ソフトウェア基礎論 19－8
プログラミング言語　9－8
（1986. 12. 12）

並列処理言語 Oc の検証系の構想

Plan of Verification System based on Parallel Programming Language Oc

枚田正宏
Masahiro Hirata
筑波大学　数学系
Institute of Mathematics, University of Tsukuba

あらまし　著者は，並列プログラムの検証系を立案中である。これは等価変換に基づいている。このシステムでは仕様もプログラムも言語 Oc で書かれる。本論文では，このシステムの基本的な概念を説明する。それは Oc の2つの側面である。一方は，仕様記述言語であり，他方はプログラミング言語である。

Abstract: We are planning a new verification system of parallel programs. It is based on the equivalence transformation. In the system, specifications and programs are written in the language Oc. In this paper, we explain the basic concept of our system. That is dual aspect of Oc. One is as a specification language and the other is as a programming language.

Introduction

We are planning a verification system of parallel programs. It is based on the equivalence transformation. In the system, specifications and programs are written in the language Oc. In this paper, we explain the basic concept of our system. It is the dual aspect of Oc.

Oc is a very simple parallel programming language. But program in it accept dual interpretations; one is as specification (in the other word, 'formal system' [Sato 1985]), other is as a parallel program [Hirata 1986].

In the next section, first aspect as the specification language is explained. In the section 2, the other aspect of Oc as a programming language is defined. At last section, we show the one property of Oc and its influence on the specifications of system programs.

2

1. Oc as Specification Language

Syntax

The syntax of Oc is the same as pure Prolog encoded in Symbolic Expression (S-exp), but don't use the terminology of it, because it is misleading about the relation between the language and formal logics.

```
term ::=  constant | variable |
          '(' term '.' term ')'.
conclusion     ::=  term.
assumptions    ::=  '(' { term } ')'.
inference rule ::=
     '(' conclusion '.' assumptions ')'.
formal system  ::=
   '(' { inference rule } ')'.
```

In the syntax, {e} denotes for some n (n≥0), n-times repetition of something that is denoted by e.

In this paper, variables are denoted by strings beginning with capital letters. Other strings that do not include the characters '(', '.', ')' and ' ' denote constants.

We use the usual abbreviations of S-exp. For example, (a b) and (X Y . Z) are abbreviations of (a . (b . ())) and (X . (Y . Z)), respectively.

We sometime call a inference rule 'axiom', if it has null assumptions.

Example (Natural Number)

The following is a formal system UN expressing the unary natural number and successor function and equality function on it.

```
(
((natural ()))
((natural (* . N))
     (natural N))

((successor N S)
     (natural N) (S :=: (* . N)))

((() = ()))
(((* . X) = (* . Y))
     (natural X) (natural Y) (X = Y))
)
```

Definition of Proof

For given formal system FS, we define the partial proofs of a term f with assumptions, recursively, as follows.

1. f is a partial proof of f itself with assumptions (f).
2. When a member A of assumptions AS of a partial proof P of f is an instance of a conclusion C of an inference rule L in FS (i.e. for some substitution S, A=S(C)), the following S-exp P1 is a partial proof of f: S-expression that is the same as P, except that A in it is replaced by S(L)
3. When a member A of assumptions AS of a partial proof P of f is a form (X :=: Y) where X and Y is arbitrary terms, and there is a most general unifier S of X and Y that is not change any

variable in f, the S-exp S(P) is a partial proof of f with assumptions S(AS-{A}).

We call a partial proof with no assumption 'proof', and a term which has proof 'theorem' of the system, respectively.

Example

The following are the partial proofs of (successor (*) (* *)) in the above formal system UN:

```
(successor () (*))
     with (successor () (*)),

((successor () (*))
     (natural ())
     ((*) :=: (*)))
with ((natural ()) ((*) :=: (*))),

((successor () (*))
     ((natural ()))
     ((*) :=: (*))) with ().
```

The last one is a proof. Its usual graphical representation is the following:

```
   (natural ())      ((*) :=: (*))
 ------------------------------------
        (successor () (*)).
```

2. Oc as Parallel Programming Language

Syntax

We change terminology.

```
process   ::=  term.
guard     ::=  term.
body      ::=  '(' { process } ')'.
clause    ::=  '(' guard '.' body ')'.
program   ::=  '(' { clause } ')'.
```

Operational Semantics

Each state of Oc is a pair of the control state and a data state. A control state is a list of processes. A data states is a substitution [Hirata 1984]. If the control state is the null list, that state is called final. A computation is a change from a given state into the final state through repeated state transition. The final data state is called the result of the computation. State transition means concurrent application of the following rules to each process P in a state.

rule 1. (receiving)
if P is an instance of a guard of some clause, substitute the embedded body processes of the clause for P in control state; Note: variables local to the body should be replaced by fresh ones.

rule 2. (broadcasting)
if P is of the form (t1 :=: t2), remove P from the control state; if t1 and t2 are unifiable, merge the most general unifier S into the data state and apply S to all other processes within a finite time; if not unifiable, abort all the computation at once.

Example (Filter)

```
(
((Filter () F O)
     (O :=: ()))
((filter (X . I) F O)
     (F X A)
     (filter_decide A X I F`O))

((filter_decide true X I F O)
     (O :=: (X . P))
     (filter I F P))
((filter_decide false X I F O)
     (filter I F O))
)
```

3. Interference Freeness and Its Influence

The following is an obvious but important property of the operational semantics of Oc: After an process P in a control state become to be an instance of guard G of a clause, unless P is not removed, this property is preserved by the rest of the computation.

This property means the interference freeness ([Hoare 1972], [Owicki 1976]) of the possibility of transforming of processes. By the property, we can conclude that the strongest post-condition for a given pre-condition is disjunction (with respect to all possible computations) of the conjunction of following ones:

1. a pre-condition,
2. all guards of successful transitions in the computation, and,
3. equivalence relation generated by the result of the computation.

In the current version of Oc, the internal feature is only the unification (X :=: Y). (Theoretically, the version is sufficiently strong in the sense that all recursive functions are encodable.) More over, every extension of the language should preserve the above property. Because, this property permits partial analysis of the compound parallel program. Modularity is one of the best tools for the construction of complex systems.

By the above limit, we can not introduce some features common in usual Prolog systems, for example, var(X) (i.e. recognizer of assignment to X) or strong simulate feature (i.e. control other process to suspend and/or eliminate from the control state). These features are sometime necessary for writing system programs.

We treat this problem as the following.

0. Every process can modity only the data state. it can access neither its program nor the control state.
1. The controlled processes and its controller in not in the same control state.

2. If the former's state should be fully analyzable by the latter, the former considered to be simulated the latter.
3. In the above case, all the former's program, the control state and the data state should be encoded into the latter's state.

One example of second stuation is user process U and the manager process M for U in the operating system. M initiate U and accept the system request of U and resend it to some other system process of charge, if necessary. Sometimes, M must be abort U because of some cause. In the principle, M can access (the coding of) the program of U as the data.

Conclusion

We represent dual semantics of Oc. Because of the aspect, We may use Oc to describe both the data states and the algorithms. But, there is one deep gap between them. It is the assignability of the variables.

We hope that the explicit direction of the information flows of Doc [Hirata 1978a] make an uniform semantics for the both purposes.

Reference

[Hirata 1986] Hirata M., Letter to Editor, SIGPLAN Notice, May 1986
[Hirata 1986a] Hirata M., Programming Language Doc and its Self-Description, or, X=X is considered harmful, in Proc. of 3rd National Conf. of Japan Society of Software Science and Technology, pp. 69-72
[Hoare 1972] Hoare C. A. R., Toward a theory of parallel programming, in Operating Systems Techniques, Hoare and Perott (Eds.), Academic Press, New York, 1972
[Owicki 1976] Owicki S. and Gries D., Verifying Properties of Parallel Programs: An Axiomatic Approach, Comm. of ACM, Vol. 19, No. 5, May, 1976
[Sato 1985] Sato M.,Theory of Symbolic Expressions, II, Publ. RIAMS, Kyoto Univ., 1985