Abstract. By extending the underlying data structure by new elements, we also extend the input/output relation generated by a program i.e., no existing run is killed, and no new one lying entirely in the old structure is created. We investigate this stability property for the weak second order semantics derived from nonstandard time models. It turns out that the light face, i.e., parameterless collection principle always induces stable semantics, but the bold face one may be unstable. We give an example where an elementary extension kills a 'bold face run', showing also that the light face semantics is strictly weaker than the bold face one.

1. Definitions, notation

While we refrain from explaining the intuition behind the concepts used in this paper (which, nevertheless, can be found in [5], [8], [1] etc.), we give a detailed explanation of the ideas. The definitions given in this section are rather general and do not necessarily follow the standard lines, but this allows us to avoid a couple of technical difficulties arising otherwise.

First of all, we identify programs with the state transition relations they induce. Therefore a program $\pi$ is a (binary) relation on the set of states which we shall denote by $D$ i.e., $\pi \subseteq D \times D$. If $p$ and $q$ are states and $(p, q) \in \pi$, then we say that $q$ is (one of the) successor state(s) of $p$. A computation is a sequence of successor states, the first element in the sequence is the starting or initial state, the last one is the halting state of the computation. (At the moment we do not bother with the possibility that a computation might have a continuation i.e., a halting state is not necessarily a state without successor.) Since we allow more than one successor states, this definition allows us to speak about non-deterministic programs, too.

To reason about a program, we also have to known the data on which the program works. Usually this is a structure with constant, relation and function symbols collected into a similarity type (or signature, or ranked alphabet). The interpretations of these symbols represent the "built in" features of the executing computer. We have no tools to speak about the fine structure of our programs, therefore we lift the data structure into a structure on the states. This usually is quite straightforward: a state can be identified with the vector consisting of the values stored in the variables together with some indication.

* The research was partially carried out while the author visited McGill University, Montreal, Canada, and was supported in part by the Natural and Sciences and Engineering Research Council of Canada
which part of the program works. Thus the structure on states is the “Cartesian product” of the data structure. This lifting, however, may result in a richer structure if knowing exactly where the control is, requires extra information. This happens, for example, if recursive procedures, or infinite (but recursively enumerable) flow-chart diagrams are allowed. If we restrict ourselves to finite flow-chart programs, or while-programs, then the lifting yields absolutely no new information, i.e., every statement about the states can be translated back into a statement about data and vice versa.

So we assume that on the set $D$ of states we are given a first-order structure $D$ of similarity type $\tau$. By a first-order formula we always mean one of type $\tau$ with equality. Also we assume that the state transition relation $\pi$ is a relation symbol in $\tau$, which is equivalent to the assumption that $\pi$ is definable by a first-order formula.

At this point the reader is advised to check that his/her favorite programming language lies in this general framework. We emphasize that $\pi$ is not the input/output relation, which, generally, is not first-order expressible, but the next-state relation. The program text is an exact definition of the next-state relation: if the control is here, what does the computer do next? The requirement that this relation is first-order definable can be worded as follows. One can get no extra information about the data structure (using first-order formulas only) with the help of the state transition relation as a new binary relation. This not being the case, the program text itself, without any reference to its behavior or correctness, would give information about the data structure. This is obviously undesirable.

From now on we fix the similarity type $\tau$, and the program (i.e., binary relation) $\pi \in \tau$. For a $\tau$-type structure $D$, a (nondeterministic) computation is a finite sequence $\langle s_0, \ldots, s_n \rangle$ of states, i.e., elements of $D$ so that for all $i < n$, $D \models s_i \pi s_{i+1}$. Now fix the state $u \in D$. The (standard) run of $\pi$ starting from $u$ is the set of all states which can be reached by some computation having $u$ as its first element. In other words, the run is the smallest subset of states containing $u$ and closed under $\pi$:

$$R(u) = \bigcap \{ X \subseteq D : u \in X \text{ and } X \text{ is closed under } \pi \}.$$ 

We shall write $D \models \text{cl}(X)$ to denote that $X \subseteq D$ is closed under $\pi$, i.e., $\text{cl}(X)$ is an abbreviation for

$$\forall x \forall y (x \in X \land x \pi y \rightarrow y \in X).$$

Using this, $R = R(u)$ is the run defined above if and only if

$$D \models u \in R \land \text{cl}(R) \land \forall X (u \in X \land \text{cl}(X) \rightarrow R \subseteq X).$$

Here the second-order variable $X$ runs through $P(D)$, the collection of all subsets of $D$. The input/output relation generated by $\pi$ consists of all pairs of states $(u, v)$ such that $v \in R(u)$. This semantics is just an alternative of the so-called fixed-point semantics since $R(u)$ is, in some sense, a fixed point of $\pi$. 