Denotational Semantics of a Simple Imperative Language Using Intensional Logic

Marc Bender
`bendermm@mcmaster.ca`

CAS 706 - Programming Languages
Instructor: Dr. Jacques Carette
Dept. of Computing and Software
McMaster University

Introduction

The standard approach to references in denotational semantics is to introduce *addresses* into the semantic domain. This method is inelegant when dealing with blocks and procedures with parameters passed by reference.

Intensional logic is used for reasoning about *senses* (or *intensions*) of expressions.

In imperative programming languages, it's very useful for dealing with the concept of *references* (as we will see).

We will devise a system of intensional logic ***IL*** and use it to translate a simple Algol-like language, as in Janssen [1986] and Hung [1990].

Sense and Denotation

When we use an *identifier* that corresponds to some object, there is an inherent ambiguity as to what we mean:

- what the object *denotes* (its *extension*), or
- the object itself.

For example, consider the identifier 'the temperature' in the following expressions:

- 'the temperature is 20°'
- 'the temperature is rising'

In the first occurence, we're making a statement about the *current* temperature. In the second, we mean the temperature itself (as a function of time).

To gain a better understanding of this situation, let's try to assign a "type" to the term in question. From 'the temperature is 20°' we could infer that 'the temperature' has type degrees.

Say we now take a measurement of the temperature, and discover that it really is 20° — i.e. 'the temperature' $=$ '20°'. We evaluate the predicate by substitution:

'20° is 20°'

which is true, as we'd expect. Now let's try to substitute into 'the temperature is rising':

'20° is rising'

This clearly makes no sense. Somehow we've introduced a "type mismatch" by substituting *the same value for the same term*.

Referential Transparency Vs. Referential Opacity

We have the following important principle:

Principle of Substitutivity of Equals:
Substituting one term for another with the same meaning in an expression should not alter the meaning of that expression. In symbols:

$$[\![x]\!] = [\![y]\!] \quad \Rightarrow \quad [\![t]\!] = [\![t\,[x/y]]\!]$$

Clearly this principle is very important in logic and mathematics.

We call situations where the principle holds (e.g. ‘the temperature is 20°’) *referentially transparent*; otherwise (e.g. ‘the temperature is rising’) they are *referentially opaque*.

Our goal is to restore the principle of substitutivity in opaque contexts.

Referential Opacity in Imperative Languages

Suppose we have the program variables `x` and `y` which correspond to different memory locations, but both store the value `2`.

Compare the assignment statements

```
z := y;      z := x;
```

Clearly they have the same result, that is to set `z` to `2`. So the principle of substitutivity holds in this case; the RHS of an assignment statement is referentially transparent.

Now consider these:

```
y := 1;      x := 1;
```

Obviously they have different effects. So the LHS of an assignment statement is referentially opaque.

We will solve this problem by translating statements into *intensional logic*.

Our System of Intensional Logic *IL*

By *state* we will mean the internal memory configuration of the computer. We introduce three new operators for dealing with sense and denotation in imperative programming languages.

- **Extension** $^\vee$
  $^\vee x$ gives us the value stored in location x in the *current* state. Intuitively this corresponds to *dereferencing* x. For example, if the value currently stored in x is 2, then
  $$^\vee x = 2.$$
- **State switcher** $\langle \cdot / \cdot \rangle$
  $\langle x/n \rangle$ changes the state by overwriting the value in location x with n. This of course models *assignment*.

Combining the two, we intuitively get

$$\langle x/n \rangle {^\vee x} = n$$

- **Intension** $\wedge$

  Intension is *almost* the inverse of extension , i.e. ${}^{\vee}{}^{\wedge}t = t$ for any t, but ${}^{\wedge}{}^{\vee}t = t$ is not true in general.

  It can be interpreted (approximately) as a “thunk” or “loaded reference”. This is far more easily demonstrated than explained — we start with a simple example.

$$x = {}^{\wedge}2$$

  means that x is roughly a “virtual location” that holds 2:

$${}^{\vee}x = {}^{\vee}{}^{\wedge}2 = 2.$$

  Intension becomes more interesting when it’s applied to entire expressions:

$$y = {}^{\wedge}({}^{\vee}x + {}^{\vee}x)$$

  When we dereference y, we evaluate ${}^{\vee}x$ in the *present context*. For example:

$$\langle x/1\rangle {}^{\vee}y = \langle x/1\rangle({}^{\vee}x + {}^{\vee}x) = 1 + 1 = 2$$

The Formalized System ***IL***

IL is a higher order typed logic. The types $\tau, \ldots$ are generated by

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

where N and B are the types of *natural numbers* and *booleans*, S is the type of *states* and $\tau_1 \to \tau_2$ is the type of *functions*.

Notice that S is a *hidden* type — there can be no terms of type S in ***IL***.

For each type τ we have

- a set of *variables* of type τ
- a set of *constants* of type τ, and
- the *domain* of τ, D_τ. e.g. $D_{\mathsf{B}} = \{\mathbf{T}, \mathbf{F}\}$, $D_{\mathsf{N}} = I\!N$

A *valuation* ρ is a function that assigns to each variable and constant of type τ a value in D_τ. Constants are obviously always assigned the same values.

We define the special constants

- $\textbf{\textit{Loc}}_g = \{X, Y, \ldots : (\mathsf{S} \longrightarrow \mathsf{N})\}$ "global" locations
- $\textbf{\textit{Array}} = \{A, \ldots : (\mathsf{N} \longrightarrow (\mathsf{S} \longrightarrow \mathsf{N}))\}$ arrays
- $\textbf{\textit{Loc}}_a = \{A(n) \mid A \in \textbf{\textit{Array}} \wedge n \in \mathbb{N}\}$ array elements
- $\textbf{\textit{Loc}} = \textbf{\textit{Loc}}_g \cup \textbf{\textit{Loc}}_a$

The global locations and array elements are disjoint sets, that is they never *share* locations.

Now define ***State*** to be the set of all states, i.e. functions $\sigma : \textbf{\textit{Loc}} \longrightarrow \mathbb{N}$.

The meaning of an ***IL*** expression $\alpha : \tau$ is then defined by structural recursion, relative to a state σ and valuation ρ:

$$[\![\alpha]\!]\sigma\rho.$$

We won't deal with the semantics here, for a full development see Hung [1990].

IL has the usual

- arithmetic operations on N $+, -, \times, \ldots,$
- relations on N $<, =, \ldots,$
- propositional connectives on B $\vee, \wedge, \neg, \ldots,$
- quantifiers $\forall, \exists$,
- function application $\cdot(\cdot)$,
- lambda abstraction

plus our *modal* operators

- ${}^\wedge\alpha : (\mathsf{S} \to \tau)$ for $\alpha : \tau$,
- ${}^\vee\alpha : \tau$ for $\alpha : (\mathsf{S} \to \tau)$, and
- $\langle x/e \rangle \alpha : \tau$ for $\alpha : \tau$, $x : (\mathsf{S} \to \mathsf{N})$ and $e : \mathsf{N}$.

Rigidity and β-Conversion

One of the key features of ***IL*** is its restricted form of β-conversion. First we define what it means for a term to be *rigid*.

Definition (Rigidity):
A term α is rigid if its value is the same in all states, i.e.

$$[\![\alpha]\!]\sigma_1 = [\![\alpha]\!]\sigma_2 \quad \textit{for all} \quad \sigma_1,\, \sigma_2 \in \textbf{\textit{State}}$$

Theorem (β-Conversion):
For any ***IL*** *term $(\lambda x \cdot \alpha)(\beta)$, if either*

(1) β is rigid, or

(2) no free occurrence of x in α lies within the scope of $^\wedge$ or $\langle\cdot/\cdot\rangle$, then

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

where $\cong$ denotes semantic equivalence.

Our Programming Language

In order to demonstrate translation into ***IL***, we'll use a simple Algol-like language. It consists of

- `v`: simple variables `x`, `y`, . . . and indexed variables `a[e]`, . . . ,
- `e`: arithmetical expressions and `b`: boolean expressions,
- `S`: program statements, generated by:

$$
\begin{aligned}
\mathtt{S} ::= \;& \mathtt{skip} \\
\mid\; & \mathtt{v\ :=\ e} \\
\mid\; & \mathtt{S}_1\mathtt{;\ S}_2 \\
\mid\; & \mathtt{if\ b\ then\ S}_1\mathtt{\ else\ S}_2\mathtt{\ fi} \\
\mid\; & \mathtt{begin\ alias\ x\ =\ v;\ S\ end} \\
\mid\; & \mathtt{begin\ new\ x\ :=\ e;\ S\ end}
\end{aligned}
$$

Backward Predicate Transformation

Consider the Hoare triple

$$\{P\}\,\mathsf{S}\,\{Q\}.$$

We consider assertions P, Q that extend the boolean expressions of our programming language by allowing *quantifiers*.

Above, Q has type B. Consider $q = {}^\wedge Q$, which has type $(\mathsf{S} \longrightarrow \mathsf{B})$.

q is a *state predicate*. Intuitively, it partitions ***State*** into two classes, that is, states that satisfy Q and states that don't.

A program statement S is translated as a *backward predicate transformer* $\mathsf{S}' : ((\mathsf{S} \longrightarrow \mathsf{B}) \longrightarrow \mathsf{B})$. For any assertion Q,

$$\mathsf{S}'(q)$$

is the *weakest precondition* that satisfies Q after the execution of S.

Translation into Intensional Logic

We now translate the programming language into ***IL***. The translation operator is $'$.

Variables
We associate with each simple variable x an ***IL*** variable $x' : (S \longrightarrow N)$ Similarly an array variable a is translated as $a' : (N \longrightarrow (S \longrightarrow N))$.

Expressions
The translations of both arithetical expressions e and boolean expressions b are defined by structural induction on e and b; for example,

$$(e_1 + e_2)' = e_1' + e_2'.$$

An important point is that variables are *dereferenced* when viewed as a part of an expression, e.g. if $e \equiv x$ then

$$e' = {}^{\vee}x'$$

Translation of Statements

In what follows, q is always assumed to be of type $(\mathsf{S} \longrightarrow \mathsf{B})$.

To illustrate the approach, we start by translating the trivial case:

$$(\texttt{skip})' = \lambda q \cdot {}^{\vee} q.$$

If we apply this to some assertion Q, we get

$$(\texttt{skip})'({}^{\wedge}Q) = (\lambda q \cdot {}^{\vee} q){}^{\wedge}Q = Q$$

as we expect. Assignment isn't much more complicated:

$$(\texttt{x := e})' = \lambda q \cdot \langle \mathsf{x}'/\mathsf{e}' \rangle {}^{\vee} q$$

This justifies the intuitive connection of the state-switcher with the assignment statement.

Our translation of statement concatenation is

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

Compare this with the backward predicate transformer $\mathtt{S}'(q)$.

The conditional is translated as follows:

$$(\mathtt{if\ b\ then\ S}_1\ \mathtt{else\ S}_2\ \mathtt{fi})'$$
$$= \lambda q \cdot (\mathtt{b}' \wedge \mathtt{S}_1'(q)) \vee (\neg\mathtt{b}' \wedge \mathtt{S}_2'(q)).$$

Now to the `alias`-block. Its function is to make `x` refer to the same location as `v` in `S`. This is where ***IL*** shines; the translation really couldn't be simpler:

$$(\mathtt{begin\ alias\ x\ =\ v;\ S\ end})' = (\lambda\mathtt{x}' \cdot \mathtt{S}')(\mathtt{v}')$$

The magic behind this lies in our restrictive β-conversion rule. $\mathtt{v}'$ can't be substituted into any intensional contexts unless it's rigid.

Memory Allocation and the new-block

Translation of the `new`-block requires some sort of mechanism to provide new memory locations "on demand". An ***IL*** array is well suited to the task.

We will use two special ***IL*** variables $j : (\mathsf{S} \longrightarrow \mathsf{N})$ and $\ell : (\mathsf{N} \longrightarrow (\mathsf{S} \longrightarrow \mathsf{N}))$, where j is a counter used to index the *allocator array* ℓ.

$\ell({}^{\vee} j)$ provides us with the current "empty location", and incrementing j *allocates* a new cell. By adhering to the policy of always incrementing j *immediately* after making use of $\ell({}^{\vee} j)$, we ensure that cells are never reallocated (to avoid unnecessary complication).

We will use artificial program variables `l` and `j` for illustrative purposes — they are completely invisible to the programmer. Naturally we set $\mathtt{l}' = \ell$ and $\mathtt{j}' = j$.

The intuition behind the translation of the `new`-block is that we can model `begin new x := e; S end` as follows:

```
begin alias x = l[j]; (j := j + 1; x := e; S) end
```

We proceed by

(1) “getting” the location of `l[j]` by aliasing `x` to it,

(2) incrementing `j` (as per our policy), then

(3) setting `x` to `e`.

The translation is:

$$(\texttt{begin new x := e; S end})' = (\lambda \mathsf{x}' \cdot \langle j/{}^{\vee}j+1\rangle\langle \mathsf{x}'/\mathsf{e}'\rangle \mathsf{S}')(\ell({}^{\vee}j)')$$

This last is my own contribution to the theory. The translation of the `new`-block in Hung [1990] is

$$\lambda q \cdot \exists n \big[\langle \ell(n)/\mathsf{e}'\rangle(((\lambda \mathsf{x}' \cdot \mathsf{S}')(\ell(n)))({}^{\wedge}[n = {}^{\vee}j \wedge \langle j/{}^{\vee}j+1\rangle {}^{\vee}q]))\big].$$

Final Notes

Many aspects of the theory were either skimmed or omitted altogether from this presentation, for time and complexity reasons.

One interesting result that deserves mention is the *state-switcher free precondition*. By using a number of *state-switcher reductions*, Hung [1990] proves that the precondition can always be expressed in the assertion language alone, without using state switchers.

Also interesting is the application of the theory to constructs involving “double intensionality”, such as pointers and pass-by-name parameters.

References

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

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

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

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

J.I. Zucker and H.K. Hung [1991], *Program Semantics, Intensional Logic and Compositionality*, in *Proceedings of the First Montreal Workshop on Programming Language Theory*, April 1991.