Yale University
Department of Computer Science

Single Assignment Semantics for Imperative Programs

Bjorn Lisper

YALEU/DCS/TR-703
May 1989

This paper is supported in part by The Office of Naval Research under Contract No. N00014-86-K-0310. It was written in August of 1988 and has just become a Yale TR.

Single Assignment Semantics for Imperative Programs

Björn Lisper

Computer Science Department
Yale University
New Haven, CT06520

Abstract

We give semantics for imperative while-programs by assigning a *set of single-assignments* and a *value function* to each program. The semantics is shown to be consistent with Dijkstra's predicate transformers. Every program execution is seen to generate a set of events with data dependencies. The events are the single-assignments and the data dependencies between them are given by common input/output variables. Program parts that always generate isomorphic sets can sometimes be effectively parallelized, using known methods for mapping sets of events with static dependencies to a space-time. Thus, the semantics can be helpful for finding techniques for compilation of while-programs to fine-grain parallel and pipelined target architectures. It also shows the relationship between imperative languages and single-assignment languages.

1 Introduction

The advent of parallel computers has created new demands on programming languages and their implementation, both from a theoretical and a practical point of view. In order to utilize the extended computing power offered by parallel hardware, new techniques are required for specifying and analyzing the task to be done. In areas like scientific computing, where the tasks are computation-intensive, it is essential that the parallel hardware is utilized well.

Fine-grain parallelism is of interest in many contexts; one of them is the aforementioned scientific computing. A theoretically appealing family of languages, suitable for fine-grain parallelity, is *single-assignment languages* [1,23]. In such a language, a statement is executed only once. Therefore, a variable can be assigned only once, thus explaining the term "single-assignment". Since variables are assigned only once, they stand for *values*, not memory locations. Whether two statements in a single-assignment program can be executed in parallel or not depends on if any values produced by one statement are used by the other. If there are, then there is a *data dependency* from the first statement to the second and they must be executed in that way. If not, they can be executed in parallel. Thus, parallellity is implicitly given by input/output variable relationships.

For various reasons, single-assignment languages are not frequently used in practice. Instead, fine-grain parallel programming is usually done using some conventional imperative language (read: FORTRAN). Even though such languages are of a sequential nature, parallellism can be introduced. This can be done either by explicit language constructs, like "**in parallel do**" or vector operations, or by a compile-time analysis of the dependencies in the program, to detect implicit parallelism [14,15,16,27,28,31].

Semantics for imperative languages is usually based on a global state transition model, where states are maps from program variables to values. The model may be explicit (the "interpretive" and the "computational" model, [12]), implicit through propositional formulas [7,11] or predicate

transformers [6], or even more implicit as in denotational semantics (for an introduction, see [21]). See also [9]. These semantics will, however, not provide a suitable support for the dependence analysis preceding the possible parallelization of the program. This is since global state transition models of this kind are inherently sequential.

In this paper we will develop an operational semantics for a simple but nontrivial class of imperative programs. Standard methods for dependence analysis of imperative programs consider dependencies between imperative statements, which necessitates the introduction of several dependence types such as *data dependence*, *antidependence* and *output dependence* [14,27,28,31]. This obscures the analysis and restricts the parallelism that can be detected. The complications arise from the fact that an imperative statement can be executed several times. The situation becomes a lot simpler if we consider dependencies between *statement executions* instead of statements. Every statement execution takes *values* as inputs and occurs only once. If the atomic statements are assignments, the statement executions can indeed be seen as single-assignments. Thus, the only dependence between statement executions will be pure data dependencies. Our semantics can be seen as a description of how a program will generate a set of statement executions. An imperative program with assignment statements will therefore in principle generate a single-assignment program. The semantics does have a state model, but the states are, rather than simple maps from program variables to values, recordings of the execution as sets of single-assignments. Thus, a state contains the true dependencies between the statement executions that have taken place so far.

This approach also has another interesting implication from a compiler-construction point of view. If it is possible to predict that a certain state will be reached, then the execution of statements up to that point can be scheduled, according to the partial ordering given by the data dependencies, already at the time of prediction. Cases where future states can be predicted are straight-line code and loops with branching-free bodies, where the steps and limits of the loop indices are known at compile-time. Thus, their execution can be scheduled at compile-time. An interesting, recently developed class of scheduling methods, seemingly applicable for this, are the so-called *space-time mapping* methods. These methods were originally developed for the purpose of synthesizing synchronous hardware [3,4,13,17,19,20,18,24,25,26,29,30]. It follows that such parts of a program can be scheduled to execute, with an absolute minimum of overhead, on a tightly coupled synchronous part of the system such as a vector pipeline or a systolic array. The approach therefore seems particularly relevant when fine-grain parallelism is desired.

Let us finally mention, that the *static single assignment form* (SSA form) of a program has been considered recently for use in optimizing compilers for conventional computers [2,5]. The SSA form is essentially a description of how the single-assignments are generated, very much in accordance with our semantics.

2 Preliminaries

We assume that the reader is familiar with the basic concepts *variables*, *formal expressions* and *polynomials* of universal algebra [8]. The expression composed from a function symbol f, with arity $n(f)$, and $n(f)$ expressions $\mathbf{p}_1, \ldots, \mathbf{p}_{n(f)}$, is denoted $f \bullet \langle \mathbf{p}_1, \ldots, \mathbf{p}_{n(f)} \rangle$. $varset(\mathbf{p})$ is the set of variables in the expression $\mathbf{p}$. ϕ denotes the *natural homomorphism* that maps every expression to its corresponding polynomial.

A *substitution* σ in X is a partial function from the variables in X to expressions over X. σ can naturally be extended to general expressions: $\sigma(\mathbf{p})$ is the expression obtained, when all occurrences in $\mathbf{p}$of variables x for which $\sigma(x)$ is defined are replaced with $\sigma(x)$. The *domain* of σ, $dom(\sigma)$, is the set of all variables x for which $\sigma(x)$ is defined. The *range* of σ, $rg(\sigma)$, is the set of all variables

that occurs in any $\sigma(x)$, i.e. $\bigcup(\, varset(\sigma(x)) \mid x \in dom(\sigma)\,)$. A *finite* substitution is defined only for a finite number of variables.

Definition 1 *(Substitution composition) For all substitutions σ_1, σ_2*

$$\sigma_1 \circ \sigma_2 = \{\, x \leftarrow \sigma_1(\sigma_2(x)) \mid x \in dom(\sigma_2)\,\} \cup \sigma_1|_{dom(\sigma_1)\setminus dom(\sigma_2)}.$$

$\sigma_1|_{dom(\sigma_1)\setminus dom(\sigma_2)}$ is σ_1 restricted to the variables in $dom(\sigma_1)$ but not in $dom(\sigma_2)$. It is not hard to show, that $\sigma_1 \circ \sigma_2(\mathbf{p}) = \sigma_1(\sigma_2(\mathbf{p}))$ for all expressions $\mathbf{p}$. Thus, composition of substitutions is essentially composition of functions. For more detail, see [19].

Given a substitution σ in X, define the relation $\prec_\sigma$ on X by: $x \prec_\sigma y$ iff $x \in varset(\sigma(y))$. We say that σ is *causal* iff $\prec_\sigma^+$ is well-founded (a strict partial order and no infinite decreasing chains exist). For any causal σ, we define ψ_σ as the unique substitution with domain $dom(\sigma)$ such that $\psi_\sigma = \psi_\sigma \circ \sigma$. ψ_σ is well-defined due to the well-foundedness of $\prec_\sigma$. Call a substitution σ *closed* iff $rg(\sigma) \subseteq dom(\sigma)$:

Lemma 1 *If σ is causal and closed, then $rg(\psi_\sigma) = \emptyset$.*

Proof. By induction over $\prec_\sigma$. Consider all x in $dom(\psi_\sigma) = dom(\sigma)$. If x is minimal, then $\sigma(x)$ is a ground term (since $rg(\sigma) \subseteq dom(\sigma)$). Thus, $\psi_\sigma(x) = \psi_\sigma \circ \sigma(x) = \sigma(x)$, and $varset(\psi_\sigma(x)) = \emptyset$. If x is not minimal, assume that $varset(\psi_\sigma(x')) = \emptyset$ for all $x' \prec_\sigma x$. Then $varset(\psi_\sigma(x)) = varset(\psi_\sigma \circ \sigma(x)) =$ (by [19, lemma D6]) $= \bigcup(\, varset(\psi_\sigma(x')) \mid x' \prec_\sigma x\,)$. By the induction hypothesis, this union is empty. The lemma follows. ■

Let σ be causal and closed. Let b be a boolean expression such that $varset(b) \subseteq dom(\sigma)$. Then we write $\sigma \models b$ iff $\psi_\sigma(b) = true$. (Note, that $\psi_\sigma(b)$ always will be a ground term under the given conditions.)

3 A formalization of single-assignments

Single-assignment languages [1,23] have been considered for a while as a promising class of languages for programming parallel machines. Their main attractive feature is the implicit parallellism; two statements need to be executed in sequence only if there is a data dependence between them. This is due to two basic restrictions put on these programs: the first is the *single-assignment rule*, that states that a variable can be assigned at most once. Thus, a variable represents a *value* and not a memory location. The second is that there are not to be any cycles in the dependence chain. Therefore, a single assignment program can in principle be evaluated bottom up, successively, by executing assignment statements as their right-hand side variables become available.

In this paper, we will adopt causal substitutions as a mathematical model of single-assignments. Any pair $\langle x, C(x)\rangle$ in such a substitution C can be interpreted as an assignment $x := C(x)$. Since these assignments are part of a substitution (i.e. a partial function from variables) they adhere to the single assignment rule. The causality ensures that there are no infinite dependence chains downwards. The relation $\prec_C$ can be seen as a *data dependence relation*: if $x \prec_C y$, then the assignment producing y uses x as input and thus the one producing x must precede it in time. If $x \not\prec_C y$ and $y \not\prec_C x$, then they can take place concurrently.

Let C be a causal substitution. Variables in $dom(C)$ are called *assigned variables* and the others are called *free variables.* The interpretation is that free variables provide inputs to the system from

somewhere outside. Assigned variables, on the other hand, are successively assigned their values during the computation. A causal substitution can be considered as an equation system, that successively defines the assigned variables in terms of the free variables. We can give very simple *semantics* to causal substitutions when regarded as a set of single-assignments:

Definition 2 *Let C be a causal substitution. $\psi_C(x)$ is called the* output expression of x under C. *The* output function of x under C *is* $\phi(\psi_C(x))$.

The meaning of every assigned variable is thus a function, a polynomial, in the free variables. Note that the recursive definition of ψ_C simulates the computation by a construction of the corresponding expressions as the computation proceeds; for any free variable x it holds that $\psi_C(x) = x$ (i.e. symbolic input of the corresponding value) and if x is assigned, then, for all $x' \prec_C x$, $\psi_C(x')$ is substituted for x' in $C(x)$ in order to form $\psi_C(x)$. This is a symbolic evaluation of $C(x)$ with inputs x'.

4 A simple imperative language

In this section we will introduce a simple imperative language, that essentially consists of **while** programs [9,12] enriched with **if..then..else**-statements. We will also give a standard weakest precondition semantics.

4.1 Definition of the language

Since we are interested in the mathematical properties of programs rather than syntax-related issues, we define the programs in the language to be mathematical objects rather than syntactical. Let X be a set of *program variables* ranging over some domain D, let $B(X)$ be the set of formal boolean-valued expressions over X, and let us call the corresponding language $L(X)$:

- *skip* $\in L(X)$ (empty statement).
- If σ is a finite substitution in X, then $\sigma \in L(X)$ (concurrent assignment).
- If $l, l' \in L(X)$, then $l; l' \in L(X)$ (concatenation).
- If $b \in B(X)$ and if $l, l' \in L(X)$, then **if** b **then** l **else** $l' \in L(X)$ (binary choice).
- If $b \in B(X)$ and if $l \in L(X)$, then **while** b **do** $l \in L(X)$ (iteration).

Properties of programs in $L(X)$ can be proved by induction on program structure, according to the inductive definition above.

4.2 Weakest precondition semantics

Following Dijkstra [6] we can give weakest precondition (wp) semantics to $L(X)$. In the following $b \in B(X)$, $l, l' \in L(X)$, and Q denotes an arbitrary predicate in program variables:

(wp1) $wp(skip, Q) \iff Q$.

(wp2) $wp(\sigma, Q) \iff \sigma(Q)$.

(wp3) $wp(l; l', Q) \iff wp(l, wp(l', Q))$.

(wp4) $wp(\textbf{if } b \textbf{ then } l \textbf{ else } l', Q) \iff (b \implies wp(l,Q)) \wedge (\neg b \implies wp(l',Q))$.

(wp5) Define $H_0(Q) \iff Q \wedge \neg b$ and for all $k > 0$ $H_k(Q) \iff wp(l, H_{k-1}(Q)) \vee (Q \wedge \neg b)$. Then $wp(\textbf{while } b \textbf{ do } l, Q) \iff \exists k \geq 0[H_k(Q)]$.

(wp2) is the obvious generalization of wp for assignment statements to concurrent assignments. (wp4), (wp5) are obtained from Dijkstra's wp's for the guarded commands IF and DO restricted to deterministic binary choice and iteration respectively. (wp5) can be defined equivalently as follows:

(wp5') Define $G_0(Q) \iff Q \wedge \neg b$ and $G_k(Q) \iff wp(l, G_{k-1}(Q))$ for all $k > 0$. Then $wp(\textbf{while } b \textbf{ do } l, Q) \iff \exists k \geq 0[G_k(Q)]$.

The proof of the equivalence between (wp5) and (wp5') is omitted here. It relies on that the **while** statement is deterministic. As an immediate spinoff from (wp2) we obtain the following result for composing chains of concurrent assignment statements.

Proposition 1 $wp(\sigma_1; \ldots; \sigma_n, Q) \iff wp(\sigma_1 \circ \cdots \circ \sigma_n, Q)$ *for all predicates* Q *and concurrent assignments* $\sigma_1, \ldots, \sigma_n$.

Proof. The result follows immediately from a repeated application of (wp3), (wp2), and the associativity of substitution composition. ■

Proposition 1 can be used to transform chains of assignments into one equivalent assignment. The resulting right-hand expressions can then be treated at compile-time to yield a maximally efficient evaluation pattern. This is an important technique in parallelizing compilers [31] and it can also be used in conventional compilers to improve the performance for techniques such as detection of common subexpressions. The composed substitution $\sigma_1 \circ \cdots \circ \sigma_n$ can be successively computed using 1.

5 Single assignment semantics

SA (single-assignment) semantics is an operational semantics, that models execution by constructing a causal substitution as the execution goes on. Every execution of an assignment statement adds single-assignments to the previously generated causal substitution. Variables representing the old *values* of the program variables are substituted for the program variables in the expressions in the right-hand side of the assignment. Distinct variables, representing new values of assigned program variables, are substituted for these in the left-hand side. The result is distinct single-assignments.

If the above is to work, we must be able to generate fresh variables every time a program variable is assigned a new value. Thus, we postulate a countable set sX, such that $X \subset sX$, of so-called *value variables* and an injective *successor function* $s: sX \to sX \setminus X$. s will provide a distinct variable each time a fresh variable is needed. s can also be seen as a relation on sX: $x' \; s \; x'' \iff x'' = s(x')$. The transitive closure s^+ of s is easily seen to be a strict partial order (which consists of one linear order for each $x \in X$). For any set of variables $Y \subseteq sX$, we define $s^+(Y) = \{ x \mid y \; s^+ \; x \wedge y \in Y \}$. From the above, we deduce the following lemma:

Lemma 2 *For any* $x, y \in sX$, $x \notin s^+(\{x\})$, *and furthermore* $x \neq y \implies s^+(\{x\}) \cap s^+(\{y\}) = \emptyset$.

We must at every point keep track of which value variable that represents the current value of every program variable. To do this we will use *value functions*, which are injective functions $X \to sX$. Note that a value function formally is a substitution.

Definition 3 *The set of* states for X, $S(X)$, *is the set of pairs* $\langle C, v\rangle$ *where* C *is a causal, closed substitution in* sX, v *is a value function* $X \to sX$ *and the following holds:*

1. $v(X) \subseteq dom(C)$.
2. $s^+(v(X)) \cap dom(C) = \emptyset$.

Note, that lemma 1 and 2 in definition 3 above implies that $\psi_C \circ v(x)$ is a ground term for any $x \in X$. Thus, a state binds every program variable to a value. The SA-semantics for a program in $L(X)$ can now be given by the aid of a partial function:

$$\mathcal{S}{:}(S(X) \times L(X)) \to S(X).$$

$\mathcal{S}$ maps pairs of states and programs to states and is thus a state transition function. For any state t and program l, we define $\langle C_{\mathcal{S}}(t,l), v_{\mathcal{S}}(t,l)\rangle = \mathcal{S}(t,l)$. In the following definition, $\langle C, v\rangle$ ranges over $S(X)$. As usual, $b \in B(X)$ and $l, l' \in L(X)$.

(SA1) $\mathcal{S}(\langle C, v\rangle, \mathit{skip}) = \langle C, v\rangle$.

(SA2)
$$C_{\mathcal{S}}(\langle C, v\rangle, \sigma) = C \cup \{\, s(v(x)) \leftarrow v \circ \sigma(x) \mid x \in dom(\sigma)\,\},$$
$$v_{\mathcal{S}}(\langle C, v\rangle, \sigma)(x) = \begin{cases} s(v(x)), & x \in dom(\sigma) \\ v(x), & x \notin dom(\sigma). \end{cases}$$

(SA3) $\mathcal{S}(\langle C, v\rangle, l; l') = \mathcal{S}(\mathcal{S}(\langle C, v\rangle, l), l')$.

(SA4) $\mathcal{S}(\langle C, v\rangle, \textbf{if } b \textbf{ then } l \textbf{ else } l') = \begin{cases} \mathcal{S}(\langle C, v\rangle, l), & C \models v(b) \\ \mathcal{S}(\langle C, v\rangle, l'), & C \models \neg v(b). \end{cases}$

(SA5) $\mathcal{S}(\langle C, v\rangle, \textbf{while } b \textbf{ do } l) = \begin{cases} \mathcal{S}(\langle C, v\rangle, l; \textbf{while } b \textbf{ do } l), & C \models v(b) \\ \langle C, v\rangle, & C \models \neg v(b). \end{cases}$

Note that since b in (SA4) and (SA5) is a boolean expression in X, the truth value of $C \models v(b)$ will always be well-defined for any state $\langle C, v\rangle$. Then $\mathcal{S}(\langle C, v\rangle, l)$, as will be seen later, is defined exactly when l terminates with "input" $\langle C, v\rangle$.

For a given state t and statement l, define $\mathcal{S}^0(t,l) = t$ and $\mathcal{S}^n(t,l) = \mathcal{S}(\mathcal{S}^{n-1}(t,l), l)$ for $n > 0$; define further $\langle C^n_{\mathcal{S}}(t,l), v^n_{\mathcal{S}}(t,l)\rangle = \mathcal{S}^n(t,l)$. The following lemma will be used when proving properties about $\mathcal{S}(\langle C, v\rangle, \textbf{while } b \textbf{ do } l)$.

Lemma 3 $\mathcal{S}(\langle C, v\rangle, \textbf{while } b \textbf{ do } l)$ *is defined exactly when there exists some* $k \geq 0$ *such that:*

- *For all* $0 \leq j < k$, $C^j_{\mathcal{S}}(\langle C, v\rangle, l) \models v^j_{\mathcal{S}}(\langle C, v\rangle, l)(b)$
- $C^k_{\mathcal{S}}(\langle C, v\rangle, l) \models \neg v^k_{\mathcal{S}}(\langle C, v\rangle, l)(b)$
- $\mathcal{S}^k(\langle C, v\rangle, l)$ *is well-defined.*

Furthermore, $\mathcal{S}(\langle C, v\rangle, \textbf{while } b \textbf{ do } l) = \mathcal{S}^k(\langle C, v\rangle, l)$ *when defined.*

Proof. The lemma is shown using a least fixpoint argument. (SA5) can be seen as a recursive definition of a function f in the arguments $\langle C, v\rangle$, b, l:

$$\begin{aligned} f(\langle C, v\rangle, b, l) &= \mathit{if}(C \models v(b), f(\mathcal{S}(\langle C, v\rangle, l), b, l), \langle C, v\rangle) \\ &= \tau[f](\langle C, v\rangle, b, l), \end{aligned}$$

rewriting (SA5) using (SA3) and the *if*-function, that returns its second argument if its first is true and its third otherwise.

We first show that $f(\langle C, v\rangle, b, l)$ always is defined when some k according to above exists. $\mathcal{S}$ is assumed to be naturally extended, and it is thus monotonic [22]. Furthermore, the well-definedness of $\mathcal{S}^k(\langle C, v\rangle, l)$ implies the same for $\mathcal{S}^{k-1}(\langle C, v\rangle, l)$ and recursively for $\mathcal{S}^j(\langle C, v\rangle, l)$, for $0 \leq j < k$. Thus, the truth values of the conditions of form $C' \models v'(b)$ are also well-defined, for $\langle C', v'\rangle = \mathcal{S}^j(\langle C, v\rangle, l)$, $0 \leq j \leq k$. It follows that we will not have to worry about the monotonicity of "$\models$". The *if*-function is monotonic. Thus, the least fixpoint f^* of the functional τ exists and we take $\mathcal{S}(\langle C, v\rangle, \textbf{while } b \textbf{ do } l) = f^*(\langle C, v\rangle, b, l)$. Define the *approximation functions* f^j, $j \geq 0$ by: $f^0 = \Omega$ and $f^j = \tau[f^{j-1}]$ for $j > 0$. We obtain, denoting $C^i_{\mathcal{S}}(\langle C, v\rangle, l)$ by $C^i_{\mathcal{S}}$ and $v^i_{\mathcal{S}}(\langle C, v\rangle, l)$ by $v^i_{\mathcal{S}}$,

$$\begin{aligned}
f^0(\langle C, v\rangle, b, l) &= \omega \\
f^1(\langle C, v\rangle, b, l) &= \mathit{if}(C \models v(b), f^0(\mathcal{S}(\langle C, v\rangle, l), b, l), \langle C, v\rangle) \\
&= \mathit{if}(C^0_{\mathcal{S}} \models v^0_{\mathcal{S}}(b), \omega, \langle C^0_{\mathcal{S}}, v^0_{\mathcal{S}}\rangle) \\
f^2(\langle C, v\rangle, b, l) &= \mathit{if}(C \models v(b), f^1(\mathcal{S}(\langle C, v\rangle, l), b, l), \langle C, v\rangle) \\
&= \mathit{if}(C^0_{\mathcal{S}} \models v^0_{\mathcal{S}}(b), \mathit{if}(C^1_{\mathcal{S}} \models v^1_{\mathcal{S}}(b), \omega, \langle C^1_{\mathcal{S}}, v^1_{\mathcal{S}}\rangle), \langle C^0_{\mathcal{S}}, v^0_{\mathcal{S}}\rangle) \\
&\vdots
\end{aligned}$$

It is not hard to see that if k fulfils the conditions, then $f^{k+1}(\langle C, v\rangle, b, l) = \mathcal{S}^k(\langle C, v\rangle, l) \neq \omega$. It follows that $f^*(\langle C, v\rangle, b, l) = f^{k+1}(\langle C, v\rangle, b, l) = \mathcal{S}^k(\langle C, v\rangle, l)$ (otherwise f^* cannot be a l.u.b. of the approximation functions).

If, on the other hand, we assume that $\mathcal{S}(\langle C, v\rangle, \textbf{while } b \textbf{ do } l) = f^*(\langle C, v\rangle, b, l)$ is defined, then there must be some $k \geq 0$ such that $f^{k+1}(\langle C, v\rangle, b, l) = f^*(\langle C, v\rangle, b, l) \neq \omega$ (again since f^* is a l.u.b.). Furthermore, there must be a least such k. It follows that for all $0 \leq j < k$ holds that $C^j_{\mathcal{S}} \models \neg v^j_{\mathcal{S}}(b)$, $C^k_{\mathcal{S}} \models v^k_{\mathcal{S}}(b)$, and $f^k(\langle C, v\rangle, b, l) = \mathcal{S}^{k-1}(\langle C, v\rangle, l)$. ■

Define, for any $l \in L(X)$, $l^0 = \mathit{skip}$ and $l^k = l^{k-1}; l$ for $k > 0$. It is easily seen that $\mathcal{S}(t, l^k) = \mathcal{S}^k(t, l)$. So if $\mathcal{S}(t, \textbf{while } b \textbf{ do } l)$ is defined, then $\mathcal{S}(t, \textbf{while } b \textbf{ do } l) = \mathcal{S}(t, l^k)$ for some k. In induction proofs of properties of $\mathcal{S}$, the step for **while** b **do** l can then be proved using the result for *skip*, the induction hypothesis for l, and repeatedly the result for concatenation.

We will now justify definition 3 of states, by showing that $S(X)$ is closed under $\mathcal{S}$:

Theorem 1 *If $t \in S(X)$, then, for all $l \in L(X)$, holds that $\mathcal{S}(t, l) \in S(X)$ whenever defined.*

Proof. By induction over program structure. Let $\langle C, v\rangle = t$.

- $l = \mathit{skip}$: trivially true.
- $l = \sigma$: denote $\{\, s(v(x)) \leftarrow v \circ \sigma(x) \mid x \in \mathit{dom}(\sigma) \,\}$ by c. By (SA2) follows that

$$v_{\mathcal{S}}(\langle C, v\rangle, \sigma)(\mathit{dom}(\sigma)) \subseteq s(v(X)) \tag{1}$$

and

$$v_{\mathcal{S}}(\langle C, v\rangle, \sigma)(X \setminus \mathit{dom}(\sigma)) \subseteq v(X). \tag{2}$$

Injectivity of $v_{\mathcal{S}}(\langle C, v\rangle, \sigma)$ follows from (1), (2), injectivity of v, injectivity of s and $s(v(X)) \cap v(X) = \emptyset$, which follows from property 1 and 2 of definition 3. Injectivity of v and s ensures

that c is a well-defined substitution. C is by assumption a well-defined substitution. $dom(c) \subseteq s^+(v(X))$ and property 2 then implies that

$$dom(c) \cap dom(C) = \emptyset, \tag{3}$$

i.e. $C_{\mathcal{S}}(\langle C, v\rangle, \sigma) = c \cup C$ is a well-defined substitution. We show *causality* of $c \cup C$ by showing that no cycles are introduced in $\prec_C$ when c is added (since c is finite, this suffices to ensure well-foundedness). By (3) and the closedness of C follows that $dom(c) \cap rg(C) = \emptyset$. Furthermore, since $rg(c) \subseteq v(X)$, 1 and 2 in definition 3 yields $dom(c) \cap rg(c) = \emptyset$. It follows that for any $x \in dom(c)$ there is no x' such that $x \prec_{c \cup C} x'$. Thus, no cycles are introduced. *Closedness* of $c \cup C$ follows from the closedness of C, $dom(C) \subseteq dom(c \cup C)$ and $rg(c) \subseteq v(X) \subseteq dom(C)$. 1 in definition 3 for $\mathcal{S}(\langle C, v\rangle, \sigma)$ follows, for $X \setminus dom(\sigma)$, from the same property for C, and for $dom(\sigma)$ it follows from $v_{\mathcal{S}}(\langle C, v\rangle, \sigma)(dom(\sigma)) = dom(c) \subseteq dom(c \cup C)$. Property 2 for $\mathcal{S}(\langle C, v\rangle, \sigma)$, finally, is proved by the following: since $s^+(v_{\mathcal{S}}(\langle C, v\rangle, \sigma)(X)) \subseteq s^+(v(X))$ (easily proved from (SA2)), it holds, by property 2 for $\langle C, v\rangle$, that $s^+(v_{\mathcal{S}}(\langle C, v\rangle, \sigma)(X)) \cap dom(C) = \emptyset$. Furthermore (SA2) yields $dom(c) \subseteq v_{\mathcal{S}}(\langle C, v\rangle, \sigma)(X)$. Lemma 2 then implies that $s^+(v_{\mathcal{S}}(\langle C, v\rangle, \sigma)(X)) \cap dom(c) = \emptyset$. It follows that $s^+(v_{\mathcal{S}}(\langle C, v\rangle, \sigma)(X)) \cap dom(c \cup C) = \emptyset$.

- Inductive cases: for $l'; l''$ the result follows directly from the induction hypothesis on l' and l''. For **if** b **then** l' **else** l'' it also follows from the induction hypothesis, observing that the truth value $C \models v(b)$ is well-defined. The validity for **while** b **do** l' follows from a repeated application of the result for concatenation.

■

We will now use $\mathcal{S}$ to define the meaning of a program when started from "scratch", with the program variables initialized to some values. Let id_X be the identity function on the set of program variables X (note that id_X is a value function) and let *init* be a function $X \to D$ that binds every program variable to some value in D. *init* can be interpreted as a substitution in sX, and it is causal and closed since $rg(init) = \emptyset$. $id_X(X) = X = dom(init)$ and lemma 2 implies that $s^+(id_X(X)) \cap dom(init) = \emptyset$. Thus, $\langle init, id_X\rangle$ is a valid state according to definition 3.

Definition 4 *The meaning of a program l in $L(X)$ is, for every possible init, $\mathcal{S}(\langle init, id_X\rangle, l)$.*

By theorem 1 follows that $\mathcal{S}(\langle init, id_X\rangle, l)$ always is a state when defined. Therefore, $\mathcal{S}$ really deserves the name single-assignment semantics; for any given initialization of the program variables, i.e. input to the program, the execution will be described as a set of single-assignments, that is: the causal substitution $C_{\mathcal{S}}(\langle init, id_X\rangle, l)$.

6 SA semantics and wp semantics

In this section we will prove a result regarding the relation between SA semantics and weakest preconditions. It essentially says that the truth values of a program variable predicate Q and $wp(l, Q)$ are the same, when taking into account the changes in the values of the program variables given by $\mathcal{S}(\langle C, v\rangle, l)$. Since this is the expected behavior of a predicate transformer, the result shows that SA semantics is consistent with wp semantics. First we need a simple lemma:

Lemma 4 *$C \subseteq C_{\mathcal{S}}(\langle C, v\rangle, l)$ for all $\langle C, v\rangle$, l.*

Proof. The result follows trivially from induction over program structure (the only case when single-assignments are added to C is the assignment statement). ■

Theorem 2 *For all states $\langle C, v\rangle$, all $l \in L(X)$ and all program variable predicates Q,*

$$C_{\mathcal{S}}(\langle C, v\rangle, l) \models v(wp(l, Q)) \iff v_{\mathcal{S}}(\langle C, v\rangle, l)(Q)$$

whenever $\mathcal{S}(\langle C, v\rangle, l)$ is defined.

Proof. By induction over program structure.

- $l = skip$: $v(wp(skip, Q)) \iff v(Q) \iff v_{\mathcal{S}}(\langle C, v\rangle, skip)(Q)$ for any $\langle C, v\rangle$.
- $l = \sigma$: $v(wp(\sigma, Q)) \iff v(\sigma(Q)) \iff v \circ \sigma(Q)$. By (SA2),

  $$C_{\mathcal{S}}(\langle C, v\rangle, \sigma) \models v \circ \sigma(x) = v_{\mathcal{S}}(\langle C, v\rangle, \sigma)(x)$$

  when $x \in dom(\sigma)$. When $x \notin dom(\sigma)$, it holds that $v \circ \sigma(x) = v(x) = v_{\mathcal{S}}(\langle C, v\rangle, \sigma)(x)$. Thus, $v(wp(\sigma, Q)) \iff v \circ \sigma(Q) \iff v_{\mathcal{S}}(\langle C, v\rangle, \sigma)(Q)$.

In the inductive cases we assume that the theorem holds for l' and l''. We further assume that the equalities in $C_{\mathcal{S}}(\langle C, v\rangle, l)$ hold.

- $l = l'; l''$: then

$$\begin{aligned} C_{\mathcal{S}}(\langle C, v\rangle, l) &= C_{\mathcal{S}}(\langle C, v\rangle, l'; l'') \\ &= C_{\mathcal{S}}(\mathcal{S}(\langle C, v\rangle, l'), l'') \\ &= C_{\mathcal{S}}(\langle C_{\mathcal{S}}(\langle C, v\rangle, l'), v_{\mathcal{S}}(\langle C, v\rangle, l')\rangle, l'') \\ &\supseteq C_{\mathcal{S}}(\langle C, v\rangle, l'), \end{aligned} \tag{4}$$

  so then the equalities in $C_{\mathcal{S}}(\langle C, v\rangle, l')$ hold as well. Thus,

$$\begin{aligned} v(wp(l'; l'', Q)) &\iff wp(l', wp(l'', Q)) \\ &\iff \text{by the induction hypothesis for } l', \text{ lemma 4, (4)} \\ &\iff v_{\mathcal{S}}(\langle C, v\rangle, l')(wp(l'', Q)) \\ &\iff \text{by the induction hypothesis for } l'' \\ &\iff v_{\mathcal{S}}(\langle C_{\mathcal{S}}(\langle C, v\rangle, l'), v_{\mathcal{S}}(\langle C, v\rangle, l')\rangle, l'')(Q) \\ &\iff v_{\mathcal{S}}(\langle C, v\rangle, l'; l'')(Q). \end{aligned}$$

- $l =$ **if** b **then** l' **else** l'': here the proof is by a case analysis. By (SA4)

$$\begin{cases} C_{\mathcal{S}}(\langle C, v\rangle, l) = C_{\mathcal{S}}(\langle C, v\rangle, l'), & C \models v(b) \\ C_{\mathcal{S}}(\langle C, v\rangle, l) = C_{\mathcal{S}}(\langle C, v\rangle, l''), & C \models \neg v(b) \end{cases} \tag{5}$$

  and

$$\begin{cases} v_{\mathcal{S}}(\langle C, v\rangle, l) = v_{\mathcal{S}}(\langle C, v\rangle, l'), & C \models v(b) \\ v_{\mathcal{S}}(\langle C, v\rangle, l) = v_{\mathcal{S}}(\langle C, v\rangle, l''), & C \models \neg v(b). \end{cases} \tag{6}$$

Thus,

$$\begin{aligned} v(wp(l,Q)) &\iff (v(b) \implies v(wp(l',Q))) \wedge (\neg v(b) \implies v(wp(l'',Q))) \\ &\iff \text{by the induction hypothesis, (5)} \\ &\iff (v(b) \implies v_{\mathcal{S}}(\langle C,v\rangle,l')(Q)) \wedge (\neg v(b) \implies v_{\mathcal{S}}(\langle C,v\rangle,l'')(Q)) \\ &\iff \text{by (6)} \\ &\iff (v(b) \implies v_{\mathcal{S}}(\langle C,v\rangle,l)(Q)) \wedge (\neg v(b) \implies v_{\mathcal{S}}(\langle C,v\rangle,l)(Q)) \\ &\iff v_{\mathcal{S}}(\langle C,v\rangle,l)(Q). \end{aligned}$$

- $l =$ **while** b **do** l': Since we only consider the case when $\mathcal{S}(\langle C,v\rangle,l)$ is defined we can use lemma 3. Furthermore, by (wp5'), $wp(l,Q) \iff \exists k \geq 0[G_k]$, where $G_0 \iff Q \wedge \neg b$ and $G_k \iff wp(l',G_{k-1})$ for $k > 0$. There exists a k such that G_k exactly when there exists a *least* such k, thus

$$wp(l,Q) \iff \exists k \geq 0[G_k \wedge \forall 0 \leq i < k[\neg G_i]] \tag{7}$$

We now prove the following by induction over j: for all j,

$$C^j_{\mathcal{S}}(\langle C,v\rangle,l') \models v(G_j) \iff v^j_{\mathcal{S}}(\langle C,v\rangle,l')(Q \wedge \neg b) \tag{8}$$

$j = 0$: $v(G_0) \iff v^0_{\mathcal{S}}(\langle C,v\rangle,l')(Q \wedge \neg b)$.

$j > 0$: assume as induction hypothesis for j that

$$C^j_{\mathcal{S}}(\langle C,v\rangle,l') \models v(G_j) \iff v^j_{\mathcal{S}}(\langle C,v\rangle,l')(Q \wedge \neg b)$$

for all $\langle C,v\rangle$. From a repeated application of lemma 4 follows that $C_{\mathcal{S}}(\langle C,v\rangle,l') \subseteq C^{j+1}_{\mathcal{S}}(\langle C,v\rangle,l')$. Thus it follows, from the induction hypothesis on l', that

$$C^{j+1}_{\mathcal{S}}(\langle C,v\rangle,l') \models v(G_{j+1}) \iff v_{\mathcal{S}}(\langle C,v\rangle,l')(G_j). \tag{9}$$

Directly from the induction hypothesis on j follows that

$$C^j_{\mathcal{S}}(\mathcal{S}(\langle C,v\rangle,l'),l') \models v_{\mathcal{S}}(\langle C,v\rangle,l')(G_j) \iff v^j_{\mathcal{S}}(\mathcal{S}(\langle C,v\rangle,l'),l')(Q \wedge \neg b),$$

or

$$C^{j+1}_{\mathcal{S}}(\langle C,v\rangle,l') \models v_{\mathcal{S}}(\langle C,v\rangle,l')(G_j) \iff v^{j+1}_{\mathcal{S}}(C,v,l')(Q \wedge \neg b). \tag{10}$$

(9), (10) finally gives

$$C^{j+1}_{\mathcal{S}}(\langle C,v\rangle,l') \models v(G_{j+1}) \iff v^{j+1}_{\mathcal{S}}(\langle C,v\rangle,l')(Q \wedge \neg b).$$

We now apply (8) to $v(wp(l,Q))$ according to (wp5') for $j = k$ and for $j = i$, $i < k$. If we furthermore observe that $C^i_{\mathcal{S}} \subseteq C^k_{\mathcal{S}}$ for all $i < k$, then we obtain, denoting $v^j_{\mathcal{S}}(\langle C,v\rangle,l')$ by $v^j_{\mathcal{S}}$,

$$\begin{aligned} v(wp(l,Q)) &\iff v(\exists k[G_k \wedge \forall i < k[\neg G_i]]) \\ &\iff \exists k[v(G_k) \wedge \forall i < k[\neg v(G_i)]] \\ &\iff \exists k[v^k_{\mathcal{S}}(Q \wedge \neg b) \wedge \forall i < k[\neg v^i_{\mathcal{S}}(Q \wedge \neg b)]] \\ &\iff \exists k[v^k_{\mathcal{S}}(Q) \wedge \neg v^k_{\mathcal{S}}(b) \wedge \forall i < k[(\neg v^i_{\mathcal{S}}(Q)) \vee v^i_{\mathcal{S}}(b)]]. \end{aligned} \tag{11}$$

But, according to lemma 3,

$$\exists k'[\forall i < k'[v^i_{\mathcal{S}}(b)] \wedge \neg v^{k'}_{\mathcal{S}}(b) \wedge \mathcal{S}(\langle C,v\rangle,l) = \mathcal{S}^{k'}(\langle C,v\rangle,l')]. \tag{12}$$

Both k and k' above are uniquely determined and it follows that $k = k'$. Thus, $v^k_{\mathcal{S}}(b)$ will be false, all $v^i_{\mathcal{S}}(b)$ will be true and from (11), (12) we obtain

$$v(wp(l,Q)) \iff v^k_{\mathcal{S}}(Q) \iff v^{k'}_{\mathcal{S}}(Q) \iff v_{\mathcal{S}}(\langle C,v\rangle,l)(Q).$$

■

At a first sight theorem 2 may seem a little surprising, since it essentially states that Q and $wp(l,Q)$ are equivalent. Note, though, that this equivalence holds only under the equalities in the causal substitution $C_{\mathcal{S}}(\langle C,v\rangle,l)$. In $v(wp(l,Q))$, the variables in $v(X)$ are substituted for the corresponding program variables and in Q the variables in $v_{\mathcal{S}}(\langle C,v\rangle,l)(X)$ are substituted for them. These variables are linked to each other through the equalities in $C_{\mathcal{S}}(\langle C,v\rangle,l)$.

7 Second order expressions

The language $L(X)$ treated so far lacks one important feature, namely *second order expressions* by which we mean expressions that evaluate not to values, but to variables. Examples are pointer expressions and subscripted array variables. In this section we will augment $L(X)$ and its semantics with second order expressions.

7.1 The general case

At this stage, all we will assume about the second order expressions is that they form a set $h(X)$ and that there is an *evaluation function* $\epsilon\colon (h(X)\times(X\to D))\to X$, that for every "machine state" (i.e. function $X\to D$) assigns a (first order) variable to every second order expression. Note that every state $\langle C,v\rangle$ defines a function $X\to D$ through $\psi_C\circ v$. First we define "mixed expressions", that are formed from both ordinary, first order expressions and second order expressions. We also extend ϵ to mixed expressions:

Definition 5 *$H(X)$ is defined by:*

- $E(X)\subseteq H(X)$.
- $h(X)\subseteq H(X)$.
- *If f is an operator, with arity $n(f)$, in the underlying algebra, and if $\mathbf{p}_1,\ldots,\mathbf{p}_{n(f)}$ belong to $H(X)$,then $f\bullet\langle\mathbf{p}_1,\ldots,\mathbf{p}_{n(f)}\rangle\in H(X)$.*

ϵ is extended to a function $(H(X)\times(X\to D))\to X$ in the following way ($F\in X\to D$): for all $\mathbf{p}\in H(X)$,

- *If $\mathbf{p}\notin h(X)$ and $\mathbf{p}\in X$, then $\epsilon(\mathbf{p},F)=\mathbf{p}$.*
- *If $\mathbf{p}\notin h(X)$ and $\mathbf{p}=f\bullet\langle\mathbf{p}_1,\ldots,\mathbf{p}_{n(f)}\rangle$, then $\epsilon(\mathbf{p},F)=f\bullet\langle\epsilon(\mathbf{p}_1,F),\ldots,\epsilon(\mathbf{p}_{n(f)},F)\rangle$.*
- *If $\mathbf{p}\in h(X)$, then $\epsilon(\mathbf{p},F)$ is given as before.*

A *second order substitution* is a partial function $h(X)\cup X\to H(X)$. In our augmented version of $L(X)$, we take the concurrent assignment statements to be second order substitutions. Furthermore, we now allow the logical expressions in **if**...**then**-statements and **while**-statements to be second order expressions. The resulting language is called $L_+(X)$.

Definition 6 *For any second order substitution σ, its* evaluated substitution *with respect to $F\in X\to D$ is $\{\,\epsilon(h,F)\leftarrow\epsilon(\sigma(h),F)\mid h\in dom(\sigma)\,\}$. It is denoted by $\epsilon(\sigma,F)$.*

In order to avoid problems with improperly defined substitutions, we assume that every $\epsilon(h,F)$ in definition 6 evaluates to a distinct variable for the substitutions under consideration. Note that this assumption always is fulfilled, if assignments of second order expressions are restricted from concurrent assignments to assignment of a single variable.

7.2 SA semantics for $L_+(X)$

In this subsection we will define SA semantics for $L_+(X)$. The difference to the SA semantics for $L(X)$ is that every expression, since it now may contain second order expressions, must be evaluated with respect to the current state instead of being used directly in the definition. Thus, (SA1) and (SA3) remain the same. In (SA4) and (SA5), $v(b)$ is replaced by $v(\epsilon(b, \psi_C \circ v))$. (With a slight abuse of notation, $\psi_C \circ v$ denotes the function $X \to D$ that it defines.) A new version of (SA2) is given below.

$$\textbf{(SA2}_+\textbf{)} \quad \begin{array}{l} C_{\mathcal{S}}(\langle C, v\rangle, \sigma) = C \cup \{\, s(v(x)) \leftarrow v \circ \epsilon(\sigma, \psi_C \circ v)(x) \mid x \in dom(\epsilon(\sigma, \psi_C \circ v)) \,\}, \\ v_{\mathcal{S}}(\langle C, v\rangle, \sigma) = \begin{cases} s(v(x)), & x \in dom(\epsilon(\sigma, \psi_C \circ v)) \\ v(x), & x \notin dom(\epsilon(\sigma, \psi_C \circ v)). \end{cases} \end{array}$$

7.3 Array element variables

An important kind of second order expressions are subscripted array elements. The usual approach is to treat arrays as the basic entities instead of the elements, so that for instance an assignment of an array element is seen as an operation on the array as a whole [6]. Here we will instead use the approach above. We will consider an array element $a(\mathbf{i})$ as a second order expression, formed by the array name a and the mixed expression $\mathbf{i}$. The variables in X that the evaluation function ϵ maps to are of the form $a(i)$, where $i \in I_a$, the *index set* of a. I_a is a subset of D. It can for instance be a finite, consecutive subset of the integers. ϵ is defined as follows:

Definition 7 *For all array names a, mixed expressions $\mathbf{i}$ of proper sort and functions $F: X \to D$, $\epsilon(a(\mathbf{i}), F) = a(\epsilon'(\mathbf{i}, F))$, where $\epsilon': (H(X) \times (X \to D)) \to D$, is given by:*

- *If $\mathbf{i} \notin h(X)$ and $\mathbf{i} \in X$, then $\epsilon'(\mathbf{i}, F) = F(\mathbf{i})$.*
- *If $\mathbf{i} \notin h(X)$ and $\mathbf{i} = f \bullet \langle \mathbf{p}_1, \ldots, \mathbf{p}_{n(f)} \rangle$, then $\epsilon'(\mathbf{i}, F) = f \bullet \langle \epsilon'(\mathbf{p}_1, F), \ldots, \epsilon'(\mathbf{p}_{n(f)}, F) \rangle$.*
- *If $\mathbf{i} = a(\mathbf{i}')$, then $\epsilon(\mathbf{i}, F) = F(a(\epsilon'(\mathbf{i}', F)))$.*

It is assumed, for a given expression $a(\mathbf{i})$, that $\epsilon'(\mathbf{i}, F)$ will always evaluate to a value in I_a. Whenever there can be no ambiguity, we will write $a(i)$ for $\epsilon(a(\mathbf{i}), F)$.

7.4 SA semantics vs. wp semantics for arrays

We will now show the counterpart to theorem 2 for programs with arrays. In order to be able to formulate the new theorem, we must extend the evaluation function ϵ to assertions about program variables. This is readily done. For simplicity, we will only consider assignments where a single variable is assigned.

The weakest precondition for assignments of arrays is usually defined with the aid of a function, that maps triples of arrays, index values and array element values to new arrays [10]. The value of the new array is defined as follows: $(a : i : e)(i) = e$, and $(a : i : e)(j) = a(j)$ whenever $i \neq j$. The weakest precondition for the assignment statement $a(\mathbf{i}) \leftarrow \mathbf{p}$ is then defined, using this function:

$$\textbf{(wp2}_+\textbf{)} \quad wp(a(\mathbf{i}) \leftarrow \mathbf{p}, Q) \iff \{a \leftarrow (a : \mathbf{i} : \mathbf{p})\}(Q).$$

Note that the substitution $\{a \leftarrow (a : \mathbf{i} : \mathbf{p})\}$ operates on array names, not on program variables.

Theorem 3 *For all states $\langle C, v\rangle$, all $l \in L_+(X)$ and all program variable predicates Q,*

$$C_{\mathcal{S}}(\langle C, v\rangle, l) \models v(\epsilon(wp(l, Q), \psi_C \circ v)) \iff v_{\mathcal{S}}(\langle C, v\rangle, l)(\epsilon(Q, \psi_C \circ v))$$

whenever $\mathcal{S}(\langle C, v\rangle, l)$ is defined.

Proof sketch. The only case that needs to be nontrivially reconsidered, is the assignment statement $a(\mathbf{i}) \leftarrow \mathbf{p}$ for array elements. Consider

$$v(\epsilon(wp(a(\mathbf{i}) \leftarrow \mathbf{p}, Q), \psi_C \circ v)) = v(\epsilon(\{a \leftarrow (a : \mathbf{i} : \mathbf{p})\}(Q), \psi_C \circ v)).$$

The interesting subterms of Q are of the form $a(\mathbf{i}')$. For every such subterm, the corresponding subterm of $v(\epsilon(\{a \leftarrow (a : \mathbf{i} : \mathbf{p})\}(Q), \psi_C \circ v))$ is

$$v(\epsilon(\{a \leftarrow (a : \mathbf{i} : \mathbf{p})\}(a(\mathbf{i}')), \psi_C \circ v)) = v(\epsilon((a : \mathbf{i} : \mathbf{p})(\mathbf{i}'), \psi_C \circ v)).$$

By the definition of $(a : \mathbf{i} : \mathbf{p})$, if $\epsilon'(\mathbf{i}, \psi_C \circ v) = \epsilon'(\mathbf{i}', \psi_C \circ v)$, then this term equals $v(\epsilon(\mathbf{p}, \psi_C \circ v))$, otherwise it equals $v(\epsilon(a(\mathbf{i}'), \psi_C \circ v)) = v(a(i'))$.

Consider now

$$v_{\mathcal{S}}(\langle C, v\rangle, a(\mathbf{i}) \leftarrow \mathbf{p})(\epsilon(Q, \psi_C \circ v)).$$

$dom(\epsilon(a(\mathbf{i}) \leftarrow \mathbf{p}, \psi_C \circ v)) = \{\epsilon(a(\mathbf{i}), \psi_C \circ v)\} = \{a(i)\}$, thus, for any subterm $a(\mathbf{i}')$ of Q, ($\mathrm{SA2}_+$) implies that if $a(i') = a(i)$ (which is the case exactly when $\epsilon'(\mathbf{i}', \psi_C \circ v) = \epsilon'(\mathbf{i}, \psi_C \circ v)$), then the corresponding subterm of $v_{\mathcal{S}}(\langle C, v\rangle, a(\mathbf{i}) \leftarrow \mathbf{p})(\epsilon(Q, \psi_C \circ v))$ equals $s(v(a(i)))$, otherwise it equals $v(a(i'))$. But by ($\mathrm{SA2}_+$), $C_{\mathcal{S}}(\langle C, v\rangle, a(\mathbf{i}) \leftarrow \mathbf{p})$ contains the equality

$$s(v(a(i))) \leftarrow v(\epsilon(\mathbf{p}, \psi_C \circ v)).$$

Thus, in all cases, the interesting subterms of $v(\epsilon(wp(a(\mathbf{i}) \leftarrow \mathbf{p}, Q), \psi_C \circ v))$ and $v_{\mathcal{S}}(\langle C, v\rangle, a(\mathbf{i}) \leftarrow \mathbf{p})(\epsilon(Q, \psi_C \circ v))$ will be equal under the equalities in $C_{\mathcal{S}}(\langle C, v\rangle, a(\mathbf{i}) \leftarrow \mathbf{p})$. It follows that the assertions themselves must be equivalent under those conditions.

A more formal proof could be carried out by induction over program assertions and their subexpressions. ■

8 Conclusions

We present a new semantics for a imperative languages. The semantics is for reasons of presentation defined only for a simple language, $L_+(X)$, but it can be extended to richer languages. It describes how an imperative program, for a given input, generates a set of single-assignments (a causal substitution) during its execution. Every execution of an assignment statement generates single-assignments, one for each assigned variable. The semantics thus demonstrates the relationship between imperative and single-assignment languages. We show that the semantics always is well-defined for all possible executions. We also show that it is consistent with conventional weakest precondition semantics.

Since the execution order of single-assignments is limited by the data dependencies only, SA semantics also indicates, in a simple way, how the execution of an imperative program can be parallelized. This is done in the same way as for a single-assignment program. Statement executions without data dependencies can be carried out in parallel. Present methods for finding parallelity in imperative programs consider dependencies between statements, rather than statement executions. Therefore, other dependence types have to be introduced. This makes the analysis unnecessarily cumbersome and restrictive. We believe that SA semantics can aid the development of new techniques for compiling imperative programs to parallel machines, especially in the light of the recent development of space-time mapping methods. A crucial point here is that we make the effort to define the semantics also for programs with arrays. A correct and efficient treatment of arrays is essential for the compilation of scientific code. We also think that SA semantics may be helpful in the further development of single-assignment languages, since it shows what the basic imperative control constructs translates into in a single-assignment context.

9 Acknowledgements

I would like to thank prof. David Gries for his kindness to supply me with references about the formal treatment of assignment of arrays. I would also like to thank the referees for the very initiated and detailed comments on the paper. Their suggestions have been most helpful. This work was in part supported by the Office of Naval Research, under contract N00014-86-K-0564, while the author was with the Department of Computer Science at Yale University.

References

[1] W. B. Ackerman. Data flow languages. *Computer*, 15:15–25, February 1982.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proc. POPL*, pages 1–11, ACM, January 1988.

[3] P. K. Cappello and K. Steiglitz. *Unifying VLSI Array Design with Linear Transformations of Space-Time.* Research Report TRCS83-03, Dept. Comput. Sci., UCSB, 1983.

[4] M. C. Chen. Transformation of parallel programs in Crystal. In H.-J. Kugler, editor, *INFORMATION PROCESSING 86*, pages 455–462, Elsevier Publishers B.V. (North-Holland), 1986.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proc. POPL*, pages 25–35, ACM, January 1989.

[6] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[7] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Proc. Symp. Applied Mathematics,* vol. 19: *Mathematical Aspects of Computer Science*, pages 19–32, American Mathematical Society, Providence, R.I., 1967.

[8] G. Grätzer. *Universal Algebra.* Springer–Verlag, New York, NY, 1979.

[9] I. Greif and A. R. Meyer. Specifying the semantics of while programs: a tutorial and critique of a paper by Hoare and Lauer. *ACM Trans. Program. Lang. Syst.*, 4(1):44–82, January 1982.

[10] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 583, October 1969.

[12] C. A. R. Hoare and P. E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Acta Inf.*, 3:135–153, 1974.

[13] P. Hudak, J-M. Delosme, and I. C. F. Ipsen. *ParLance: A Para-Functional Programming Environment for Parallel and Distributed Computing.* Research Report YALEU/DCS/RR-524, Dept. Comput. Sci., Yale University, March 1987.

[14] D. J. Kuck. A survey of parallel machine organization and programming. *Computing Surveys*, 9(1):29–59, March 1977.

[15] D. J. Kuck, Y. Muraoka, and S. C. Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Trans. Comput.*, C-21:1293–1310, December 1972.

[16] L. Lamport. The parallel execution of DO loops. *Comm. ACM*, 17:83–93, February 1974.

[17] J. Li, M. C. Chen, and M. F. Young. *Design of Systolic Algorithms for Large Scale Multiprocessors.* Technical Report YALEU/DCS/TR-513, Dept. Comput. Sci., Yale University, February 1987.

[18] B. Lisper. Synthesis and equivalence of concurrent systems. *Theoretical Computer Science*, 58:183–199, 1988.

[19] B. Lisper. *Synthesis of Synchronous Systems by Static Scheduling in Space-time.* Volume 362 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, May 1989.

[20] B. Lisper. *Time-Optimal Synthesis of Systolic Arrays with Pipelined Cells.* Research Report YALEU/DCS/RR-560, Dept. Comput. Sci., Yale University, September 1987.

[21] M. G. Main. A powerdomain primer. *Bullentin of the European Association for Theoretical Computer Science*, (33):115–147, October 1987.

[22] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974.

[23] J. R. McGraw. The VAL language: description and analysis. *ACM Trans. Program. Lang. Syst.*, 4(1):44–82, January 1982.

[24] W. L. Miranker and A. Winkler. Spacetime representations of computational structures. *Computing*, 32:93–114, 1984.

[25] D. I. Moldovan. On the analysis and synthesis of VLSI algorithms. *IEEE Trans. Comput.*, C-31:1121–1126, October 1982.

[26] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms in fixed size systolic arrays. *IEEE Trans. Comput.*, C-35:1–12, January 1986.

[27] D. A. Padua, D. J. Kuck, and D. H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Trans. Comput.*, C-29(9):763–776, September 1980.

[28] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Comm. ACM*, 29(12):1184–1201, December 1986.

[29] P. Quinton. *The Systematic Design of Systolic Arrays.* Research Report RR 216, INRIA, Rennes, July 1983.

[30] S. V. Rajopadye, S. Purushotaman, and R. Fujimoto. *On Synthesizing Systolic Arrays from Recurrence Relations with Linear Dependencies.* Detailed summary, Dept. Comput. Sci., University of Utah, 1986.

[31] R. A. Towle. *Control and data dependence for program transformations.* PhD thesis, Dept. Comput. Sci., University of Illinois at Urbana-Champaign, March 1976.