Andrei Mantsivoda

THE SEMANTICS OF FLANG

1. Introduction

In this paper we present denotational and operational semantics of Flang. Flang is a functional-logic language with constraints [3]. The semantics of Flang are based on the ideas of Σ-programming [2] and concurrent constraint programming [6]. The denotational semantics of Flang programs is defined using the notion of the least fixed point. A Σ+-machine introduced below describes the procedural semantics of Flang.

2. Σ-programs

In this section we introduce the notion of a Σ-program (which is a theoretical analogue of a program written in Flang) and describe its denotational semantics. Let $\mathcal{L}$ be a set of functional and predicate symbols and $\mathfrak{M}$ a model of the language $\mathcal{L}$. We augment $\mathcal{L}$ with the set of new predicate symbols $\{p_1, \ldots, p_k\}$ and denote $\mathcal{L}^* = \mathcal{L} \cup \{p_1, \ldots, p_k\}$. The new predicate symbols will denote predicates defined by Σ-programs.

**Definition 2.1.** [Term and Σ-formula] A variable and a constant are terms. If $t_1, \ldots, t_n$ are terms and $f$ is n-ary functional symbol, then $f(t_1, \ldots, t_n)$ is a term.

An atomic formula is an expression of the form $p(t_1, \ldots, t_n)$, where $p$ is an n-ary predicate symbol and $t_i$ are terms. We say that $p$ is a predicate symbol of this formula.

(i) An atomic formula is a Σ-formula; (ii) if $F$ and $G$ are Σ-formulas and $H$ is a formula of the language $\mathcal{L}$ without unbounded quantifiers, then $F \land G$, $F \lor G$, $H \rightarrow F$, $(\exists x \in y)F$, $(\forall x \in y)F$, $(\exists x)F$ are Σ-formulas; (iii) no other formulas.
A formula $F$ is a formula of the language $\mathcal{L}$ ($\mathcal{L}^*$) if all functional and predicate symbols occurring in $F$ belong to $\mathcal{L}$ (to $\mathcal{L}^*$). Let $\text{Var}(F)$ be a set of all free variables of $F$. The expression $P \Leftrightarrow F$ is a *predicate definition* if $P$ is an atomic formula with a user-defined predicate symbol and $F$ is a $\Sigma$-formula of the language $\mathcal{L}^*$ such that $\text{Var}(F) \subseteq \text{Var}(P)$. A *$\Sigma$-program* is a set $\{D_1, \ldots, D_k, F_1, \ldots, F_m\}$ such that $D_i$ are definitions and $F_i$ are $\Sigma$-formulas. The $F_i$s are queries (constraints). We denote programs by $\Gamma, \Delta, \ldots$

The notion of validity of a formula in the model $\mathcal{R}$ is defined as usual. Since variables in $\Sigma$-formulas can denote not only elements of the model $\mathcal{R}$ but also sets of elements, the definition of the denotational semantics of $\Sigma$-formulas needs an extension of $\mathcal{R}$ which can be obtained by introducing a hereditary finite superstructure over $\mathcal{R}$ [1].

**Theorem 2.1.** Let $\mathcal{R}$ be a model of the language $\mathcal{L}$. Then for any $\Sigma$-program $\Gamma$ of the language $\mathcal{L}^*$ the corresponding operator $\mathcal{P}_\Gamma$ has a least fixed point.

$\Gamma(\mathcal{R})$ will denote an extension of $\mathcal{R}$ in which all $p_i \in \mathcal{L}^* \setminus \mathcal{L}$ are defined as least fixed points of $\mathcal{P}_\Gamma$.

### 3. $\Sigma^+$-machine

In this section we introduce an abstract machine computing $\Sigma$-programs. This machine describes an ‘actual’ procedural semantics of $\Sigma$-programs. A $\Sigma^+$-machine is a particular case of a $\text{cc}$-machine [6]. $\Sigma$-formulas play in the $\Sigma^+$-machine a role of constraints. The store of the $\Sigma^+$-machine contains a $\Sigma$-program. Logically the store is equivalent to the conjunction of the formulas it contains. The $\Sigma^+$-machine is transforming its store until the moment when it is possible to extract a solution from information collected in the store of the machine.

All transition rules for $\Sigma^+$-machine can be split into two parts – the *closure rules* and the *choice making rules*. From now on $\{F \mid \Gamma\}$ denotes a store $\Gamma$ augmented by a formula $F$. 

27
Closure Rules

(∀-elimination)

\[ \{ (\exists x) F \mid \Gamma \} \mapsto \{ F \mid \Gamma \} \]

if there are no formulas in \( \Gamma \) containing \( x \) (otherwise, \( x \) must be renamed)

\[ \{ (\exists x \in \{ t_1, \ldots, t_k \}) F \mid \Gamma \} \mapsto \{ (x = t_1 \lor \ldots \lor x = t_k), F \mid \Gamma \} \]

\[ \{ (\exists x \in \emptyset) F(x) \mid \Gamma \} \mapsto \{ \text{false} \} \]

\( x = t_1 \lor \ldots \lor x = t_k \) is called the domain disjunction for \( x \)

(ask-operation: \( \rightarrow \)-elimination)

\[
\begin{align*}
\Gamma \vdash F & \quad \{ F \rightarrow G \mid \Gamma \} \mapsto \{ G \mid \Gamma \} \\
\{ F \mid \Gamma \} \vdash \text{false} & \quad \{ F \rightarrow G \mid \Gamma \} \mapsto \Gamma
\end{align*}
\]

(local consistency)

\[
\begin{align*}
\Delta \subseteq \Gamma \quad \{ F \mid \Delta \} \vdash \text{false} & \quad \{ F \lor G \mid \Gamma \} \mapsto \{ G \mid \Gamma \}
\end{align*}
\]

(constuctive disjunction – a sketch)

\[
\begin{align*}
\Delta \subseteq \Gamma \quad \{ F \mid \Delta \} \overset{\ast}{\mapsto} \Lambda_1 \quad \{ G \mid \Delta \} \overset{\ast}{\mapsto} \Lambda_2 & \quad \{ F \lor G \mid \Gamma \} \mapsto \{ F \lor G \mid \Gamma \} \sqcup (\Lambda_1 \cap \Lambda_2)
\end{align*}
\]

In the last two rules \( \Delta \) denotes the set of the domain disjunctions for all the variables occurring in \( F \lor G \). The symbol \( \overset{\ast}{\mapsto} \) denotes the closure operator and \( \Gamma \sqcup \Delta \) and \( \Gamma \cap \Delta \) denote the informal notions of union and intersection of portions of information, respectively. Unfortunately, we can not give here the precise definition of the constructive disjunction rule, since it is complicated and needs a lot of space. We also omit some closure rules.

The key property of the closure rules is that they together define in the \( \Sigma^+ \)-machine the entailment relation \( \vdash \).
The Choice Making Rule

(∨-elimination)

\[ \{F \lor G \mid \Gamma\} \mapsto \{G \mid \Gamma\} \]

We do not distinguish \( F \lor G \) and \( G \lor F \).

The \( \lor \)-elimination rule is the only non-equivalent rule in the \( \Sigma^+ \)-machine. The above form of the choice making rule brings about the exhaustive search strategy of computation. To reduce the exhaustive search, the \( \Sigma^+ \)-machine applies the local consistency and constructive disjunction rules.

The depth-first search with chronological backtracking is the main strategy of searching for a solution in the \( \Sigma^+ \)-machine. Applications of the choice rule play the role of choice points. Backtracking starts when the constraint \texttt{false} appears in the store. In this case the machine returns to the last application of the choice making rule and selects for considerations another alternative of disjunction.

4. Completeness and Soundness

In this section we consider the problems of soundness and completeness for the abstract \( \Sigma^+ \)-machine introduced above.

**Definition 4.1.** Let \( \Gamma = \{G_1, \ldots, G_k, D_1, \ldots, D_m\} \), where \( G_i \) are \( \Sigma \)-formulas and \( D_i \) are definitions, be a \( \Sigma \)-program and \( \mathcal{R} \) be a model of the same language. We say that \( \Gamma \) is \( \Sigma^+ \)-realizable if there exists a sequence of state transitions transforming \( \Gamma \) into the store \( \{D_1, \ldots, D_m\} \) containing only predicate definitions. \( F \) is \( \Sigma^+ \)-realizable by a \( \Sigma \)-program \( \Gamma \), if the set \( \{F \mid \Gamma\} \) is \( \Sigma^+ \)-realizable.

The following theorem shows that the \( \Sigma^+ \)-machine is sound:

**Theorem 4.1.** [soundness] Let \( \Gamma \) be a set of predicate definitions and \( F \) a \( \Sigma \)-formula. If \( \{F \mid \Gamma\} \) is \( \Sigma^+ \)-realizable, then \( \Gamma(\mathcal{R}) \models F \).

The next theorem is concerned with a very important case when the \( \Sigma^+ \)-machine is complete:
Theorem 4.2. \([\Sigma^+]\)-completeness] If \(R\) is a finite model then any \(\Sigma\)-formula is \(\Sigma^+\)-realizable by a program \(\Gamma\) iff \(\Gamma(R) \models F\).

5. Conclusion

Understanding a logical formula as a process and the machine store as the set of processes gradually gathering information about the solution to be found, is very fruitful from both the theoretical and practical points of view. It is also very important that this approach permits very efficient implementation [4], [5].

References


Chair of Information Systems
Irkutsk State University
664003 Irkutsk
Russia