Program Semantics, Intensional Logic and Compositionality

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

Hing-Kai Hung
380 Campus Drive
Snyder, NY 14226, USA
hung@cs.buffalo.edu

Abstract

We apply intensional logic to the semantics of an Algol-like programming language. This associates with expressions their 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". This approach has been applied in three areas: (1) assignments (Janssen and Van Emde Boas 1977); (2) blocks and procedures with parameters passed by value and by reference (Hung 1990); and (3) assignments to pointers, and parameters passed by name (Hung and Zucker 1991). It is shown how this approach gives an elegant semantics which is simple, compositional and implementation independent.

1 Introduction

1.1 Denotational semantics; Compositionality. Assume (within some general framework) we have a number of *syntactic domains* $\Delta, \ldots,$ and corresponding *semantic domains* $\mathcal{D}, \ldots,$ with a *meaning function*

$$[\![\cdot]\!] : \ \Delta \ \longrightarrow \ \mathcal{D}.$$

Assume also that the *compound expressions* of the syntax are formed from the atomic expressions by a fixed set of *syntactic operators*.

The meaning function $[\![\cdot]\!]$ is said to satisfy the *Compositionality Principle* if for every such n-ary syntactic operator $\Phi : \ \Delta_1 \times \cdots \times \Delta_n \ \longrightarrow \ \Delta_{n+1}$, there exists a corresponding function $\Psi : \ \mathcal{D}_1 \times \cdots \times \mathcal{D}_n \ \longrightarrow \ \mathcal{D}_{n+1}$ such that for every compound expression $\Phi(\delta_1, \ldots, \delta_n) \in \Delta_{n+1}$

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

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.

The Compositionality Principle thus says that "the meaning of a compound expression is a function of the meanings of its parts". The idea can be found in the writings of Frege. It is carefully discussed in Janssen [1986, Chapter 1], where it is used in the framework of many-sorted algebras and homomorphisms (*op. cit.*, Chapter 2). Janssen gives many arguments for accepting this principle: usefulness, elegance, generality, comprehensibility, power, value as a heuristic tool, and others.

Its significance for the correctness and verification of concurrent systems is discussed in Zwiers [1989].

Consider a simple Algol-like programming language, with assignments, sequential composition and conditionals. Whether we define the meaning $[\![\mathtt{S}]\!]$ of a program `S` in this language as a *state transformer* (De Bakker [1980]) or as a *predicate transformer* (forward or backward) (Dijkstra [1976]) it is not hard to see that the resulting semantics is *compositional*.

Consider now, however, the language extended by *procedures* with *value* and *reference (address) parameters*, and by *blocks*. For such a language, compositional semantics is more difficult to define. De Bakker [1980], for example, uses "syntactic application", which is non-compositional.

So how can we "restore" compositionality in such a framework? The answer is in the use of *intensional logic*, as we will now see.

1.2 Sense and Denotation. Frege [1892] distinguished between two notions of meaning: (1) *denotation* (or *reference* or *extension*), and (2) *sense* (or *intension*). We illustrate the application of each with a well-known example. Note first that *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. Consider, for example, the sentence

(S1) I see the morning star now

which has a definite denotation (= truth value) (relative to a particular valuation of the indexicals 'I' and 'now'). Now the phrases 'the morning star' and 'the evening star' have the same denotation (= the planet Venus), so substituting 'the evening star' for 'the morning star' in (S1) should result in a sentence with the same truth value:

(S2) I see the evening star now

which is the case. But now consider the sentence

(S3) I believe that the morning star is the morning star

which is obviously true. Now, however, if we substitute 'the evening star' for (say) the second occurrence of 'the morning star' in (S3), we get a sentence

(S4) I believe that the morning star is the evening star

which may very well be false!

The explanation for this (apparent) violation of the principle of substitutivity of equals is that the context after a verb of attitude (expressing a belief etc.) is *referentially opaque* (compared to a *referentially transparent* context such as '... is visible'). Frege's insight was that the meaning of an expression in an opaque context is given *not* by its denotation, but by its *sense*. Now the expressions 'morning star' and 'evening star' have *different senses* (since we can easily imagine a world in which they refer to different objects), and hence *different meanings in an opaque*

context. Hence (S3) and (S4) do *not* give a counterexample to the principle of substitutivity of equals!

To summarize this discussion: taking the *senses* of expressions to be their meanings in opaque contexts *restores* the principle of substitutivity of equals, and (hence) compositionality.

1.3 Intensional logic. In order to reason about senses or intensions (as opposed to denotations or extensions), we use intensional logic. An early axiomatization of intensional logic was given by Church [1951]. Carnap [1947] gave a semantics, in which the *sense* of an expression is a function from "states" to (extensional) values. Kripke [1963] gave a more mathematically developed semantics, in which the *sense* of an expression is a function from *possible worlds* to (extensional) values.

Montague [1974] gave a compositional semantics for a fragment of English, including opaque contexts, by defining a translation of it into a system of intensional logic.

Janssen & Van Emde Boas [1977] and Janssen [1986] applied Montague's approach to an imperative programming language — essentially a fragment of Algol. Their insight was that certain contexts in such a language can be viewed as opaque contexts; for example, the *left hand side of an assignment*.

EXAMPLE. Suppose the locations $\mathtt{x}$ and $\mathtt{y}$ each store the value 2. Compare first the two assignments

$$\mathtt{x} := \mathtt{x} + 1 \qquad \text{and} \qquad \mathtt{x} := \mathtt{y} + 1. \tag{1}$$

They both have the same effect ($\mathtt{x} = 3,\ \mathtt{y} = 2$).

But compare now the two assignments

$$\mathtt{x} := \mathtt{x} + 1 \qquad \text{and} \qquad \mathtt{y} := \mathtt{x} + 1. \tag{2}$$

They have different effects. (The first results in $\mathtt{x} = 3,\ \mathtt{y} = 2$ and the second in $\mathtt{x} = 2,\ \mathtt{y} = 3$).

So substitutivity of equals works in (1), but not in (2). From this we infer that the *rhs (right hand side)* of an assignment is *referentially transparent*, but the *lhs (left hand side)* is *referentially opaque*.

The solution is to define the the *sense* of a program variable (or identifier) of type N as a *function* from "possible worlds" to numbers, interpreting "possible worlds" as *computation states*.

This amounts to taking the *sense* of an identifier as a *memory address* or *location*, the contents of which depends on the state of the computation; and its *denotation* as the (current) contents.

REMARK. In common progam language terminology, the lhs of an assignment is said to denote a *reference* (or location), and the rhs an *object*. The assignment causes the reference to "possess" the object as its value, and the object can then be accessed through the reference by *dereferencing*. This is quite a different concept of "reference" from the one used in philosophy (denotation) that we have been discussing.

Janssen and Van Emde Boas proceeded by translating the programming language into a system of higher-order intensional logic. The semantics thus inherited by the programming language turns out to be compositional!

1.4 An overview of this paper. The approach has been applied for fragments of Algol in three areas:

(1) Assignments (Janssen & Van Emde Boas [1977]);

(2) Blocks and procedures with parameters, passed by *value* and by *reference* (Hung [1989, 1990]);

(3) Assignments to pointer identifiers; parameters passed by *name* (Hung & Zucker [1991]).

We will consider each of these three areas in turn: in Sections 2, 3 and 4 respectively. (We have already touched on the first in §1.3.) In each case, as we will see, one gets a semantics which is *compositional* and *implementation independent*.

This paper is intended as a brief survey of these areas; details can be found in the above references and in a forthcoming publication.

We thank Peter Grogono, Garrel Pottinger and Bob Tennent for helpful remarks following the talk on which this paper is based.

2 The formal system *IL* of intensional logic

2.1 Syntax. We define a typed system ***IL*** of *intensional logic*. The *types* $\tau, \ldots$ are generated according to the definition:

$$\tau \ ::= \ \mathsf{N} \mid \mathsf{B} \mid (\tau_1 \to \tau_2) \mid (\mathsf{S} \to \tau)$$

where N and B are the basic types of *numbers* and *booleans* (or *propositions*), and S is the type of *states*. (Note that S is a "hidden type" — there are no terms of type S.) For each type τ there is a countable set $\boldsymbol{Var}_\tau$ of variables of type τ. There are also *constants* of certain types, including the constant 0. Typed *expressions* are generated, according to the typing rules, from these variables and constants, by the *arithmetical operations* of $+$, $-$, $\times$; the relations of '$=$' and '$<$' on the numbers; the propositional connectives and existential quantifier; the operations of *application* $\cdot(\cdot)$ and *lambda abstraction* at all types; and the *modal operators* $^\wedge$ (intension), $^\vee$ (extension) and $\langle \cdot / \cdot \rangle$ (state switcher).

DEFINITION. *Reference variables* $x, \ldots$ are variables of type $(\mathsf{S} \to \mathsf{N})$. There is also a countable set ***Ref*** of *reference constants* $X, \ldots$ of type $(\mathsf{S} \to \mathsf{N})$. (The reference variables and constants will represent the number "references" or locations).

REMARKS. (1) For an expression $\alpha : \tau$ (*i.e.*, α *of type* τ), the expression $^\wedge\alpha$ gives the *sense* or *intension* of α, *i.e.*, its values in all states.

(2) For $\alpha : (\mathsf{S} \to \tau)$, $^\vee\alpha$ gives the *extension* of α, *i.e.*, its value in the current state. This can be thought of as the *dereferencing* of α (see Remark in §1.3).

(3) For $\alpha : \tau$, reference variable or constant ξ and expression $e : \mathsf{N}$, the *state switcher expression* $\alpha\langle \xi / e \rangle : \tau$ gives an "explicit representation of syntactic substitution". (See the Substitution Lemma in §2.4.)

2.2 Semantics. Let N be the set of natural numbers, and $\mathrm{B} = \{\mathtt{tt}, \mathtt{ff}\}$ the set of truth values. Then define ***State*** to be the set of *states*, *i.e.*, functions $\sigma : \ \boldsymbol{Ref} \ \to \mathrm{N}$.

For each type τ there is a domain D_τ of objects of type τ: $D_\mathsf{N} = \mathrm{N}$, $D_\mathsf{B} = \mathrm{B}$, $D_{\tau_1 \to \tau_2}$ is a (suitable) subset of the set of functions from D_{τ_1} to D_{τ_2}, and $D_{\mathsf{S} \to \tau}$ is a (suitable) subset of the set of functions from ***State*** to D_τ.

A *valuation* is a functions ρ from $\boldsymbol{Var}_\tau$ to D_τ (for all τ).

For $\alpha : \tau$, we define the *meaning of α relative to a state σ and valuation ρ*, $[\![\alpha]\!]\sigma\rho$, by structural induction on α. The full definition can be found in Hung [1990] or Hung & Zucker [1991]. We give some of the more interesting clauses. (For $k \in \boldsymbol{Var}_{\mathsf{N}}$ and $\boldsymbol{n} \in \mathrm{N}$, $\rho\{k/\boldsymbol{n}\}$ denotes the *variant of ρ at k with respect to n*, and similarly for $\sigma\{X/\boldsymbol{n}\}$.

$$
\begin{aligned}
[\![X]\!]\sigma\rho &= \lambda\sigma' \cdot \sigma'(X) \\
[\![x]\!]\sigma\rho &= \rho(x) \\
[\![\exists k\phi]\!]\sigma\rho &= \begin{cases} \mathtt{tt} & \text{if } [\![\phi]\!]\sigma\rho\{k/\boldsymbol{n}\} \text{ for some } \boldsymbol{n} \in \mathrm{N} \\ \mathtt{ff} & \text{otherwise} \end{cases} \\
[\![{}^{\wedge}\alpha]\!]\sigma\rho &= \lambda\sigma'[\![\alpha]\!]\sigma'\rho \\
[\![{}^{\vee}\alpha]\!]\sigma\rho &= ([\![\alpha]\!]\sigma\rho)\sigma \\
[\![\alpha\langle X/e\rangle]\!]\sigma\rho &= [\![\alpha]\!]\sigma\{X/[\![e]\!]\sigma\rho\}\rho
\end{aligned}
$$

Note that the interpretation of terms is "two-dimensional": Quantification and λ-abstraction are interpreted by means of *valuations* ρ, and modal operators by means of *states* σ.

2.3 Translation of programming language into *IL*. Let ***ProgLang*** be the simple programming language of §1.1. It contains

(1) *Variables* $\mathtt{u}, \mathtt{v}, \ldots$, consisting of both *simple variables* $\mathtt{x}, \mathtt{y}, \ldots$ and *indexed variables* $\mathtt{a}[\mathtt{e}], \ldots$

(2) *Arithmetical expressions* $\mathtt{e}, \ldots$

(3) *Booleans* $\mathtt{b}, \ldots$

(4) *Program statements* $\mathtt{S}, \ldots$, generated by:

$$\mathtt{S} \ ::= \ \mathtt{v} := \mathtt{e} \mid \mathtt{S}_1; \mathtt{S}_2 \mid \mathsf{if}\ \mathtt{b}\ \mathsf{then}\ \mathtt{S}_1\ \mathsf{else}\ \mathtt{S}_2\ \mathsf{if}$$

(5) *Assertions* $\phi, \ldots$, or first order formulae, extending booleans by having quantifiers.

(Full details can be found in Hung [1990] or Hung & Zucker [1991].)

We describe a translation of ***ProgLang*** into ***IL***. (Again, we only give the more interesting cases. A full definition can be found in the above references.)

First, we associate, with each *simple variable* $\mathtt{x}$ of type $\mathtt{N}$, an ***IL*** variable $\mathtt{x}' : (\mathtt{S} \to \mathtt{N})$, and with each *array variable* $\mathtt{a}$, an ***IL*** variable $\mathtt{a}' : (\mathtt{N} \to (\mathtt{S} \to \mathtt{N}))$.

Next, for each *arithmetical expression* $\mathtt{e}$ and *boolean* $\mathtt{b}$, we define their translations $\mathtt{e}' : \mathtt{N}$ and $\mathtt{b}' : \mathtt{B}$ respectively, by structural induction on $\mathtt{e}$ and $\mathtt{b}$. For example, the translation of $\mathtt{x}$ (viewed now as an arithmetical expression) is ${}^{\vee}\mathtt{x}' : \mathtt{N}$ (its "dereferenced" value!), and similarly, the translation of $\mathtt{a}[\mathtt{e}]$ is ${}^{\vee}(\mathtt{a}'(\mathtt{e})) : \mathtt{N}$.

Next, a *statement* $\mathtt{S}$ is translated as a "backward predicate transformer" $\mathtt{S}' : ((\mathtt{S} \to \mathtt{B}) \to \mathtt{B})$. (Note that a (state) predicate has type $(\mathtt{S} \to \mathtt{B})$.) This is also defined by structural induction on $\mathtt{S}$. For example:

$$(\mathtt{v} := \mathtt{e})' \ = \ \lambda q^{(\mathtt{S}\to\mathtt{N})} \cdot q\langle \mathtt{v}'/\mathtt{e}' \rangle$$

(note the use of the state switcher!), and

$$(\mathtt{S}_1; \mathtt{S}_2)' \ = \ \lambda q^{(\mathtt{S}\to\mathtt{N})} \cdot \mathtt{S}_1'({}^{\wedge}(\mathtt{S}_2'(q))).$$

The semantics of a program statement is then that induced by its translation. Now since (*i*) the translation is a *homomorphism* of the term algebras, and (*ii*) the semantics of ***IL*** is compositional, therefore the induced semantics of ***ProgLang*** is compositional. (See Janssen [1986, Chapter 2].)

We have dealt with the first of the three areas described in §1.4. Before moving on to the second, we will show that one can derive more from the translation, namely expressibility of the weakest precondition.

2.4 State switcher elimination; Expressibility of weakest precondition.

LEMMA (SUBSTITUTION LEMMA). *For any assertion* ϕ*, reference variable* $\mathtt{x}$ *and arithmetical expression* $\mathtt{e}$*,*

$$\phi'\langle \mathtt{x}'/\mathtt{e}'\rangle \;\simeq\; (\phi[\mathtt{x}/\mathtt{e}])',$$

where '$\simeq$*' denotes semantic equivalence relative to instantiation of reference variables by reference constants.*

The proof is by structural induction on ϕ. It uses a number of "state switcher reductions".

This lemma shows that the state switcher functions correctly as an explicit representation of *syntactic substitution*. The fact that it also correctly represents *semantic substitution* is shown by its semantic definition (see the last equation in §2.2).

THEOREM (EXPRESSIBILITY OF WEAKEST PRECONDITION). *For any statement* S *and assertion* ϕ *we can find an assertion* ψ *such that*

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

The proof uses the Substitution Lemma.

DISCUSSION. It is trivial to apply a transformer associated with a statement $\mathtt{S}$ to a postcondition ϕ to get a precondition in ***IL***. (That is just $\mathtt{S}'(\hat{}\phi')$ in the statement of the theorem.) The interesting question is whether the precondition is *expressible in our assertion language* (by ψ, in the statement of the theorem.) The significance of state switcher elimination and the Substitution Lemma is that (in the context of this theorem) it provides a *state switcher free* form of $\mathtt{S}'(\hat{}\phi')$, which is the translation of such a precondition.

3 Blocks and procedures with value and reference parameters

3.1 The problem. Consider the procedure declaration:

$$\mathtt{P} \Leftarrow \langle\, \mathsf{val}\ \mathtt{x};\ \mathsf{ref}\ \mathtt{y}\,\rangle\, \mathtt{y} := \mathtt{x}\ \mathsf{end}$$

Suppose the locations u and v each store the value 2. Compare first the three procedure calls

$$\mathtt{P}(\mathtt{u},\mathtt{w}), \qquad \mathtt{P}(\mathtt{v},\mathtt{w}), \qquad \mathtt{P}(2,\mathtt{w}).$$

They all have the same effect $(\mathtt{w} \leftarrow 2)$.

But compare now the two calls

$$\mathtt{P}(0,\mathtt{u}), \qquad \mathtt{P}(0,\mathtt{v}).$$

They have different effects. (The first results in $\mathtt{u} = 0,\ \mathtt{v} = 2$ and the second in $\mathtt{u} = 2,\ \mathtt{v} = 0$.)

So a *value* parameter is referentially *transparent*, and a *reference* parameter is referentially *opaque*.

Compositional semantics for pass-by-reference is therefore difficult. In fact, it also turns out to be difficult for pass-by-value! For both of these, De Bakker [1980] uses "syntactic application", which is non-compositional.

The solution, as we will see, is again by means of a translation into ***IL***.

There is a corresponding problem for blocks, and a corresponding solution, as we will also see.

3.2 Syntax of procedures and blocks. We adjoin to ***ProgLang*** *procedure identifiers* of two sorts: *pass-by-value* $\mathtt{P}^{\mathtt{v}}, \ldots,$ and *pass-by-reference* $\mathtt{P}^{\mathtt{r}}, \ldots.$

We also have *procedure bodies* of both sorts:

$$\begin{aligned} \mathtt{B}^{\mathtt{v}} &\equiv \langle\, \mathsf{val}\ \mathtt{x} \,\rangle\, \mathtt{S}\ \mathsf{end} \\ \mathtt{B}^{\mathtt{r}} &\equiv \langle\, \mathsf{ref}\ \mathtt{x} \,\rangle\, \mathtt{S}\ \mathsf{end} \end{aligned}$$

and *procedure declarations*

$$\mathtt{D} \equiv \mathtt{P}^{\mathtt{v}} \Leftarrow \mathtt{B}^{\mathtt{v}} \mid \mathtt{P}^{\mathtt{r}} \Leftarrow \mathtt{B}^{\mathtt{r}}$$

We also adjoin *blocks* of two sorts:

$$\begin{aligned} \mathtt{K}^{\mathtt{v}} &\equiv \mathsf{begin\ new}\ \mathtt{x} := \mathtt{e};\ \mathtt{S}\ \mathsf{end} \\ \mathtt{K}^{\mathtt{r}} &\equiv \mathsf{begin\ alias}\ \mathtt{x} := \mathtt{u};\ \mathtt{S}\ \mathsf{end} \end{aligned}$$

Our approach in these definitions is in accordance with the *Correspondence Principle* of Tennent [1981, §9.1]: "For any parameter mechanism, an analogous definition mechanism is possible, and *vice versa*." Here "definition mechanism" can be interpreted as "block mechanism". So 'new' blocks correspond to 'value' parameters and 'alias' blocks to 'reference' parameters. (We will extend the application of this principle in Section 4.)

And correspondingly, the the rhs of the assignment is referentially *transparent* in the 'new' block, and referentially *opaque* in the 'alias' block.

The syntax of *program statements* is extended by:

$$\mathtt{S} ::= \quad \ldots \mid \mathtt{P}^{\mathtt{v}}(\mathtt{e}) \mid \mathtt{P}^{\mathtt{r}}(\mathtt{u}) \mid \mathtt{K}^{\mathtt{v}} \mid \mathtt{K}^{\mathtt{r}}$$

and *programs* $\mathtt{R}$ are defined by

$$\mathtt{R} \equiv \langle\, \vec{\mathtt{D}}; \mathtt{S} \,\rangle$$

where $\vec{\mathtt{D}}$ a vector of procedure declarations.

We assume our programs are *closed*, *i.e.*, all procedures called in $\mathtt{S}$ are declared in $\vec{\mathtt{D}}$. We also forbid (for now) recursive procedures. (This is just to avoid having to deal with fixpoint domain theory. We believe there is no difficulty in extending our semantic treatment to these.)

3.3 Rigidity; β-conversion. Before we define the translation of the extended ***ProgLang*** to ***IL***, we need a little of the theory of ***IL***.

DEFINITION. A term α of ***IL*** is *rigid* if its value is the same in all states, *i.e.*, $[\![\alpha]\!]\sigma_1 = [\![\alpha]\!]\sigma_2$ for all states σ_1, σ_2.

Versions of the following theorem on β-conversion in ***IL*** can be found in Dowty, Wall & Peters [1981], Gallin [1975] and Janssen [1986].

THEOREM (β-CONVERSION). *For any* ***IL****-term* $(\lambda x \cdot \alpha)(\beta)$*, if either of the following conditions holds: (1)* β *is rigid; (2) no free occurrence of* x *in* α *lies in the scope of* $^\wedge$ *or* $\langle \cdot / \cdot \rangle$*; then*

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

(Here '$\cong$' means semantic equivalence.)

Let us see what goes wrong with β-conversion when neither condition above holds.

EXAMPLE. Let $\alpha = \lambda k \cdot {}^\wedge(k = 1) : (\mathsf{N} \to (\mathsf{S} \to \mathsf{B}))$, let σ_0 be the "current state" and let $x_1 : (\mathsf{S} \to \mathsf{N})$ be any variable such that $\rho(x_1)(\sigma_0) = 1$ Consider the term $\alpha \cdot {}^\vee x_1$. Now β-conversion of this term would lead to ${}^\wedge({}^\vee x_1 = 1)$. Note that this β-conversion violates both restrictions in the above theorem. But, as can easily be seen, $[\![\alpha \cdot {}^\vee x_1]\!]\sigma_0\rho = \boldsymbol{\lambda}\sigma \cdot \mathtt{tt}$, the constantly true predicate, whereas $[\![{}^\wedge({}^\vee x_1 = 1)]\!]\sigma_0\rho = \boldsymbol{\lambda}\sigma \cdot \big(\rho(x_1)(\sigma) = 1\big)$.

Note that this problem is related to the one discussed in §1.2 (substituting non-rigid terms in "intensional" (= opaque) contexts).

For later use we also state

PROPOSITION ($^\vee{}^\wedge$-CANCELLATION). *For any* ***IL****-term* α*,*

$${}^\vee{}^\wedge\alpha \;\cong\; \alpha.$$

3.4 Translation of programming language into *IL*. We extend the translation of §2.3. We first consider blocks.

We have two special ***IL***-variables, $j : (\mathsf{S} \to \mathsf{N})$ and $l : (\mathsf{N} \to (\mathsf{S} \to \mathsf{N}))$ for the translation of new-blocks. Intuitively, l is an array and j is an index which points to next available element in l.

The 'alias' block is simple:

$$\mathsf{begin\ alias\ x := u;\ S\ end} \;\longmapsto\; (\lambda \mathtt{x}' \cdot \mathtt{S}')(\mathtt{u}'),$$

the 'new' block less so:

$$\begin{aligned}\mathsf{begin\ new\ x := e;\ S\ end} \;\longmapsto\; \lambda q \cdot \exists n\big[&(((\lambda \mathtt{x}' \cdot \mathtt{S}')(l(n))) \\ &({}^\wedge[n = {}^\vee j \wedge {}^\vee q\langle j/{}^\vee j + 1\rangle]))\langle l(n)/\mathtt{e}'\rangle\big]\end{aligned}$$

Intuitively, this says that there is a number $n = {}^\vee j$ such that execution of the 'new' block is equivalent to the execution of the following statement:

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

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

REMARK. A suggestion for a simpler translation of 'new' blocks might be

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

By simple type checking, we would discover that this is incorrectly typed, and come up instead with

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

But this is also incorrect, because $\hat{}\,\mathtt{e}'$ is rigid and hence permits a β-conversion (by the Theorem in §3.2), which (by $\check{}\,\hat{}$-cancellation) will result in $(\mathtt{S}[\mathtt{x}/\mathtt{e}])'$, which is clearly incorrect, since this could result in substituting $\mathtt{e}$ for $\mathtt{x}$ on the lhs of an assignment in $\mathtt{S}$!

The translation of procedures, with the two sorts of parameters, proceeds similarly. With each procedure identifier we associate an ***IL***-variable, whose value is a function from parameters to predicate transformers. The type of value parameters is $\mathtt{N}$, and the type of reference parameters is $(\mathtt{S} \to \mathtt{N})$. The translation thus has the form:

$$\begin{aligned} \mathtt{P}^{\mathsf{v}} \quad &\longmapsto \quad p^{\mathsf{v}} : (\mathtt{S} \to (\mathtt{N} \to \tau)) \\ \mathtt{P}^{\mathsf{r}} \quad &\longmapsto \quad p^{\mathsf{r}} : (\mathtt{S} \to ((\mathtt{S} \to \mathtt{N}) \to \tau)) \end{aligned}$$

where $\tau = ((\mathtt{S} \to \mathtt{B}) \to \mathtt{B})$, the type of predicate transformers. Then procedure calls are translated by:

$$\begin{aligned} \mathtt{P}^{\mathsf{v}}(\mathtt{e}) \quad &\longmapsto \quad \check{}\,p^{\mathsf{v}}(\mathtt{e}') \\ \mathtt{P}^{\mathsf{r}}(\mathtt{u}) \quad &\longmapsto \quad \check{}\,p^{\mathsf{r}}(\mathtt{e}') \end{aligned}$$

Procedure bodies are translated by:

$$\begin{aligned} \langle \mathsf{val\ x} \rangle\, \mathtt{S\ end} \quad &\longmapsto \quad \lambda m \cdot \lambda q \cdot \exists n \big[(((\lambda \mathtt{x}' \cdot \mathtt{S}')(l(n))) \\ &\qquad\qquad (\hat{}\,[n = \check{}\,j \wedge \check{}\,q\langle j/\check{}\,j + 1\rangle]))\langle l(n)/m\rangle\big] \\ \langle \mathsf{ref\ x} \rangle\, \mathtt{S\ end} \quad &\longmapsto \quad \lambda \mathtt{y}' \cdot \mathtt{S}' \end{aligned}$$

Note the analogy with the translations of the corresponding blocks. This gives semantic support to the Correspondence Principle.

Finally, programs are translated by:

$$\langle \mathtt{P}_1 \Leftarrow \mathtt{B}_1 \ldots, \mathtt{P}_k \Leftarrow \mathtt{B}_k\ ;\ \mathtt{S} \rangle \quad \longmapsto \quad (\lambda p_1 \cdot\ (\ldots (\lambda p_k \cdot \mathtt{S}')(\hat{}\,\mathtt{B}'_k) \ldots))(\hat{}\,\mathtt{B}'_1)$$

The semantics of a program is then that induced by its translation. Again, since the translation is a *homomorphism* of the relevant term algebras, and the semantics of ***IL*** is compositional, the induced semantics of ***ProgLang*** is compositional.

Again (as in §2.4), we can prove a *Substitution Lemma* for state switchers, and *expressibility* of the weakest precondition.

4 Pointers; Pass by name; Macro blocks

We extend ***ProgLang*** with a number of concepts involving "double intensionality".

4.1 Pointers. First, we introduce *pointer variables* $\mathtt{z}, \ldots,$ which can occur on the lhs of an assignment, either in the form $\mathtt{z} := \mathtt{v}$, or in the "dereferenced" form $\mathsf{dref\ z} := \mathtt{e}$. Also, $\mathsf{dref\ z}$ may form part of an arithmetical expression.

With each pointer variable z we associate an ***IL***-variable $\mathsf{z}' : (\mathsf{S} \to (\mathsf{S} \to \mathsf{N}))$. Then the translation of assignments is extended to the cases:

$$\begin{array}{rcl} \mathsf{z} := \mathsf{v} & \longmapsto & \lambda q \cdot {}^{\vee}q\langle \mathsf{z}'/\mathsf{v}' \rangle \\ \mathsf{dref\ z} := \mathsf{e} & \longmapsto & \lambda q \cdot {}^{\vee}q\langle {}^{\vee}\mathsf{z}'/\mathsf{e}' \rangle \end{array}$$

Note that the second case is merely a special case of our previous rule (§2.3) for translating assignments.

Further, the translation of $\mathsf{dref\ z}$ (viewed as an arithmetical expression) is ${}^{\vee\vee}\mathsf{z} : \mathsf{N}$, a "doubly dereferenced" version of z.

4.2 Procedures with pass-by-name parameters. We introduce *pass-by-name* procedure identifiers $P^{\mathsf{n}}, \ldots,$ with procedure bodies

$$B^{\mathsf{n}} \ \equiv\ \langle\, \mathsf{name\ z} \,\rangle\, \mathsf{S\ end},$$

and declarations $P^{\mathsf{n}} \Leftarrow B^{\mathsf{n}}$. The formal 'name' parameter z may occur in the "extensionalized" form $\mathsf{expand\ z}$ inside the body S.

For the translation into ***IL***, we associate with each formal name parameter z an ***IL***-variable $\mathsf{z}' : (\mathsf{S} \to (\mathsf{S} \to \mathsf{N}))$, and with each procedure identifier P^{n} an ***IL***-variable $p^{\mathsf{n}} : (\mathsf{S} \to ((\mathsf{S} \to (\mathsf{S} \to \mathsf{N})) \to \tau))$, where, again, $\tau = ((\mathsf{S} \to \mathsf{B}) \to \mathsf{B})$.

Then $\mathsf{expand\ z}$ is translated as ${}^{\vee}\mathsf{z}'$, and pass-by-name bodies by:

$$\langle\, \mathsf{name\ z} \,\rangle\, \mathsf{S\ end} \quad \longmapsto \quad \lambda \mathsf{z}' \cdot \mathsf{S}'.$$

4.3 Macro blocks. In accordance with the Correspondence Principle, we introduce 'macro' blocks to match pass-by-name parameters:

$$K^{\mathsf{n}} \ \equiv\ \mathsf{begin\ macro\ z} := \mathsf{v};\ \mathsf{S\ end}.$$

where, again, S may contain the "extensionalized" form $\mathsf{expand\ z}$.

The translation of macro blocks is analogous to that for pass-by-name parameters:

$$\mathsf{begin\ macro\ z} := \mathsf{v};\ \mathsf{S\ end} \quad \longmapsto \quad (\lambda \mathsf{z}' \cdot \mathsf{S}')({}^{\wedge}\mathsf{v}').$$

4.4 Results. As before, we obtain compositional semantics for the extended language. We also obtain a *Substitution Lemma* for state switchers and (hence) *expressibility* of the weakest precondition, by non-trivial extensions of the relevant proofs for the language of Section 3.

5 Future work

We plan to extend our work in the following directions:

- Compare our semantics with *operational semantics* for the languages considered here;
- Extend our approach to *higher order procedures*, *i.e.*, procedures with *procedure parameters*;
- Apply this method also to *object-oriented languages*, for which compositional semantics seem to be lacking.

References

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

R. Carnap [1947], *Meaning and Necessity* (2nd edition 1956), The University of Chicago Press.

A. Church [1951], "A formulation of the logic of sense and denotation," in *Structure, Method and Meaning: Essays in Honor of Henry M. Sheffer*, P. Henle *et al.*, ed., Liberal Arts Press, New York.

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

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

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

D. Gallin [1975], *Intensional and Higher-Order Modal Logic*, North-Holland.

H. K. Hung [1989], "Application of intensional logic to program semantics," in *Proceedings of the 7th Amsterdam Colloquium, December 1989*.

——— [1990], *Compositional Semantics and Program Correctness for Procedures with Parameters*, PhD Thesis, Computer Science Department, SUNY-Buffalo, Technical Report 90-18.

H. K. Hung & J. I. Zucker [1991], "Semantics of pointers, referencing and dereferencing with intensional logic," in *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science, July 1991, Amsterdam*, 127–136.

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

T. M. V. Janssen & P. van Emde Boas [1977], "On the proper treatment of referencing, dereferencing and assignment," in *4th International Colloquium on Automata, Languages and Programming, University of Turku, Finland*, A. Salomaa & M. Steinby, eds., Lecture Notes in Computer Science #52, Springer-Verlag, 282–300.

S. A. Kripke [1963], "Semantical considerations on modal logic," *Acta Philosophica Fennica* 16, 83–94.

R. Montague [1974], "The proper treatment of quantification in ordinary English," in *Formal Philosophy: Selected Papers of Richard Montague*, R.H. Thomason, ed., Yale University Press, 247–270.

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

J. Zwiers [1989], *Compositionality, Concurrency and Partial Correctness: Proof Theories on Networks of Processes, and Their Relationships*, Lecture Notes in Computer Science #321, Springer-Verlag.