IEICE TRANS. FUNDAMENTALS, VOL.E89–A, NO.6 JUNE 2006

PAPER Special Section on Papers Selected from ITC-CSCC 2005

A Behavioral Specification of Imperative Programming Languages*

Masaki NAKAMURA[†a)], ***Member***, **Masahiro WATANABE**[††], ***and*** **Kokichi FUTATSUGI**[†], ***Nonmembers***

SUMMARY In this paper, we give a denotational semantics of imperative programming languages as a CafeOBJ behavioral specification. Since CafeOBJ is an executable algebraic specification language, not only execution of programs but also semi-automatic verification of programs properties can be done. We first describe an imperative programming language with integer and Boolean types, called *IPL*. Next we discuss about how to extend *IPL*, that is, *IPL* with user-defined types. We give a notion of equivalent programs, which is defined by using the notion of the behavioral equivalence of behavioral specifications. We show a sufficient condition for the equivalence relation of programs, which reduces the task to prove programs to be equivalent.

key words: *semantics of imperative programs, behavioral specification, CafeOBJ*

1. Introduction

Since software written by imperative programming languages (C, Java, etc) are widely used in various fields, it is important for the safety of our social life to give a formal semantics to imperative programs and to verify desired properties. There are three kinds of semantics of imperative programs: denotational semantics, such as D. Scott and C. Strachey's denotational semantics in which a program denotes a partial function from inputs to outputs [7], axiomatic semantics, such as Hoare logic in which a triple of a precondition, a program and a postcondition is used [6], and operational semantics, which describes a way of executing programs. Recently, an algebraic denotational semantics of imperative programs has been proposed [4]. It showed that algebraic specifications are very useful to describe and verify imperative programs. After that, Hidden algebra, an extension of classical algebra, has been proposed [3], [5]. A specification based on Hidden algebra, called a behavioral specification, treats both system description and data description in a same framework rigorously and easily. In this paper we describe an behavioral specification for an algebraic semantics of an imperative programming language.

In the next section we introduce CafeOBJ algebraic specification language, which supports behavioral specifications and a rewrite engine to verify properties of a specification semi-automatically [2], [8]. In Sect. 3, we give a behavioral specification for a semantics of imperative programs. The imperative language we describe is called *IPL*. *IPL* has fundamental sentences of imperative programs: assignments, conditionals, iterations. We give a syntax and a semantics of *IPL*. In Sect. 4, we discuss about executability of *IPL* and verification of properties of programs written in *IPL*. We formalize equivalence relation on programs and show an efficient sufficient condition for program equivalence. In Sect. 5, we give a way to define user-defined types to *IPL*. By using this, we can introduce a kind of classes of object-oriented languages to *IPL*. In Sect. 6, we discuss about related works, and conclude our work in Sect. 7.

2. Preliminaries: CafeOBJ in a Nutshell

CafeOBJ specifications are divided into those with the tight denotation and the loose denotation: they are used to describe abstract data types and systems behaviors respectively.

Specification for data description A specification with the tight denotation is described as `mod!{...}`. Sorts, operations, equations, conditional equations are declared as `[...]`, `op` (`ops`), `eq`, and `ceq`, respectively. A model of a CafeOBJ specification is an algebra where sorts and operations interpreted as carrier sets and operations on the sets, and each equation should hold. A model of a specification with the tight denotation is an initial algebra (or an initial model). An initial model is a model which has the unique morphism to any other model. Since initial models are isomorphic, essentially there is only one initial model. Each carrier set of the initial model has only elements defined in the specification (no junk), and only elements which are equivalent in the specification are equivalent in the initial model (no confusion).

Example 2.1: The following is a specification of integers: an example of specifications with the tight denotation.

```
mod! INTEGER{
 [Int]
 op 0 : -> Int
 op s : Int -> Int
 op p : Int -> Int
 eq p(s(I:Int))= I.
 eq s(p(I:Int))= I.
```

Manuscript received September 5, 2005.
Manuscript revised December 27, 2005.
Final manuscript received February 14, 2006.

[†]The authors are with the School of Information Science, Japan Advanced Institute of Science and Technology, Nomi-shi, 923-1292 Japan.
[††]The author is with Production Engineering Research Laboratory, Hitachi, Ltd., Yokohama-shi, 244-0817 Japan.

*This paper was presented at Session TD3: Computer Systems & Applications (2) 3 of ITC-CSCC 2005.

a) E-mail: masaki-n@jaist.ac.jp

DOI: 10.1093/ietfec/e89–a.6.1558

```
}
```

Since `mod!` stands for the tight denotation, `INTEGER` denotes only the initial model, that is, the set of integers, where terms `0`, $\mathtt{s}^n(\mathtt{0})$ and $\mathtt{p}^n(\mathtt{0})$ correspond to integers 0, n and $-n$ respectively. Note that any term of `Int`, which may have both `s` and `p`, equals to `0`, $\mathtt{s}^n(\mathtt{0})$ or $\mathtt{p}^n(\mathtt{0})$, e.g. $\mathtt{s}(\mathtt{p}(\mathtt{s}(\mathtt{0}))) = \mathtt{s}(\mathtt{0})$.

Behavioral specification As opposed to tight specifications, the model of a specification with the loose denotation, described by `mod*`, is the set of all algebras satisfying the axiom (equations). A behavioral specification is a specification with the loose denotation which has **a hidden sort** (`*[...]*`) and **behavioral operations** (`bop` or `bops`). A non-hidden sort is called a visible sort. In a system description by a behavioral specification, a hidden sort denotes a set of states of the system and visible sorts denote data types used in the system. Behavioral operations are divided into **observations** and **actions**. The system is hidden, which means it cannot be observed and modified directly. Only observations can observe it and only actions can modify it. In other words, the models of a behavioral specification are those satisfying a behavior defined by the observations and the actions. The style of a behavioral specification is object-oriented as opposed to the data-oriented classical algebraic specification style. In OO terminology, observations correspond to attributes and actions correspond to methods.

Example 2.2: The following is a specification of a buffer: an example of behavioral specifications.

```
mod* BUFFER{
 pr(INTEGER)
 *[Buf]*
 op init : -> Buf
 bop val : Buf -> Int
 bops up dn : Buf -> Buf
 eq val(init) = 0 .
 eq val(up(X:Buf)) = s(val(X)).
 eq val(dn(X:Buf)) = p(val(X)).
}
```

`BUFFER` denotes a buffer which can keep an integer, where the specification for integers is imported by `pr(INTEGER)`. An observation `val` observes an integer kept by the buffer. A constant `init` stands for an initial buffer with 0. An action `up` (or `dn`) increases (or decreases) the integer one by one. Since `mod*` stands for the loose denotation, the denotation of `BUFFER` is the set of all models, i.e. all possible implementations satisfying it, e.g. implementation by arrays, stacks, etc.

3. Specification of Imperative Programs

The language described in this paper is called *IPL*. We call the hidden sort of a behavioral specification of *IPL* **Semantic system**. The observations are references of the value of an expression, and the actions are execution of a program. It is a variant of D .Scott and C .Strachey's denotational semantics [7].

Syntax of *IPL* is defined by `EXP` and `PGM`.

```
mod! EXP{
pr(VAR + EQ-INT)
[Var Int < Exp]
[Bool < Tst]
op _+_ : Exp Exp -> Exp
op _*_ : Exp Exp -> Exp ...
op _=_ : Exp Exp -> Tst ...
op not_ : Tst -> Tst
}
```

`EXP` describes a syntax of integer and Boolean expression by importing CafeOBJ built-in specifications `INT` and `BOOL`[†]. Sort declaration `[Var Int < Exp]` means that `Exp` of the integer expression is defined as a super sort of `Var` and `Int`. Thus, a variable (or an integer) is an expression. Operations +, *, etc are constructors of integer expressions. Similarly, Sort `Tst` of the Boolean expression is a super sort of `Bool`. For example, `X + 7` and `X + 1 > Y` are examples of integer expressions and Boolean expressions respectively.

```
mod! PGM{
pr(EXP)
[BPgm < Pgm]
op _:=_ : Var Exp -> BPgm
op _;_ : Pgm Pgm -> Pgm {assoc}
op if_then_else_fi : Tst Pgm Pgm -> Pgm
op while_do_od : Tst Pgm -> Pgm
}
```

`PGM` describes a syntax of programs. Assignments are basic programs, e.g. `X := X + 1`. When `P1` and `P2` are programs and `T` is a Boolean expression, then `P1 ; P2` (compositions)[††], `if T then P1 else P2 fi` (conditionals) and `while T do P1 od` (iterations) are programs.

Semantics of *IPL* is defined through Semantic system. For a given state of Semantic system, each integer (or Boolean) expression has its value, observed by `bop _[_] : Sys Exp -> Int` (or `bop _[_] : Sys Tst -> Bool`). `S[E]` denotes the value of an expression `E` at a state `S` of Semantic system. Roughly speaking an observation by `E` is calculation of `E`. E.g. `S[1 + 2 + 3]` is `6` for each `S`. `S[1 + X * 3]` is `7` for each `S` where Variable `X` is assigned to `2`, i.e. `S[X] = 2`. Applying programs changes a state of the system. It corresponds to an action `bop _;_ : Sys BPgm -> Sys`. State "`S ; P`" means the result of applying Program `P` to State `S`.

The following is a specification for a semantics of *IPL*:

```
mod* SEM{
```

[†] `INT` is a built-in specification with Sort `Int`, integer constants (`1`, `2`, ...) and operations (+, -, ...). `EQ-INT` is an extension of `INT`, where an equality predicate `=i=` on `Int` is added. `QID` is a built-in specification with Sort `Oid` and quoted identifiers (`'X`, `'Y`, ...). `VAR` is an extension of `QID`, where Sort `Qid` is renamed `Var` and an equality predicate `=v=` on `Var` is defined. Roughly speaking, the equality predicates are defined as follows: i_1 `=i=` i_2 (or v_1 `=v=` v_2) is reduced into `true` if the arguments are same after reducing them. Otherwise `false`.

[††] An operation attribute `assoc` means the operation is associative, i.e. `(P ; Q) ; R = P ; (Q ; R)`. Parentheses can be omitted, like `P ; Q ; R`.

```
  pr(PGM) *[Sys]*
  bop _[_] : Sys Exp -> Int
  bop _[_] : Sys Tst -> Bool
  bop _;_ : Sys BPgm -> Sys
  var S : Sys .
  var I : Int .
  var B : Bool . ...
  eq S[I] = I .
  eq S[E1 + E2] = (S[E1]) + (S[E2]) . ...
  eq S[E1 = E2] = (S[E1]) =i= (S[E2]) . ...
  eq S[B] = B .
  eq S[not T] = not S[T] . ...
  eq  S ; (X := E1)[X] = S[E1] .
  ceq S ; (X := E1)[Y] = S[Y] if not X=v=Y .
  beq S ; (P1 ; P2) = (S ; P1) ; P2 .
  bceq S ; if T then P1 else P2 fi = S ; P1
  if S[T].
  bceq S ; if T then P1 else P2 fi = S ; P2
  if not S[T].
  bceq S ; while T do P1 od = (S ; P1) ;
          while T do P1 od if S[T] .
  bceq S ; while T do P1 od = S if not S[T] .
 }
```

The axiom part (a set of equations) is divided into two parts: that for expressions (from `eq S[I] = I` to `eq S[not T] = not S[T]`) and that for programs (the others)[†]. The value of an expression is equivalent to a term which is composed of integers and the values of variables. For example,

`S[X + (2 * 3)]`
= `S[X]` + `S[2 * 3]`
= `S[X]` + `S[2]` × `S[3]`
= `S[X]` + 2 × 3
= `S[X]` + 6.

If we know the values of all variable, each expression is equivalent to an integer value. For example, if `S[X] = 7` then `S[X + (2 * 3)] = 13`.

Consider first two equations of the axiom for programs, which define a semantics of assignment programs. `S ; (X := E)` stands for the result of applying `X := E` to State `S`. The equations describe the value `S ; (X := E)[Y]` for each variable `X`. The first equation means that it equals to the value of `E` of the original state `S` when `X = Y`. The second equation means that it equals to the value of `Y` of the original state `S` when $X \neq Y$. Equation `S ; (P1 ; P2) = (S ; P1) ; P2` means that a sequence of programs is executed from the front. Next two conditional equations define a semantics of conditionals. The value of the condition part `T` at `S` is evaluated first, and then `P1` is executed if it is true, otherwise `P2` is executed. The last two equations are for iteration programs. If `S[T]` is true then `S ; while T do P1 od` is equivalent to `(S ; P1) ; while T do P1 od`. Next `(S ; P1)[T]` should be checked. Until `S[T]` is false, `P1` is executed again and again, i.e. a state `S` is updated to the new state `S ; P1` repeatedly.

4. Execution and Verification

Because CafeOBJ is an executable algebraic specification language, we can execute programs of *IPL*. Execution of CafeOBJ is done according to term rewriting [1]. In the execution stage, each equation is regarded as a left-to-right rewrite rule, and an input term is reduced into a term which has no redex (which is a subterm a rewrite rule can be applied).

The following is an example of program executions which calculate 2^{10}.

```
SEM> reduce S ;
 'X := 10 ;
 'Y := 1 ;
 while (not 'X = 0)
 do
   ('Y := 'Y * 2);
   ('X := 'X - 1)
 od
 ['Y]
-- reduce in SEM : (((S ; ('X := 10...
 ...
1024 : NzNat
```

We can do not only execution of programs but also verification of some desired properties semi-automatically. Before showing an example of verification, we give a property which helps us to do verification, especially those for program equivalence.

One of the special features of a behavioral specification is the notion of the behavioral equivalence.

Definition 4.1: [3] The states s_1 and s_2 are behaviorally equivalent, denoted by $s_1 \sim s_2$, if and only if they can not be distinguished by any observations and actions, i.e. $o(a_1(\cdots a_n(s_1)\cdots)) = o(a_1(\cdots a_n(s_2)\cdots))$ for each observation o and sequence of actions $a_1, \ldots, a_n$.

The behavioral equivalence of *IPL* means the equivalence relation of states of Semantic system. By using the notion of the behavioral equivalence we define an equivalence relation for programs. A program is regarded as a function taking a state and returning a new state. Thus, programs are equal when the result states of applying the programs are equal.

Definition 4.2: Programs p_1 and p_2 are equal, denoted by $p_1 \approx p_2$, if and only if $s; p_1 \sim s; p_2$ for each state s.

Verifying given programs to be equal directly is hard since we should consider all expressions and sequences of programs. According to the definitions, to prove programs equal, i.e. $p_1 \approx p_2$, we should check `S;` p_1; `P1`; $\cdots$; `Pn[E]` = `S;` p_2; `P1`; $\cdots$; `Pn[E]` for any expression `E`, sequence of basic programs `P1`, $\ldots$, `Pn`, state `S`. To reduce the task we show the following useful sufficient condition.

[†]`beq` (`bceq`) stands for a (conditional) behavioral equation, which will be explained later.

Theorem 4.3: In SEM, $s_1 \sim s_2$ if and only if s_1[X] = s_2[X] for each variable X. Thus, $p_1 \approx p_2$ if and only if (S; p_1) [X] = (S; p_2) [X] for each variable X and state S.

To prove Theorem 4.3, we introduce the notion of behavioral congruence relations [3], [5].

Definition 4.4: [3] A binary relation R on states is behaviorally congruent if and only if
(1) $s\ R\ s' \Rightarrow o(s) = o(s')$ for each observation o.
(2) $s\ R\ s' \Rightarrow a(s)\ R\ a(s')$ for each action a.

Proposition 4.5: [3] $s\ R\ s' \Rightarrow s \sim s'$.

Now we prove Theorem 4.3 through Proposition 4.5.
Proof of Theorem 4.3: The latter part is trivial if the former part holds. We prove that $s_1 \sim s_2 \Leftrightarrow \forall$X.$s_1$[X] = s_2[X].
[⇒]: Trivial
[⇐]: We show that $s_1\ R\ s_2 \overset{\text{def}}{\Longleftrightarrow} \forall$X.$s_1$[X] = s_2[X] is behaviorally congruent.
(1) Each observation can be written as _[E] or _[T] for an integer expression E and a Boolean expression T. We first prove $\forall$E.s_1[E] = s_2[E] by the structural induction on E.
Base Case :
When E is a variable X, it is trivial from the assumption. When E is an integer I, s_1[I] = I = s_2[I].
Induction Step :
In the case of E = E1 + E2,
s_1[E]
= s_1[E1 + E2]
= s_1[E1] + s_1[E2]
= s_2[E1] + s_2[E2] from I.H.
= s_2[E1 + E2]
= s_2[E] .
The other cases (E = E1 * E2, etc) are similar to this case. Next we show $\forall$T.s_1[T] = s_2[T] by the structural induction on T where T is an arbitrary Boolean expression. The proof is similar to the case of E. In the case of T = E1 = E2,
s_1[T]
= s_1[E1 = E2]
= s_1[E1] =i= s_1[E2]
= s_2[E1] =i= s_2[E2] from the case of E
= s_2[E1 = E2]
= s_2[T] .
In the case of T = not(T1),
s_1[T]
= s_1[not(T1)]
= not s_1[T1]
= not s_2[T1] from I.H.
= s_2[not(T1)]
= s_2[T] .
(2) Each action can be written as _ ; Y := E for a variable Y and an integer expression E. We show $\forall$X. (s_1; Y := E)[X] = (s_2; Y := E)[X] under the assumption of $\forall$X.s_1[X] = s_2[X].
If X = Y, then
(s_1;(X := E))[X]
= s_1[E]
= s_2[E] from (1)
= (s_2;(X := E))[X]
If X ≠ Y, then
(s_1;(Y := E))[X]
= s_1[X]
= s_2[X] from the assumption
= (s_2;(Y := E))[X]
Thus, R is behaviorally congruent, and $s_1 \sim s_2$. □

Thanks to Theorem 4.3 we can verify the equivalence of programs by CafeOBJ easily, which is one of the significant benefits of this paper since the existing algebraic semantics [4] does not treat equivalence of programs well.

Verification of CafeOBJ specification is done by describing a proof score which consists of

```
open SP
op c1 :  -> Elt . ··· .  op ci :  -> Elt .
eq l1 = r1 . ··· .  eq lj = rj .
red t1 . ··· .  red tk .
close
```

where SP is a name of a module where the proof is done. $\overrightarrow{l_j = r_j}$ and $\vec{t_k}$ are an antecedent and a consequent of the proof respectively[†]. Terms $\vec{l_j}$,$\vec{r_j}$ and $\vec{t_k}$ may have constant operations $\vec{c_i}$ as bound variables in the proof. Thus, the meaning of this proof score is an implication $\forall c_1, \ldots, c_i \in$ Elt. $l_1 = r_1 \wedge \cdots \wedge l_j = r_j \Rightarrow t_1 \wedge \cdots \wedge t_k$. If the CafeOBJ system returns true for each reduction, then the proposition is guaranteed to be correct.

More complicated verification can be obtained by combining implication propositions. A case splitting of a proof of P is done by separating P into $Q \Rightarrow P$ and $\neg Q \Rightarrow P$. A use of lemma to prove a proposition P is done as follows: find a suitable proposition (lemma) Q, prove Q, and prove $Q \Rightarrow P$ after proving Q. The following is a way to prove $\forall I \in$ Int.$P(I)$ by the induction on the structure of terms:

```
open INTEGER
red P(0)
op i :  -> Int .
eq P(i) = true .
red P(s(i)).
red P(p(i)).
close
```

First reduction is the base case and the remaining part is the induction step.

Whether a suitable proof score can be described or not is depending on the ability of the verifier. The CafeOBJ system is a tool to support that task. From the experience, difficulty of describing a proof score depends on a property of a proposition to be proved rather than the size of the proposition (or a program). Only one reduction may verify a large proposition. Thousands of lines may be needed to prove a simple proposition.

Example 4.6: Figure 1 is a verification of equivalence of two swap programs defined differently. The swap programs are exchange the value of Variable 'X into the value of Vari-

[†]We often write a sequence of $a_1, \ldots, a_n$ as $\vec{o_i}$.

input

```
open SEM .
ops swap1 swap2 : -> Pgm .
eq swap1 = ('T := 'X) ; ('X := 'Y) ; ('Y := 'T) .
eq swap2 = ('X := 'X + 'Y) ; ('Y := 'X - 'Y) ; ('X := 'X - 'Y) .

op s : -> Sys . op z : -> Var .
eq z =v= 'X = false . eq z =v= 'Y = false . eq z =v= 'T = false .
...
red s ; (swap1 ; 'T := 0) ['X]  == s ; (swap2 ; 'T := 0) ['X] .
red s ; (swap1 ; 'T := 0) ['Y]  == s ; (swap2 ; 'T := 0) ['Y] .
red s ; (swap1 ; 'T := 0) ['T]  == s ; (swap2 ; 'T := 0) ['T] .
red s ; (swap1 ; 'T := 0) [ z]  == s ; (swap2 ; 'T := 0) [ z] .
close
```

output

```
-- opening module SEM.. done.__*
-- reduce in %SEM : (s ; (swap1 ; ('T := 0))) [ 'X ]...
true : Bool ...
-- reduce in %SEM : (s ; (swap1 ; ('T := 0))) [ 'Y ]...
true : Bool ...
-- reduce in %SEM : (s ; (swap1 ; ('T := 0))) [ 'T ]...
true : Bool ...
-- reduce in %SEM : (s ; (swap1 ; ('T := 0))) [ z ]...
true : Bool ...
```

Fig. 1 Verification of program equivalence.

able 'Y vice verse. Variable 'T is a temporary space for the swap program. swap1 is a direct definition of the swap program. swap2 is defined by using a mathematical property over integers. This proof score tries to verify that swap1 ; 'T := 0 and swap2 ; 'T := 0 are equivalent programs.

State s and Variable z mean an arbitrary state and an arbitrary variable in this proof score. Three equations stand for the assumption that Variable z is different from each of Variables 'X, 'Y and 'T.

By reduction command (red), we can do equational reasoning. The first reduction tries to prove that the value of 'X of the state after applying the program swap1 ; 'T := 0 to s is equivalent to that of swap2 ; 'T := 0. Note that the last reduction tries to prove the same thing for Variable z which is different from the other variables. All reductions return true when loading this proof score into CafeOBJ system. Thus four equations hold, which means that s ; swap2 ; 'T := 0 [x] = s ; swap2 ; 'T := 0 [x] for each state s and variable x. Therefore swap1 ; 'T := 0 and swap2 ; 'T := 0 are equivalent from Theorem 4.3.

5. User-Defined Types

In this section we give a way to add user-defined types other than integer and Boolean types. When a user desires to add a new data type which can be described as a specification for a data description (mod!{...}), it is described and is imported to *IPL*, like pr(EQ-INT) in EXP. It is straightforward to define such data types. In this section, we focus on a way to define a kind of classes in object-oriented terminology[†].

We explain a rough sketch to introduce Class Buffer in *IPL*. We use BUFFER shown in Sect. 2.2 (with replacing the imported module Integer into EQ-INT). A module BVAR is a module of buffer variables, where B I is a buffer variable for each integer I and there exists an equality predicate =b=. BEXP, BPGM and BSEM are extensions of EXP, PGM and SEM. In the following we only show the addition part.

```
mod! BEXP{ pr(VAR + BVAR + BUFFER) ...
 op val : BVar -> Exp ...
}
mod! BPGM{ pr(BEXP) ...
 ops up dn  : BVar -> BPgm ...
}
mod* BSEM{ pr(BPGM) ...
 bop _[_] : Sys BVar -> Buf ...
 vars B1 B2 : BVar ...
 eq S[val(B1)] = val(S[B1]) . ...
 beq  S ; up(B1) [B1] = up( S [B1]  ) .
 bceq S ; up(B1) [B2] = S[B2] if B1 =b= B2 .
 beq  S ; dn(B1) [B1] = dn( S [B1]  ) .
 bceq S ; dn(B1) [B2] = S[B2] if B1 =b= B2 .
 ...
 eq   S ; (up(B1)) [X:Var]  = S[X] .
```

[†]Strictly speaking, the notion of a class is not included in "imperative programs." However, we hope our method will be useful to describe and verify practical programs. For that purpose, we think introducing the object-oriented techniques in *IPL* is very important. The introducing a class is a first step for *IPL* to be an OO language.

```
  eq   S ; (dn(B1)) [X:Var]  = S[X] .
  beq  S ; X := E1 [B1] = S[B1] .
}
```

In BEXP, val is added, and val(B) is a new expression for each buffer variable B. In BPGM, new basic programs up(B) and dn(B) are added. In BSEM, the meaning of those expressions and programs are defined. Semantics are given through BUFFER. The first equation in BSEM means that the value of "Expression" val(B1) is the integer kept in "Buffer" S[B1]. The next two equations mean that the value of "Buffer variable" B2 after applying "Program" up(B1) is the buffer up (S[B2]) if B1 = B2, otherwise, it is S[B2] itself. Note that S[B2] is a buffer defined in BUFFER. The meaning of Program dn(B1) is similar with that of up(B1). The last three equations define the remaining cases. A basic program up(B) (or dn(B)) does not change a value of "Integer variable" X and an assignment program does not change a value of "Buffer variable" B1.

Example 5.1: The following proof score shows that the value of an expression val (B 0) after applying a program up (B 0) twice is 2 under the assumption that the observed values of all buffer are init, which returns true.

```
open BSEM .
op s : -> Sys .
eq s [B I:Int] = init .
red s ; up(B 0) ; up(B 0) [val (B 0)] == 2 .
close .
```

We generalize the way to define a buffer class. We give a way to introduce a class, which can be defined as a behavioral specification, to *IPL*.

1. Assume a behavioral specification BSP, which has a hidden sort H, observations $\vec{o}_i$ and actions $\vec{a}_j$.
2. Define a specification HVAR of variables for H and import HVAR and BSP to a specification of expressions HEXP. In HEXP, declare operations o_i : HVar → Exp for each observation o_i : H → V in BSP, that means that new expressions $o_i(h)$ are added to *IPL* where h is a variable of HVAR.
3. In a specification of programs HPGM, declare operations a_i : HVar → Bpgm for each action a_i : H → H in BSP, that means that $a_i(h)$ is a basic program in *IPL*.
4. Lastly, declare equations to define semantics of those expressions and programs in a specification of semantics HSEM. For each expression $o_i(h)$, declare eq S[$o_i(h)$] = o_i(S[h]). For each program $a_i(h)$, declare beq S ; $a_i(h)$ [h] = a_i(S[h]) and bceq S ; $a_i(h)$ [h'] = S[h'] if h =h= h'. In addition to those equations, we need the following equations: eq S ; $a_i(h)$ [X] = S[X] and beq S ; X := E1 [h] = S[h]. The former means that the value of "an integer variable" X is unchanged by a program $a_i(h)$, and the latter means that the value of a variable h of HVar is unchanged by an assignment program.

We revise an equivalence relation on programs. After introducing a class to *IPL* according to our methodology, a new variable (of a class) is added. Thus Theorem 4.3 should be reconsidered because only integer variables are considered as it is. It is modified as follows:

Theorem 5.2: In $HSEM$, $s_1 \sim s_2$ if and only if s_1[X] = s_2[X] for each integer variable X and s_1[H] ~ s_2[H] for each class variable H . Thus, $p_1 \approx p_2$ if and only if (S; p_1) [X] = (S; p_2) [X] for each integer variable X and state S and (S; p_1) [H] ~ (S; p_2) [H] for each class variable H and state S.

Proof. Similar with the proof of Theorem 4.3. □

Note that observed values S[H] of class variables H should be compared by a behavioral equivalence for the imported behavioral specification BSP since they are of a hidden sort in BSP.

In BUFFER, we can easily prove that $s_1 \sim s_2$ if and only if val(s_1) = val(s_2). When the equivalence of each observed values implies the behavioral equivalence in the imported behavioral specification BSP, the following property holds.

Theorem 5.3: If $s_1 \sim s_2$ if and only if $\forall o_i.o_i(s_1) = o_i(s_2)$ in BSP then in $HSEM$, $p_1 \approx p_2$ if and only if (S; p_1) [X] = (S; p_2) [X] for each integer variable X and state S and (S; p_1) [o_i(H)] = (S; p_2) [o_i(H)] for each observation o_i, class variable H and state S.

Proof. Straightforward from Theorem 5.2. □

Now we try to prove the programs up(B 0); dn(B 0) and dn(B 0); up(B 0) to be equivalent. We open BSEM, declare an arbitrary state s and label the above programs updn and dnup. We declare a trivial lemma, whose proof is omitted, on integers.

```
open BSEM .
op s : -> Sys .
ops updn dnup : -> Pgm .
eq updn = up(B 0) ; dn(B 0) .
eq dnup = dn(B 0) ; up(B 0) .
eq p(s(I:Int)) = s(p(I)) .
```

First we try to prove the observed values of the expression val(B 0) after applying the programs to be equivalent.

```
red s ; updn[val(B 0)] == s ; dnup[val(B 0)].
```

Next we try to prove the observed values of the expression val(B i) after applying the programs to be equivalent where B i stands for the all other buffer variable B i (≠ B_0).

```
op i : -> Int .
eq i =i= 0 = false .
red s ; updn[val(B i)] == s ; dnup[val(B i)].
```

Lastly we try to prove the observed values of an arbitrary integer variable x after applying the programs to be equivalent.

```
op x : -> Var .
red s ; updn [x] == s ; dnup [x] .
close
```

All reductions return true in CafeOBJ System. Thus we can conclude those programs are equivalent from Theorem 5.3.

Example 5.4: We can introduce arrays to *IPL*. The following is a behavioral specification of arrays.

```
mod* ARRAY{
  pr(EQ-INT)
  *[Array]*
  op _[_] : Array Int -> Int
  op _[_]=_ : Array Int Int -> Array
  var A : Array
  vars I J K  : Int
  ceq (A[I]= J)[K] = J    if I =i= K .
  ceq (A[I]= J)[K] = A[K] if not I =i= K .
}
```

An array `A` is indexed by integers and array elements are also integers. `A[I]` is an integer of `I`-th element of `A`. `A[I]= J` is an array where `A[I]` is changed into `J` and the others are unchanged. The following is an example of execution.

Input:

```
open ARRAY .
op a : -> Array .
eq a[I:Int] = 0 .
red a[1] .
red (a[1]= 10)[1] .
close
```

Output:

```
-- opening module ARRAY.. done._*
-- reduce in %ARRAY : a [ 1 ]
0 : Zero
-- reduce in %ARRAY : (a [ 1 ] := 10) [ 1 ]
10 : NzNat
```

We re-define specifications `EXP`, `PGM` and `SEM` to add an array expression and an array program to *IPL*. We name those re-defined specifications `AREXP`, `ARPGM` and `ARSEM` as follows:

```
mod! AREXP{ ...
 pr(VAR + ARVAR + EQ-INT + ARRAY)
 op _[_] : ArVar Exp -> Exp
 ...
}
mod! ARPGM{ ...
 op _[_]=_ : ArVar Int Exp -> BPgm
 ...
}
mod* ARSEM{ ...
 bop _[_] : Sys ArVar -> Array
 vars A1 A2 : ArVar
 ...
 eq S[ A1[E1] ] = (S[A1])[ S[E1] ] .
 beq  S ; (A1[I]= E1)[A1]
 = (S[A1])[ S[I] ]= S[E1] .
 bceq S ; (A1[I]= E1)[A2] = S[A2]
 if not (A1 =a= A2) .
 eq   S ; (A1[I]= E1)[X]  = S[X] .
 beq  S ; X := E1 [A1] = S[A1] .
}
```

The following is an example of execution on `ARSEM`. For an array variable `(A 1)`, first insert `1` into the first cell (0), next insert the value of the first element plus `2` into the second cell (1), lastly observe the second cell (1) of the result array, which should be `3`.

```
select ARSEM .
ARSEM> red S ;
   (A 1) [0]= 1 ;
   (A 1) [1]= ((A 1)[0]) + 2
[(A 1) [1]] .
-- reduce in ARSEM : ((S ; (A 1 ...)))
...
3 : NzNat
```

6. Related Work

A specification of *IPL* is an algebraic semantics for imperative programs. Our work has the following advantages over other non-algebraic semantics for imperative programs.

Applicable: The specification does not fix a model (an implementation). For example, a famous semantics using a storage [7] restricts a target implementation into those using a storage. Our model can be applied to all models (implementations) satisfying the axiom of our behavioral specification.

Executable: Thanks to CafeOBJ rewrite engine based on the term rewriting system, not only specifying imperative programs but also executing them can be done. Moreover we can verify a desired properties, especially equivalence of programs, semi-automatically.

Readable: Equational specifications are easy to understand since all definitions and verification are constructed by equations. We can easily review the result of a proof score. When verification fails, it is easy to identify a reason of the failure by tracing failed execution of the proof. The trace helps us to discover a needed lemma, etc. Behavioral specifications describe only behaviors rather than detail structures. Thus, specifications can be very simple and short. Almost all specifications we have described for *IPL* are in the above sections.

Extensible: As we showed in the section of user-defined types, *IPL* can be extended by importing specifications. That is one of the most important benefits of the module-system of CafeOBJ specification languages. Besides the user-defined types, as another proof of extensibility of *IPL* we also succeeded to introduce concurrent programs to *IPL*, by importing a specification of non-deterministic integers.

One of the most relevant researches to our work is algebraic semantics of imperative programs by Goguen and Malcom [4]. Main differences are (1) that we give the semantics by using behavioral specifications (i.e. based on Hidden algebra), and (2) that we propose a way to introduce a kind of classes of OO languages to *IPL*. Most important advantage of our work is to formalize an equivalence

relation on programs by using the notion of the behavioral equivalence, and to propose a sufficient condition (Theorem 4.3, 5.2, and 5.3) for program equations, which gives us a way to prove program equivalence easily.

7. Conclusion

We gave an algebraic semantics of imperative programs through Hidden algebra, that is, a behavioral specification of *IPL* was given. We formalized an equivalence relation on programs and proposed a property to reduce a task of proving program equivalence. We also gave a way to introduce a new class to *IPL*. In our specification, a class we can treat is restricted, that means that only classes described as a behavioral specification can be treated. A future work is to give a way to introduce complete classes of OO languages, in which we can describe message passing between objects, etc.

References

[1] F. Baader and T. Nipkow, Term rewriting and all that, Cambridge University Press, 1998.
[2] R. Diaconescu and K. Futatsugi, CafeOBJ Report, World Scientific, 1998.
[3] R. Diaconescu and K. Futatsugi, "Behavioural coherence in object-oriented algebraic specification," The Journal of Universal Computer Science, vol.6, no.1, pp.74–96, 2000.
[4] J. Goguen and G. Malcolm, Algebraic Semantics of Imperative Programs, MIT Press, 1996.
[5] J. Goguen and G. Malcolm, "Hidden agenda," Theoretical Computer Science, vol.245, no.1, pp.55–101, 2000.
[6] C.R.A. Hoare, "An axiomatic basis for computer programming," Communications of the Association for Computing Machinery, vol.12, no.10, pp.576–580, 1969.
[7] D. Scott and C. Strachey, "Towards a mathematical semantics for computer languages," Proc. 21st Symposium on Computers and Automata, pp.19–46, 1971.
[8] A.T. Nakagawa, T. Sawada, and K. Futatsugi, CafeOBJ User's Manual – ver.1.4.2 –, http://www.ldl.jaist.ac.jp/cafeobj/, 1999.

Masaki Nakamura received his Ph.D. degree from Japan Advanced Institute of Science and Technology (JAIST) in 2002. Since 2002, he has been an associate of School of Information Science of JAIST. His current research interests include term rewriting, algebraic specification, verification systems, and formal method.

Masahiro Watanabe received his M.IS. degree from Japan Advanced Institute of Science and Technology in 2005. Since 2005, he has been employed in Hitachi Ltd.

Kokichi Futatsugi is a professor at Graduate School of Information Science, JAIST (Japan Advanced Institute of Science and Technology), Ishikawa, Japan. His research interest includes formal methods, software requirements and specifications, and specification languages. An important part of his research activities is done on CafeOBJ specification language. He is a member of the advisory board of Journal of Higher-Order and Symbolic Computation (www.wkap.nl/journals/hosc), and the editorial board of Journal of Object Technology (www.jot.fm) and Journal of Applied Logic (www.elsevier.com/locate/jal).