Dataflow Equations for Parallel Programs that Share Memory

Deliverable 2.11.1

The COMPARE Consortium

Release:	1.1
Date:	January 17, 1994
Status:	release
Confidentiality:	public
Reference:	GMD-1101-dfepp

Abstract

Traditional data flow analysis methods are designed for sequential programs. Hence they may fail when applied to control flow parallel imperative programs that share memory and are based on the MIMD computer model. Current approaches mostly use a copy-in/copy-out semantics, when entering/exiting a process, to model shared memory.

To avoid this restriction, this paper extends the notion of program execution paths. Selecting some specific paths out of the set of all possible paths, allows to give simple data flow equations which are proved to be equal to the meet over all path solution. Since these data flow equations are extensions of the sequential ones, they fit very well to the traditional optimization methods.

An example shows that the code generator of a compiler as well as a reordering assembler needs this kind of data flow analysis to avoid unnecessary memory barrier instructions and to produce correct instruction reorderings, respectively.

Another paper is currently under work (actually it's already present, but only in german) which extends this theory so that it can be used with the control flow graph representation of a source program.

Contents

1 Introduction

To exploit the power of todays' processors, optimizations like common subexpression elimination, constant folding, dead code elimination etc. must be performed for parallel programs as well as for sequential ones. Optimizing a program requires analyzing it, and this is often done by applying data flow equations to the program. Traditional data flow analysis methods are designed for sequential programs. Hence they may fail when applied to control flow parallel programs. [Midkiff $^{et\ al}$ 90] presents some examples, where traditional analyzing and optimizing techniques fail when applied to parallel programs.

Current approaches in analyzing the data flow of parallel programs have either a restricted model of shared memory, or even disallow it. [Reif 84] investigates the data flow of communicating processes, but these do not share memory. Processes communicate solely through synchronous channels. [Srinivasan $^{et\ al}$ 91a] describes an efficient method of computing the Static Single Assignment form [Cytron $^{et\ al}$ 89] for explicitly parallel programs with `wait` clauses. The parallel sections must be data independent, except where explicit synchronization is used. [Srinivasan $^{et\ al}$ 91b, Wolfe $^{et\ al}$ 91] introduce the *Parallel Control Flow Graph* and the *Parallel Precedence Graph* which may form the basis of concrete optimizing algorithms. [Chow $^{et\ al}$ 92] use abstract interpretation as framework to obtain program properties, like side effects, data dependencies, object lifetimes and concurrent expressions. [Grunwald $^{et\ al}$ 93] present a solution for the reaching definition problem, both with and without synchronization. But they restrict themselves to PCF FORTRAN (defined by the Parallel Computing Forum) standard conforming programs, which means access to shared variables is done only at synchronization points. For process start and process end they assume a copy in/copy out semantics.

Our investigation is based on an imperative language with explict control flow parallelism, dynamic process creation, and shared memory. As computing model we assume a MIMD (multiple instruction, multiple data) system, where each process is executed on a separate processor. Each processor runs independently of each other and has its own set of registers, which are invisible for other processors. All processors access a shared memory, without the above mentioned restrictions. The main result of this paper is an extension of the well known sequential data flow equations covering *forward/backward* and *may/must* data flow problems:

1. Theorem 1 shows which data flow information reaches a statement (in the parallel context), and
2. theorems 2 and 3 give the information which reaches the end of the in parallel executed statements.

Section 2 defines the language we base our investigation on, section 3 presents the idea which leads to the main result, and sections 4 and 5 shows the theorems.

1.1 The classical data flow problems

The four “classical” data flow problems are classified into *may* and *must* problems and the direction of information propagation (*forward / backward*) (cf. table 1). If the information which reaches a program point comes from the preceding statements, the problem is called a *forward problem*; if it comes from the following statements, it is called a *backward problem*. If the information has to be available in all predecessors (successors) the problem is called a *must problem*, if it has to be available in at least one, it is a *may* problem [Hecht 77].

Usually, the data flow information is computed over a control flow graph. Another possibility is to use the structure tree of the program [Babich $^{et\ al}$ 78, Aho $^{et\ al}$ 86]. We chose this approach, since it allows easier formulation.

In this paper we consider only forward problems, for backward problems the results may be stated in a similar way.

	Must	May
forward	Available Expressions	Reaching Definitions
backward	Very Busy Expressions	Live Variables

Table 1: Classification of the "classical" data flow problems.

The equations are always stated using the sets $gen[S], kill[S], in[S], out[S]$. *gen* is the set of the informations generated by statement S and reaching its end. *kill* is the set of informations invalidated by S and still invalid at the end of S. *in* represents the informations reaching S, and *out* the set of informations reaching the end of S. *gen* and *kill* are defined in terms of the underlying data flow problem, e.g. the set of generated definitions of a variable or set of computed expressions. For example in the reaching definition problem, the assign statement `id := expr` generates this definition of variable `id` and invalidates all other definitions of this variable.

The most important equations are the ones for propagating the information from one statement to the next in sequential execution: $\mathtt{S}_1\mathtt{;}\ \mathtt{S}_2$. For these there is no distinction between *must* and *may* problems and the information is propagated as shown in table 2. The equations for the other sequential statements are given in appendix A.

$$\begin{aligned} gen[S] &= gen[S_2] \cup gen[S_1] - kill[S_2]^1 & (1)\\ kill[S] &= kill[S_2] \cup kill[S_1] - gen[S_2] & (2)\\ in[S_1] &= in[S] \\ in[S_2] &= out[S_1] \\ out[S] &= out[S_2] \end{aligned}$$

Table 2: Data flow equation for $\mathtt{S ::= S}_1\mathtt{;\ S}_2$

The following equation holds for the sequential statements [Aho $^{et\ al}$ 86]:

$$out[S] = gen[S] \cup in[S] - kill[S] \tag{3}$$

1.2 What could go wrong?

This section shows the potential problems, when applying sequential data flow analysis to an explicit parallel program. The small program[2] executes the processes P_1 and P_2 in parallel. It is intuitively clear that $critical_1$ and $critical_2$ are never executed at the same time.

A simple-minded optimizer could perform the following "optimizations" (which would be correct in sequential contexts):

- Propagate `a = 0` and `b = 0` to `IF a = 0` and `IF b = 0` respectively.
- Then the expressions could be statically evaluated to `TRUE`.
- Dead code elimination removes the `IF` and `ELSE` parts.

⇒ Both, $critical_1$ and $critical_2$ are executed!

But even without traditional optimizations performed by the compiler, things could go wrong when using an assembler which does instruction scheduling (reordering), to better use the processor's internal parallelism (i.e. the pipelined processing of instructions).

[1]If a and b are sets, then $a - b$ is the set difference and is defined as $a - b := a \cap \overline{b}$.

[2][Lamport 79] (shown in table 3) presents this problem concerning the design of parallel computers.

[2][Lamport 79] presents this problem concerning the design of parallel computers.

```
            a := 0; b := 0;
                 PAR
       (P_1)          |       (P_2)
a := 1;               | b := 1;
IF b = 0              | IF a = 0
THEN critical_1;      | THEN critical_2;
       a := 0;        |        b := 0;
ELSE else_1           | ELSE else_2
END                   | END
                 END
```

Table 3: Simple parallel program.

The non-optimized code of process body P_1 on a typical RISC processor is given in table 4. The instruction scheduler could now decide to reorder the instructions, e.g. to insert another instruction between a register load and an immediately following register use instruction (e.g. `ldc 1,r0; st r0,a`) which results in the code for P_1 shown in table 5a. In this case, it can happen that *critical*$_1$ and *critical*$_2$ are executed both, as shown in table 5b!

ldc	1, r0	Load constant `1` into register `r0`.
st	r0, a	Store the content of register `r0` in memory at address `a`.
ld	b, r1	Load content of memory at address `b` into a register.
cmp	r1, 0	Compare a register with a constant, set condition code.
jeq	then$_1$	Conditional branch to **then**$_1$, if condition code *equal* set.
code of else$_1$		
...		

Table 4: Non-optimized code for Process P_1

ldc	1, r0
ld	b, r1
st	r0, a
cmp	r1, 0
jeq	then$_1$
...	

5a

time	t$_1$	t$_2$	t$_3$	t$_4$	...	
Processor$_1$:	ldc 1, r0;	ld b, r1; *(r1 = 0)*	st r0, a;	cmp r1, 0;	...	*critical*$_1$
Processor$_2$:	ldc 1, r0;	ld a, r1; *(r1 = 0)*	st r0, b;	cmp r1, 0;	...	*critical*$_2$

5b

Table 5: Code and execution of reordered code

Even worse, some processors (like the *Dec Alpha Chip* [DEC 92]) are able to reorder the memory accesses to different addresses to some degree. Hence, even the unchanged code could give the wrong result. To avoid this situation, the *Dec Alpha Chip* offers a *memory barrier* instruction, which delays the processor until all memory requests are fulfilled. In our example this instructions must follow every memory access, which results in a great slow-down of the program speed.

On a system with distributed memory, the shared memory access may implemented by calls to the operating system, which transports a value from the memory it is stored in to the destination where it is needed. If these calls are asynchronously performed (e.g. the memory fetch is separated into two calls: a non blocking *ask_for_value(address)* and a blocking *wait_for_value(address))*, the same problem arises.

[Lamport 79] offers a solution which is formalized by [Afek $^{et\ al}$ 93]. Memory access have to fulfill the two conditions:

1. Each processor executes the memory access in the order specified by the program.
2. All access to a single memory cell are executed in a first-in-first-out queue.

It is obvious, that these conditions are too restrictive, since optimizations of "really" independent memory accesses are forbidden.

The base of these transformations is information like the *reaching definitions* or *available expressions*. The reason for the above shown problems is that they use the wrong information, i.e. the information was calculated in a "sequential context", not considering the parallelism expressed in the program.

2 The sample language

A simple imperative language will be used in this paper, having loops, conditional statements, and a statement to execute other statements in parallel. (explict control flow parallelism). Replicators allow dynamic process creation, and processes share memory.

```
Prog          ::=  Stmt
Stmt          ::=  Identifier ":=" Expr | Stmt//";" |
                   IF Expr THEN Stmt ELSE Stmt END|
                   REPEAT Stmt UNTIL Expr | PAR ProcessBody//"|" END.
ProcessBody   ::=  [Replicator] Stmt//";" .
Replicator    ::=  "[" Identifier ":" LowerBound TO UpperBound "]" .
LowerBound    ::=  Expr .
UpperBound    ::=  Expr .
Expr          ::=  usual expressions.
```

`Prog` is the root symbol of the grammar. `Stmt//";"` is a list of statements separated by a semicolon. `[Replicator]` stands for an optional `Replicator` part.

The `PAR` statement executes all processes specified by `ProcessBody` in parallel and independently. The processes executing this `PAR` statement is suspended until all child processes have terminated. A `ProcessBody` is a list of statements which may be replicated. That is: `max(UpperBound - LowerBound + 1, 0)` processes are forked which all execute the statements following the replicator. Each replicated process gets its private copy of the replicator variable `Identifier`, which has in each replicated process a unique value in the range `[LowerBound .. UpperBound]`. Replicated processes are also called *forall loops* in other languages. Each variable can be accessed in each process. No automatic synchronization is done for the access.

Procedures are not contained in the language, since the analysis and optimizing problems can be solved in the usual way.

3 The idea

A single run of a program may be seen as an execution of sequence of assign statements, $\langle s_1; \ldots; s_n\rangle$, starting with s_1 and ending with s_n. The s_i in the sequence are selected by some "magic", i.e. by the conditionals, loops, and par's[3]. If the program terminates, the sequence is finite. To compute the data flow information, reaching a statement in this sequence is straight forward: Use the equations for `S ::= S1; S2` of table 2.

But now the meaning of data flow information is to state facts about any program run, not a specific one. Hence all possible sequences, or *paths*, must be considered. The following formulas show how all program paths are computed[4]. If *path* is defined as the set of all assign statement sequences, $path := \{\langle s_1; \ldots; s_n\rangle | s_i \text{is an assignment}\}$, then $p^n \in path$ is defined as the n-fold concatenation of path p, and $p^+ = \bigcup_{i=1}^{n} p^i$, with some n. The set $paths[S]$ is now defined as the set of all paths, produced by statement S. For example, $paths[Prog]$ is the set of all possible paths of a program.

$$paths[S] = \begin{cases} concat(paths[S'], paths[S'']) & : S ::= S'; S'' \\ \{\langle S\rangle\} & : S \equiv \text{assign statement} \\ paths[S'] \cup paths[S''] & : S ::= \texttt{IF E THEN S}' \texttt{ ELSE S}'' \texttt{ END} \\ paths^+[S'] & : S ::= \texttt{REPEAT S}' \texttt{ UNTIL E} \\ merge(paths[S'], paths[S'']) & : S ::= \texttt{PAR S}' \texttt{ | S}'' \texttt{ END} \end{cases}$$

Where *merge* returns all paths which may be generated by a `PAR` statement:
$merge(\langle s'_1; \ldots; s'_{n'}\rangle, \langle s''_1; \ldots; s''_{n''}\rangle) :=$

$$\left\{ \langle s_1; \ldots; s_{n'+n''}\rangle \;\middle|\; \begin{array}{l} \forall i < j \in \{1, \ldots, n'\} \exists k, l \in \{1, \ldots, n' + n''\} : s_k = s'_i, s_l = s'_j \Rightarrow k < l \;\; or \\ \forall i < j \in \{1, \ldots, n''\} \exists k, l \in \{1, \ldots, n' + n''\} : s_k = s''_i, s_l = s''_j \Rightarrow k < l \end{array} \right\}$$

merge mixes the paths of the branches of a `PAR` statement such that the order of the statements in one branch is obeyed in the merged path, but between two statements $s'_i; s'_{i+1}$ in a path of one branch the merged path may contain statements of the other branch: e.g. $s'_i; s''_k; s'_{i+1}$. *merge* may be extended to take as arguments set of paths: $merge : 2^{path} \times 2^{path} \rightarrow 2^{path} : merge(P', P'') := \bigcup_{p' \in P', p'' \in P''} merge(p', p'')$

It is clear, several runs of a sequential program execute with the same input always the same path. But for a parallel program there are several paths possible.

Depending on the *may/must* property of the data flow problem, the data flow information is the union (may) or intersection (must) of the information computed for all paths, reaching a statement S:

$$\boxed{\begin{array}{lcll} info[S] & = & \bigwedge\limits_{p \in prefix(paths[Prog], S)} info[p] & (4) \\ info[S] & = & \bigwedge\limits_{p \in paths[S]} info[p] & (5) \end{array}}$$

$\bigwedge$ and $\wedge$ stands, depending on the problem, for either set union or set intersection. $prefix(paths[Prog], S)$ is the set of paths, reaching statement S. Equation (4) is used when the information depends on the preceding statements (like *in*), (5) is used if this not the case, like for *gen* and *kill*. For *out* (5) can be used too, since it depends on *in*, for which the other equation is used. These equations correspond to the so called *meet over all paths* solution of the data flow problem [Kildall 73].

Since for a single path it is known how to compute the data flow information, the algorithm to get the data flow information reaching a statement S is now clear. However, it has a big

[3]This is possible due to the interleaving semantics of the language.

[4]For the sake of simplicity, they are given only for `PAR` statements with two branches and no replicators. The generalization is straight forward.

drawback: the number of paths may be exponential in the number of conditionals or even infinite if the program contains loops.

A better solution would be: find a simple formula $F_{info}[S]$ computing the data flow information *info[S]* based only on the structure tree of the program, and proof that this formula returns the same as this *meet over all path* solution.

The next sections shows the main result of this paper:

1. it is shown which information reaches a statement (in the parallel context), and
2. which information reaches the end of a `PAR` statement.

For the first result, two sets $(in, \widetilde{in})$ are used to represent the data flow information reaching a statement, instead of only one in the sequential case.

The second result is based on the fact that it is sufficient to consider some specific paths instead of all possible paths. For these specific paths, a simple formula $F_{info}[S]$ can be given and it returns the same results as the belonging meet over all path solution.

For the `PAR` S_1 `|` S_2 `END` statement with the two branches S_1 and S_2, these specific paths are: *concat*$(paths[S_1], paths[S_2])$ and *concat*$(paths[S_2], paths[S_1])$, i.e. the paths resulting from the concatenated execution of $S_1; S_2$ and $S_2; S_1$.

For example: Analysing the `PAR` s_1`;` s_2 `|` s_3 `END` statement, for which the s_i are simple statements, only the two paths $\langle s_1; s_2; s_3\rangle$ and $\langle s_3; s_1; s_2\rangle$ are needed for the analysis. The third possible path $\langle s_1; s_3; s_2\rangle$ need not to be considered.

For `PAR` statements with n branches the paths from the $n!$ concatenated statements $S_{i_1}; \ldots; S_{i_n}$, where $\vec{\imath} = (i_1, \ldots, i_n) \in perm(1, n)$ is a permutation of the numbers $1, \ldots, n$, are used.

If the sample language would have a process synchronisation statement, the number of paths could be reduced, since some path never occur in any program run. Hence not considering this kind of statement, produces a "worst case" data flow information which is still correct.

Including a `goto` statements doesn't change the ideas, it complicates only the description. This is due to the fact that we are using the structure tree and not a control flow graph for representing a program in our presentation of the ideas.

4 Which information reaches a statement (in the parallel context)

In the sequential case only one set is defined to represent the information reaching a statement. In the parallel case two sets are needed:

in[S] The information propagated along the edges of the structure tree, i.e. without considering the effect of the statements executed parallel to S. That is the information which comes from the statements executed always before S.

$\widetilde{in}$**[S]** The set of informations reaching S, which stems from sequentially before S executed statements and from statements executed in parallel to S.

It is obvious, that $\widetilde{in}[S] = \bigwedge_{p \in prefix(paths[Prog], S)} out[p]$. The same distinction can be made for the set of information reaching the end of a statement, but $\widetilde{out}[S]$ is not of interest, hence only $out[S]$ is used.

The set $sibl[S]$ (sibling) contains all assign statements which may be executed in parallel to S.

Theorem 1 (Information reaching a statement) *For an arbitrary statement S holds*[5]*:*

[5] For the sake of simplicity,the following precedence rules are used: $a \wedge b - c = a \wedge (b - c)$ and $\bigwedge_i a_i - \bigwedge_i b_i = (\bigwedge_i a_i) - (\bigwedge_i b_i)$.

(a) if the problem is a may problem:

$$\bigcup_{p \in prefix(paths[Prog],S)} out[p] = in[S] \cup \bigcup_{s \in sibl[S]} gen[s]$$

(b) if the problem is a must problem:

$$\bigcap_{p \in prefix(paths[Prog],S)} out[p] = in[S] - \bigcup_{s \in sibl[S]} kill[s]$$

Proof (Theorem 1a) The proof is done in two steps "$\subseteq$" and "$\supseteq$":

"$\subseteq$" It has to be shown: $\bigcup_{p \in prefix(paths[Prog],S)} out[p] \subseteq in[S] \cup \bigcup_{s \in sibl[S]} gen[s]$

Let $x \in \bigcup_{p \in prefix(paths[Prog],S)} out[p]$, then there is in a path $p \in prefix(paths[Prog], S)$ a statement s_p which generates x and x is not invalidated later in p. The assignment statements in p stem either from program statements, which are executed sequentially before S, or from statements which may be executed in parallel to S[6] (see the definition of $paths[S]$). If s_p is a statement which is always executed before S then $x \in in[S]$ holds. If s_p a statement which may be executed in parallel to S, then $x \in \bigcup_{s \in sibl[S]} gen[s]$.

"$\supseteq$" It has to be shown: $in[S] \cup \bigcup_{s \in sibl[S]} gen[s] \subseteq \bigcup_{p \in prefix(paths[Prog],S)} out[p]$

Let $x \in in[S]$ and $x \notin \bigcup_{s \in sibl[S]} gen[s]$, then the claim holds obviously.

Let $x \in \bigcup_{s \in sibl[S]} gen[s]$ be generated by a statement $s \in sibl[S]$. Since there is for every $s \in sibl[S]$ a path p_s which executes s as its last statement before executing S, and $p_s \in prefix(paths[Prog], S)$, the claim holds.

■

Proof (Theorem 1b) Lemma: Let $x \in \bigcap_{p \in prefix(paths[Prog],S)} out[p]$, then $x \in in[S]$ holds. That is, if x reaches S, then x is generated by a statement which is executed always before S. Otherwise there would be a statement $s \in sibl[S]$ which generates x and since $x \in \bigcap_{p \in prefix(paths[Prog],S)} out[p]$, s must be executed in all paths before S, which contradicts $s \in sibl[S]$.

"$\subseteq$" It has to be shown: $\bigcap_{p \in prefix(paths[Prog],S)} out[p] \subseteq in[S] - \bigcup_{s \in sibl[S]} kill[s]$

Let $x \in \bigcap_{p \in prefix(paths[Prog],S)} out[p]$, hence $x \in in[S]$. Now it has to be shown that $x \notin \bigcup_{s \in sibl[S]} kill[s]$: Assume: there is a statement $s \in sibl[S]$ which kills x. Then we can construct a path p_s, where s is the last statement before S, and hence x is not part of $out[p_s]$. Since $p_s \in prefix(paths[Prog], S)$, x is not part of $\bigcap_{p \in prefix(paths[Prog],S)} out[p]$ which contradicts our initial assumption. Hence if $x \in \bigcap_{p \in prefix(paths[Prog],S)} out[p]$ it cannot be killed by a statement executed in parallel to S.

"$\supseteq$" It has to be shown: $in[S] - \bigcup_{s \in sibl[S]} kill[s] \subseteq \bigcap_{p \in prefix(paths[Prog],S)} out[p]$

Because $x \in in[S] - \bigcup_{s \in sibl[S]} kill[s]$ on all paths[7] $p' := p/\{$statements executed sequentially before $S\}$ with $p \in prefix(paths[Prog], S)$, x is generated by a statement $s_{p'}$ and and

[6] When speaking in the context of paths from statements which are executed in parallel to others, we mean that these statements are contained in a PAR statement in different branches, or if the branch is replicated, the same branch too.

[7] For a path p and a set of statements S, p/S is defined to be the restricted path containing only statements of the set S.

$x \in out[p']$. Since $x \notin \bigcup_{s \in sibl[S]} kill[s]$, x is not killed by statements which may be executed in parallel to S. Hence x is generated on path p by $s_{p'}$ and it is not killed in p after executing $s_{p'}$, since p contains compared to p' only additional statements which may be executed in parallel to S. Hence $x \in \bigcap_{p \in prefix(paths[Prog], S)} out[p]$.

■

5 Which information reaches the end of the PAR statement

Theorem 2 *Data flow equations for the* `PAR` *statement with n branches*
For the `S ::= PAR` S_1 `|` ... `|` S_n `END` *statement, and no process body is replicated, it holds:*

if out *is a may-problem*

$$\begin{aligned} gen[S] &= \bigcup_{j=1}^{n} gen[S_j] \\ kill[S] &= \bigcup_{j=1}^{n} kill[S_j] - \bigcup_{k=1}^{n} gen[S_k] \\ out[S] &= gen[S] \cup \left(in[S] - \bigcup_{j=1}^{n} kill[S_j] \right) \end{aligned}$$

if out *is a must-problem*

$$\begin{aligned} gen[S] &= \bigcup_{j=1}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k] \\ kill[S] &= \bigcup_{j=1}^{n} kill[S_j] \\ out[S] &= gen[S] \cup \left(in[S] - \bigcup_{j=1}^{n} kill[S_j] \right) \end{aligned}$$

All proofs follow the same road: Compute the data flow information for the "special paths", resulting from the $n!$ statement sequences $S_{i_1}; \ldots ; S_{i_n}$: $\bigwedge_{\vec{\imath} \in perm(1,n)} info[S_{i_1}; \ldots ; S_{i_n}]$ and show that this equal to the meet over all paths solution $\bigwedge_{p \in paths[S]} info[p]$, which is defined to be $info[S]$.

First we give the data flow equations for the "special paths":

$$\bigcup_{\vec{\imath} \in perm(1,n)} gen[S_{i_1}; \ldots ; S_{i_n}] = \bigcup_{j=1}^{n} gen[S_j] \tag{6}$$

$$\bigcap_{\vec{\imath} \in perm(1,n)} gen[S_{i_1}; \ldots ; S_{i_n}] = \bigcup_{j=1}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k] \tag{7}$$

$$\bigcup_{\vec{\imath} \in perm(1,n)} kill[S_{i_1}; \ldots ; S_{i_n}] = \bigcup_{j=1}^{n} kill[S_j] \tag{8}$$

$$\bigcap_{\vec{\imath}\in perm(1,n)} kill[S_{i_1};\ldots;S_{i_n}] = \bigcup_{j=1}^{n} kill[S_j] - \bigcup_{k=1}^{n} gen[S_k] \tag{9}$$

$$\bigcup_{\vec{\imath}\in perm(1,n)} out[S_{i_1};\ldots;S_{i_n}] = \bigcup_{j=1}^{n} gen[S_j] \cup \left(in[S] - \bigcup_{k=1}^{n} kill[S_k]\right) \tag{10}$$

$$\bigcap_{\vec{\imath}\in perm(1,n)} out[S_{i_1};\ldots;S_{i_n}] = \left(\bigcup_{j=1}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k]\right) \cup \left(in[S] - \bigcup_{l=1}^{n} kill[S_l]\right) \tag{11}$$

Proof (Equations 6, 8) Since:
$gen[S_1;\ldots;S_n] \overset{(1)}{=} gen[S_n]\cup(gen[S_{n-1}]\cup(gen[S_{n-2}]\cup-\ldots)-kill[S_{n-1}])-kill[S_n] \subseteq \bigcup_{j=1}^{n} gen[S_j]$
and
$gen[S_1;\ldots;S_n] \overset{(1)}{=} gen[S_n]\cup(gen[S_{n-1}]\cup(gen[S_{n-2}]\cup-\ldots)-kill[S_{n-1}])-kill[S_n] \supseteq gen[S_n]$,
then
$\bigcup_{\vec{\imath}\in perm(1,n)}(gen[S_{i_n}]) \subseteq \bigcup_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}] \subseteq \bigcup_{\vec{\imath}\in perm(1,n)}(\bigcup_{j=1}^{n} gen[S_j])$ and hence
$\bigcup_{k=1}^{n} gen[S_k] \subseteq \bigcup_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}] \subseteq \bigcup_{j=1}^{n} gen[S_j]$ which is the desired result.
The same is done for $kill[S]$. ∎

Proof (Equations 7, 9) The proof is given for *gen*, equation (9) is proved similarly. The proof shows the two set inclusions: "$\subseteq$" and "$\supseteq$"

"$\subseteq$" To be shown: $\bigcap_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}] \subseteq \bigcup_{j=1}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k]$.

$\bigcap_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}] \subseteq \bigcup_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}] = \bigcup_{j=1}^{n} gen[S_j]$.

Let $x \in \bigcap_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}]$ then x can not be killed by any statement $S_k, 1 \le k \le n$. Otherwise, assume x will be killed by a statement S_k, then there is a permutation $\vec{\imath}$, where $i_n = k$, i.e. S_k is the last statement. Hence $x \notin gen[S_{i_1};\ldots;S_{i_n}]$, which contradicts the assumption, that x is generated by all permuted statement sequences.

Hence $x \in \bigcup_{j=1}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k]$.

"$\supseteq$" To be shown: $\bigcup_{j=1}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k] \subseteq \bigcap_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}]$.

Let $x \in \bigcup_{j=1}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k]$ then x is not invalidated by any statement $S_k, 1 \le k \le n$, but generated by some $S_j, 1 \le j \le n$. Hence $x \in \bigcap_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}]$.

∎

Proof (Equation 10)

$$\begin{aligned}
&\bigcup_{\vec{\imath}\in perm(1,n)} out[S_{i_1};\ldots;S_{i_n}]\\
&\quad \overset{(3)}{=} \bigcup_{\vec{\imath}\in perm(1,n)}(gen[S_{i_1};\ldots;S_{i_n}] \cup in[S_{i_1};\ldots;S_{i_n}] - kill[S_{i_1};\ldots;S_{i_n}])\\
&\quad \text{since } \forall\vec{\imath}\in perm(1,n) : in[S_{i_1};\ldots;S_{i_n}] = in[S] \text{ and } (6),(20),(9),(26)\\
&\quad = \bigcup_{j=1}^{n} gen[S_j] \cup (in[S] - \bigcup_{k=1}^{n} kill[S_k]) \cup (in[S] \cap \bigcup_{l=1}^{n} gen[S_l])\\
&\quad = \bigcup_{j=1}^{n} gen[S_j] \cup (in[S] - \bigcup_{k=1}^{n} kill[S_k])
\end{aligned}$$

∎

Proof (Equation 11) The proof shows the two set inclusions: "$\subseteq$" and "$\supseteq$"

"$\subseteq$" To be shown:
$\bigcap_{\vec{\imath}\in perm(1,n)} out[S_{i_1};\ldots;S_{i_n}] \subseteq (\bigcup_{j=1}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k]) \cup (in[S] - \bigcup_{l=1}^{n} kill[S_l])$.

$$\begin{aligned}
\bigcap_{\vec{\imath}\in perm(1,n)} out[S_{i_1};\ldots;S_{i_n}] &\overset{(3)}{=} \\
&\bigcap_{\vec{\imath}\in perm(1,n)}(gen[S_{i_1};\ldots S_{i_n}] \cup in[S_{i_1};\ldots;S_{i_n}] - kill[S_{i_1};\ldots;S_{i_n}]) \\
\subseteq\ & (\bigcap_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}])\cup \\
& (\bigcap_{\vec{\imath}\in perm(1,n)}(in[S_{i_1};\ldots;S_{i_n}] - kill[S_{i_1};\ldots;S_{i_n}])) \\
& \text{since } \forall\vec{\imath}\in perm(1,n) : in[S_{i_1};\ldots;S_{i_n}] = in[S] \text{ and } (19),(7),(8) \\
=\ & (\bigcup_{j=1}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k]) \cup (in[S] - \bigcup_{l=1}^{n} kill[S_l])
\end{aligned}$$

"$\supseteq$" To be shown:
$(\bigcup_{j=1}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k]) \cup (in[S] - \bigcup_{l=1}^{n} kill[S_l]) \subseteq \bigcap_{\vec{\imath}\in perm(1,n)} out[S_{i_1};\ldots;S_{i_n}]$.
Let $x \in (\bigcup_{j=1}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k]) \cup (in[S] - \bigcup_{l=1}^{n} kill[S_l])$ then

1. $x \in \bigcup_{j=1}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k]$ or
2. $x \in in[S] - \bigcup_{l=1}^{n} kill[S_l]$.

In both cases x is never killed by a statement $S_i, 1 \le i \le n$. In the first case x is generated by some statement S_j and since not killed, it is contained in $out[S_{i_1};\ldots;S_{i_n}]$. In the second case it it reaches the start of the statement sequence $S_{i_1};\ldots;S_{i_n}$ and since not killed by it, $x \in out[S_{i_1};\ldots;S_{i_n}]$.

■

Now the correctness of the formula is proved, i.e. that they return the same result as the meet over all paths solutions.

$$\bigcup_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}] = \bigcup_{p\in paths[S]} gen[p] \quad \text{if } gen \text{ is a may-problem} \tag{12}$$
$$\bigcap_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}] = \bigcap_{p\in paths[S]} gen[p] \quad \text{if } gen \text{ is a must-problem} \tag{13}$$
$$\bigcup_{\vec{\imath}\in perm(1,n)} kill[S_{i_1};\ldots;S_{i_n}] = \bigcup_{p\in paths[S]} kill[p] \quad \text{if } kill \text{ is a may-problem} \tag{14}$$
$$\bigcap_{\vec{\imath}\in perm(1,n)} kill[S_{i_1};\ldots;S_{i_n}] = \bigcap_{p\in paths[S]} kill[p] \quad \text{if } kill \text{ is a must-problem} \tag{15}$$
$$\bigcup_{\vec{\imath}\in perm(1,n)} out[S_{i_1};\ldots;S_{i_n}] = \bigcup_{p\in paths[S]} out[p] \quad \text{if } out \text{ is a may-problem} \tag{16}$$
$$\bigcap_{\vec{\imath}\in perm(1,n)} out[S_{i_1};\ldots;S_{i_n}] = \bigcap_{p\in paths[S]} out[p] \quad \text{if } out \text{ is a must-problem} \tag{17}$$

Proof (Equations 12, 14) The proof is given for *gen*, equation (14) is proved similarly. The proof shows the two set inclusions: "$\subseteq$" and "$\supseteq$"

"$\subseteq$" To be shown: $\bigcup_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}] \subseteq \bigcup_{p\in paths[S]} gen[p]$.

$\bigcup_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}] = \bigcup_{\vec{\imath}\in perm(1,n)} \left(\bigcup_{p\in path[S_{i_1};\ldots;S_{i_n}]} gen[p]\right)$

Since a path $p \in path[S_{i_1};\ldots;S_{i_n}]$ is a path of the set $paths[S]$, the inclusion holds.

"$\supseteq$" To be shown: $\bigcup_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}] \overset{(6)}{=} \bigcup_{j=1}^{n} gen[S_j] \supseteq \bigcup_{p\in paths[S]} gen[p]$.

Let $x \in \bigcup_{p\in paths[S]} gen[p]$ then there is a path $p \in paths[S]$ with $x \in gen[p]$. In p there is a statement s_p with $x \in gen[s_p]$ and s_p cannot be followed by a statement in p, which kills x. Now p has the form $p = \langle X; s_p; Y\rangle$, then in the subpath Y there is no statement which invalidates x. Let s_p be a statement from $S_j, 1 \leq j \leq n$: then it must be shown that $x \in gen[S_j]$. If p is restricted to the statements of S_j, then Y/S_j contains only statements of S_j. Since Y contains no statements which kill x, this is true for Y/S_j. Hence $p/S_j \in paths[S_j]$ implies $x \in gen[S_j]$.

■

Proof (Equations 15, 13) The proof is given for *kill*, equation (13) is proved similarly. The proof shows the two set inclusions: "$\subseteq$" and "$\supseteq$"

"$\subseteq$" To be shown: $\bigcap_{\vec{\imath}\in perm(1,n)} kill[S_{i_1};\ldots;S_{i_n}] \subseteq \bigcap_{p\in paths[S]} kill[p]$.

Let $x \in \bigcap_{\vec{\imath}\in perm(1,n)} kill[S_{i_1};\ldots;S_{i_n}] \overset{(9)}{=} \bigcup_{j=1}^{n} kill[S_j] - \bigcup_{k=1}^{n} gen[S_k]$ then there is $j, 1 \leq j \leq n$ with $x \in kill[S_j] - \bigcup_{k=1}^{n} gen[S_k]$, i.e. x is killed by S_j, and x is not generated by any other branch of the `PAR` statement.

Let p an arbitrary path of the set $paths[S]$. Since $p_j = p/S_j$ is a path from the set $paths[S_j]$, $x \in kill[p_j]$ holds, and s_{p_j} is the statement killing x on p_j, and after that statement x is not generated on p_j. p has now the form $\langle X; s_{p_j}; Y\rangle$. In Y there are no statements from S_j, which generate x (otherwise $x \notin kill[S_j]$) and p cannot contain statements from $S_k, 1 \leq k \leq n, j \neq k$ which generate x (otherwise $x \notin \bigcup_{k=1,j\neq k}^{n} gen[S_j]$), and hence there is no statement in Y generating x. Now we have for any path p: $x \in kill[p]$ and hence $x \in \bigcap_{p\in paths[S]} kill[p]$.

"$\supseteq$" To be shown: $\bigcap_{\vec{\imath}\in perm(1,n)} kill[S_{i_1};\ldots;S_{i_n}] \supseteq \bigcap_{p\in paths[S]} kill[p]$.

$\bigcap_{\vec{\imath}\in perm(1,n)} kill[S_{i_1};\ldots;S_{i_n}] = \bigcap_{\vec{\imath}\in perm(1,n)} \left(\bigcap_{p\in path[S_{i_1};\ldots;S_{i_n}]} kill[p]\right)$

Since a path $p \in path[S_{i_1};\ldots;S_{i_n}]$ is a path of the set $paths[S]$ the inclusion holds.

■

Proof (Equations 16, 17) If *out* a *may(must)-problem*, then obviously *gen* is a *may(must)-* and *kill* a *must(may)-problem*. Using the previous results, when calculating
$\bigcup_{\vec{\imath}\in perm(1,n)} out[S_{i_1};\ldots;S_{i_n}]$ and $\bigcap_{\vec{\imath}\in perm(1,n)} out[S_{i_1};\ldots;S_{i_n}]$, and the fact that for all permutations $\vec{\imath} \in perm(1,n), in[S_{i_1};\ldots;S_{i_n}] = in[S]$ holds, demonstrates the goals (see appendix B).
■

Theorem 3 *Data flow equations for the* `PAR` *statement with replicators*
For the `S ::= PAR` S_1 `|` S_2 `| ... |` S_n `END` *statement with possibly replicated branches it holds:*

(a) if `upb`$_i$ `- lwb`$_i$ `+ 1 > 0` *the equations of theorem 1 and theorem 2*[8] *remain valid.*
(b) if `upb`$_i$ `- lwb`$_i$ `+ 1` $\geq$ `0` *the following equations hold*[9]*:*

[8] $Sibl[S_i]$ of the replicated statement S_i now contains also its own statements.
[9] n.r.: not replicated

If out *is a may-problem (upb_i - lwb_i + 1 $\geq$ 0):*

$$\begin{aligned}
\widetilde{in}[S] &= in[S] \cup \bigcup_{s \in sibl[S]} gen[s] \\
gen[S] &= \bigcup_{j=1}^{n} gen[S_j] \\
kill[S] &= \bigcup_{j=1, S_j \; n.r.}^{n} kill[S_j] - \bigcup_{k=1}^{n} gen[S_k] \\
out[S] &= gen[S] \cup \left(in[S] - \bigcup_{j=1, S_j \; n.r.}^{n} kill[S] \right)
\end{aligned}$$

If out *is a must-problem (upb_i - lwb_i + 1 $\geq$ 0):*

$$\begin{aligned}
\widetilde{in}[S] &= in[S] - \bigcup_{s \in sibl[S], s \; n.r.} kill[s] \\
gen[S] &= \bigcup_{j=1, S_j \; n.r.}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k] \\
kill[S] &= \bigcup_{j=1}^{n} kill[S_j] \\
out[S] &= gen[S] \cup \left(in[S] - \bigcup_{j=1}^{n} kill[S_j] \right)
\end{aligned}$$

Proof (Theorem 3a) We have to show that, if S_i is replicated then $info$`[PAR S`$_1$ `|` ... `| S`$_j$ `| S`$_{j+1}$ `|` ... `|S`$_n$ `END]` = $info$`[PAR S`$_1$ `|` ... `| S`$_j$ `| S`$_j$ `| S`$_{j+1}$ `|` ... `|S`$_n$ `END]` holds. Working out the equations for *may-* and *must-problems* using theorem 2 shows this. The proof of equation 1 is still valid (see appendix B). ■

Proof (Theorem 3b) The proofs are performed by calculating $info$`[PAR` ... `| S`$_j$ `|` ... `END]` $\wedge$ $info$`[PAR` ... `| S`$_k$ `|` ... `END]`, where in the S_j the replicators are assumed to generate more than one process and in S_k they are omitted, i.e. generate no process. ■

6 Conclusions

Without doubt, control flow parallel programs have to be analyzed and optimized as well as sequential ones. Since the traditional data flow analysis methods are designed in the context of sequential programs, they have to be adapted to fit into the parallel programming paradigm.

But instead of restricting either the shared memory model in the language, or force the user to instruct the compiler not to modify certain regions of code (by e.g. specifying a variable to be *volatile* or using pragmas) the equations presented in this paper may be used. Since they are an extension of the sequential ones, they may be easyly integrated into an existing optimizer.

Solutions to other data flow problems than those which have been presented, can be derived by the method shown in the proofs: Concatenate the process bodies, permute them, calculate a formula for the union/intersection of these $n!$ statement sequences and proof the equality with the meet over all paths solution, using the proof techniques given in the previous sections.

References

[ACM89] ACM. *16. ACM Symposium on Principles of Programming Languages*, January 1989. Austin, Texas.

[ACM92] ACM. *19. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1992. Albuquerque, New Mexico.

[Afek $^{et\ al}$ 93] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.

[Aho $^{et\ al}$ 86] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers; Principles, Techniques and Tools.* Addison-Wessley Publishing Company, 1986.

[Babich $^{et\ al}$ 78] Wayne A. Babich and Mehdi Jazayeri. The method of attributes for data flow analysis. *Acta Informatica*, 10:345–272, 1978. Part I Exhaustive Analysis, Part II Demand Analysis.

[Banerjee $^{et\ al}$ 91] U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors. *Languages and Compilers for Parallel Computing*, volume 598 of *Lecture Notes in Computer Science.* Springer Verlag, Heidelberg, New York, August 1991.

[Chow $^{et\ al}$ 92] Jyh-Herng Chow and Williams Ludwell III Harrison. Compile-time analysis of parallel programs that share memory. In *[ACM92]*, pages 130–141, 1992.

[Cytron $^{et\ al}$ 89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *[ACM89]*, pages 25–35, 1989.

[DEC 92] Digital Equipment Cooperation DEC. *Alpha Architecture Handbook.* DEC, USA, February 1992. Preliminary.

[Grunwald $^{et\ al}$ 93] Dirk Grunwald and Harini Srinivasan. Data flow equations of explicitly parallel programs. In *PPoPP 93.* ACM SIGPLAN NOTICES, 1993.

[Hecht 77] Matthew S. Hecht. *Flow analysis of computer programs.* North Holland, 1977.

[Kildall 73] Gary A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, October 1973.

[Lamport 79] Leslie Lamport. How to make a multiprocessor computer that correctly executs multi process programs. *IEEE Transactions on Computers*, c-28(9):690–691, September 1979.

[Midkiff $^{et\ al}$ 90] S.P. Midkiff and David A. Padua. Issues in the optimization of parallel programs. In *Proceedings of the 1990* International Conference on Parallel Processing, pages 105–113, Volume II, August 1990.

[Reif 84] John H. Reif. Data flow analysis of distributed communicating processes. Technical Report TR-12-83, Harvard University, Center for Research in Computing Technology, September 1984.

[Srinivasan $^{et\ al}$ 91a] Harini Srinivasan and Dirk Grunwald. An efficient construction of parallel static single assignment form for structured parallel programs. Technical Report CU-CS-564-91, University of Colorado at Boulder, Department of Computer Science, December 1991.

[Srinivasan $^{et\ al}$ 91b] Harini Srinivasan and Michael Wolfe. Analysing programs with explicit parallelism. In *[Banerjee $^{et\ al}$ 91]*, 1991.

[Wolfe $^{et\ al}$ 91] Michael Wolfe and Harini Srinivasan. Data structures for optimizing programs with explicit parallelism. In *[Zima 91]*, pages 139–156, 1991.

[Zima 91] Hans Zima, editor. *Parallel Computing, 1. Int. ACPC Conference Salzburg, Austria*, volume 591 of *Lecture Notes in Computer Science*. Springer Verlag, Heidelberg, New York, September 1991.

Appendix

A Data flow equations for the `IF` and `REPEAT` statement

For the `S ::= IF E THEN S`$_1$ `ELSE S`$_2$ `END` statement the equations look like:

$$\begin{array}{rcl} in[S_1] &=& in[S] \\ in[S_2] &=& in[S] \\ gen[S] &=& gen[S_1] \cup gen[S_2] \\ kill[S] &=& kill[S_1] \cap kill[S_2] \\ out[S] &=& out[S_1] \cup out[S_2] \end{array} \qquad \begin{array}{rcl} in[S_1] &=& in[S] \\ in[S_2] &=& in[S] \\ gen[S] &=& gen[S_1] \cap gen[S_2] \\ kill[S] &=& kill[S_1] \cup kill[S_2] \\ out[S] &=& out[S_1] \cap out[S_2] \end{array}$$

If *out* is a May-problem — If *out* is a Must-problem

For the statement `S ::= REPEAT S`$_1$ `UNTIL E` holds:

$$\begin{array}{rcl} in[S_1] &=& in[S] \cup gen[S_1] \\ gen[S] &=& gen[S_1] \\ kill[S] &=& kill[S_1] \\ out[S] &=& out[S_1] \end{array} \qquad \begin{array}{rcl} in[S_1] &=& in[S] - kill[S_1] \\ gen[S] &=& gen[S_1] \\ kill[S] &=& kill[S_1] \\ out[S] &=& out[S_1] \end{array}$$

If *out* is a May-problem — If *out* is a Must-problem

It is important to see that no iteration is needed to compute these sets as needed if the equations are formulated over the control flow graph [Aho $^{et\ al}$ 86].

B Complete proofs

Proof (Equation 16)

$$
\begin{array}{cl}
\multicolumn{2}{l}{\bigcup_{\vec{\imath}\in perm(1,n)} out[S_{i_1};\ldots;S_{i_n}] \overset{(3)}{=}} \\
 & \bigcup_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}] \cup \bigcup_{\vec{\imath}\in perm(1,n)}(in[S_{i_1};\ldots;S_{i_n}] - kill[S_{i_1};\ldots;S_{i_n}]) \\
\multicolumn{2}{l}{\text{since } \forall\vec{\imath}\in perm(1,n) : in[S_{i_1};\ldots;S_{i_n}] = in[S]} \\
= & \bigcup_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}] \cup \bigcup_{\vec{\imath}\in perm(1,n)}(in[S] - kill[S_{i_1};\ldots;S_{i_n}]) \\
\overset{(20)}{=} & \bigcup_{\vec{\imath}\in perm(1,n)} gen[S_{i_1};\ldots;S_{i_n}] \cup in[S] - \bigcap_{\vec{\imath}\in perm(1,n)} kill[S_{i_1};\ldots;S_{i_n}] \\
\overset{(12),(15)}{=} & \bigcup_{p\in paths[S]} gen[p] \cup in[S] - \bigcap_{p\in paths[S]} kill[p] \\
\overset{(19)}{=} & \bigcup_{p\in paths[S]} gen[p] \cup \bigcup_{p\in paths[S]}(in[S] - kill[p]) \\
\multicolumn{2}{l}{\text{since } \forall_{p\in paths[S]} : in[p] = in[S]} \\
= & \bigcup_{p\in paths[S]}(gen[p] \cup in[p] - kill[p]) \\
\overset{(3)}{=} & \bigcup_{p\in paths[S]} out[p]
\end{array}
$$

■

Proof (Equation 17) $\bigcap_{\vec{\imath}\in perm(1,n)} out[S_{i_1};\ldots;S_{i_n}] \overset{(11)}{=} \bigcup_{j=1}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k] \cup in[S] - \bigcup_{l=1}^{n} kill[S_l]$
$\overset{(7),(13),(8),(14)}{=} gen[S] \cup in[S] - kill[S] \overset{(3)}{=} out[S] \overset{\text{by definition}}{=} \bigcap_{p\in paths[S]} out[p]$

■

Proof (Theorem 3b) Obviously: $(a): \quad \bigcup_{j=1,S_j\ n.r.}^{n} gen[S_j] \subseteq \bigcup_{k=1}^{n} gen[S_k]$
$(b): \quad \bigcup_{j=1,S_j\ n.r.}^{n} kill[S_j] \subseteq \bigcup_{k=1}^{n} kill[S_k]$ and $(c): \quad \overline{\bigcup_{j=1}^{n} kill[S_j]} \subseteq \overline{\bigcup_{k=1,S_k\ n.r.}^{n} kill[S_k]}$.

$\widetilde{in}$ **May** Obviously, since `(a)`.

$\widetilde{in}$ **Must** Since `(b)` together with (22).

***gen/kill* May** Obviously since, `(a)` and `(b)`, respectively.

***gen/kill* Must** It is only shown for *gen*, *kill* is proved similarly.

$$
\begin{array}{l}
(\bigcup_{j=1}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k]) \cap (\bigcup_{l=1,S_l\ n.r.}^{n} gen[S_l] - \bigcup_{i=1,S_i\ n.r.}^{n} kill[S_i]) \overset{(18)}{=} \\
\bigcup_{j=1}^{n} gen[S_j] \cap \overline{\bigcup_{k=1}^{n} kill[S_k]} \cap \bigcup_{l=1,S_l\ n.r.}^{n} gen[S_l] \cap \overline{\bigcup_{i=1,S_i\ n.r.}^{n} kill[S_i]} = \\
\bigcup_{j=1,S_j\ n.r.}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k], \text{ since } (\mathtt{a}), (\mathtt{c}).
\end{array}
$$

***out* May:**

$$
\begin{array}{l}
(\bigcup_{j=1}^{n} gen[S_j] \cup in[S] - \bigcup_{k=1}^{n} kill[S_k]) \cup (\bigcup_{l=1,S_l\ n.r.}^{n} gen[S_l] \cup in[S] - \bigcup_{i=1,S_i\ n.r.}^{n} kill[S_i]) = \\
\bigcup_{j=1}^{n} gen[S_j] \cup in[S] - \bigcup_{k=1,S_k\ n.r.}^{n} kill[S_k], \text{ since } (\mathtt{c}) \text{ and } (22).
\end{array}
$$

***out* Must:** $(\bigcup_{j=1}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k] \cup in[S] - \bigcup_{l=1}^{n} kill[S_l]) \cap$

$$
\begin{array}{l}
\qquad (\bigcup_{j=1,S_j\ n.r.}^{n} gen[S_j] - \bigcup_{k=1,S_k\ n.r.}^{n} kill[S_k] \cup in[S] - \bigcup_{l=1,S_l\ n.r.}^{n} kill[S_l]) = \\
\bigcup_{j=1,S_j\ n.r.}^{n} gen[S_j] - \bigcup_{k=1}^{n} kill[S_k] \cup in[S] - \bigcup_{l=1,S_l}^{n} kill[S_l]. \text{ see } gen \text{ - Must and } (\mathtt{b}), (\mathtt{c}) \\
\text{and } (22).
\end{array}
$$

■

C Frequently used formulas

The following equations for set differences hold, a, b, c, b_1, b_2 are sets:

$$
\begin{array}{rcll@{\qquad}rcll}
a - b & = & a \cap \overline{b} & (18) & a \cap (a - b) & = & a - b & (23) \\
a - (b \cup c) & = & (a - b) \cap (a - c) & (19) & a \cup (a - b) & = & a & (24) \\
a - (b \cap c) & = & (a - b) \cup (a - c) & (20) & (a - b) - c & = & (a - c) - b = a - (b \cup c) & (25) \\
(a \cup b) - c & = & (a - c) \cup (b - c) & (21) & a - (b - c) & = & (a - b) \cup (a \cap c) & (26) \\
b_1 \subseteq b_2 & \Rightarrow & (a - b_2) \subseteq (a - b_1) & (22) & a - (b \cup c) & = & a - b \text{ if } a \cap c = \emptyset & (27)
\end{array}
$$

D An Example

This example shows the definitions reaching the statements in and after a `PAR` statement.

```
                               definition      definitions reaching this statement
 a := 0;                          (1)          a:{}      b:{}       c:{}    d:{}
 b := 0;                          (2)          a:{1}     b:{}       c:{}    d:{}
 d := ...;                        (3)          a:{1}     b:{2}      c:{}    d:{}
 PAR
   a := 1;                        (4)          a:{1}     b:{2,8,10} c:{9}   d:{3,11}
   IF b = 0                                    a:{4}     b:{2,8,10} c:{9}   d:{3,11}
   THEN
       c := critical_1();         (5)          a:{4}     b:{2,8,10} c:{9}   d:{3,11}
       a := 0;                    (6)          a:{4}     b:{2,8,10} c:{5,9} d:{3,11}
   END;
   d := f(d);                     (7)          a:{4,6}   b:{2,8,10} c:{5,9} d:{3,11}
 |
   b := 1;                        (8)          a:{1,4,6} b:{2}      c:{5}   d:{3,7}
   IF a = 0                                    a:{1,4,6} b:{8}      c:{5}   d:{3,7}
   THEN
       c := critical_2();         (9)          a:{1,4,6} b:{8}      c:{5}   d:{3,7}
       b := 0;                    (10)         a:{1,4,6} b:{8}      c:{5,9} d:{3,7}
   END;
   d := f(d);                     (11)         a:{1,4,6} b:{8,10}   c:{5,9} d:{3,7}
 END;
 ...                                           a:{4,6}   b:{8,10}   c:{5,9} d:{7,11}
```