The Guarded Command Language (GCL)

The GCL is intended to be a simple, abstract imperative programming language. The constructs here can easily be mapped into C/C++/Java etc.

Let $S, S_0, S_1, \ldots$ be statements, E be an expression, $B, B_0, \ldots$ be boolean expressions, x be any identifier and T be any type. Then, the syntax of statements in GCL is defined as follows:

$$
\begin{array}{rcll}
S & ::= & \mathbf{skip} & \textit{no-op} \\
 & | & x := E & \textit{assignment} \\
 & | & S_1; S_2 & \textit{sequencing} \\
 & | & \mathbf{if}\ B_0 \rightarrow S_0 \ [\!]\ B_1 \rightarrow S_1 \ [\!]\ \ldots \ [\!]\ B_n \rightarrow S_n\ \mathbf{fi} & \textit{selection} \\
 & | & \mathbf{do}\ B \rightarrow S\ \mathbf{od} & \textit{iteration} \\
 & | & [\![\mathbf{var}\ x : T; S]\!] & \textit{block}
\end{array}
$$

Notes

- We usually assume that the evaluation of an expression E has no *side-effects*, so we disallow, for example, pre-increment and post-increment expressions (`++x`, `--x`) etc.
- Often we use logical operators for and/or in boolean expressions ($B_1 \wedge B_2$, $B_1 \vee B_2$). The semantics of these in GCL is *strict* unlike the standard C/C++/Java non-strict boolean operators: `&&, ||`. Use an `if` statement in GCL to get the non-strict semantics.
- The assignment statement is sometimes generalised to allow multiple simultaneous assignments:

  $$x_0, x_1, \ldots, x_n \ := \ E_0, E_1, \ldots, E_n$$

  which means: evaluate each of $E_0, E_1, \ldots, E_n$ and then assign the resulting values to $x_0, x_1, \ldots, x_n$ respectively.
- The `if` statement above is somewhat generalised - a mix of a normal `if` and a `case` statement. The standard `if/else` construct is represented by:

  $$\mathbf{if}\ B_0 \rightarrow S_0 \ [\!]\ \neg B_0 \rightarrow S_1\ \mathbf{fi}$$

  and the simple `if/then` construct (i.e. with no else) is

  $$\mathbf{if}\ B_0 \rightarrow S_0 \ [\!]\ \neg B_0 \rightarrow \mathbf{skip}\ \ \mathbf{fi}$$
- Our version of the `if` statement is intended to be deterministic, in that we test each of $B_0, B_1, \ldots B_n$ in turn. It is also possible to define a version of `if` that simply executes *one* of the S_i corresponding to a true B_i.
- You will often see the iteration construct generalised to look like the `if` statement; for example:

  $$\mathbf{do}\ B_0 \rightarrow S_0 \ [\!]\ B_1 \rightarrow S_1 \ [\!]\ \ldots \ [\!]\ B_n \rightarrow S_n\ \mathbf{od}$$

  meaning: on each iteration, find the first B_i that's true, and execute the corresponding S_i; the loop terminates when none of the B_i are true. Since this is pretty much the equivalent of putting an `if` inside our version of the `do` loop (and since it's not often used), we'll stick with the simpler version above.

References

All of the following are in section 005.1 of the NUIM library:

- *Programming : the derivation of algorithms* by Anne Kaldewaij
- *Program derivation : the development of programs from specifications* by Geoff Dromey
- *The science of programming* by David Gries
- *A discipline of programming* by Edsger Wybe Dijkstra

Hoare Triples

The meaning of a triple of the form: $\{P\}\,S\,\{Q\}$ is that:

All executions of S starting in a state satisfying P will terminate in a state satisfying Q.

While we will be using this to prove programs correct, it is also usable as a mechanism for defining the **formal semantics** of a programming language.

Programming Rules

We assume in what follows that none of the expressions throws an exception when evaluated.

- **no-op**
  $\{P\}\,\textbf{skip}\ \{Q\}$ is equivalent to $(P \Rightarrow Q)$
- **assignment**
  $\{P\}\,x := E\,\{Q\}$ is equivalent to $(P \Rightarrow Q[x := E])$
- **sequencing**
  $\{P\}\,S; T\,\{Q\}$ is equivalent to saying that:
  a predicate R exists such that $\{P\}\,S\,\{R\}$ and $\{R\}\,T\,\{Q\}$
- **selection**
  $\{P\}\,\textbf{if}\ B_0 \rightarrow S_0 \,[]\, B_1 \rightarrow S_1 \,[]\, \cdots \,[]\, B_n \rightarrow S_n\ \textbf{fi}\,\{Q\}$ is equivalent to:
  $P \Rightarrow (B_0 \vee \ldots \vee B_n)$ and
  $\forall i : N | 0 \leq i \leq n \cdot \{P \wedge \neg(B_0 \wedge \ldots \wedge B_{i-1}) \wedge B_i\}\, S_i\, \{Q\}$
- **iteration**
  If there exists a loop invariant P such that:
  (i) $\{P \wedge B\}\,S\,\{P\}$
  (ii) $(P \wedge \neg B) \Rightarrow Q$
  and there is some integer function t such that:
  (iii) $(P \wedge B) \Rightarrow (t \geq 0)$
  (iv) $\{P \wedge B \wedge (t = C)\}\,S\,\{P \wedge (t < C)\}$
  then:
  $\{P\}\,\textbf{do}\ B \rightarrow S\ \textbf{od}\,\{Q\}$
- **block**
  $\{P\}\,[\![\textbf{var}\ x : T; S]\!]\,\{Q\}$ is equivalent to $\{P\}\,S\,\{Q\}$
  provided that x does not occur free in either P or Q.

Logical Rules

- **No Miracles:**
  $\{P\}\,S\,\{false\}$ is equivalent to $(P \iff false)$
- **You can always strengthen a precondition:**
  $(P_0 \Rightarrow P)$ and $\{P\}\,S\,\{Q\}$ implies $\{P_0\}\,S\,\{Q\}$
- **You can always weaken a postcondition:**
  $\{P\}\,S\,\{Q\}$ and $(Q \Rightarrow Q_0)$ implies $\{P\}\,S\,\{Q_0\}$
- **Putting two postconditions together:**
  $\{P_1\}\,S\,\{Q\}$ and $\{P_2\}\,S\,\{Q\}$ is equivalent to $\{P_1 \vee P_2\}\,S\,\{Q\}$
- **Putting two preconditions together:**
  $\{P\}\,S\,\{Q_1\}$ and $\{P\}\,S\,\{Q_2\}$ is equivalent to $\{P\}\,S\,\{Q \wedge Q_2\}$
- **Hiding a variable**
  If P is some predicate that uses some local variable x,
  then you can always "hide" this variable by existentially quantifying it.
  Thus P can be changed to $\exists \hat{x} : N \cdot P[x := \hat{x}]$ provided that $\hat{x}$ is some new variable.