Semantics of Pointers, Referencing and Dereferencing with Intensional Logic

Hing-Kai Hung

380 Campus Drive #3
Snyder, NY 14226, USA
hung@cs.buffalo.edu

Jeffery I. Zucker

Department of Computer Science & Systems
McMaster University
Hamilton, Ont. L8S 4K1, Canada
zucker@maccs.dcss.mcmaster.ca

Abstract

We apply intensional logic to the semantics of an Algol-like programming language. This associates with expressions their senses, or meanings relative to "possible worlds", here interpreted as machine states. These meanings lie in the semantic domains of a higher order typed intensional logic.

The great advantage of this approach is that it preserves compositionality of the meaning function, even in "opaque contexts".

Earlier work in this direction, by Janssen and Van Emde Boas, dealt with the semantics of assignments to simple variables, indexed variables and pointers, without, however, considering "dereferenced" pointers on the left or right hand side of assignments. More recent work by Hung applied this approach to the semantics of blocks and procedures with parameters (passed by value, by reference and by name).

The present work extends this approach to pointers, including dereferenced pointers on both sides of assignments. It is shown how this approach gives an elegant solution to the problem of pointer semantics, which is simple, compositional and implementation independent.

1 Background

1.1 Pointers and References

Most imperative programming languages such as Algol 68, Pascal and C, have a basic command called *assignment* which is of the form

$$expr_1 := expr_2.$$

The expression $expr_1$ delivers a *reference* and the expression $expr_2$ delivers an object. The assignment causes the reference to possess the object as its single value, and the object can then be accessed through the reference by an action called *dereferencing* [Reynolds 70]. As the subsequent computation may cause the reference to possess different values, $expr_1$ is called a *variable* when it is an identifier. In implementation, references are usually addresses of memory locations, and the action of fetching the memory contents corresponds to dereferencing. Note that the concepts of *reference* and *variable* as used here are different, respectively, from those of *reference* (denotation) as used in philosophy, and *variable*, as used in mathematical logic.

To provide the advantages of local updatable states and data sharing in functional programming, references can sometimes be created and manipulated. Both Standard ML [Reade 89] and Scheme have the primitive **ref** which creates updatable references to objects. Only assignments can be used to update the references.

A reference, which on dereferencing returns another reference, is called a *pointer*. Indirect addressing is just the implementation of pointers.

1.2 Blocks, Procedures and Parameter Passing

A *block* is a definition mechanism in which identifiers are declared and the declared identifiers are then visible inside the statement which follows the declarations. Nested blocks are possible, so that the same identifier name can be redeclared inside the nested declarations. *Procedure declarations* allow the programmer to package computations and parameterize their behavior. Our syntax for these is as follows.

Blocks :

begin new x := e ; S **end**

begin alias y = v ; S **end**

begin macro z = v ; S **end**

In the *new-block* above, the new identifier x is declared with the initial value e and S is the body (statement) of the block. The statement S is also referred to as the *scope* of the identifier x. It is the region of text over which x is in effect. This resembles λ-binding in the lambda calculus and quantification in first order

*This research was supported by the National Science Foundation under grant no. DCR-8504296, by a grant from the Science & Engineering Research Board of McMaster University, and by a grant from the National Sciences and Engineering Research Council of Canada

logic. In the *alias-block*, y is set equal to the reference v and hence both y and v possess the same object. The *macro-block* behaves the same as the statement obtained by replacing all occurrence of (**expand** z) in S by v. We call z a *macro name* and (**expand** z) a name (macro) *expansion*.

Procedure Declarations :

$$\mathtt{G}^v \Leftarrow < \textbf{value } \mathtt{x} > \quad \mathtt{S} \textbf{ end}$$

$$\mathtt{G}^r \Leftarrow < \textbf{ref } \mathtt{y} > \quad \mathtt{S} \textbf{ end}$$

$$\mathtt{G}^n \Leftarrow < \textbf{name } \mathtt{z} > \quad \mathtt{S} \textbf{ end}$$

where $\mathtt{G}^v$, $\mathtt{G}^r$, $\mathtt{G}^n$ are procedure identifiers, S the procedure bodies (statements), x is a pass-by-value parameter, y is a pass-by-reference parameter, and z is a pass-by-name parameter. All the parameters in the declaration are called *formal parameters*. They have scopes over the procedure body. Procedures can be invoked by procedure calls, say $\mathtt{G}^v(1+2)$, $\mathtt{G}^r(\mathtt{a})$, $\mathtt{G}^n(\mathtt{b})$. We call 1+2, a, b the *actual parameters*.

We consider three different parameter passing mechanisms, as follows. (1) *Pass by Value*: The value possessed by the actual parameter is copied and possessed by the formal parameter. (2) *Pass by Reference*: The actual parameter is a reference. The formal parameter shares the same reference as the actual parameter and both of them possess the same object. (3) *Pass by Name*: This can be thought of as a textual substitution of all free occurrences of the formal parameter in the procedure body by the actual parameter.

We adopt a design principle for programming languages called the *Correspondence Principle*, which states that for any parameter passing mechanism, there should be a corresponding definition (block) mechanism, and vice versa [Tennent 81]. Here pass-by-value corresponds to new-block, pass-by-reference corresponds to alias-block, and pass-by-name corresponds to macro-block. This principle will be illustrated by the semantic correspondences between blocks and procedures (Section 4).

1.3 Compositionality Principle

A *meaning* for expressions of a language is a function $[\![\cdot]\!]$ mapping syntactic entities to semantic entities. Let $\triangle$ be a syntactic domain, and $\mathcal{D}$ the corresponding semantic domain. For any $\delta \in \triangle$, we write $[\![\delta]\!] \in \mathcal{D}$ to denote the meaning of δ. We assume that all compound expressions of a language are constructed from primitive expressions by finitely many applications of certain *syntactic operators*.

The meaning function of the language is said to satisfy the *compositionality principle* if for every such n-ary syntactic operator $\Phi : \triangle_1 \times \ldots \times \triangle_n \rightarrow \triangle_{n+1}$, there exists a function $\Psi : \mathcal{D}_1 \times \ldots \times \mathcal{D}_n \rightarrow \mathcal{D}_{n+1}$ such that for every compound expression $\Phi(\delta_1, \ldots, \delta_n) \in \triangle_{n+1}$ formed from the components $\delta_1 \in \triangle_1, \ldots, \delta_n \in \triangle_n$ with the syntactic operator Φ,

$$[\![\Phi(\delta_1, \ldots, \delta_n)]\!] \;=\; \Psi([\![\delta_1]\!], \ldots, [\![\delta_n]\!])$$

This principle says that the meaning of a compound expression is obtained from the meanings of its parts. It is carefully formulated and discussed in [Janssen 86]. Its significance for the correctness and verification of concurrent systems is discussed in [Zwiers 89].

1.4 Earlier Approaches

The semantics of pointers and references are difficult to handle [Manna & Waldinger 81]. One of the reasons is that it is simpler to deal with expressions whose values are *rigid* (§3.3). However, references can be updated and may possess different values at different states, and so fail to be rigid in general.

Motivated by the compositionality of intensional logic and the fact that the left hand side of an assignment statement is referentially opaque, [Janssen & Van Emde Boas 77] and [Janssen 86] applied intensional logic to the semantics of assignments. Assignments to number pointers were considered, but not dereferenced pointers on the left or right hand side of assignments.

In the [Milne & Strachey 76] approach, *addresses* are introduced as domain objects. The interpretation of program expressions is relative to *environments* and *stores*. The environments map identifiers into addresses (L-values), and the stores map addresses into values (R-values). Such interpretation is very complicated when *aliasing* occurs, that is, when two identifiers possess the same address. It also has a disadvantage that two program expressions which although behave the same operationally, may have different meanings [Brookes 85]. Using the intensional logic approach, the treatment for two identifiers possessing the same address is like that for two variables possessing the same value, and no complexity is added to the interpretation.

Blocks and procedures are also difficult to handle — most semantic attempts involve modeling local memory allocation in terms of stacks. Interesting approaches include [Brookes 85], [Oles 85], [Meyer & Sieber 88]. In all these models, however, pointers and parameter passing mechanisms were not considered.

[Wegner 89] provided a uniform treatment of blocks, data types, and pointers, in terms of transformations defined using pushouts and reducts of algebra. Declarations of procedures were not, however, considered.

Parameter passing was modeled by syntactic application in [De Bakker 80]. The semantics, however, is then not compositional.

Reynolds' specification logic [Reynolds 81] did not deal with pass-by-reference parameters.

Pursuing the direction of [Janssen & Van Emde Boas 77] and [Janssen 86], [Hung 89] and [Hung 90, Chapter 2] observed that there are also intensional contexts within parameter passing, and achieved a compositional semantics for a language with *blocks*, *procedures* and *parameters*. Pointers were, however, not considered because of the difficulties in obtaining "state switcher free" conditions (§3.2).

1.5 Achievement of Present Work

We use intensional logic to develop a compositional semantics which models *pointers* and *references*, in addition to blocks, non-recursive procedures, and the three parameter passing mechanisms considered above.

Although pointers were considered in [Janssen & Van Emde Boas 77], they occur there only on the left hand side of assignments, since there is no dereferencing facility there. We show how dereferencing can be incorporated into the formalism (see Section 4), thus permitting dereferenced pointers on the left and right hand sides of assignments.

The inclusion of pointers in our programming language in this way is important from the viewpoint of programming language theory. It leads to non-trivial modifications in the formulations and proofs of the main results (see Section 5) compared with [Hung 90], as well as to the new Substitution Lemma (Lemma 6) of Section 5.

We will see how this approach gives an elegant solution to the problem of pointer semantics, which is simple, compositional and implementation independent.

2 Introduction

2.1 Intensional Logic

There are two important notions in connection with the meaning of expressions in natural languages: (*i*) its *denotation*, *reference* or *extension*, and (*ii*) its *sense* or *intension*. For example, the denotation of a sentence is its truth value.

Frege was one of the first who investigated the notion of sense [Frege 92]. Here is a well-known example. Compositionality implies the principle of "substitutivity of equals", which states that substituting one term for another with the same meaning in an expression should not alter the meaning of that expression. But consider the sentence

Necessarily the morning star is the morning star.

The sentence is obviously true. Now the terms 'the morning star' and 'the evening star' have the same denotation, the planet Venus, but different senses, since we can easily imagine a world in which they refer to different objects. Replacing the second 'the morning star' by 'the evening star' in the above sentence, we get

Necessarily the morning star is the evening star,

which is false! Hence the substitution of a co-denotational name in the sentence does not preserve its meaning (here: truth value). The text after 'Necessarily' is said to be *referentially opaque* (as opposed to *referentially transparent*, in which the truth value under such a substitution would be preserved). Frege's insight was to propose that the meaning of a term in an opaque context is *not* its denotation, but its sense. Since 'the morning star' and 'the evening star' have different meanings in this context (i.e., senses), we do *not* have a counterexample here to the principle of substitutivity of equals.

Frege's insight might be summarized thus: taking senses as meanings in opaque contexts, we *restore* the principle of substitutivity of equals, and hence *compositionality of meaning*.

By means of intensional logic [Church 51], we can reason about intensions or senses of expressions. A mathematical semantics for intensional logic was given by Kripke [Kripke 63], who defined the sense of an expression as a function from *possible worlds* to (extensional) values.

Montague [Montague 74] used Frege's insight to give a compositional semantics for a fragment of English, including opaque contexts, by means of a translation of that fragment into intensional logic.

Next, Janssen and Van Emde Boas [Janssen & Van Emde Boas 77, Janssen 86] used a similar approach to provide a compositional semantics for a fragment of an Algol-like programming language. Their insight was that the left-hand side of an assignment can again be viewed as an opaque context, and a compositional semantics can be given, similarly, by a translation of the fragment into intensional logic. In this case, the *sense* of an expression (say an identifier of number type) is again a function from "possible worlds" into objects (numbers), where each "possible world" is now a *computation state*. Thus the meaning of the identifier can be taken as a *memory address* or *location*.

Extending this approach, Hung [Hung 89, 90] provided a compositional semantics for procedures with parameters (passed e.g. by value and by reference), noticing now that *pass-by-reference* provides an opaque context.

In the present paper, we extend this method to deal with yet other opaque contexts: assignments to *pointer identifiers*, and *pass-by-name*. Again, we provide compositional semantics for these.

Our framework for intensional logic, **IL** (along the lines of [Montague 74], [Gallinn 75], [Dowty *et al.* 81] and [Janssen 86]) is a higher order typed logic. There are two basic types, $\mathtt{N}$ and $\mathtt{B}$, which correspond respectively to the natural numbers and booleans (truth values). The higher order types of **IL** are compound types of the form $(\tau_1 \to \tau_2)$ which are the types of functions from τ_1 objects to τ_2 objects. Senses of expressions have type $(\mathtt{S} \to \tau)$, which is the type of functions from possible worlds or states ($\mathtt{S}$ objects) to τ objects. **IL** does not allow possible worlds as domain objects. (The type $\mathtt{S}$ is "hidden").

To repeat, our major motivation in using intensional logic is to provide a *compositional* semantics for a programming language. Full discussions of these issues of intensionality and compositionality in programming languages can be found in [Janssen 86] and [Hung 89, 90].

2.2 Syntax of Our Programming Language

The syntax of our language is now described. Let x, y range over number identifiers, p over pointers, z over macro names, m over number constants, a over array identifiers, G over procedure identifiers. The syntax of program expressions is as follows.

Number References
v ::= x | a[e] | (**expand** z) | (**dref** p)
Number Terms
e ::= m | v | (e_1 + e_2) | (e_1 − e_2) | (e_1 × e_2) | **if** b **then** e_1 **else** e_2 **fi** | ...
Booleans
b ::= true | (e_1 = e_2) | (e_1 > e_2) | ...
Blocks
K ::= **begin new** x := e ; S **end** | **begin alias** y = v ; S **end** | **begin macro** z = v ; S **end**
Statements
S ::= **skip** | v := e | p := v | (S_1 ; S_2) | **if** b **then** S_1 **else** S_2 **fi** | K | G^v(e) | G^r(v) | G^n(v)
Procedure Identifiers
G ::= G^v | G^r | G^n
Procedure Bodies
B ::= B^v | B^r | B^n
where
$B^v \equiv$ < **value** x > S **end**
$B^r \equiv$ < **ref** y > S **end**
$B^n \equiv$ < **name** z > S **end**
Procedure Declarations
D ::= < $G^i \Leftarrow B^i$ > where $i \in \{v, r, n\}$
Programs
R ::= < $\vec{D}$; S >

In the above, (**dref** p) is the *dereference* of the pointer p. (This is a memory address in most language implementations.) The statement p:=v is a *pointer assignment* statement.

We will only consider *well-formed programs*, in which all the procedures are declared before their calls appear, the macro expansions (**expand** z) occur only inside the relevant macro-blocks or pass-by-name procedure bodies, and recursive procedures and nested procedure declarations are not allowed. We do not consider recursive procedures because we do not want to deal with fixpoint domain theory. Similarly, for the purpose of avoiding uninteresting complexity, we do not consider arrays of pointers, or pointers to pointers. We believe there is no difficulty in extending our semantic treatment to these [Janssen 86].

2.3 Correctness Formulas

We consider *correctness formulas* (Hoare formula) of the form { ϕ_1 } S { ϕ_2 }, where S is a program (statement), ϕ_1 is a first order formula which describes the computer states *before* execution of S, and ϕ_2 is a formula which describes the computer states *after* execution of S. When the correctness formula is valid, we write $\models$ { ϕ_1 } S { ϕ_2 }. We consider *total* correctness, since our programs will always halt. The syntax of our assertion language is as follows.

Number Variables $n, \ldots$
Number Terms
$e ::= n \mid$ m $\mid$ x $\mid$ a[e] $\mid$ (**dref** p) $\mid (e_1 + e_2) \mid (e_1 - e_2) \mid (e_1 \times e_2) \mid$ *if* b *then* e_1 *else* e_2 *fi*
Booleans
$b ::=$ *true* $\mid (e_1 = e_2) \mid (e_1 > e_2) \mid \neg b \mid b_1 \wedge b_2$
Assertions
$\phi ::= b \mid \neg\phi \mid (\phi_1 \wedge \phi_2) \mid \exists n \phi$
Hoare Formulas
$F ::=$ { ϕ_1} S{ ϕ_2} | { ϕ_1} R{ ϕ_2}

Note that we exclude name expansion (**expand** z) from the number terms because we do not want to use macro names in assertions. We use $\phi_1 \rightarrow \phi_2$ as an abbreviation for $\neg(\phi_1 \wedge \neg\phi_2)$, and $\phi_1 \vee \phi_2$ as an abbreviation for $\neg(\neg\phi_1 \wedge \neg\phi_2)$. Some examples of valid correctness formulas are:

$$\models \{\exists n\ \mathtt{x} + 1 = 2 \times n\}\ \mathtt{x{:=}x+1}\ \{\exists n\ \mathtt{x} = 2 \times n\}$$

$$\models \{\mathtt{i} = 0 \wedge \mathtt{a}[0] = 0 \wedge \mathtt{a}[1] = 0\}\ \mathtt{a[a[i]]{:=}1}\ \{\mathtt{a[a[i]]} = 0\}$$

Suppose we have the procedure declaration

double ⇐ < **ref** y > p := w; y := y + y **end**

Then

$$\models \{\mathtt{w} = 1 \wedge \mathtt{x} = 1\}\ \mathtt{p{:=}x; double((dref\ p))}\ \{\mathtt{w} = 1 \wedge \mathtt{x} = 2\}.$$

More interesting examples can be found in [Hung 89, 90].

3 Intensional Logic

3.1 Possible Worlds

We identify possible worlds in the context of intensional logic as *states*, or the possible memory configurations of computers. In programming languages, dereferencing a number reference yields a number. We therefore interpret references as functions from states to numbers, of type (S → N). *Arrays* are rows of references, or functions of type (N → (S → N)). A *pointer* is a reference which yields a reference when dereferenced. Hence a pointer to a number has, in a given state, a reference as value. We therefore treat pointers as functions from states to references, of type (S → (S → N)) [Janssen & Van Emde Boas 77]. A *macro name* denotes all possible values of the name in different contexts. We therefore treat it as the sense of a reference, so that it also has type (S → (S → N)).

We denote the set of states (possible worlds) by $\mathcal{S}tate$ and its elements by $\sigma, \ldots$, the set of natural numbers by $\mathcal{N}$, the set of boolean values by $\mathcal{B}$= {***true***,***false***}. Let VAR_τ be the set of variable symbols of type τ, CON_τ the set of constant symbols of type τ, and ME_τ the set of meaningful **IL** expressions of type τ. We have three disjoint countable sets:

$REF \subseteq CON_{(\mathsf{S} \to \mathsf{N})}$, $ARRAY \subseteq CON_{(\mathsf{N} \to (\mathsf{S} \to \mathsf{N}))}$, and $PTR \subseteq CON_{(\mathsf{S} \to (\mathsf{S} \to \mathsf{N}))}$, where REF is the set of *reference symbols*, $ARRAY$ is the set of *array symbols* and PTR is the set of *pointer* symbols. Further, the set REF is a union of two disjoint sets REF_s and REF_a, where REF_s are reference symbols for free simple variables in programs, and REF_a are reference symbols for array elements. Intuitively, these sets of constant symbols will play the roles of *memory addresses* or *locations*.

For each $A \in ARRAY$ and $\boldsymbol{k} \in \mathcal{N}$, we can associate a reference symbol $A^{\boldsymbol{k}} \in REF_a$ such that

$$A_1 \not\equiv A_2 \text{ or } \boldsymbol{k}_1 \neq \boldsymbol{k}_2 \;\Rightarrow\; A_1^{\boldsymbol{k}_1} \not\equiv A_2^{\boldsymbol{k}_2}.$$

We define the structure of $\mathcal{S}tate$ in two stages using $\mathcal{S}tate_0, \mathcal{S}tate_1$ as follows.

The set $\mathcal{S}tate_0$ is a non-empty set of functions $\sigma : REF \to \mathcal{N}$, closed under *variant* or *update*, where for any $\sigma \in \mathcal{S}tate_0$, $X \in REF$, and $\boldsymbol{n} \in \mathcal{N}$, the variant of σ at X, denoted by $\sigma\{X/\boldsymbol{n}\}$, is defined by:

$$\sigma\{X/\boldsymbol{n}\}(X) = \boldsymbol{n}$$

and

$$\sigma\{X/\boldsymbol{n}\}(Y) = \sigma(Y) \quad \text{for } Y \not\equiv X.$$

The semantics of REF and $ARRAY$ are then as follows. For all $X \in REF$, $[\![X]\!]_0$ is a function from $\mathcal{S}tate_0$ to $\mathcal{N}$ given by

$$[\![X]\!]_0(\sigma) = \sigma(X).$$

For all $A \in ARRAY$, $[\![A]\!]_0$ is a function from $\mathcal{N}$ to $\mathcal{S}tate_0 \to \mathcal{N}$ given by

$$[\![A]\!]_0(\boldsymbol{k})(\sigma) = [\![A^{\boldsymbol{k}}]\!]_0(\sigma).$$

Let $\mathbf{Ref}_0 = \{[\![X]\!]_0 \mid X \in REF\}$ be the set of *references*.

The set $\mathcal{S}tate_1$ is a non-empty set of functions $\sigma : PTR \to \mathbf{Ref}_0$ closed under variants. The meaning of any $P \in PTR$ is:

$$[\![P]\!]_1(\sigma) = \sigma(P).$$

Finally, the set of states $\mathcal{S}tate = \mathcal{S}tate_0 \oplus \mathcal{S}tate_1$ is the set of functions $\sigma : REF \cup PTR \to \mathcal{N} \cup \mathbf{Ref}_0$ such that

$$\sigma\lceil REF \in \mathcal{S}tate_0$$
$$\sigma\lceil PTR \in \mathcal{S}tate_1.$$

The semantics of REF, $ARRAY$ and PTR relative to $\mathcal{S}tate$ can then be defined accordingly (see §3.2).

This construction can be used to define layers of references $\mathbf{Ref}_1$, $\mathbf{Ref}_2, \ldots$, which allow us to extend the semantic values to hierarchies of pointers to pointers, as in [Janssen & Van Emde Boas 77].

Note that there are one-one correspondences between the symbols and their meanings:

Lemma 1

$$X_1 \not\equiv X_2 \;\Rightarrow\; [\![X_1]\!] \neq [\![X_2]\!]$$
$$A_1 \not\equiv A_2 \;\Rightarrow\; [\![A_1]\!] \neq [\![A_2]\!]$$
$$P_1 \not\equiv P_2 \;\Rightarrow\; [\![P_1]\!] \neq [\![P_2]\!]$$

Proof: By the property of state variants. □

3.2 Syntax and Semantics

We now define the formal system **IL** of intensional logic. The symbol set involves + (number plus) ,−, ×,= (number equality), < (less than), ¬ (negation), ∧ (conjunction), ∃ (existential quantifier over numbers) λ (functional abstraction), $\cdot(\cdot)$ (function application), $\hat{}$(intension operator), $\check{}$(extension operator) and $\cdot <\cdot>$ (state switcher). We call $\hat{}$, $\check{}$, and $\cdot <\cdot>$ the *state* or *modal operators*. The terms of **IL** are built up from the basic terms (constants and variables) by the operators in the symbol set, according to the typing rules. An expression is said to be *state switcher free* if it does not contain any state switcher.

A function which maps **IL** variables to values of correct type is called a *valuation*. The set of valuations is denoted by $\mathcal{V}al$ with elements $\rho, \ldots$. The value function of the expressions in **IL** is defined relative to states (possible worlds) σ and valuations ρ. Let x range over **IL** variables, α, β over any **IL** expressions, η over expressions of type $(\mathsf{S} \to (\mathsf{S} \to \mathsf{N}))$, ξ over expressions of type $(\mathsf{S} \to \mathsf{N})$, ε over expressions of type N and $\boldsymbol{k}$ over $\mathcal{N}$. Also let $\rho\{x/\boldsymbol{d}\}$ denote the *function variant* of ρ. The class of expressions in **IL** and their meanings are defined inductively as follows.

1. $[\![0]\!]\sigma\rho = \mathbf{0}$, $[\![1]\!]\sigma\rho = \mathbf{1}$, $[\![true]\!]\sigma\rho = \boldsymbol{true}$
2. $[\![X]\!]\sigma\rho = \boldsymbol{\lambda}\sigma' \cdot \sigma'(X)$
3. $[\![A^{\boldsymbol{k}}]\!]\sigma\rho = \boldsymbol{\lambda}\sigma' \cdot \sigma'(A^{\boldsymbol{k}})$
4. $[\![A]\!]\sigma\rho = \boldsymbol{\lambda k} \in \mathcal{N} \cdot \boldsymbol{\lambda}\sigma' \cdot \sigma'(A^{\boldsymbol{k}})$
5. $[\![P]\!]\sigma\rho = \boldsymbol{\lambda}\sigma' \cdot \sigma'(P)$
6. $[\![x]\!]\ \sigma\rho = \rho(x)$ where $x \in VAR$
7. $[\![\varepsilon_1 + \varepsilon_2]\!]\ \sigma\rho = [\![\varepsilon_1]\!]\sigma\rho + [\![\varepsilon_2]\!]\sigma\rho$
   and similar for − and ×
8. $[\![\varepsilon_1 = \varepsilon_2]\!]\ \sigma\rho = \begin{cases} \boldsymbol{true} & \text{if } [\![\varepsilon_1]\!]\sigma\rho = [\![\varepsilon_2]\!]\sigma\rho \\ \boldsymbol{false} & \text{otherwise} \end{cases}$
   and similar for <.
9. $[\![\neg\varphi]\!]\ \sigma\rho = \begin{cases} \boldsymbol{false} & \text{if } [\![\varphi]\!]\sigma\rho = \boldsymbol{true} \\ \boldsymbol{true} & \text{otherwise} \end{cases}$
10. $[\![\varphi_1 \wedge \varphi_2]\!]\ \sigma\rho$
    $= \begin{cases} \boldsymbol{true} & \text{if } [\![\varphi_1]\!]\sigma\rho = \boldsymbol{true}, \\ & \text{and } [\![\varphi_2]\!]\sigma\rho = \boldsymbol{true} \\ \boldsymbol{false} & \text{otherwise} \end{cases}$

11. $[\![if\ \varphi\ then\ \varepsilon_1\ else\ \varepsilon_2\ fi]\!]\ \sigma\rho$
$= \begin{cases} [\![\varepsilon_1]\!]\sigma\rho & \text{if } [\![\varphi]\!]\sigma\rho = \boldsymbol{true} \\ [\![\varepsilon_2]\!]\sigma\rho & \text{otherwise} \end{cases}$

12. $[\![\exists n\varphi]\!]\ \sigma\rho$
$= \begin{cases} \boldsymbol{true} & \text{if } [\![\varphi]\!]\sigma\rho\{\ n/\boldsymbol{n}\} = \boldsymbol{true} \\ & \text{for some } \boldsymbol{n} \in \mathcal{N} \\ \boldsymbol{false} & \text{otherwise} \end{cases}$

13. $[\![\lambda x \cdot \alpha]\!]\ \sigma\rho = \boldsymbol{\lambda d} \cdot [\![\alpha]\!]\sigma\rho\{\ x/\boldsymbol{d}\}$ where $\boldsymbol{d} \in \mathcal{D}$

14. $[\![\alpha(\beta)]\!]\ \sigma\rho = [\![\alpha]\!]\sigma\rho([\![\beta]\!]\sigma\rho)$

15. $[\![\hat{}\alpha]\!]\ \sigma\rho = \boldsymbol{\lambda}\sigma' \cdot [\![\alpha]\!]\sigma'\rho$

16. $[\![\check{}\alpha]\!]\ \sigma\rho = ([\![\alpha]\!]\sigma\rho)(\sigma)$

17. $[\![\alpha{<}\xi/\varepsilon{>}]\!]\ \sigma\rho$
$= \begin{cases} [\![\alpha]\!]\sigma\{\ X/[\![\varepsilon]\!]\sigma\rho\}\ \rho & \text{if } [\![\xi]\!]\sigma\rho = [\![X]\!] \\ & \text{for some } X \in REF \\ [\![\alpha]\!]\sigma\rho & \text{otherwise} \end{cases}$

18. $[\![\alpha{<}\eta/\xi{>}]\!]\ \sigma\rho$
$= \begin{cases} [\![\alpha]\!]\sigma\{\ P/[\![X]\!]\}\ \rho & \text{if } [\![\xi]\!]\sigma\rho = [\![X]\!] \\ & \text{and } [\![\eta]\!]\sigma\rho = [\![P]\!] \\ & \text{for some } X \in REF, \\ & \text{and some } P \in PTR \\ [\![\alpha]\!]\sigma\rho & \text{otherwise} \end{cases}$

Note that the interpretation of terms is "two-dimensional". Quantification and λ-abstraction are interpreted by means of *valuations* ρ, and state operators are interpreted by means of *states* σ. By Lemma 1, X and P (in clauses 17 and 18) are unique if they exist. Intuitively, $\check{}\alpha$ means the value of α in the *current* state, $\hat{}\alpha$ means the *sense* of α. As we can see, the value of $\alpha{<}X/\varepsilon{>}$ in a state σ is the same as the value of α in the state $\sigma\{\ X/[\![\varepsilon]\!]\sigma\}$. (This is essentially the Substitution Lemma for State Switchers: Lemma 5 in Section 5). The state switcher is thus a semantic representation of the syntactic substitution in finding weakest pre-conditions. It will be used in the translations of assignment statements (see Section 4).

3.3 Rigidity

The value of an **IL** term may vary according to different worlds. A term whose value remains the same in all worlds is called a *rigid designator* [Kripke 63]. In our formalism, a term α is defined to be *rigid* if for all states σ_1, σ_2, ρ, $[\![\alpha]\!]\ \sigma_1\rho = [\![\alpha]\!]\ \sigma_2\rho$.

In the λ-calculus, we can simplify formulas of the form $(\lambda x \cdot \alpha)(\beta)$ by substituting the arguments β for the free occurrences of x in α. (This is *β-conversion.*) This will not be correct in general, if we interpret the execution of a procedure call as the effect of executing the procedure body obtained by replacing all the free occurrences of the formal parameter by the argument (actual parameter). The problem lies in the translation of pass-by-reference, in the substitution of non-rigid terms into "modal" contexts. (In **IL**, only a restricted form of β-conversion is valid: see Theorem 1 below).

Note that constants and variables are rigid in this formalism. In general, program expressions are translated into non-rigid **IL** expressions (see Section 4). For example the expression $a(\check{}x + \check{}y)$ (translation of `a[x+y]`) is non-rigid. One of our techniques in reasoning with parameter passing is to replace non-rigid terms by rigid terms while preserving the overall meaning.

3.4 Theorem on β-conversion

Let α, β range over **IL** terms, σ over $\mathcal{S}tate$ and ρ over $\mathcal{V}al$. Let $\alpha[x/\beta]$ denote the term obtained by replacing all free occurrences of x in α by β.

Definition Two terms α, β are *semantically equivalent*, denoted by $\alpha \cong \beta$, if for all σ, ρ $[\![\alpha]\!]\sigma\rho = [\![\beta]\!]\sigma\rho$.

Theorem 1 (β-conversion) For any term $(\lambda x \cdot \alpha)(\beta)$, if either one of the following conditions holds: (1) β is rigid, (2) no occurrence of x in α lies inside the scope of $\hat{}$or $<>$; then

$$(\lambda x \cdot \alpha)(\beta) \ \cong\ \alpha[x/\beta]$$

Proof: By structural induction on α. See [Hung 90]. □

This says that if *either* the argument β is rigid, *or* no modal operators are applied to the bound variable x, then function applications can be modeled by syntactic substitution. This extends the corresponding β-conversion theorem in [Janssen 86, Dowty *et al.* 81].

For later use, we state

Lemma 2 ($\check{}\hat{}$-cancellation) For any $\alpha \in ME_\tau$,

$$\check{}\hat{}\alpha \ \cong\ \alpha.$$

Proof: By the semantic definitions of $\hat{}$ and $\check{}$. □

4 Translations

The program semantics is given by translating the expressions of the programming language into the expressions in **IL**. The meaning is then obtained by interpreting the **IL** expressions. Since number identifiers are references and we want to treat formal parameters as lambda variables, we translate number identifiers as variables in $VAR_{(\mathsf{S} \to \mathsf{N})}$. We take an array as a row of references. (This is the approach taken by [Wagner 89], in which an array is a tuple of "locations", but different from that in [Janssen 86], in which an array is an intension of a row of numbers). An array identifier `a` is therefore translated into a variable a in $VAR_{(\mathsf{N} \to (\mathsf{S} \to \mathsf{N}))}$. Similarly, a pointer `p` is translated into a variable p in $VAR_{(\mathsf{S} \to (\mathsf{S} \to \mathsf{N}))}$, and a macro name `z` is translated into a variable z in $VAR_{(\mathsf{S} \to (\mathsf{S} \to \mathsf{N}))}$. Realizing that a subset of $\mathcal{S}tate$ can be characterized by a state predicate of type $(\mathsf{S} \to \mathsf{B})$, we regard a program as a backward state predicate transformer [Dijkstra 76], which we will show is expressible in **IL** (Theorem 2 in Section 5). Intuitively, it transforms a predicate

(postcondition), which defines some set of states *after* the execution of the program, to the predicate (precondition) which defines the set of all those states *before* its execution which result in the given postcondition. We follow Janssen's approach in that the translations of statements and programs have type $((\mathsf{S} \to \mathsf{B}) \to \mathsf{B})$. Procedure names are translated into functional variables whose values are functions from parameters to predicate transformers. The parameter type for pass-by-value is a number N, the parameter type for pass-by-reference is a reference $(\mathsf{S} \to \mathsf{N})$, and the parameter type for pass-by-name is the sense of a reference $(\mathsf{S} \to (\mathsf{S} \to \mathsf{N}))$. We have two special variables $j \in VAR_{(\mathsf{S} \to \mathsf{N})}$ and $l \in VAR_{(\mathsf{N} \to (\mathsf{S} \to \mathsf{N}))}$ for the translation of new-blocks. Intuitively, l is a linear array and j is an index which points to next available element in l.

We give the translation as follows. From now on, let $q \in VAR_{(\mathsf{S} \to \mathsf{B})}$ (state predicate variables), $n, m \in VAR_{\mathsf{N}}$. For any expression E, its translation is denoted by E'.

4.1 Translations of Program Expressions

Identifiers

To each identifier we associate a unique variable of **IL**.

$$
\begin{array}{rcl}
\mathtt{x}' & \equiv & x \in VAR_{(\mathsf{S} \to \mathsf{N})} \\
\mathtt{z}' & \equiv & z \in VAR_{(\mathsf{S} \to (\mathsf{S} \to \mathsf{N}))} \\
\mathtt{a}' & \equiv & a \in VAR_{(\mathsf{N} \to (\mathsf{S} \to \mathsf{N}))} \\
\mathtt{p}' & \equiv & p \in VAR_{(\mathsf{S} \to (\mathsf{S} \to \mathsf{N}))} \\
\mathtt{G}^{v\prime} & \equiv & g^v \in VAR_{\mathsf{S} \to (\mathsf{N} \to ((\mathsf{S} \to \mathsf{B}) \to \mathsf{B}))} \\
\mathtt{G}^{r\prime} & \equiv & g^r \in VAR_{\mathsf{S} \to ((\mathsf{S} \to \mathsf{N}) \to ((\mathsf{S} \to \mathsf{B}) \to \mathsf{B}))} \\
\mathtt{G}^{n\prime} & \equiv & g^n \in VAR_{\mathsf{S} \to ((\mathsf{S} \to (\mathsf{S} \to \mathsf{N})) \to ((\mathsf{S} \to \mathsf{B}) \to \mathsf{B}))}
\end{array}
$$

Number References $(ME_{(\mathsf{S} \to \mathsf{N})})$

$$
\begin{array}{rcl}
\mathtt{x}' & & \text{as above} \\
(\mathtt{a}[\mathtt{e}])' & \equiv & \mathtt{a}'(\mathtt{e}') \\
(\textbf{dref } \mathtt{p})' & \equiv & \check{}\mathtt{p}' \\
(\textbf{expand } \mathtt{z})' & \equiv & \check{}\mathtt{z}'
\end{array}
$$

Note the extensionalizing effect in the translations of (**dref** p) and (**expand** z).

Number Terms (ME_{N})

$$
\begin{array}{rcl}
\mathtt{m}' & \equiv & \overline{m} \in CON_{\mathsf{N}} \\
\mathtt{v}' & \equiv & \check{}\mathtt{v}' \\
 & & (\mathtt{v}' \text{ as above}) \\
(\mathtt{e}_1 \ + \ \mathtt{e}_2)' & \equiv & (\mathtt{e}_1' \ + \ \mathtt{e}_2')
\end{array}
$$

(Similarly for $-$ and $\times$)

$$
(\textbf{if } \mathtt{b} \textbf{ then } \mathtt{e}_1 \textbf{ else } \mathtt{e}_2 \textbf{ fi})' \equiv \mathit{if}\ \mathtt{b}'\ \mathit{then}\ \mathtt{e}_1'\ \mathit{else}\ \mathtt{e}_2'\ \mathit{fi}
$$

Note that the translation of $\mathtt{v}'$, considered as a *number term*, is the *extensional value* of its translation when considered as a *number reference*. Also, the number term (**dref** p) is translated into $\check{}\check{}p$.

Boolean Terms (ME_{B})

$$
\begin{array}{rcl}
\mathtt{true}' & \equiv & \mathit{true} \\
(\mathtt{e}_1 \ = \ \mathtt{e}_2)' & \equiv & (\mathtt{e}_1' \ = \ \mathtt{e}_2') \\
 & \cdots &
\end{array}
$$

Statements $(ME_{((\mathsf{S} \to \mathsf{B}) \to \mathsf{B})})$

Note that $q \in VAR_{(\mathsf{S} \to \mathsf{B})}$.

$$
\begin{array}{rcl}
\textbf{skip}' & \equiv & \lambda q \cdot \check{}q \\
(\mathtt{v} := \mathtt{e})' & \equiv & \lambda q \cdot \check{}q{<}\mathtt{v}'/\mathtt{e}'{>} \\
(\mathtt{p} := \mathtt{v})' & \equiv & \lambda q \cdot \check{}q{<}\mathtt{p}'/\mathtt{v}'{>} \\
(\mathsf{S}_1; \mathsf{S}_2)' & \equiv & \lambda q \cdot \mathsf{S}_1'(\hat{}(\ \mathsf{S}_2'(q))\ ) \\
\textbf{if } \mathtt{b} \textbf{ then } \mathsf{S}_1 \textbf{ else } \mathsf{S}_2 \textbf{ fi}' & \equiv & \lambda q \cdot (\mathtt{b}' \wedge \mathsf{S}_1'(q)) \ \vee \\
 & & \qquad (\neg \mathtt{b}' \wedge \mathsf{S}_2'(q)) \\
\mathtt{G}^v(\mathtt{e})' & \equiv & \check{}\mathtt{G}^{v\prime}(\mathtt{e}') \\
\mathtt{G}^r(\mathtt{v})' & \equiv & \check{}\mathtt{G}^{r\prime}(\mathtt{v}') \\
\mathtt{G}^n(\mathtt{v})' & \equiv & \check{}\mathtt{G}^{n\prime}(\hat{}\mathtt{v}')
\end{array}
$$

Note how the state switchers are used with the assignment statements.

Blocks $(ME_{((\mathsf{S} \to \mathsf{B}) \to \mathsf{B})})$

$$
\begin{array}{l}
(\textbf{begin new } \mathtt{x} := \mathtt{e} \ ; \ \mathsf{S} \textbf{ end})' \ \equiv \\
\quad \lambda q \cdot \exists n[(((\lambda \mathtt{x}' \cdot \mathsf{S}')(l(n))) \\
\qquad (\hat{}[n = \check{}j \wedge \check{}q{<}j/\check{}j + 1{>}])){<}l(n)/\mathtt{e}'{>}]
\end{array}
$$

where $n \in VAR_{\mathsf{N}}$, $\mathtt{j}' \equiv j \in VAR_{(\mathsf{S} \to \mathsf{N})}$ and $\mathtt{l}' \equiv l \in VAR_{(\mathsf{N} \to (\mathsf{S} \to \mathsf{N}))}$ (see explanation below).

$$
(\textbf{begin alias } \mathtt{y} = \mathtt{v} \ ; \ \mathsf{S} \textbf{ end})' \equiv (\lambda \mathtt{y}' \cdot \mathsf{S}')(\mathtt{v}')
$$

$$
(\textbf{begin macro } \mathtt{z} = \mathtt{v} \ ; \ \mathsf{S} \textbf{ end})' \equiv (\lambda \mathtt{z}' \cdot \mathsf{S}')(\hat{}\mathtt{v}')
$$

Procedure Bodies

$$
\begin{array}{l}
(< \textbf{value } \mathtt{x} > \mathsf{S} \textbf{ end})' \ \equiv \\
\quad \lambda m \cdot \lambda q \cdot \exists n[(((\lambda \mathtt{x}' \cdot \mathsf{S}')(l(n))) \\
\qquad (\hat{}[n = \check{}j \wedge \check{}q{<}j/\check{}j + 1{>}])){<}l(n)/m{>}]
\end{array}
$$

(see explanation below).

$$(< \textbf{ref } \mathtt{y} > \mathsf{S} \textbf{ end})' \equiv \lambda \mathtt{y}' \cdot \mathsf{S}'$$

$$(< \textbf{name } \mathtt{z} > \mathsf{S} \textbf{ end})' \equiv \lambda \mathtt{z}' \cdot \mathsf{S}'$$

Programs

$$\begin{aligned} &(<< \mathtt{G}_1 \Leftarrow \mathsf{B}_1, \ldots, < \mathtt{G}_k \Leftarrow \mathsf{B}_k >; \mathsf{S}_0 >)' \\ \equiv\ &(\lambda \mathtt{G}_1' \cdot (\ldots (\lambda \mathtt{G}_k' \cdot \mathsf{S}_0')(\hat{}\mathsf{B}_k') \ldots))(\hat{}\mathsf{B}_1') \end{aligned}$$

The above translations show that our semantics is compositional. They also illustrate the semantic correspondence between pass-by-value and new-block, pass-by-reference and alias-block, and pass-by-name and macro-block, as predicated by the Correspondence Principle (§1.2).

Explanation The translation of a macro-block results in a β-redex, because $\hat{}\mathtt{v}'$ is rigid. Hence, β-reduction can be carried out as given by Theorem 1. Since $(\textbf{expand } \mathtt{z})'$ is $\check{}z$, we can apply Lemma 2 to eliminate $\check{}\hat{}$ operators after the β-reduction, and we can then establish

$$(\textbf{begin macro } \mathtt{z} = \mathtt{v} \ ; \ \mathsf{S} \textbf{ end})' \cong (\mathsf{S}[\mathtt{z}/\mathtt{v}])'.$$

A straightforward suggestion for a translation of new-blocks would be

$$(\textbf{begin new } \mathtt{x} := \mathtt{e} \ ; \ \mathsf{S} \textbf{ end})' \equiv (\lambda \mathtt{x}' \cdot \mathsf{S}')(\mathtt{e}').$$

By simple type checking, we would discover that it is incorrectly typed and came up with

$$(\textbf{begin new } \mathtt{x} := \mathtt{e} \ ; \ \mathsf{S} \textbf{ end})' \equiv (\lambda \mathtt{x}' \cdot \mathsf{S}')(\hat{}\mathtt{e}').$$

This is also incorrect because $\hat{}\mathtt{e}'$ is rigid and permits a β-reduction, and after $\check{}\hat{}$eliminations it will result in

$$(\textbf{begin new } \mathtt{x} := \mathtt{e} \ ; \ \mathsf{S} \textbf{ end})' \cong (\mathsf{S}[\mathtt{x}/\mathtt{e}])'$$

which is clearly incorrect, since this could result in substituting $\mathtt{e}$ for $\mathtt{x}$ on the left hand side of an assignment in S.

Our translation of new-blocks relies on an array of *new* references. Intuitively, there is a number $n = \check{}j$ such that execution of the new-block is equivalent to the execution of the following statement:

$$\mathtt{l}[n] := \mathtt{e};\ \mathsf{S}[\mathtt{x}/\mathtt{l}[n]];\ \mathtt{j} := \mathtt{j} + 1,$$

where $\mathsf{S}[\mathtt{x}/\mathtt{l}[n]]$ denotes the replacement of all free $\mathtt{x}$ in S by $\mathtt{l}[n]$.

The translation of procedure bodies with value parameters works similarly.

Note that in our semantics, α-equivalence (renaming of bound variables) holds for the formal parameters in the procedure bodies and also for the new, alias, and macro identifiers in blocks. This is a good property of static scope semantics, and is not always the case with dynamic scope semantics [Hung 90, Chapter 3].

4.2 Translations of Assertions

Number Variables (VAR_{N})

To each number variable n we associate a unique **IL** variable n:

$$n' \equiv n$$

The translations of number and boolean terms are same as the above.

Assertions (ME_{B})

$$\begin{aligned} b' &\quad \text{as above} \\ (\neg\phi)' &\equiv \neg\phi' \\ (\phi_1 \wedge \phi_2)' &\equiv \phi_1' \wedge \phi_2' \\ (\exists n \phi)' &\equiv \exists n' \phi' \end{aligned}$$

Hoare Formulas (ME_{B})

$$\begin{aligned} (\{\ \phi\}\ \mathsf{S}\{\ \psi\}\ )' &\equiv \phi' \rightarrow \mathsf{S}'(\hat{}\psi') \\ (\{\ \phi\}\ \mathsf{R}\{\ \psi\}\ )' &\equiv \phi' \rightarrow \mathsf{R}'(\hat{}\psi') \end{aligned}$$

5 Results

Definition Two terms α, β are equivalent relative to *instantiation*, denoted by $\alpha \simeq \beta$, if for any set of distinct $X_1, \ldots X_k \in REF_s$, distinct $A_1, \ldots, A_j \in ARRAY$, distinct $P_1, \ldots, P_l \in PTR$,

$$\alpha[\vec{x}, \vec{a}, \vec{p}/\vec{X}, \vec{A}, \vec{P}] \cong \beta[\vec{x}, \vec{a}, \vec{p}/\vec{X}, \vec{A}, \vec{P}]$$

where $\vec{x}$,$\vec{a}$,$\vec{p}$ include all the free variables in α and β.

This notion of equivalence $\simeq$ is needed for the statements of Lemmas 5, 6 and Theorem 2 below, since the state switcher reductions of Lemma 4, which are used in the proofs, apply only to *constants*.

5.1 State-Switcher Reductions

Lemma 3 (Reduction Lemma 1) For all $\xi \in ME_{(\mathsf{S} \rightarrow \mathsf{N})}$, $\varepsilon, \varepsilon_1, \varepsilon_2 \in ME_{\mathsf{N}}$, $\eta \in ME_{(\mathsf{S} \rightarrow (\mathsf{S} \rightarrow \mathsf{N}))}$,

1. $C{<}\xi/\varepsilon{>} \cong C$ for all $C \in CON$
2. $v{<}\xi/\varepsilon{>} \cong v$ for all $v \in VAR$
3. $(\varepsilon_1 + \varepsilon_2){<}\xi/\varepsilon{>} \cong \varepsilon_1{<}\xi/\varepsilon{>} + \varepsilon_2{<}\xi/\varepsilon{>}$
   similar for $-, \times, =, <, \neg, \wedge$, *if-then-else-fi*.
4. $(\exists n\varphi){<}\xi/\varepsilon{>} \cong \exists n[\varphi{<}\xi/\varepsilon{>}]$
   where $n \notin \textbf{free}(\xi) \cup \textbf{free}(\varepsilon)$
5. $(\lambda x \cdot \alpha){<}\xi/\varepsilon{>} \cong \lambda x \cdot (\alpha{<}\xi/\varepsilon{>})$
   where $x \notin \textbf{free}(\xi) \cup \textbf{free}(\varepsilon)$
6. $\alpha(\beta){<}\xi/\varepsilon{>} \cong \alpha{<}\xi/\varepsilon{>}(\beta{<}\xi/\varepsilon{>})$
7. $(\hat{}\alpha){<}\xi/\varepsilon{>} \cong \hat{}\alpha$

The same results hold for state switchers $<\eta/\xi>$.

Proof: By the semantic definition of state switchers. □

Lemma 4 (Reduction Lemma 2) For all $X, Y \in REF$, $X \not\equiv Y$, $A, B \in ARRAY$, $A \not\equiv B$, $P, P_1 \in PTR$, $P \not\equiv P_1$, $\varepsilon, \delta \in ME_{\mathsf{N}}$, $\xi, \xi_1 \in ME_{(\mathsf{S} \to \mathsf{N})}$,

1. $(\check{}X)<X/\varepsilon> \cong \varepsilon$
2. $(\check{}Y)<X/\varepsilon> \cong \check{}Y$
3. $(\check{}X)<(A(\delta))/\varepsilon> \cong \check{}X$
4. $(\check{}(A(\delta)))<X/\varepsilon> \cong \check{}(A(\delta<X/\varepsilon>))$
5. $(\check{}(A(\delta')))<(A(\delta))/\varepsilon> \cong$
   $if\ (\delta'<(A(\delta))/\varepsilon> = \delta)$
   $then\ \varepsilon\ else\ \check{}(A(\delta'<(A(\delta))/\varepsilon>))\ fi$
6. $(\check{}(B(\delta')))<(A(\delta))/\varepsilon> \cong \check{}(B(\delta<(A(\delta))/\varepsilon>))$
7. $\check{}X<P/\xi> \cong \check{}X$
8. $\check{}(A(\delta))<P/\xi> \cong \check{}(A(\delta<P/\xi>))$
9. $(\check{}P)<P/X> \cong X$
10. $(\check{}P)<P/A(\delta)> \cong A(\delta)$
11. $(\check{}P)<P/\check{}P_1> \cong \check{}P_1$
12. $(\check{}P_1)<P/\xi> \cong \check{}P_1$
13. $(\check{}\check{}P)<\check{}P/\varepsilon> \cong \varepsilon$
14. Suppose ξ is rigid, then
    $\alpha<\xi_1/\varepsilon_1><P/\xi>$
    $\cong \alpha<P/\xi><\xi_1<P/\xi>/\varepsilon_1<P/\xi>>$
15. $\alpha<\xi/\varepsilon><P/\check{}P_1>$
    $\cong \alpha<P/\check{}P_1><\xi<P/\check{}P_1>/\varepsilon<P/\check{}P_1>>$

Proof: By the semantic definition of state switchers. □

Lemma 5 (Substitution Lemma 1 for State Switchers) Suppose ϕ is an assertion which contains no pointer dereferencing, and $\mathtt{d}$, $\mathtt{e}$ are number terms, then

$$\phi'<\mathtt{x}'/\mathtt{e}'> \simeq (\phi[\mathtt{x}/\mathtt{e}])',$$

and we can find an assertion ψ such that

$$\phi'<\mathtt{a}[\mathtt{d}]'/\mathtt{e}'> \simeq \psi'.$$

Proof: By structural induction on ϕ, and Lemmas 3 and 4 (Parts 1–6). □

This shows that we have correctly found a semantic representation for the syntactic substitution in [Hoare 69].

Lemma 6 (Substitution Lemma 2 for State Switchers) For any assertion ϕ, pointer $\mathtt{p}$, and number reference $\mathtt{v}$,

$$\phi'<\mathtt{p}'/\mathtt{v}'> \simeq (\phi[(\mathbf{dref}\ \mathtt{p})/\mathtt{v}])'.$$

Proof: By structural induction on ϕ, and Lemmas 3 and 4 (Parts 7–15). □

This is the new "pointer version" of Lemma 5.

5.2 State Switcher Free Pre-Condition

Program statements are interpreted as state predicate transformers. It is very easy to apply a transformer associated with a statement to a post-assertion to get a pre-assertion which contains state switchers. It is more interesting to know whether a *state switcher free* pre-assertion can be found.

Theorem 2 (Expressibility of the Weakest Precondition) (*a*) For any statement S which contains no procedure calls and with all its pointers initialized to number identifiers or array elements, and for any assertion ψ, we can find a assertion ϕ such that

$$\mathsf{S}'(\hat{}\psi') \simeq \phi'.$$

(*b*) For any well-formed program R which has all its pointers initialized to number identifiers or array elements, and any assertion ψ, there exists an assertion ϕ such that

$$\mathsf{R}'(\hat{}\psi') \simeq \phi'.$$

The proof, which uses the above Substitution Lemmas for state switchers, will be given in a forthcoming publication.

The significance of a *state-switcher free* formula of type $\mathtt{B}$ of **IL** is that (in the context of Theorem 2) it is the *translation* of an *assertion* in our assertion language. This Theorem thus shows that the weakest precondition is expressible in the assertion language.

6 Conclusion

It will be interesting to extend this approach to procedures with *procedures as parameters*, passed by value, reference, or name, as in [Kfoury & Urzyczyn 89].

References

[De Bakker 80] De Bakker J.W., *Mathematical Theory of Program Correctness*, Prentice-Hall, 1980.

[Brookes 85] Brookes S.D., A Fully Abstract Semantics for an Algol-like Language with Sharing, Technical report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, 1985.

[Church 51] Church A., A Formulation of the Logic of Sense and Denotation. In: *Structure, Method and Meaning*, ed. Henle P. *et. al*, 1951.

[Dijkstra 76] Dijkstra E.W., *A Discipline of Programming*, Prentice-Hall, 1976.

[Dowty, Wall & Peters 81] Dowty D.R., Wall R.E., Peters S., *Introduction to Montague Semantics*, D. Reidel, 1981.

[Frege 92] Frege G., Über Sinn und Bedeutung, *Zeitschrift für Philosophie und philosophische Kritik* **100**, 1892, 25–50. Translated as "On sense and Reference" in *Translations from the Philosophical Writings of Gottlob Frege*, ed. Geach P.T. & Black M., Basil Blackwell, Oxford, 1977, 56–85.

[Gallin 75] Gallin D., *Intensional and Higher-Order Modal Logic With Applications to Montague Semantics*, North-Holland Mathematics Studies **19**, 1975.

[Hoare 69] Hoare C.A.R., An Axiomatic Basis for Computer Programming, *Communications of ACM*, **12**, 1969, 576–583.

[Hung 89] Hung H.K., Application of Intensional Logic to Program Semantics, *Proceedings of the 7th Amsterdam Colloquium*, December 1989, Amsterdam.

[Hung 90] Hung H.K., *Compositional Semantics and Program Correctness for Procedures with Parameters* Ph.D. Thesis, SUNY-Buffalo, Computer Science Dept. Technical Report 90-18, 1990.

[Janssen, Van Emde Boas 77] Janssen T.M.V., Van Emde Boas P., On the Proper Treatment of Referencing and Dereferencing and Assignment, *Automata, Languages, and Programming (Proc.4th Colloq. Turku)*, Lecture Notes in Computer Science **52**, Springer-Verlag, 1977, 282–300.

[Janssen 86] Janssen T.M.V., *Foundations and Applications of Montague Grammar. Part 1: Philosophy, Framework, Computer Science*, CWI Tracts **19**, Centre for Mathematics and Computer Science, Amsterdam, 1986.

[Kfoury & Urzyczyn 89] Kfoury A.J., Urzyczyn P., Algol-like Languages With Higher-Order Procedures and Their Expressive Power, *Logic at Botik '89, Symposium on Logical Foundation of Computer Science*, Springer-Verlag, 1989, 186–199.

[Kripke 63] Kripke S.A., Semantical Considerations on Modal Logic, *Acta Philosophica Fennica* **16**, 1963, 83–94.

[Manna & Waldinger 81] Manna Z., Waldinger R., Problematic Features of Programming Languages: A Situational-Calculus Approach, *Acta Informatica* **16**, 1981, 371–426.

[Meyer & Sieber 88] Meyer A.R., Sieber K., Towards Fully Abstract Semantics for Local Variables: Preliminary Report, *ACM 15th Symposium on Principles of Programming Languages*, 1988, 191–203.

[Milne & Strachey 76] Milne R., Strachey C., *A Theory of Programming Language Semantics*, Parts 1 & 2, Chapman and Hall, 1976.

[Montague 74] Montague R., The Proper Treatment of Quantification in Ordinary English. In: *Formal Philosophy: Selected Papers of Richard Montague*, ed. Thomason R.H., Yale University Press, 1974, 247–270.

[Oles 85] Oles F.J., Type Algebras, Functor Categories and Block Structure, *Algebraic Methods in Semantics*, ed. Nivat M., Reynolds J.C., Cambridge University Press, 1985.

[Reade 89] Reade C., *Elements of Functional Programming*, Addison-Wesley, 1989.

[Reynolds 70] Reynolds J.C., GEDANKEN: A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept, *Communications of the ACM* **13**, 1970, 308–319.

[Reynolds 81] Reynolds J.C., *The Craft of Programming*, Prentice-Hall, 1981.

[Tennent 81] Tennent R.D., *Principles of Programming Languages*, Prentice-Hall, 1981.

[Wagner 89] Wagner E.G., On Declarations, *IBM Research Report RC-14657*, May 1989.

[Zwiers 89] Zwiers J., *Compositionality, Concurrency and Partial Correctness: Proof Theories on Networks of Processes, and Their Relationships*, Springer-Verlag, 1989.