3D-3

Language for Modular Programming on Term Rewriting Systems

Hideki Yamanaka
International Institute for Advanced Study of Social Information Science,
FUJITSU LIMITED

Introduction

In a few years, term rewriting systems[5] becomes popular and common as a computation model, especially a model for functional programming languages. Because it has special features, e.g. its denotational or algebraic semantics is corresponding to its operational semantics under some condition. Nevertheless it has no features of modularization for the structural programming, so it is claimed that term rewriting systems have the same computation power as turing machines and we can program anything by term rewriting systems, but we may not program large scale one because term rewriting systems have no modularization mechanism in themselves.

In a couple of years, the direct sum system of term rewriting systems[5][6][7] that is a disjoint union of term rewriting systems; renaming symbols appearing in them so that no name conflict can arise, was proposed for a key to the solution. But the direct sum system is too weak to modularize term rewriting systems because it is so hard to divide the programming into pieces that process shared resources, i.e. in the direct sum system, a module can not know any data representation of other modules.

In this paper we propose a programming language MT for which we may describe a term rewriting system by a set of small scale ones. We will show the structural programming, i.e. the modularization, in term rewriting systems later.

Syntax of MT

MT is designed for programmers to be able to program in styles of top-down and bottom-up manner. In top-down style programming, we may use usual functional programming ways and program recursive descendantly from the top level function to bottom functions. In bottom-up style programming, we may use procedural programming ways and make up a program by combining many modules together, which have been programmed in top-down style. In short, we program a large program by gathering many small modules in the language MT. Although the language has many procedural features: the modularity and so on, but yet its semantics is purely functional and mathematical. Now we start to explain its syntax.

A program is a set of modules. A module is a pair of a signature and equations roughly. (Note that each module is a usual term rewriting system.) In detail, it is shown as a BNF representation as follows:

$\langle program \rangle$ ::= $\langle global\text{-}struct \rangle \langle config \rangle \langle modules \rangle^*$
$\langle global\text{-}struct \rangle$ ::= '*global*' '{' $\langle signature \rangle$ '}'
$\langle config \rangle$ ::= '*init*' '{' $\langle term \rangle$ ';' '}'
$\langle module \rangle$ ::= $\langle term \rangle$ '[' $\langle parm\text{-}declar \rangle$ ']' '{' $\langle init \rangle \langle signature \rangle \langle equation \rangle^+$ '}'
$\langle parm\text{-}declar \rangle$ ::= $\langle parm\text{-}list \rangle$ '−' $\langle sort \rangle$
$\langle parm\text{-}list \rangle$ ::= | $\langle non\text{-}null\text{-}parms \rangle$ ',' $\langle parm \rangle$
$\langle non\text{-}null\text{-}parms \rangle$ ::= $\langle parm \rangle$ | $\langle non\text{-}null\text{-}parms \rangle$ ',' $\langle parm \rangle$
$\langle parm \rangle$ ::= $\langle var \rangle$ ':' $\langle sort \rangle$
$\langle init \rangle$ ::= '*init*' ':' $\langle term \rangle$ ';'
$\langle signature \rangle$ ::= | $\langle signature \rangle \langle functionality \rangle$
$\langle functionality \rangle$ ::= $\langle function \rangle$ ':' $\langle arity \rangle$ ';'
$\langle arity \rangle$ ::= $\langle sort\text{-}list \rangle$ '−' $\langle sort \rangle$
$\langle sort\text{-}list \rangle$ ::= | $\langle non\text{-}null\text{-}sorts \rangle$ ',' $\langle sort \rangle$
$\langle non\text{-}null\text{-}sorts \rangle$ ::= $\langle sort \rangle$ | $\langle non\text{-}null\text{-}sorts \rangle$ ',' $\langle sort \rangle$
$\langle equation \rangle$ ::= $\langle term \rangle$ '=' $\langle term \rangle$ ';'
$\langle term \rangle$ ::= $\langle var \rangle$ | $\langle function \rangle$ '(' $\langle args \rangle$ ')'
$\langle args \rangle$ ::= | $\langle non\text{-}null\text{-}args \rangle$
$\langle non\text{-}null\text{-}args \rangle$ ::= $\langle term \rangle$ | $\langle non\text{-}null\text{-}args \rangle$ ',' $\langle term \rangle$
$\langle vars \rangle$ ::= | $\langle non\text{-}null\text{-}vars \rangle$ ',' $\langle var \rangle$
$\langle sort \rangle$ ::= $\langle alfabet \rangle^+$
$\langle function \rangle$ ::= $\langle alfabet \rangle^+$
$\langle var \rangle$ ::= $\langle alfabet \rangle^+$
$\langle alfabet \rangle$ ::= $[0-9] | [a-zA-Z]$

Limitations:

1. $\langle term \rangle$ must be well-formed to $\langle parm\text{-}declar \rangle$ and $\langle arity \rangle$.
2. Any proper subterm without variables of $\langle term \rangle$ in $\langle module \rangle$ must be declared in $\langle global\text{-}struct \rangle$.
3. $\langle term \rangle$ at the head of $\langle module \rangle$ and left hand side $\langle term \rangle$ of $\langle equation \rangle$ must be linear.
4. $\langle module \rangle$ and $\langle equation \rangle$ must be non-overlapping.

Semantics of MT

A MT program P has an initial configuration $Conf$, a special module $Glob$, said a global module, and other modules $M_1, M_2, \cdots, M_n$. So a program

$$P = \langle Conf, Glob, M_1, M_2, \cdots, M_n \rangle$$

Each module M_i has a configuration $conf_i$, a signature Σ_i and a set of rules R_i, so $M_i = \langle conf_i, \Sigma_i, R_i \rangle$, where $conf_i = \langle N_1, N_2 \rangle$: N_1 is a term laid ahead of its module and N_2 is a term laid *init* part of its module. Note that we sometimes write M_0 as $\langle Conf, Glob \rangle$ because $\langle Conf, Glob \rangle$ is a special case of modules.

Language for Modular Programming on Term Rewriting Systems
Hideki Yamanaka
FUJITSU LIMITED

Here we start to define a semantic domain $\mathfrak{D} = \langle \mathfrak{Conf}, \Sigma, \mathfrak{I}, \mathfrak{X}, \mathfrak{R} \rangle$ and semantic function $\mathfrak{M} : P \rightarrow \mathfrak{D}$. we define $\mathfrak{M}$ as a quintuple $\langle \delta, \theta, \mathcal{V}, \alpha, \beta \rangle$ At first, δ is a map that extracts sorts from a term using the fact $t \in T[Glob \cup X]$ as follows:

$$\delta(f(t_1, t_2, \cdots, t_m)) = \{f : s_1, s_2, \cdots, s_m \rightarrow s_{m+1}\}$$

$$\delta(x) = \{x : s\} \quad \text{from } [\cdots, x : s, \cdots] \text{ in } \langle parm\text{-}declar \rangle$$

where $s_i = Sort(t_i)$. And it is extended to

$$\tilde{\delta}((N_1, N_2)) = \delta(N_1)$$

At second, θ is a map that annotates each signature with its module number. Let

$$\Sigma_i = \{f_k : s_{k1}, s_{k2}, \cdots, s_{kn_k} \rightarrow s_{kn_k+1}\}$$

,then θ maps it to

$$\theta(\Sigma_i) = \{f_k^i : s_{k1}^i, s_{k2}^i, \cdots, s_{kn_k}^i \rightarrow s_{kn_k+1}^i\}$$

At third, $\mathcal{V}$ is a map that collects variables in a term as follows:

$$\mathcal{V}(N) = \{\{x \mid x \text{ is a variable symbol occurring in } N \text{ whose sort is } s\} \mid s \in S\}$$

And it is extended to

$$\tilde{\mathcal{V}}((N_1, N_2)) = \mathcal{V}(N_1) \cup \mathcal{V}(N_2)$$

At fourth,β is as follows:

$$\beta(f(t_1, t_2, \cdots, t_n), i) = \alpha(f, i)(\beta(t_1, i), \beta(t_2, i), \cdots, \beta(t_n, i))$$

$$\beta(x, i) = \Delta_s(\alpha(x, i))$$

where $x \in X_s$. And β is extended as follows:

$$\tilde{\beta}((N_1, N_2), i) = \{(\alpha(N_1, 0), \Delta_s(\beta(N_2, i)))\}$$

where s is the sort of N_2. At last, we define α that annotates all symbols with module numbers as follows:

$$\alpha(v, i) = \begin{cases} v^i & \text{if } v \text{ is defined in } M_i \\ \Delta_s^0 & \text{if } v \text{ is a } \Delta_i \\ v^0 & \text{otherwise} \end{cases}$$

where s is also the sort of Δ_i. And it is extended to

$$\alpha'(M_i, i) = \tilde{\beta}(conf_i, i) \cup \alpha(R_i, i)$$

Note that the domain of α's is extended from symbols to terms and rules.

Now we get the semantic function as follows:

Let $\Sigma_0 = Glob \cup \delta(conf_1) \cup \delta(conf_2) \cup \cdots \cup \delta(conf_n)$

,then we get

$$\Sigma = \theta(\Sigma_0) \cup \theta(\Sigma_2) \cup \cdots \cup \theta(\Sigma_n)$$

And an initial configuration; the start point of computation, is

$$\mathfrak{Conf} = \alpha(Conf)$$

$\mathfrak{I}$ is a S-indexed family of predefined input variables that are denumerable. In this system $\mathfrak{I}$ is treated as a S-indexed family of new constants. the constants can allow to be placed only in $\langle Conf_0 \rangle$ to be used for inputs from users. And a S-indexed family of all variable symbols used in this system is

$$\mathfrak{X} = \tilde{\mathcal{V}}(conf_0) \cup \tilde{\mathcal{V}}(conf_1) \cup \cdots \cup \tilde{\mathcal{V}}(conf_n)$$

Finally we get a set of rules that is maybe infinite is

$$R = \bigcup_{i \in I} \alpha'(M_i) \bigcup \{\{(\Delta_s(t), t) \mid t \in T[Glob \cup \mathfrak{I} \cup \{\Delta_{s'} \mid s' \in S_0\}]_s\} \mid s \in S_0\}$$

Note that Δ_s is the abbreviation of Δ_s^0 and $\cup$ is not the union of sets but is the union of S-indexed family of sets.

Discussion

Many languages based on term rewriting systems, i.e. HOPE[1], OBJ2[2], MIRANDA[8] and so on, have been developed. But these languages only have a modularization mechanism by the abstract type constructor or the parameterization[3] but have no identifier management mechanisms like stack. In short, these languages can not have any identifier localization mechanism.

The localization is the key to make programmers to program large scale ones, It has been pointed out from the view of the structural programming. So we must be necessary for the mechanism, so that we may program large scale ones using term rewriting systems.

As far as we know, MT is the first language that treats the identifier management mechanism in a framework of term rewriting systems. Moreover MT has a feature that partially computes modules, so that each module of MT may become efficient by incremental partial computation. In short, we have design MT, so that MT might be the key to the theory of reusability of programs.

Acknowledgement

The author would like to thank Professor Y.Inagaki, Dr. T.Naoi and other members of Inagaki laboratory at Nagoya University for their valuable comments.

References

[1] Burstall,R.M. *et al* : HOPE: An Experimental Applicative Languages, The 1980 LISP Conference, pp.136-143, Stanford Univ. (1980).

[2] Futatsugi,K. *et al* : Priciples of OBJ2, Proc. 12th ACM POPL (1985).

[3] Goguen,J.A. *et al* : An Initial Algebra Approach to The Specification, Correctness and Implementation of Abstract Data Types, Current Trend in Programming Methodology IV (Ed. Yeh,R.), pp.80-149 (1978).

[4] Hoffman,C.M. *et al* : Programming with Equations, ACM TOPLAS Vol.4, No.1, pp.83-112 (1982).

[5] Klop,J.W. : Term Rewriting Systems: a tutorial, Note CS-N8701, Centre for Mathematics and Computer Science, Amsterdam (1987).

[6] Tovama,Y. : On the Church-Rosser Property for the Direct Sum of Term Rewriting Systems, J.ACM Vol.34, No.1, pp.128-143 (1987).

[7] Toyama,Y. : Counterexamples to Termination for the Direct Sum of Term Rewriting Systems, Info.Proc.Lett. 25, pp.141-143 (1987).

[8] Turner,D. : Miranda: a non-strict functional laguage with polymorphic types, LNCS Vol.201 (1985).