may invoke the new process,
(3) and a goal A affects the environment as θ_g^+ followed by a sequence of reactive behaviors represented by Δ which is obtained from interleaving reactive behaviors generated by all sub-goals of A, that is, A may perform the computation represented by Δ after A gets the binding θ_g, or
(4) when A meets with the binding θ incompatible with all bindings to solve the guard $\{A{=}H\} \cup G$ for all clauses, A will suspend. This situation is called a *reduction failure*.

Lemma 4.1 *Let P be an FGHC program and G be a goal. The function $T_{P,G}$ is continuous, i.e., $T_{P,G}(lub(X)) = lub(T_{P,G}(X))$ for any directed subset X of Den_P*

Proof: It is proved in a similar way to the proof of continuity of the semantic function of a standard logic program. (See pp.37-38 [Lloyd 1987].) ∎

From Lemma 4.1, $T_{P,G}$ has the least fixpoint, $lfp(T_{P,G})$, and $lfp(T_{P,G}) = glb\{X \mid T_{P,G}(X) \subseteq X\}$. Furthermore, $lfp(T_{P,G}) = T_{P,G}\uparrow\omega$.

Definition 4.12 (Topdown Semantics)
Let P be a FGHC program and G be a goal. Then $lfp(T_{P,G})$ is called a *topdown semantics* of P with G, and denoted by $[\![P]\!]_G$.

5 Relation between Operational Semantics and Fixpoint Semantics

In this section, we show that the topdown semantics defined in Section 4 is closely related to the operational semantics of FGHC introduced in Section 3.

Theorem 5.1 (Soundness) *Let P be an FGHC program and G_0 be a goal. If $(G_0, \Delta) \leadsto_P$, then $(G_0, \Delta) \in [\![P]\!]_{G_0}$.*

Proof (A Sketch of the Proof):
Let k be a length of Δ, denoted by $|\Delta|$. The proof is by induction on the length k.

If $k = 0$, then the theorem is trivial since $(G, \Box)$ is always correct.

Otherwise, i.e., $k > 0$, suppose $\Delta = \Delta' \cdot \delta$ such that $|\Delta'| =\geq 0$.

If G_0 is a unification goal, then the theorem is trivial.

Otherwise, G_0 is a non-unification goal. Now, since $(G_0, \Delta) \leadsto_P$ hold, $(G_0, \Delta') \leadsto_P$ holds. By the induction hypothesis, since $|\Delta'| < k$ and $(G_0, \Delta') \leadsto_P$, $(G_0, \Delta') \in [\![P]\!]_{G_0}$.

Hence, from the definition of T_{P,G_0}, $\exists \theta_g^+ (\Delta' = \theta_g^+ \cdot \Delta'')$ such that $(H :- G \mid B) \in P$ and $\theta_g = mgu(\{A = H\} \cup G)$ and $\forall B_i \in B \exists (B_i \theta_g, \Delta_i) \in [\![P]\!]_{G_0}$ and $\Delta'' \in int(\Delta_1, \ldots, \Delta_n)_{|var(G_0)}$. Now we have $(G_0, \theta_g^+ \cdot \Delta'') \leadsto_P$ and $\Delta'' \in int(\Delta_1, \ldots, \Delta_n)_{|var(G_0)}$. Then we can get Δ_i such that $(B_i \theta_g, \Delta_i) \leadsto_P$ by selecting a unit reaction from the only i-th argument (i.e., Δ_i) in the definition of int.

Suppose that the last transition of $(G_0, \Delta' \cdot \delta) \leadsto_P$ is a transition on a sub-goal of $B_i \theta_g$. Then $(B_i \theta_g, \Delta_i \delta) \leadsto_P$.

Since $k > |\Delta'| > |\Delta''| \geq |\Delta_i|$, $k > |\Delta'| \geq |\Delta_i \delta|$.

By induction hypothesis again, since $(B_i \theta_g, \Delta_i \delta) \leadsto_P$, $(B_i \theta_g, \Delta_i \delta) \in [\![P]\!]_{G_0}$.

Therefore, from the definition of T_{P,G_0}, since $\Delta' \delta = int(\Delta_1, \ldots, \Delta_i, \ldots, \Delta_n)$, $(G_0, \Delta' \delta) \in [\![P]\!]_{G_0}$. ∎

In Theorem 5.1 we show that any most general correct atom reaction (G_0, Δ) w.r.t. a program P is in the topdown semantics $[\![P]\!]_{G_0}$. In general it is necessary to prove the only-if part of the theorem (usually called *Completeness Theorem*), and we think this is possible by introducing a kind of downward closure of (A, Δ) by using the *'more general than' relation* in Section 4.1, as ***subsumption relation*** in [Falaschi *et al.* 1990]. This, however, is beyond the scope of this paper. Because Theorem 5.1 is sufficient to guarantee the correctness of the framework of abstract interpretation based on the topdown semantics since we want to use this semantics as a collecting semantics.

6 General Framework for Abstract Interpretation

In this section we briefly review a general framework of abstract interpretation for programs whose semantics can be defined from a fixpoint approach, and some conditions to guarantee that the abstract interpretation is 'safe' for the semantics.

When a standard semantics is given by the least fixpoint of some semantic function, an abstract semantics is given by another semantic function obtained by directly abstracting the concrete semantic function such that the *safe* relation exists between their two semantics.

6.1 Concrete Fixpoint Semantics

Suppose that the meaning of a program P is given by the least fixpoint of a *(concrete) semantic function T_P*, denoted by $lfp(T_P)$, where $T_P : Den \to Den$ is a continuous function and Den is a powerset of D, called a ***concrete domain***, such that each element of D expresses a concrete computation state of the program. For example, in an ordinary logic program, is an Herbrand Base. And Den forms a complete lattice with regard to the set inclusion ordering $\subseteq$ on Den. Then, the least fixpoint of T_P exists and we can get it by $lfp(T_P) = T_P \uparrow \omega$.

Definition 6.1 (Concrete Semantics)
$[\![P]\!] = lfp(T_P)$ is called the *least fixpoint semantics* of a program P. Especially, we call it the *concrete semantics* of a program P since the semantics is obtained from the concrete semantic function T_P

6.2 Abstract Fixpoint Semantics

We define an abstract fixpoint semantics by abstracting the concrete domain and the concrete semantic function introduced in 6.1.

Definition 6.2 (Abstract Domain)
Given a concrete domain D, an ***abstract domain*** $\underline{D}$ is a finite set of denotations satisfying the following conditions:

(1) every element of $\underline{D}$ represents a subset of D,
(2) $\underline{D}$ forms a complete lattice with respect to an order relation $\sqsubseteq$ defined on $\underline{D}$, and
(3) there exist two monotonic mappings, that is, ***abstraction*** $\alpha : D \to \underline{D}$ and ***concretization*** $\gamma : \underline{D} \to D$ defined as follows: $\forall \underline{d} \in \underline{D}\ (\underline{d} = \alpha(\gamma(\underline{d}))) \ \wedge\ \forall d \in D\ (d \subseteq \gamma(\alpha(d)))$

In order to define the abstract semantics of a program P, we should define (or design) a monotonic and continuous mapping of a program P; $\underline{T}_P : \underline{Den} \to \underline{Den}$, called the ***abstract semantic function***, as well as the abstract domain $\underline{D}$, corresponding to the concrete domain D and the concrete semantic function T_P of P. Then we have to define the abstract versions of various operations, e.g., a composition or an application of substitutions, used in the definition of T_P.

Definition 6.3 (Abstract Semantics)
Then the least fixpoint semantics $\underline{[\![P]\!]} = lfp(\underline{T}_P)$, obtained from the abstract semantic function $\underline{T}_P$, is called the *abstract semantics* of a program P.

Now we claim the termination property with respect to the abstract fixpoint semantics.

Lemma 6.1 *There exists the least fixpoint $lfp(\underline{T}_P)$ of $\underline{T}_P$ such that $lfp(\underline{T}_P) = \underline{T}_P \uparrow k$ for some finite k*

Lastly, we attach the following acceptable relation between the abstract semantics and the concrete semantics:

Definition 6.5 (Safeness Condition)
A *safeness condition for the abstract semantics* is as follows: $[\![P]\!] \subseteq \gamma(\underline{[\![P]\!]})$.

Lemma 6.2 *If $T_P(\gamma(\underline{d})) \subseteq \gamma(\underline{T}_P(\underline{d}))$ for all $\underline{d} \in \underline{D}$, then the abstract semantics is safe, i.e., a safeness condition holds, where $T_P(\gamma(\underline{d})) = \{T_P(d) \mid d \in \gamma(\underline{d})\}$.*

7 Applications for Analysis of FGHC Programs

In this section we show some examples of analyzing FGHC programs by using abstract interpretation based on the topdown semantics in Section 4, which is an instance of the general framework in Section 6

7.1 Moded Type Graph

The abstract domain presented here is so similar to the one based on type graphs in [Bruynooghe and Janssens 1988], that most necessary operations on the abstract domain will be well-defined similarly to [Bruynooghe and Janssens 1988][Janssens and Bruynooghe 1989].

Here we introduce a moded type graph, and show briefly that a reaction sequence and an atom reaction can be abstracted by a moded type graph.

Definition 7.1 (Moded Type Constructor and Generic Types)
A(n n-ary) *moded type constructor* is a(n n-ary) function symbol $f/n \in Func$ with a mode annotation $+$ (or $-$), denoted by $f^+_{/n}$ (or $f^-_{/n}$) or simply f^+ (or f^-), which represents a(n n-ary) function symbol f appearing in input (or output) unit reactions (respectively). Four *generic (moded) types* are an *any type*, a *variable type*, an *undefined type* and an *empty type*, denoted by $\underline{any}$, $\underline{V}$, $\_$ and $\emptyset$ respectively. An *any type* represents the set of all moded terms, both $\underline{V}$ and $\_$ represent the set of variables, and $\emptyset$ represents the empty set of terms.

Definition 7.2 (Moded Term and Moded Type)
A *moded term* is a term constructed from moded type constructors over a set of variables Var. A moded term represents the same term without all mode annotations such that a moded type constructor with $+$ (or $-$) corresponds to a function symbol appearing in an input (or an output) unit reaction. A *moded type* is a set of moded terms.

Definition 7.3 (Moded Type Graph)
A *moded type graph* is a representation of a moded type, which is a directed graph such that each node is labeled with either a moded type constructor, a generic type, or a special label 'or'.

The relation between a parent node and (possibly no) child nodes in a moded type graph $\mathcal{G}$ is defined as follows:

(1) a node labeled with $f^+_{/n}$ or $f^-_{/n} (n \geq 0)$ has n *ordered* arcs to n nodes, i.e., has n ordered child nodes,
(2) a node labeled with 'or' has n *non-ordered* arcs to n nodes ($n \geq 2$), i.e., has n non-ordered child nodes,
(3) a node labeled with a generic type has no child node,
(4) there exists at least one node, called a *root node*, such that there are paths from the root node to any other nodes in $\mathcal{G}$, and
(5) the number of occurrences of nodes with the same label on each path from the root node of $\mathcal{G}$ is bounded by a constant d, called a *moded type depth*.

Suppose that a node N tries to be newly added as a child node of N_p in a moded type graph $\mathcal{G}$. Then, if the creation of the node N violates the condition (5) in the above definition, that is, if there exist more than d numbers of nodes with the same label as N on the path from the root node to N, then the new node N will not be added to $\mathcal{G}$ as a new child node of N_p but will be shared with the farthest ancestor node of N_p with the same label as N. In such a case, a circular path must be created. (Nodes with the same label aren't shared with each other when their nodes are on different paths from root.) The restriction of (5) is the same as the *depth restriction* in [Janssens and Bruynooghe 1989]. They call a type graph satisfying the depth restriction a *restricted type graph*, and they have presented an algorithm for transforming a non-restricted type graph to restricted one.

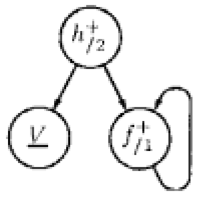
A concretization for a moded type graph with a root node N_0, denoted by $\gamma(N_0)$, is defined as follows:

(1) $\gamma(N)$ is Var if the label of N is $\underline{V}$ or $\_$,
(2) $\gamma(N)$ is the empty set $\emptyset$ if the label of N is $\emptyset$,
(3) $\gamma(N)$ is $\{f^+(t_1, \ldots, t_n) \mid t_i \in \gamma(N_i) \wedge 0 \leq i \leq n\}$ if the label of N is $f^+_{/n}$ and $N_1, \ldots, N_n$ are child nodes of N,
(4) $\gamma(N)$ is $\{f^-(t_1, \ldots, t_n) \mid t_i \in \gamma(N_i) \wedge 0 \leq i \leq n\}$ if the label of N is $f^-_{/n}$ and $N_1, \ldots, N_n$ are child nodes of N, or
(5) $\gamma(N)$ is $\gamma(N_1) \cup \ldots \cup \gamma(N_n)$ if the label of N is 'or' and $N_1, \ldots, N_n$ are child nodes of N.

A moded type graph represents a set of moded term, i.e., a moded type, defined by γ. A set of all moded types is denoted by $\underline{Term}$.

A moded type graph $\mathcal{G}$ can be also represented by an expression, called a *moded type definition*, like a context free grammar with (possibly no) non-terminal symbols, called *type variables*, and one start symbol, called a *root* *type variable*, corresponding to the root node of $\mathcal{G}$, as in [Janssens and Bruynooghe 1989]. A moded type or a mode type graph represented by a moded type definition may be referred to the root type variable.

Example 7.1 The following graph $\mathcal{G}$ is a moded type graph whose root node is labeled with $h^{+}_{/2}$:



Then the moded type graph may also be denoted by the following moded type definition:

$\tau = h^{+}(\underline{V}, \tau_1)$,
$\tau_1 = f^{+}(\tau_1)$,

where τ, τ_1 is type variable and τ is a root type variable.

This moded type definition represents a set of moded terms:

$\gamma(\tau) = \{\, h^{+}(V_1, f^{+}(V_2)),\ h^{+}(V_1, f^{+}(f^{+}(V_2))), \dots \}$

An abstraction α for a moded term satisfying the condition (3) in Definition 6.2 is also well-defined in a similar way to [Janssens and Bruynooghe 1989].

A *moded type substitution* $\underline{\theta}$ is a mapping Var to $\underline{Term}$, and is also represented by a set of assignments of variables to moded types. A concretization and an abstraction for a moded type substitution is defined:

$$\gamma(\underline{\theta}) = \{\theta \mid \forall X \in dom(\underline{\theta})\, (t \in \gamma(X\underline{\theta}) \supset (X \leftarrow t) \in \theta)$$
$$\alpha(\theta) = \{X \leftarrow \alpha(X\theta) \mid X \in dom(\theta)\}$$

And an ordering relation $\sqsubseteq$ over moded type substitutions is defined as follows: $\underline{\theta}_1 \sqsubseteq \underline{\theta}_2$ iff $\gamma(V\underline{\theta}_1) \subseteq \gamma(V\underline{\theta}_2)$ for all variables $V \in Var$.

A *moded type reaction sequence* $\underline{\Delta}$ is a sequence of moded type substitutions $\underline{\delta}_1\underline{\delta}_2 \dots \underline{\delta}_n$ such that
$\forall i, j (1 \leq i < j \leq n)(\underline{\delta}_i \sqsubseteq \underline{\delta}_j \wedge dom(\underline{\delta}_i) = dom(\underline{\delta}_j))$,
and $dom(\underline{\Delta})$ is defined $dom(\underline{\delta}_i)$. A concretization for a moded type reaction sequence, denoted by $\gamma(\underline{\delta}_1 \dots \underline{\delta}_n)$, is defined as follows:

$$\{\delta_1 \dots \delta_n \mid \delta_1 \dots \delta_n \in Rseq \wedge \Pi\delta_i \in \gamma(\delta_i)\},$$

where $\Pi\delta_i$ is a composition of substitutions $|\delta_1| \dots |\delta_i|$.

And an instantiation ordering $\sqsubseteq$ on a moded type reaction sequence is defined: $\underline{\Delta}_1 \sqsubseteq \underline{\Delta}_2$ iff $\gamma(\underline{\Delta}_1) \subseteq \gamma(\underline{\Delta}_2)$.

Definition 7.4 (Moded Type Atom Reaction)
A *moded type atom reaction* is a pair of an atom A and moded type reaction sequence $\underline{\Delta}$ such that $dom(\underline{\Delta}) \subseteq var(A)$. $\underline{Areact}$ is a set of all moded type atom reactions.

Example 7.2 Let Δ be a reaction sequence $\{X \leftarrow f(Y)\}^{+}\{Y \leftarrow g(Z)\}^{-}$. Then $\alpha(\{X \leftarrow f(Y)\}^{+}\{Y \leftarrow g(Z)\}^{-})$ is $\{X \leftarrow \tau_1\}\{X \leftarrow \tau_2\}$, where τ_1 and τ_2 is defined by the following type definitions:

$\tau_1 = f^{+}(\underline{V})$,
$\tau_2 = f^{+}(g^{-}(\underline{V}))$.

An application of a moded type substitution $\underline{\delta}$ to a moded type reaction sequence $\underline{\delta}_1 \dots \underline{\delta}_n$ is a moded type reaction sequence $\delta'_1 \dots \delta'_n$ such that $\delta'_i = lub(\delta_{i-1}, \delta_i)$ for all i $(0 \leq i \leq n)$ where $\delta_0 = \delta$.

A possible interleaving of moded type reaction sequences $\underline{int}$ can be well-defined by using the definition of possible interleaving on a concrete domain in Section 4.2. And $\underline{Den}$ is a power set of $\underline{Areact}$.

Now we can define the abstract semantic function $\underline{T}_{P,G}: \underline{Den} \rightarrow \underline{Den}$ for a program P and a goal G by using abstract operations and denotations defined above.

7.2 An Example of Detecting Multiple Writers

Consider that two goals try to instantiate a shared variable to a (possibly different) symbol(s). In such a case, the goals may cause inconsistent assignments to the same variable, which are called *multiple writers*. Recently, in the family of concurrent logic languages, several languages have been proposed that do not allow multiple writers, and many advantages have been discussed [Saraswat 1990][Ueda 1990a][Kleinman *et al.* 1991][Foster and Taylor 1989]. For examples, *moded FGHC* presented in [Ueda 1990a] has the following advantages: (1) an efficient implementation based on a message-oriented technique, (2) unification failure free, and (3) easy mode analysis. So moded FGHC seems to lead FGHC programmers into a good style of FGHC programming.

Although you can write most programs without using multiple writers, you may want to use them in *a few* cases. *Stop signal* may be one of these examples.

Stop signal is a programming technique such that, when some goal find the answer to a searching problem, the goal broadcasts a *stop signal* to any other goals which are solving the same searching problem (or its sub-problems) and the goal forces any other goals to terminate their process by instantiating a flag symbol to a variable shared by all goals. Several flaggings may occur on different goals at the same time, or some goal may broadcast a flag at any stage if a flag is not received but has been sent from other goals. In such cases, multiple writing problems may occur.

Now we show a method of detecting multiple writers as an application of the moded type inference in the previous section. The following program implements a very simple example of 'stop signal'. A subscript number of each function symbol is used to distinguish occurrences.

```
main(T,F) :- true | generate(T),search(T, F).
search(t1(_,a1,_),F) :- true | F=f1.
search(_,f2) :- true | true.
search(t2(L,b1,R),F) :- true |
                     search(L,F),search(R,F).
generate(T) :- true | T=t3(L,N,R),genNode(N),
                     generate(L),generate(R).
genNode(N) :- true | N=a2.
genNode(N) :- true | N=b2.
```

A goal `generate(T)` generates a binary tree with each node labeled with `a` or `b`, and a goal `search(T,F)` searches a node labeled with `a`. Body goals of `search/2` share the second argument as a 'stop signal'. Now we try to analyze the moded type of a goal `main(T,F)` by computing $\underline{[P]}_{\mathtt{main(T,F)}}$ on the abstract domain for the moded type. Here each moded type constructor has a subscript number. When we apply $\underline{\theta} = \{\mathtt{X} \leftarrow \mathtt{a}_1^+\}$ to $\underline{\delta} = \{\mathtt{X} \leftarrow \mathtt{a}_2^-\}$, we can get a moded type substitution $(\theta)\underline{\delta} = \{\mathtt{X} \leftarrow \mathtt{a}_{21}^-\}$. This represents a moded type $\{\mathtt{X} \leftarrow \mathtt{a}^-\}$ by engaging $\mathtt{a}_2$ to $\mathtt{a}_1$. When goals try to engage a moded type constructor with $-$ to a moded type constructor with $-$, the goals are multiple writers.

In the above program, we can compute the following moded type atom reaction in $\underline{[P]}_{\mathtt{main(T,F)}}$:

$(\mathtt{main(T,F)}, \ldots \{\mathtt{F} \leftarrow \mathtt{f}_{11}^-\} \ldots)$.

Then we can get information such that

(1) the goal `main(T,F)` may cause multiple writes, and
(2) the problematic goal is a unification goal writing $\mathtt{f}_1$, i.e. in the body of the first clause of `search`.

8 Discussions

Much research has been presented on the fixpoint approaches to the semantics for concurrent logic languages.

Atom reactions are essentially the same as *reactive clauses* introduced in *reactive behavior semantics* [Gaifman *et al.* 1989]. Since reactive behavior semantics is defined by the self-unfolding of reactive clauses, we cannot always define some reasonable abstraction of the semantics when the semantics is applied to abstract interpretation. That is, the same non-terminating problem may occur as in the example below. While using our semantics, since we define by computing all possible reaction sequences corresponding to atoms in a body at one time by *int*, such a problem does not occur.

Our semantics distinguishes *reduction failure* from *deadlock* as well as *unification failure*, although the operational semantics of FGHC say nothing w.r.t. reduction failure, that is, reduction failure is regarded as *suspension*. Then the case that a goal is reduced by no clause is distinguished from *failure* (*unification failure*), but not distinguished from deadlock. But we introduce reduction failure as a termination symbol. In a practical system of FGHC, reduction failure may be reported as a *system service* to users if the system *fortunately* detects it at run time. It is helpful to users if reduction failure can be detected since such failure causes deadlock. So, we will want to detect the possibility of reduction failure at analysis time too. This is why we must introduce reduction failure to the semantics.

In [de Boer *et.al.* 1989], they have presented a denotational and a fixpoint approach to the semantics for (non-flat) GHC. They have presented the declarative semantics based on a fixpoint approach over the semantic domain similar to our atom reaction. They have mentioned that the fixpoint semantics is sound and complete w.r.t. the operational semantics giving only the results of finite success computations. Whereas, since our approach keeps more information by using the complement of all correct input unit reactions and $\perp_{rf}$, it can be *correctly* related to the operational semantics including the cases of deadlock and finite failure.

A few works on abstract interpretation for concurrent logic programs have been presented. The approaches of [Codognet *et al.* 1990] and [Codish *et al.* 1991] are based on the operational semantics.

In [Codognet *et al.* 1990], they have presented a meta-algorithm for FCP(:) and an abstracted version of it. They also show the correctness relation of the algorithm to the operational semantics, which is defined by a transition system similar to this paper.

In [Codish *et al.* 1991], they directly abstracted a standard transition system semantics, where a set of configurations is approximated to an abstract configuration. One of the advantages of their approach is that the analysis is simple and easy to prove correct.

These two are essentially the same approaches and it is easy to understand the correspondence to the operational semantics in both approaches.

In the approach of [Codish *et al.* 1991], the termination of abstract interpretation may not be guaranteed for some programs such that a goal may infinitely generate more and more sub-goals. For example, the following program is taken from [Codish *et al.* 1991]. They must abstract the domain (i.e., configuration) too much (called *star abstraction*) in order to solve such a problem. The star abstraction is enough and not too abstract to analyze suspension. But it may not be suitable to call and/or success pattern analysis. These problems may be solved by adopting some abstraction on goals other than the star abstraction [Codish 1992].

```
producer(X) :- true |
      X=f(X1,X2), producer(X1), producer(X2).
consumer(X) :- X=f(X1,X2) |
      consumer(X1), consumer(X2).
```

But our abstract interpretation can analyze call pattern of the program, and return the following moded type atom reaction when the moded type depth is 1:

$(\mathtt{producer(X)},\ \{\mathtt{X} \leftarrow \tau_1\}\{\mathtt{X} \leftarrow \tau_2\}\{\mathtt{X} \leftarrow \tau_3\}\ )$

$\tau_1 = \mathtt{f}^+(\_,\_)$
$\tau_2 = \mathtt{f}^+(\tau_3,\_)$
$\tau_3 = \mathtt{f}^+(\tau_3,\tau_3)$

Although our *possible interleavings* may be a little difficult to define and understand, these problems can be solved by the abstraction only on the domain, i.e., reaction sequences.

9 Conclusions

We have presented a denotational semantics for FGHC which is less abstract semantics and is suitable as a basis for abstract interpretation. Since the semantics is defined by a fixpoint approach on atom reactions which represent the reactive behaviors of atoms, we can easily develop a program analysis system only to abstract a (possibly infinite) domain to a finite domain. We have also demonstrated moded type inference of FGHC programs.

Acknowledgments

I thank Kazunori Ueda and Michael Codish for valuable comments and suggestions and the referees for their helpful comments.

References

[Bruynooghe and Janssens 1988] Bruynooghe, M. and G. Janssens, "An Instance of Abstract Interpretation Integrating Type and Mode Inferencing", Proc. of the 5th International Conference and Symposium on Logic Programming, R. A. Kowalski and K. A. Bowen (eds.), pp.669–683, 1988.

[Codish and Gallagher 1989] Codish, M., J. Gallagher, "A Semantic Basis for the Abstract Interpretation of Concurrent Logic Programs", Technical Report CS89-26, November, 1989.

[Codish *et al.* 1991] Codish, M., M. Falaschi and K. Marriott, "Suspension Analysis for Concurrent Logic Programs", Proc. of the 8th International Conference on Logic Programming, Furukawa, K. (ed.), pp.331–345, 1991

[Codish 1992] Codish, M., *personal communication*, Feb, 1992

[Codognet *et al.* 1990] Codognet, C., P. Codognet and M. M. Corsini, "Abstract Interpretation for Concurrent Logic Languages", Proc. of the North American Conference on Logic Programming, S. Debray and M. Hermenegildo (eds.), pp.215–232, 1990.

[de Boer *et.al.* 1989] de Boer, F. S., J. N. Kok and C. Palamidessi, "Control Flow versus Logic: a denotational and declarative model for Guarded Horn Clauses", Proc. of Mathematical Foundations of Computer Science, A. Kreczmar and G. Mirkowska (eds.), pp.165–176, LNCS-379, Springer-Verlag, 1989.

[de Boer and Palamidessi 1990] de Boer, F. S., and C. Palamidessi, "Concurrent Logic Programming: Asynchronism and Language Comparison", Proc. of the North American Conference on Logic Programming, S. Debray and M. Hermenegildo (eds.), pp.175–194, 1990.

[Falaschi *et al.* 1989] Falaschi, M., G. Levi, M. Martelli, C. Palamidessi, "A Model-theoretic Reconstruction of the Operational Semantics of Logic Programs", Università di Pisa, Technical Report TR-32/89, 1989.

[Falaschi *et al.* 1990] Falaschi, M., M. Gabbrielli, G. Levi and M. Murakami, "Nested Guarded Horn Clauses", International Journal of Foundations of Computer Science, Vol.1, no. 3, pp.249–263, 1990.

[Foster and Taylor 1989] Foster, I. and S. Taylor, "Strand: A Practical Parallel Programming Tool", Proc. of the North American Conference on Logic Programming, E. L. Lusk and R. A. Overbeek (eds.), pp.497–512, 1989.

[Gabbrielli and Levi 1990] Gabbrielli, M. and G. Levi, "Unfolding and Fixpoint Semantics for Concurrent Constraint Logic Programs", Proc. of the 2nd International Conference on Algebraic and Logic Programs, LNCS, Nancy, France, 1990.

[Gaifman *et al.* 1989] Gaifman, H., M. J. Maher and E. Shapiro, "Reactive Behavior Semantics for Concurrent Constraint Logic Programs", Proc. of the North American Conference on Logic Programming, E. L. Lusk and R. A. Overbeek (eds.), pp.551–569, 1989.

[Gerth *et al.* 1988] Gerth, R., M. Codish, Y. Lichtenstein and E. Shapiro "Fully Abstract Denotational Semantics for Concurrent Prolog", Proc. of 3rd Annual Conference on Logic in Computer Science, IEEE, pp.320–335, 1988.

[Janssens and Bruynooghe 1989] Janssens, G. and M. Bruynooghe, "An Application of Abstract Interpretation: Integrated Type and Mode Inferencing", Report CW86, Katholieke Universiteit Leuven, April, 1989.

[Kleinman *et al.* 1991] Kleinman, A., Y. Moscowitz, A. Pnueli and E. Shapiro, "Communication with Directed Logic Variables", Proc. of the 8th Annual ACM Symposium on Principles of Programming Languages, pp.221–232, 1991.

[Lassez *et al.* 1987] Lassez, J. L., M. J. Maher, and K. Marriott, "Unification Revised", Foundations of Deductive Databases and Logic Programming, Minker, J. (ed.), Morgan Kaufmann, pp. 587–625, 1987.

[Levi 1988] Levi, G., "A New Declarative Semantics of Flat Guarded Horn Clauses", Technical Report, ICOT, 1988.

[Lloyd 1987] Lloyd, J.W., "Foundation of Logic Programming", Second, Extended Edition, Springer-Verlag, 1087.

[Murakami 1988] Murakami, M., "A Declarative Semantics of Parallel Logic Programs with Perpetual Processes", Proc. of the International Conference on FGCS'88, pp.374–388, Tokyo, 1988.

[Palamidessi 1990] Palamidessi, C., "Algebraic Properties of Idempotent Substitutions", Proc. of the 17th ICALP, pp 386–399, 1990.

[Saraswat 1990] Saraswat, V. A., K. Kahn and J. Levy, "Janus: A Step Towards Distributed Constraint Programming", Proc. of the North American Conference on Logic Programming, S. Debray and M. Hermenegildo (eds.), 1990.

[Ueda 1990a] Ueda, K. and M. Morita, "New Implementation Technique for Flat GHC", Proc. of the 7th International Conference on Logic Programming, D. H. D. Warren and P. Szeredi (eds.), pp.3–17, 1990

[Ueda 1990b] Ueda, K., "Designing a Concurrent Programming Language", Proc. of an International Conference organized by the IPSJ to Commemorate the 30th Anniversary: InfoJapan'90, pp.87–94, Tokyo, 1990.