CARP

D6.1: Verification of PENCIL

Grant Agreement:	287767
Project Acronym:	CARP
Project Name:	Correct and Efficient Accelerator Programming
Instrument:	Small or medium scale focused research project (STREP)
Thematic Priority:	Alternative Paths to Components and Systems
Start Date:	1 December 2011
Duration:	36 months
Document Type[1]:	D (Deliverable)
Document Distribution[2]:	PU (Public)
Document Code[3]:	CARP-UT-RP-001
Version:	v1.0
Editor (Partner):	Marieke Huisman (UT)
Contributors:	UT, ICL
Workpackage(s):	WP6
Reviewer(s):	RWTHA
Due Date:	30 November 2013 (revised from 31 October 2013)
Submission Date:	6 December 2013
Number of Pages:	41

[1]MD = management document; TR = technical report; D = deliverable; P = published paper; CD = communication/dissemination.

[2]PU = Public; PP = Restricted to other programme participants (including the Commission Services); RE = Restricted to a group specified by the consortium (including the Commission Services); CO = Confidential, only for members of the consortium (including the Commission Services).

[3]This code is constructed as described in the Project Handbook.

D6.1: Verification of PENCIL

Stefan Blom[1], **Saeed Darabi**[1], **Marieke Huisman**[1], **Jeroen Ketema**[2]

[1]UT, [2]ICL

REVISION HISTORY

Date	**Version**	**Author**	**Modification**
2013-10-26	0.1	J. Ketema	PENCIL semantics
2013-11-15	0.2	J. Ketema	OpenMP verification
2013-11-27	0.3	S. Blom and S. Darabi	First draft on static verification of PENCIL
2013-11-28	0.4	S. Blom and S. Darabi	Applied feedback from Wojciech Mostowski (UT)
2013-11-29	0.5	M. Huisman	Executive summary, introduction and conclusions
2013-12-01	0.6	S. Blom	Cleanup in static verification chapter
2013-12-02	0.7	S. Blom, S. Darabi and M. Huisman	Draft deliverable ready for review
2013-12-03	0.8	S. Blom	Applied review comments from Christina Jansen (RWTH)
2013-12-06	0.9	J. Ketema	Applied review comments from Christina Jansen (RWTH)
2013-12-06	1.0	M. Huisman	Final version

APPROVALS

Role	**Name**	**Partner**	**Date**
Workpackage Leader	A.F. Donaldson	ICL	6 December 2013
Project Manager	A.F. Donaldson	ICL	6 December 2013

Contents

1 Executive Summary

This deliverable presents novel research and development results on techniques for the verification of PENCIL programs. First of all, we describe the formal semantics of PENCIL. This semantics incorporates checks on the PENCIL annotations **independent** and **ivdep**, *i.e.*, if a program violates these annotations, its semantics is not defined. Second, we sketch how permission-based separation logic can be used to verify statically that a PENCIL program respects its loop annotations, *i.e.*, when a program can be verified, its behaviour is defined and different from error. Finally, the last part of this deliverable investigates whether our ideas about PENCIL verification also can be applied to OpenMP.

2 Introduction

In recent years, massively parallel accelerator processors, primarily graphics processing units (GPUs) from companies such as AMD and NVIDIA, and based on designs from ARM, have become widely available to end-users. Accelerators offer significant compute power at a low cost, and tasks such as media processing, medical imaging and eye-tracking can be accelerated to beat CPU performance by orders of magnitude.

However, low-level languages working directly on the accelerator architecture are not well suited to productive, structured programming of accelerator applications. Therefore, within the consortium the high-level accelerator programming language PENCIL is developed. PENCIL will be suitable to allow straightforward translation from different DSLs, and it supports downstream compilation into extremely efficient low-level GPU code.

Since GPU applications are used in many different areas, such as medical image processing and media processing, software errors can have serious consequences for safety or cause significant financial damage. Therefore, it is important that the PENCIL programming language has a formal semantics, and is supported by thorough verification techniques. Moreover, these verification techniques should be related to verification techniques for the low-level code, such as the verification techniques for OpenCL presented in Deliverable D6.2. This enables to establish verification of an OpenCL program from the verification of the corresponding PENCIL program. In particular, the handwritten annotations that are needed for the verification of PENCIL programs can be compiled into suitable specifications for the resulting OpenCL program.

Chapter 3 provides a formal semantics for a core of the PENCIL language. The semantics can be straightforwardly extended to full PENCIL, but at the price of clarity of presentation. The semantics keeps track of the variables read and written so far during program evaluation (in read and write sets). Moreover, to be able to identify loop dependences, within a loop, a stack of these read and write sets is maintained, to keep track of the variables that are read and written during the different iterations of a loop. The rules use the stack of read and write sets to establish the absence of unwanted dependences. If the loop has dependences that are not allowed by its annotation, then the behaviour is defined to be error. The semantics rules guarantee that

- the behaviour of loops annotated with **independent** is different from error, only when there are no loop-carried dependences, and
- the behaviour of loops annotated with **ivdep** is different from error, only when there are no backward loop-carried dependences.

Chapter 4 then sketches our idea for verification of PENCIL programs using permission-based separation logic. We focus in particular on the verification of the loop annotations **independent** and **ivdep**. Permissions are used to specify for each iteration of the loop which variables may be written and read. Additionally, it is specified how permissions are transferred between the different iterations. When a loop is independent, it does not need to transfer any permissions between different loop iterations. When a loop is annotated with **ivdep**, it may transfer permissions to next iterations of the loop. Different transfer patterns are identified that characterise forward and backward dependences. We also describe how we plan to develop tool support for our specification technique.

Finally, Chapter 5 investigates applicability of our results to OpenMP, and it discusses related work on verification of OpenMP.

Since the work in this task was dependent on the development of PENCIL, the techniques developed so far are only pen-and-paper techniques, and no results have been published yet. However, in the next period, we plan to develop tool support for our techniques.

3 Formal Semantics of PENCIL

To enable formal verification of PENCIL, it is essential that we precisely pin down the meaning of PENICL programs. To this end we define an operational semantics for the language. We focus on a subset of the PENCIL language—"idealised PENCIL"—which figures the features of PENCIL that make it unique as a language. The differences with the full language are either easily or straightforwardly removed, but only at the expense of clarity.

The differences with the full language are as follows:

- The syntax deviates somewhat from the syntax of C and PENCIL. The most notable difference is the use of the keyword **var** in variable declarations.
- The language is not typed.
- The operators that may occur in expressions are not explicitly defined.
- Short-circuit evaluation of Boolean expressions is omitted.
- Variables cannot be declared and initialised simultaneously.
- Multi-dimensional arrays cannot be declared.
- While-loops are omitted.
- No functions can be defined or called. Observe that this implies that there are no access summaries. To be able to still define the semantics of access summaries, each PENCIL program is equipped with an access summary.

In addition to the above, we assume that each variable is declared at most once. That is, variables cannot go out of scope due to variable shadowing.

3.1 Grammar

The grammar of idealised PENCIL is presented in Figure 3.1. Except for the access summaries, the correspondence with the actual PENCIL language should be obvious. Recall that variable declarations employ the **var** keyword. Moreover, recall that we do not explicitly specify the operators that may occur in expressions; in the grammar *op* is used as a placeholder.

Although we leave the form of expressions mostly unspecified, we assume we have at least the constants **true** and **false**, with the obvious meaning.

As the ordering of the accesses as described by access summaries is irrelevant, we abstract from the definition of access summaries as functions. Instead, we define them in terms of tuples of sets. The tuples are 4-tuples consisting of the following components:

- A set $\mathscr{V}$ that specifies the variables whose accesses should be tracked; only variables that are in scope at the end of the program may occur in this set. The set, which does not occur in actual PENCIL, compensates for the lack of function parameters, which in the case of actual PENCIL implicitly specify which variables should be tracked.
- A set $\mathscr{U}$ that specifies which of the variables from $\mathscr{V}$ are *used*, i.e., may be read from.

$$
\begin{array}{lcl}
\mathsf{pencil} & ::= & \textbf{pencil}\ \mathsf{access}\ \{\mathsf{stmts}\} \\
\mathsf{stmts} & ::= & \mathsf{stmt}\,;\langle\mathsf{stmts}\rangle \\
\mathsf{stmt} & ::= & \textbf{var}\ \mathsf{name} \\
 & | & \textbf{var}\ \mathsf{name}[\mathsf{expr}] \\
 & | & \textbf{kill}(\mathsf{name}) \\
 & | & \textbf{kill}(\mathsf{name}[\mathsf{expr}]) \\
 & | & \textbf{assert}(\mathsf{expr}) \\
 & | & \textbf{assume}(\mathsf{expr}) \\
 & | & \mathsf{name} = \mathsf{expr} \\
 & | & \mathsf{name}[\mathsf{expr}] = \mathsf{expr} \\
 & | & \textbf{if}\,(\mathsf{expr})\ \{\mathsf{stmts}\}\ \textbf{else}\ \{\mathsf{stmts}\} \\
 & | & \langle\mathsf{annotation}\rangle\ \textbf{for}\,(\textbf{var}\ \mathsf{name} = \mathsf{expr}; \\
 & & \qquad\qquad\qquad \mathsf{name}\,[<|\leq|>|\geq]\,\mathsf{expr}; \\
 & & \qquad\qquad\qquad \mathsf{name} \mathrel{+}= \mathsf{expr}) \\
 & & \qquad\qquad \{\mathsf{stmts}\} \\
 & | & \textbf{break} \\
 & | & \textbf{continue} \\
\mathsf{expr} & ::= & \mathit{constant} \\
 & | & \mathsf{expr}\ \mathit{op}\ \mathsf{expr} \\
 & | & \mathsf{name} \\
 & | & \mathsf{name}[\mathsf{expr}] \\
\mathsf{access} & ::= & (\mathscr{V},\mathscr{U},\mathscr{D},\mathscr{M}) \\
\mathsf{annotation} & ::= & \textbf{independent}\ \mathsf{reduction}^{*} \\
 & | & \textbf{ivdep} \\
\mathsf{reduction} & ::= & \textbf{reduction}(\mathit{op} : \mathsf{name}^{*}) \\
\mathsf{name} & ::= & \mathit{any\ valid\ C\ name}
\end{array}
$$

Figure 3.1: The grammar of idealised PENCIL

- A set $\mathscr{D}$ that specifies which of the variables from $\mathscr{V}$ are *defined*, i.e., must be written to.
- A set $\mathscr{M}$ that specifies which of the variables from $\mathscr{M}$ may be defined, i.e., may be written to.

Remark that any variable from $\mathscr{V}$ which does not occur in $\mathscr{U}$ may not be read from. Likewise, any variable from $\mathscr{V}$ that does not occur in either $\mathscr{D}$ or $\mathscr{M}$ may not be written to.

3.2 Semantics of Expressions

The operational semantics of expressions is defined in a big-step operational style. Given a tuple $\langle\sigma,e\rangle$ consisting of a store σ, i.e., a mapping from variables to values, and an expression e, the operational semantics defines the evaluation of e under σ. The values in the store can be both scalar (in the case the value of a scalar variable is being represented) and maps from integers to scalars (in case the value of an array is being represented).

The evaluation a tuple $\langle\sigma,e\rangle$ either results in a triple $\langle\sigma',v,(\mathscr{R},\mathscr{W})\rangle$ or in a special value *error* indicating that an error occurred during evaluation.

Given a tuple $\langle \sigma, e \rangle$, the elements of the triple $\langle \sigma', v, (\mathscr{R}, \mathscr{W}) \rangle$ are as follows:

- σ' is the store σ after evaluation of e.
- v is the value to which e evaluates.
- $(\mathscr{R}, \mathscr{W})$ specifies which variables were read and written during evaluation of e. The sets $\mathscr{R}$ and $\mathscr{W}$ are called the read and write set, respectively.

The rules for evaluating expressions are as follows:

- A constant c is evaluated by yielding the value of the constant:

  $$\frac{}{\vdash \langle \sigma, c \rangle \Downarrow \langle \sigma, c, (\emptyset, \emptyset) \rangle} \textsc{Constant}$$

  As no variables are accessed, the read and write sets are empty.

- An operator expression $e_1\, op\, e_2$ is evaluated by first evaluating the expression e_1 on the left-hand side of the operator, next evaluating the expression e_2 on the right-hand side of the operator, and finally applying the operator to the values obtained[1]:

  $$\frac{\vdash \langle \sigma, e_1 \rangle \Downarrow \langle \sigma', v_1, (\mathscr{R}, \mathscr{W}) \rangle \qquad \vdash \langle \sigma', e_2 \rangle \Downarrow \langle \sigma'', v_2, (\mathscr{R}', \mathscr{W}') \rangle}{\vdash \langle \sigma, e_1\, op\, e_2 \rangle \Downarrow \langle \sigma'', v_1\, op\, v_2, (\mathscr{R} \cup \mathscr{R}', \mathscr{W} \cup \mathscr{W}') \rangle} \textsc{Op}$$

  Recall from above we do not treat short-circuit evaluation of Boolean expressions. The read and write sets are simply the unions of the read and write sets obtained during evaluation of e_1 and e_2.

  Evaluation of an operator expression can fail both during evaluation of the left- and right-hand side. This gives rise to two additional rules:

  $$\frac{\vdash \langle \sigma, e_1 \rangle \Downarrow \mathit{error}}{\vdash \langle \sigma, e_1\, op\, e_2 \rangle \Downarrow \mathit{error}} \textsc{Op-Err-1}$$

  $$\frac{\vdash \langle \sigma, e_1 \rangle \Downarrow \langle \sigma', v_1, (\mathscr{R}, \mathscr{W}) \rangle \qquad \vdash \langle \sigma', e_2 \rangle \Downarrow \mathit{error}}{\vdash \langle \sigma, e_1\, op\, e_2 \rangle \Downarrow \mathit{error}} \textsc{Op-Err-2}$$

- A variable n is evaluated by obtaining its value from the store σ. Here, $n \in \mathit{Dom}(\sigma)$ denotes that n occurs in the store (i.e., is in scope).

  $$\frac{n \in \mathit{Dom}(\sigma)}{\vdash \langle \sigma, n \rangle \Downarrow \langle \sigma, \sigma(n), (\{n\}, \emptyset) \rangle} \textsc{Val}$$

  As n is read, the read set becomes equal to $\{n\}$. The write set is empty, as not variable is updated during the evaluation.

  Evaluation of a variable is erroneous in case the variable does not occur in the store (i.e., is out of scope). This can happen, e.g., because the variable was not declared or because the variable was **kill**ed (see the discussion of this statement below):

  $$\frac{n \notin \mathit{Dom}(\sigma)}{\vdash \langle \sigma, n \rangle \Downarrow \mathit{error}} \textsc{Val-Dom-Err}$$

[1] Observe that the evaluation order of the left- and right-hand side is unspecified in C and, hence, in PENCIL. We could introduce additional rules that make different evaluation orders possible, but we do not do so for brevity.

- An array access $n[e]$ is evaluated by first evaluating the expression e, obtaining a value v, and next retrieving the value of $n[v]$ from the store. This requires both the array n and the array element v to occur in the store. It may happen that an array element does not occur in the store, because **kill** may have been applied to it. Recall that array variables are maps from integers to scalars, hence $Dom(\sigma'(n))$ is well-defined.

$$\frac{\vdash \langle \sigma, e\rangle \Downarrow \langle \sigma', v, (\mathscr{R}, \mathscr{W})\rangle \qquad n \in Dom(\sigma') \qquad v \in Dom(\sigma'(n))}{\vdash \langle \sigma, n[e]\rangle \Downarrow \langle \sigma', \sigma'(n)(v), (\{n[v]\} \cup \mathscr{R}, \mathscr{W})\rangle}\ \textsc{Val-Arr}$$

The read set combines $\{n[v]\}$ with the variables being read during evaluation of e. The write set is simply taken from the evaluation of e.

As expression evaluation may fail and as either n or $n[v]$ may not occur in the store, there are three additional rules:

$$\frac{\vdash \langle \sigma, e\rangle \Downarrow error}{\vdash \langle \sigma, n[e]\rangle \Downarrow error}\ \textsc{Val-Arr-Err}$$

$$\frac{\vdash \langle \sigma, e\rangle \Downarrow \langle \sigma', v, (\mathscr{R}, \mathscr{W})\rangle \qquad n \notin Dom(\sigma')}{\vdash \langle \sigma, n[e]\rangle \Downarrow error}\ \textsc{Val-Arr-Dom-Err}$$

$$\frac{\begin{array}{c}\vdash \langle \sigma, e\rangle \Downarrow \langle \sigma', v, (\mathscr{R}, \mathscr{W})\rangle \\ n \in Dom(\sigma') \qquad v \notin Dom(\sigma'(n))\end{array}}{\vdash \langle \sigma, n[e]\rangle \Downarrow error}\ \textsc{Val-Arr-Dom-N-Err}$$

Observe that the rules as given above always yield a triple $\langle \sigma, v, (\mathscr{R}, \emptyset)\rangle$, i.e., the store is left unchanged and the write set is always empty. This would change if we would allow function calls to occur in expressions. To keep this latter fact explicit, we choose to include σ' and $\mathscr{W}$ in the triple $\langle \sigma', v, (\mathscr{R}, \mathscr{W})\rangle$.

3.3 Semantics of Programs and Statements

The operational of programs and statements is defined in a small-step operational fashion. This involves a 4-tuple $\langle \sigma, ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C})\rangle$, where the components are as follows:

- σ is a store, i.e., a mapping from program variables to values.
- ss is the sequence of statements to be evaluated.
- $(\mathscr{R}, \mathscr{W})$ is a pair of sets specifying which variables have been read and written so far during program evaluation. The sets $\mathscr{R}$ and $\mathscr{W}$ are called, respectively, the read and write sets.
- $(\mathscr{L}, \mathscr{C})$ is a tuple of stacks. The elements of these stacks are $(\mathscr{R}, \mathscr{W})$-pairs. The stacks are used to track, respectively, which variables have been accessed during previous iterations and the current iteration of the loop-nest currently being evaluated.

The operational meaning of an idealised PENCIL program

$$\textbf{pencil}\,(\mathscr{V}, \mathscr{U}, \mathscr{D}, \mathscr{M})\,\{ss\}$$

is now defined as the result of evaluating

$$\langle \sigma_e, ss, (\emptyset, \emptyset), (\texttt{empty}, \texttt{empty}) \rangle$$

according to the rules given below. Here, σ_e denotes the empty store, which assumes all variables to be out of scope, i.e., $n \notin Dom(\sigma_e)$ for all n. Moreover, `empty` denotes an empty stack.

3.3.1 Preliminaries

Stack Handling To handle the stacks in our operation semantics, we need some additional notation. Given a stack $\mathscr{S}$ of pairs (S_r, S_w), we define:

- The function $\texttt{push}(\mathscr{S}, (\mathscr{R}, \mathscr{W}))$ yields a stack $\mathscr{S}'$ which is identical to the stack $\mathscr{S}$ except that $(\mathscr{R}, \mathscr{W})$ was added as the top-most element.
- The function $\texttt{top}(\mathscr{S})$ yields the top-most element of $\mathscr{S}$.
- The function $\texttt{pop}(\mathscr{S})$ yields a stack $\mathscr{S}'$ which is equal to the stack $\mathscr{S}$ except that the top-most element has been removed.
- The function $\texttt{update\_top}(\mathscr{S}, (\mathscr{R}, \mathscr{W}))$ yields a stack $\mathscr{S}'$ which is equal to the stack $\mathscr{S}$ except that the top-most element (S_r, S_w) has been replaced by $(S_r \cup \mathscr{R}, S_w \cup \mathscr{W})$. Thus we have

  $$\texttt{update\_top}(\mathscr{S}, (\mathscr{R}, \mathscr{W})) = \texttt{push}(\texttt{pop}(\mathscr{S}), (S_r \cup \mathscr{R}, S_w \cup \mathscr{W}))$$

  with $(S_r, S_w) = \texttt{top}(\mathscr{S})$.
- The function $\texttt{clear\_top}(\mathscr{S})$ yields a stack $\mathscr{S}'$ which is equal to the stack $\mathscr{S}$ except that the top-most element has been replaced by $(\emptyset, \emptyset)$. Thus we have

  $$\texttt{clear\_top}(\mathscr{S}) = \texttt{push}(\texttt{pop}(\mathscr{S}), (\emptyset, \emptyset)) .$$
- The function $\texttt{add}(\mathscr{S}, (\mathscr{R}, \mathscr{W}))$ yields a stack $\mathscr{S}'$ equal to the stack $\mathscr{S}$ but with *each* pair (S_r, S_w) from $\mathscr{S}$ replaced by $(S_r \cup \mathscr{R}, S_w \cup \mathscr{W})$.

Loop-Carried Dependence Handling To handle loop-carried dependences in a PENCIL program, we need to establish for all loops and all iterations of those loops whether:

a. a variable that has been written in the current loop iteration was either read or written in a previous loop iteration, or

b. a variable that has been read or written in the current loop iteration was written in a previous loop iteration.

Formally, given the accumulated read and write sets (L_r, L_w) from the previous iterations of some loop and the read and write sets (C_r, C_w) from the current loop iteration we define:

$$\texttt{dependence}((L_r, L_w), (C_r, C_w)) \triangleq ((L_r \cup L_w) \cap C_w) \cup (L_w \cap (C_r \cup C_w)) .$$

Given that n is the loop-counter of the inner-loop at the end of whose iteration we currently are, then checking that there are no *loop-carried dependences* with previous iterations of the inner-loop amounts to checking that:

$$\texttt{indep}(\mathscr{L},\mathscr{C},N) \triangleq (\texttt{dependence}(\texttt{top}(\mathscr{L}),\texttt{top}(\mathscr{C})) \setminus N) = \emptyset.$$

Here, N contains n and the reduction variables of the current loop. Moreover, $\texttt{top}(\mathscr{L})$ is the accumulated $(\mathscr{R},\mathscr{W})$-pair of all previous loop iterations of the current inner-loop. And, $\texttt{top}(\mathscr{C})$ is the $(\mathscr{R},\mathscr{W})$-pair of the current loop iteration.

Let $m \in S_r \cup S_w$ and let $\texttt{stmts}((S_r,S_w),m)$ denote the statements responsible for adding m to (S_r,S_2) (in the PENCIL program under consideration). Moreover, let T_1 and T_2 be sets of statements, and write $T_1 \prec T_2$ if each statement in T_1 occurs textually before all statements in T_2. Given that n is the loop-counter an inner-loop at the end of whose iteration we currently are, then checking that there are no *backward loop-carried dependences* with previous iterations of the inner-loop amounts to checking:

$$\texttt{back\_indep}(\mathscr{L},\mathscr{C},N) \triangleq \forall m \in (\texttt{dependence}(\texttt{top}(\mathscr{L}),\texttt{top}(\mathscr{C})) \setminus N) : \texttt{stmts}(\texttt{top}(\mathscr{L}),m) \prec \texttt{stmts}(\texttt{top}(\mathscr{C}),m)$$

Here, $N = \{n\}$.

3.3.2 Termination and Out-of-Scope Variables

Following the grammar of idealised PENCIL, we will now discuss the operational semantics. We start with termination and the treatment of out-of-scope variables.

- If there is no statement to be evaluated, i.e., if the sequence of statements is empty (denoted ε), then we terminate. Upon termination it is checked that the access summary has been adhered to during evaluation and the final store is returned:

$$\frac{\mathscr{R} \cap \mathscr{V} \subseteq \mathscr{U} \qquad \mathscr{D} \subseteq \mathscr{W} \cap \mathscr{V} \qquad \mathscr{W} \cap \mathscr{V} \subseteq \mathscr{D} \cup \mathscr{M}}{\vdash \langle \sigma, \varepsilon, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C}) \rangle \to \sigma}\ \textsc{Terminate}$$

  In case the access summary is not being adhered to, this can can have three causes, which of which results in a transition to the error state *error*. Here, either some variable from the read set (which also occurs in $\mathscr{V}$) does not occur in the used set $\mathscr{U}$:

$$\frac{\mathscr{R} \cap \mathscr{V} \nsubseteq \mathscr{U}}{\vdash \langle \sigma, \varepsilon, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C}) \rangle \to \mathit{error}}\ \textsc{Terminate-Read-Err}$$

  Or, some variable from the defined set $\mathscr{D}$ does not occur in the write set:

$$\frac{\mathscr{D} \nsubseteq \mathscr{W} \cap \mathscr{V}}{\vdash \langle \sigma, \varepsilon, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C}) \rangle \to \mathit{error}}\ \textsc{Terminate-Def-Err}$$

  Or, some variable from the write set (which also occurs in $\mathscr{V}$) does not occur in the defined set $\mathscr{D}$ or the maybe defined set $\mathscr{M}$:

$$\frac{\mathscr{W} \cap \mathscr{V} \nsubseteq \mathscr{D} \cup \mathscr{M}}{\vdash \langle \sigma, \varepsilon, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C}) \rangle \to \mathit{error}}\ \textsc{Terminate-Write-Err}$$

- To handle the scoping of variables, we introduce an addition keyword **clear_scope**. This keyword is not supposed to be specified by programmers; it only exists to define the operational semantics. Evaluation of **clear_scope** removes all variables that are no longer in scope at the point the statement is executed from the store, the read and write sets, and stacks:

$$\frac{\begin{array}{c}\sigma' \text{ is identical to } \sigma,\\ \text{except that } n \notin Dom(\sigma') \text{ for any variable } n \text{ not in scope}\\ \mathscr{R}', \mathscr{W}', \mathscr{L}', \text{ and } \mathscr{C}' \text{ are identical to } \mathscr{R}, \mathscr{W}, \mathscr{L}, \text{ and } \mathscr{C},\\ \text{except that all out-of-scope variables have been removed}\end{array}}{\vdash \langle \sigma, \mathbf{clear\_scope}; ss, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C})\rangle \to \langle \sigma', ss, (\mathscr{R}',\mathscr{W}'), (\mathscr{L}',\mathscr{C}')\rangle} \text{ CLEAR}$$

3.3.3 Variable and Array Declarations

Variable and array declarations extend the store by mapping the declared variable or array to a value or values. Initially, the values are arbitrary, which we indicate by $\star$.

- Evaluation of a variable declaration **var** n extends the store with n:

$$\frac{}{\vdash \langle \sigma, \mathbf{var}\, n; ss, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C})\rangle \to \langle \sigma[n \mapsto \star], ss, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C})\rangle} \text{ VAR}$$

  Observe that declaring a variable does not mean it is being written, i.e., the write sets and stacks are left unchanged.

- Evaluation of an array declaration **var** $n[e]$ requires evaluation of the expression e, obtaining a value v. Once the expression has been evaluated the store is extended with an array n of size v, all elements of which are given an arbitrary value $((\star)_{0..v-1})$:

$$\frac{\begin{array}{c}\vdash \langle \sigma, e\rangle \Downarrow \langle \sigma', v, (\mathscr{R}',\mathscr{W}')\rangle\\ \mathscr{R}^* = \mathscr{R} \cup \mathscr{R}' \qquad \mathscr{W}^* = \mathscr{W} \cup \mathscr{W}'\\ \mathscr{C}^* = \mathtt{add}(\mathscr{C}, (\mathscr{R}',\mathscr{W}'))\end{array}}{\begin{array}{l}\vdash \langle \sigma, \mathbf{var}\, n[e]; ss, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C})\rangle\\ \qquad \to \langle \sigma'[n \mapsto (\star)_{0..v-1}], ss, (\mathscr{R}^*,\mathscr{W}^*), (\mathscr{L},\mathscr{C}^*)\rangle\end{array}} \text{ VAR-ARR}$$

  As variables may be read or written during evaluation of the expression e, the read and write sets and the stack for the current nested loop iterations are updated appropriately.

  In case evaluation of the expression yields *error*, we transition to the error state:

$$\frac{\vdash \langle \sigma, e\rangle \Downarrow error}{\vdash \langle \sigma, \mathbf{var}\, n[e]; ss, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C})\rangle \to error} \text{ VAR-ARR-ERR}$$

3.3.4 Kill Statements

The **kill** statement removes the specified variable, array, or array element from the store. Thus, e.g., after evaluating **kill**(n) we have that $n \notin Dom(\sigma)$, where σ is the store after evaluation of the statement. As a result, accessing n later in the evaluation will result in a transition to *error*, as it will be checked that $n \in Dom(\sigma)$.

- Evaluating **kill**(n) with n either a variable or an array maps n to the undefined value $\perp$ in the store:

$$\frac{}{\vdash \langle \sigma, \mathbf{kill}(n); ss, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C})\rangle \to \langle \sigma[n \mapsto \perp], ss, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C})\rangle}\ \textsc{Kill}$$

  Observe that **kill**ing a variable or array does not mean it is being read or written, i.e., the read and write sets and stacks are left unchanged. Moreover, observe that it does not matter whether $n \in Dom(\sigma)$; it is valid to evaluate **kill** in case n is not defined.

- Evaluating **kill**$(n[e])$ with n an array evaluates the expression e, obtaining a value v, and map $n[v]$ to $\perp$:

$$\frac{\begin{array}{c}\vdash \langle \sigma, e\rangle \Downarrow \langle \sigma', v, (\mathscr{R}',\mathscr{W}')\rangle \qquad n \in Dom(\sigma') \\ \mathscr{R}^* = \mathscr{R} \cup \mathscr{R}' \qquad \mathscr{W}^* = \mathscr{W} \cup \mathscr{W}' \\ \mathscr{C}^* = \mathtt{add}(\mathscr{C}, (\mathscr{R}',\mathscr{W}'))\end{array}}{\begin{array}{l}\vdash \langle \sigma, \mathbf{kill}(n[e]); ss, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C})\rangle \\ \qquad\qquad \to \langle \sigma'[n(v) \mapsto \perp], ss, (\mathscr{R}^*,\mathscr{W}), (\mathscr{L},\mathscr{C}^*)\rangle\end{array}}\ \textsc{Kill-Arr}$$

  As variables may be read or written during the evaluation of the expression e, the read and write sets and the stack for the current nested loop iterations are updated appropriately.

  Failure occurs in case evaluation of the expression results in *error*:

$$\frac{\vdash \langle \sigma, e\rangle \Downarrow \mathit{error}}{\vdash \langle \sigma, \mathbf{kill}(n[e]); ss, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C})\rangle \to \mathit{error}}\ \textsc{Kill-Arr-Err}$$

  Evaluating the statement also fails in case the array n does not exist:

$$\frac{\vdash \langle \sigma, e\rangle \Downarrow \langle \sigma', v, (\mathscr{R}',\mathscr{W}')\rangle \qquad n \notin Dom(\sigma')}{\vdash \langle \sigma, \mathbf{kill}(n[e]); ss, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C})\rangle \to \mathit{error}}\ \textsc{Kill-Arr-Dom-Err}$$

3.3.5 Assert and Assume

The **assert** and **assume** statements check whether the specified expression evaluates to **true**, after which evaluation continues with the remaining statements. In case the expression evaluates to **false** a transition to the error state occurs.

- Evaluation of **assert**(e) with e evaluating to **true** lets evaluation continue with the remaining statements ss:

$$\frac{\begin{array}{c}\vdash \langle \sigma, e\rangle \Downarrow \langle \sigma', \mathbf{true}, (\mathscr{R}',\mathscr{W}')\rangle \\ \mathscr{R}^* = \mathscr{R} \cup \mathscr{R}' \qquad \mathscr{W}^* = \mathscr{W} \cup \mathscr{W}' \\ \mathscr{C}^* = \mathtt{add}(\mathscr{C}, (\mathscr{R}',\mathscr{W}'))\end{array}}{\vdash \langle \sigma, \mathbf{assert}(e); ss, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C})\rangle \to \langle \sigma', ss, (\mathscr{R}^*,\mathscr{W}), (\mathscr{L},\mathscr{C}^*)\rangle}\ \textsc{Assert-T}$$

  As variables may be read or written during the evaluation of the expression e, the read and write sets and the stack for the current nested loop iterations are updated appropriately.

  In case the expression evaluates to **false**, we transition to the error state:

$$\frac{\vdash \langle \sigma, e\rangle \Downarrow \langle \sigma', \mathbf{false}, (\mathscr{R}',\mathscr{W}')\rangle}{\vdash \langle \sigma, \mathbf{assert}(e); ss, (\mathscr{R},\mathscr{W}), (\mathscr{L},\mathscr{C})\rangle \to \mathit{error}}\ \textsc{Assert-F}$$

Moreover, in case evaluation of the expression yields *error*, we also transition to the error state:

$$\frac{\vdash \langle \sigma, e \rangle \Downarrow error}{\vdash \langle \sigma, \textbf{assert}(e); ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \to error} \ \textsc{Assert-Err}$$

- An **assume** statement is simply rewritten to an **assert** statement to obtain the same behaviour as for **assert**s:

$$\frac{}{\begin{array}{l}\vdash \langle \sigma, \textbf{assume}(e); ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \\ \qquad\qquad \to \langle \sigma, \textbf{assert}(e); ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle\end{array}} \ \textsc{Assume}$$

We rewrite **assume**s to **assert**s, because we would like to check that the validity of the statement (like in the case of every other PENCIL-specific construct).

3.3.6 Assignments

Assignments update the store after evaluating the expression specified in the assignment.

- An assignment $n = e$ evaluates e, obtaining v, and maps n to v in the store:

$$\frac{\begin{array}{c}\vdash \langle \sigma, e \rangle \Downarrow \langle \sigma', v, (\mathscr{R}', \mathscr{W}') \rangle \qquad n \in Dom(\sigma) \\ \mathscr{R}^* = \mathscr{R} \cup \mathscr{R}' \qquad \mathscr{W}^* = \mathscr{W} \cup \mathscr{W}' \cup \{n\} \\ \mathscr{C}^* = \texttt{add}(\mathscr{C}, (\mathscr{R}', \mathscr{W}' \cup \{n\}))\end{array}}{\begin{array}{l}\vdash \langle \sigma, n = e; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \\ \qquad\qquad \to \langle \sigma'[n \mapsto v], ss, (\mathscr{R}^*, \mathscr{W}^*), (\mathscr{L}, \mathscr{C}^*) \rangle\end{array}} \ \textsc{Assign}$$

As variables may be read or written during evaluation of the expression e, the read and write sets and the stack for the current nested loop iterations are updated appropriately. And, similarly n is added, as it is being written.

Evaluation fails in case evaluation of the expression fails:

$$\frac{\vdash \langle \sigma, e \rangle \Downarrow error}{\vdash \langle \sigma, n = e; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \to error} \ \textsc{Assign-Err}$$

Evaluation of the assignment also fails in case n does not occur in the store:

$$\frac{\vdash \langle \sigma, e \rangle \Downarrow error \qquad n \notin Dom(\sigma)}{\vdash \langle \sigma, n = e; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \to error} \ \textsc{Assign-Dom-Err}$$

- An assignment $n[e_1] = e_2$, assigning a value to an array element, evaluates e_2, obtaining v_2, and next evaluates e_1 to determine to which array element v_2 should be assigned:

$$\frac{\begin{array}{c}\vdash \langle \sigma, e_2 \rangle \Downarrow \langle \sigma', v_2, (\mathscr{R}', \mathscr{W}') \rangle \qquad \vdash \langle \sigma', e_1 \rangle \Downarrow \langle \sigma'', v_1, (\mathscr{R}'', \mathscr{W}'') \rangle \\ n \in Dom(\sigma'') \qquad v_1 \in Dom(\sigma''(n)) \\ \mathscr{R}^* = \mathscr{R} \cup \mathscr{R}' \cup \mathscr{R}'' \qquad \mathscr{W}^* = \mathscr{W} \cup \mathscr{W}' \cup \mathscr{W}'' \cup \{n[v_1]\} \\ \mathscr{C}^* = \texttt{add}(\mathscr{C}, (\mathscr{R}' \cup \mathscr{R}'', \mathscr{W}' \cup \mathscr{W}'' \cup \{n[v_1]\}))\end{array}}{\begin{array}{l}\vdash \langle \sigma, n[e_1] = e_2; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \\ \qquad\qquad \to \langle \sigma''[n(v_1) \mapsto v_2], ss, (\mathscr{R}^*, \mathscr{W}^*), (\mathscr{L}, \mathscr{C}^*) \rangle\end{array}} \ \textsc{Assign-Arr}$$

As variables may be read or written during evaluation of the expressions e_1 and e_2, the read and write sets and the stack for the current nested loop iterations are updated appropriately. And, similarly $n[e_1]$ is added, as it is being written.

The assignment fails in case evaluation of e_2 fails:

$$\frac{\vdash \langle \sigma, e_2 \rangle \Downarrow \mathit{error}}{\vdash \langle \sigma, n[e_1] = e_2; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \to \mathit{error}} \text{ ASSIGN-ARR-ERR-2}$$

Similarly, the assignment fails in the case e_1 cannot be evaluated successfully:

$$\frac{\vdash \langle \sigma, e_2 \rangle \Downarrow \langle \sigma', v_2, (\mathscr{R}', \mathscr{W}') \rangle \qquad \vdash \langle \sigma', e_1 \rangle \Downarrow \mathit{error}}{\vdash \langle \sigma, n[e_1] = e_2; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \to \mathit{error}} \text{ ASSIGN-ARR-ERR-1}$$

The assignment also fails in the case n does not occur in the store:

$$\frac{\begin{array}{c}\vdash \langle \sigma, e_2 \rangle \Downarrow \mathit{error} \qquad \vdash \langle \sigma', e_1 \rangle \Downarrow \langle \sigma'', v_1, (\mathscr{R}'', \mathscr{W}'') \rangle \\ n \notin \mathit{Dom}(\sigma'')\end{array}}{\vdash \langle \sigma, n[e_1] = e_2; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \to \mathit{error}} \text{ ASSIGN-ARR-DOM-ERR}$$

Finally, the assignment fails in the case the array element to which a value is to be assigned is undefined ($v_1 \notin \mathit{Dom}(\sigma''(n))$). This is possible, as the array element may have been **kill**ed:

$$\frac{\begin{array}{c}\vdash \langle \sigma, e_2 \rangle \Downarrow \langle \sigma', v_2, (\mathscr{R}', \mathscr{W}') \rangle \qquad \vdash \langle \sigma', e_1 \rangle \Downarrow \langle \sigma'', v_1, (\mathscr{R}'', \mathscr{W}'') \rangle \\ n \in \mathit{Dom}(\sigma'') \qquad v_1 \notin \mathit{Dom}(\sigma''(n))\end{array}}{\vdash \langle \sigma, n[e_1] = e_2; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \to \mathit{error}} \text{ ASSIGN-ARR-DOM-N-ERR}$$

3.3.7 If-Statements

If-statements evaluate a Boolean expression and branch accordingly.

- In case the expression e evaluates to **true**, the if-branch is taken, i.e., ss_1 becomes the expression to be evaluated next. Moreover, it is ensured that the scope is cleared of any variables declared in ss_1 after evaluation of ss_1:

$$\frac{\begin{array}{c}\vdash \langle \sigma, e \rangle \Downarrow \langle \sigma', \mathbf{true}, (\mathscr{R}', \mathscr{W}') \rangle \\ \mathscr{R}^* = \mathscr{R} \cup \mathscr{R}' \qquad \mathscr{W}^* = \mathscr{W} \cup \mathscr{W}' \\ \mathscr{C}^* = \mathtt{add}(\mathscr{C}, (\mathscr{R}', \mathscr{W}'))\end{array}}{\begin{array}{l}\vdash \langle \sigma, \mathbf{if}(e)\, \{ss_1\}\, \mathbf{else}\, \{ss_2\}; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \\ \qquad \to \langle \sigma', ss_1; \mathbf{clear\_scope}; ss, (\mathscr{R}^*, \mathscr{W}^*), (\mathscr{L}, \mathscr{C}^*) \rangle\end{array}} \text{ IF-TRUE}$$

As variables may be read or written during evaluation of the expression e, the read and write sets and the stack for the current nested loop iterations are updated appropriately.

In case the expression e evaluates to **false**, the else-branch is taken, i.e., ss_2 becomes the expression to be evaluated next. Moreover, it is ensured that the scope is cleared of any variables declared in ss_2 after evaluation of ss_2:

$$\dfrac{\begin{array}{c}\vdash \langle \sigma, e\rangle \Downarrow \langle \sigma', \mathbf{false}, (\mathscr{R}', \mathscr{W}')\rangle \\ \mathscr{R}^* = \mathscr{R} \cup \mathscr{R}' \qquad \mathscr{W}^* = \mathscr{W} \cup \mathscr{W}' \\ \mathscr{C}^* = \mathtt{add}(\mathscr{C}, (\mathscr{R}', \mathscr{W}'))\end{array}}{\begin{array}{l}\vdash \langle \sigma, \mathbf{if}(e)\,\{ss_1\}\,\mathbf{else}\,\{ss_2\}; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C})\rangle \\ \qquad\qquad \to \langle \sigma', ss_2; \mathbf{clear\_scope}; ss, (\mathscr{R}^*, \mathscr{W}^*), (\mathscr{L}, \mathscr{C}^*)\rangle\end{array}}\ \textsc{If-False}$$

As variables may be read or written during evaluation of the expression e, the read and write sets and the stack for the current nested loop iterations are updated appropriately.

Evaluation of the if-statement fails in case evaluation of the expression fails:

$$\dfrac{\vdash \langle \sigma, e\rangle \Downarrow \mathit{error}}{\vdash \langle \sigma, \mathbf{if}(e)\,\{ss_1\}\,\mathbf{else}\,\{ss_2\}; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C})\rangle \to \mathit{error}}\ \textsc{If-Err}$$

3.3.8 For-Loops

Throughout we assume that the loop counter n is *not* **kill**ed in the loop body.

Evaluation of for-loops requires us to differentiate between the first loop iteration and later loop iterations: At the beginning of the first iteration the loop counter needs to be initialised. During later loop iterations it needs to be updated.

To differentiate between the initial loop iteration and later loop iterations we introduce an underlined version of the **for** keyword: $\underline{\mathbf{for}}$. This underlined version is not supposed to be specified by programmers; it only exists to define the operational semantics.

In case of an **independent** r annotation, $\mathit{Var}(r)$ denotes the set which contains the variable names that occur in the reduction declarations in r.

Recall from Figure 3.1 that annotations occur before before the **for** keyword and are generally denoted by the letter a.

- On the first loop iteration n is initialised and the loop condition $n\,op\,e_2$ is checked. If the loop condition evaluates to **true**, the statements ss' from the loop body will be evaluated next and the for-loop is added at the beginning of the sequence of statements to be evaluated after the loop body. Moreover, it is ensured that the scope is cleared of any variables declared in ss' after evaluation of ss'[2]:

$$\dfrac{\begin{array}{c} s = a\ \mathbf{for}(\mathbf{var}\,n = e_1; n\,op\,e_2; n \mathrel{+}= e_3)\,\{ss'\} \\ \vdash \langle \sigma, e_1\rangle \Downarrow \langle \sigma', v_1, (\mathscr{R}', \mathscr{W}')\rangle \\ \vdash \langle \sigma'[n \mapsto v_1], n\,op\,e_2\rangle \Downarrow \langle \sigma'', \mathbf{true}, (\mathscr{R}'', \mathscr{W}'')\rangle \\ \mathscr{R}^* = \mathscr{R} \cup \mathscr{R}' \cup \mathscr{R}'' \qquad \mathscr{W}^* = \mathscr{W} \cup \mathscr{W}' \cup \mathscr{W}'' \cup \{n\} \\ \mathscr{C}^+ = \mathtt{add}(\mathscr{C}, (\mathscr{R}', \mathscr{W}' \cup \{n\})) \\ \mathscr{L}^* = \mathtt{push}(\mathscr{L}, (\emptyset, \emptyset)) \qquad \mathscr{C}^* = \mathtt{add}(\mathtt{push}(\mathscr{C}^+, (\emptyset, \emptyset)), (\mathscr{R}'', \mathscr{W}''))\end{array}}{\begin{array}{l}\vdash \langle \sigma, s; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C})\rangle \\ \qquad\qquad \langle \sigma'', ss'; \mathbf{clear\_scope}; \underline{s}; ss, (\mathscr{R}^*, \mathscr{W}^*), (\mathscr{L}^*, \mathscr{C}^*)\rangle\end{array}}\ \textsc{For-True}$$

Above $\underline{s}$ denotes s with the **for**-keyword replaced by $\underline{\mathbf{for}}$. As variables may be read or written during evaluation of the expressions e_1 and e_2, the read and write sets and the

[2] Observe that n is still in scope after evaluation of ss'.

stack for the current nested loop iterations are updated appropriately. Similarly for the loop counter n, as it is being written. As a new loop is entered, the $\mathscr{L}$ and $\mathscr{C}$ stacks are extended with a new top element.

In the case the loop condition evaluates to **false** upon the first loop iteration, the loop is not entered and evaluation continues with clearing n from the scope and evaluation of ss:

$$\frac{\begin{array}{c} s = a\ \mathbf{for}(\mathbf{var}\, n = e_1;\, n\, op\, e_2;\, n \mathrel{+}= e_3)\, \{ss'\} \\ \vdash \langle \sigma, e_1 \rangle \Downarrow \langle \sigma', v_1, (\mathscr{R}', \mathscr{W}') \rangle \\ \vdash \langle \sigma'[n \mapsto v_1], n\, op\, e_2 \rangle \Downarrow \langle \sigma'', \mathbf{false}, (\mathscr{R}'', \mathscr{W}'') \rangle \\ \mathscr{R}^* = \mathscr{R} \cup \mathscr{R}' \cup \mathscr{R}'' \qquad \mathscr{W}^* = \mathscr{W} \cup \mathscr{W}' \cup \mathscr{W}'' \cup \{n\} \\ \mathscr{C}^* = \mathtt{add}(\mathscr{C}, (\mathscr{R}' \cup \mathscr{R}'', \mathscr{W}' \cup \{n\} \cup \mathscr{W}'')) \end{array}}{\begin{array}{l} \vdash \langle \sigma, s;\, ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \\ \qquad \rightarrow \langle \sigma'', \mathbf{clear\_scope};\, ss, (\mathscr{R}^*, \mathscr{W}^*), (\mathscr{L}, \mathscr{C}^*) \rangle \end{array}} \ \textsc{For-False}$$

As variables may be read or written during evaluation of the expressions e_1 and e_2, the read and write sets and the stack for the current nested loop iterations are updated appropriately. Similarly for the loop counter n, as it is being written.[3]

Evaluation of the loop fails in the case evaluation of the expression e_1 fails:

$$\frac{\begin{array}{c} s = a\ \mathbf{for}(\mathbf{var}\, n = e_1;\, n\, op\, e_2;\, n \mathrel{+}= e_3)\, \{ss'\} \\ \vdash \langle \sigma, e_1 \rangle \Downarrow \mathit{error} \end{array}}{\vdash \langle \sigma, s;\, ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \rightarrow \mathit{error}} \ \textsc{For-Err-1}$$

Similarly, evaluation of the loop fails in the case evaluation of the expression e_2 fails:

$$\frac{\begin{array}{c} s = a\ \mathbf{for}(\mathbf{var}\, n = e_1;\, n\, op\, e_2;\, n \mathrel{+}= e_3)\, \{ss'\} \\ \vdash \langle \sigma, e_1 \rangle \Downarrow \langle \sigma', v_1, (\mathscr{R}', \mathscr{W}') \rangle \\ \vdash \langle \sigma'[n \mapsto v_1], n\, op\, e_2 \rangle \Downarrow \mathit{error} \end{array}}{\vdash \langle \sigma, s;\, ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \rightarrow \mathit{error}} \ \textsc{For-Err-2}$$

- Upon any following loop iteration, the loop counter n is updated. And, if the loop condition evaluates to **true**, the statements ss' from the loop body will be evaluated next and the for-loop is re-added to the sequence of statements to enable any following iterations to be evaluated after the loop body. Moreover, it is ensured that the scope is cleared of any variables declared in ss' after evaluation of ss'.

  In addition, as a loop iteration has been completed at this point, it is checked whether the loop annotation is valid up to this point in the evaluation. In case of **independent** this means that it is checked that there are no loop-carried dependences. In case of **ivdep** it is

[3]Observe that n will immediately be removed by the **clear_scope** statement

checked that there are no backward loop-carried dependences:

$$\frac{\begin{array}{c} s = a\ \underline{\textbf{for}}(\textbf{var}\, n = e_1;\, n\, op\, e_2;\, n \mathrel{+}= e_3)\,\{ss'\} \\ \vdash \langle \sigma, n + e_3 \rangle \Downarrow \langle \sigma', v_3, (\mathscr{R}', \mathscr{W}') \rangle \\ \vdash \langle \sigma'[n \mapsto v_3], n\, op\, e_2 \rangle \Downarrow \langle \sigma'', \textbf{true}, (\mathscr{R}'', \mathscr{W}'') \rangle \\ \mathscr{R}^* = \mathscr{R} \cup \mathscr{R}' \cup \mathscr{R}'' \qquad \mathscr{W}^* = \mathscr{W} \cup \mathscr{W}' \cup \mathscr{W}'' \cup \{n\} \\ \mathscr{C}^+ = \mathtt{add}(\mathscr{C}, (\mathscr{R}', \mathscr{W}' \cup \{n\})) \\ (a = \textbf{independent}\, r) \Rightarrow \mathtt{indep}(\mathscr{L}, \mathscr{C}^+, \{n\} \cup Var(r)) \\ (a = \textbf{ivdep}) \Rightarrow \mathtt{back\_indep}(\mathscr{L}, \mathscr{C}^+, \{n\}) \\ \mathscr{L}^* = \mathtt{update\_top}(\mathscr{L}, \mathtt{top}(\mathscr{C}^+)) \\ \mathscr{C}^* = \mathtt{add}(\mathtt{clear\_top}(\mathscr{C}^+), (\mathscr{R}'', \mathscr{W}'')) \end{array}}{\begin{array}{l} \vdash \langle \sigma, s;\, ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \\ \qquad \to \langle \sigma'', ss';\, \textbf{clear\_scope};\, s;\, ss, (\mathscr{R}^*, \mathscr{W}^*), (\mathscr{L}^*, \mathscr{C}^*) \rangle \end{array}}\ \textsc{UFor-True}$$

As variables may be read or written during evaluation of the expressions e_3 and e_2, the read and write sets and the stack for the current nested loop iterations are updated appropriately. Similarly for the loop counter n which is written. As a new loop iteration is started, the top element of $\mathscr{C}$ is merged with the top element of $\mathscr{L}$. After this the top element of $\mathscr{C}$ is cleared.

In the case the loop condition evaluates to **false** with clearing n from the scope and evaluation of ss.

In addition, as a loop iteration has been completed at this point, it is checked whether the loop annotation is valid up to this point in the evaluation. In case of **independent** this means that it is checked that there are no loop-carried dependences. In case of **ivdep** it is checked that there are no backward loop-carried dependences:

$$\frac{\begin{array}{c} s = a\ \underline{\textbf{for}}(\textbf{var}\, n = e_1;\, n\, op\, e_2;\, n \mathrel{+}= e_3)\,\{ss'\} \\ \vdash \langle \sigma, n + e_3 \rangle \Downarrow \langle \sigma', v_3, (\mathscr{R}', \mathscr{W}') \rangle \\ \vdash \langle \sigma'[n \mapsto v_3], n\, op\, e_2 \rangle \Downarrow \langle \sigma'', \textbf{false}, (\mathscr{R}'', \mathscr{W}'') \rangle \\ \mathscr{R}^* = \mathscr{R} \cup \mathscr{R}' \cup \mathscr{R}'' \qquad \mathscr{W}^* = \mathscr{W} \cup \mathscr{W}' \cup \mathscr{W}'' \cup \{n\} \\ \mathscr{C}^+ = \mathtt{add}(\mathscr{C}, (\mathscr{R}', \mathscr{W}' \cup \{n\})) \\ (a = \textbf{independent}\, r) \Rightarrow \mathtt{indep}(\mathscr{L}, \mathscr{C}^+, \{n\} \cup Var(r)) \\ (a = \textbf{ivdep}) \Rightarrow \mathtt{back\_indep}(\mathscr{L}, \mathscr{C}^+, \{n\}) \\ \mathscr{L}^* = \mathtt{pop}(\mathscr{L}) \qquad \mathscr{C}^* = \mathtt{add}(\mathtt{pop}(\mathscr{C}^+), (\mathscr{R}'', \mathscr{W}'')) \end{array}}{\begin{array}{l} \vdash \langle \sigma, s;\, ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \\ \qquad \to \langle \sigma'', \textbf{clear\_scope};\, ss, (\mathscr{R}^*, \mathscr{W}^*), (\mathscr{L}^*, \mathscr{C}^*) \rangle \end{array}}\ \textsc{UFor-False}$$

As variables may be read or written during evaluation of the expressions e_3 and e_2, the read and write sets and the stack for the current nested loop iterations are updated appropriately. Similarly for the loop counter n,as it is being written. As the loop is exited, the top elements of $\mathscr{L}$ and $\mathscr{C}$ are removed.

Evaluation of the for-loop fails in case updating the loop counter fails:

$$\frac{\begin{array}{c} s = a\ \underline{\textbf{for}}(\textbf{var}\, n = e_1;\, n\, op\, e_2;\, n \mathrel{+}= e_3)\,\{ss'\} \\ \vdash \langle \sigma, n + e_3 \rangle \Downarrow error \end{array}}{\vdash \langle \sigma, s;\, ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \to error}\ \textsc{UFor-Err-3}$$

Similarly, evaluation fails in case evaluation of the loop condition fails:

$$\frac{\begin{array}{c} s = a\ \underline{\mathbf{for}}(\mathbf{var}\, n = e_1;\, n\, op\, e_2;\, n \mathrel{+}= e_3)\ \{ss'\} \\ \vdash \langle \sigma, n + e_3 \rangle \Downarrow \langle \sigma', v_3, (\mathscr{R}', \mathscr{W}') \rangle \qquad \vdash \langle \sigma'[n \mapsto v_3], n\, op\, e_2 \rangle \Downarrow \mathit{error} \end{array}}{\vdash \langle \sigma, s;\, ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \rightarrow \mathit{error}}\ \textsc{UFor-Err-2}$$

If the loop is annotated with **independent** and a loop-carried dependence is detected, we also transition to the error state:

$$\frac{\begin{array}{c} s = a\ \underline{\mathbf{for}}(\mathbf{var}\, n = e_1;\, n\, op\, e_2;\, n \mathrel{+}= e_3)\ \{ss'\} \\ \vdash \langle \sigma, n + e_3 \rangle \Downarrow \langle \sigma', v_3, (\mathscr{R}', \mathscr{W}') \rangle \\ \mathscr{C}^+ = \mathtt{add}(\mathscr{C}, (\mathscr{R}', \mathscr{W}' \cup \{n\})) \\ (a = \mathbf{independent}\, r) \wedge \neg \mathtt{indep}(\mathscr{L}, \mathscr{C}^+, \{n\} \cup \mathit{Var}(r)) \end{array}}{\vdash \langle \sigma, s;\, ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \rightarrow \mathit{error}}\ \textsc{UFor-Indep-Err}$$

Similarly, if the loop is annotated with **ivdep** and a backward loop-carried dependence is detected, we transition to the error state:

$$\frac{\begin{array}{c} s = a\ \underline{\mathbf{for}}(\mathbf{var}\, n = e_1;\, n\, op\, e_2;\, n \mathrel{+}= e_3)\ \{ss'\} \\ \vdash \langle \sigma, n + e_3 \rangle \Downarrow \langle \sigma', v_3, (\mathscr{R}', \mathscr{W}') \rangle \\ \mathscr{C}^+ = \mathtt{add}(\mathscr{C}, (\mathscr{R}', \mathscr{W}' \cup \{n\})) \\ (a = \mathbf{ivdep}) \wedge \neg \mathtt{back\_indep}(\mathscr{L}, \mathscr{C}^+, \{n\}) \end{array}}{\vdash \langle \sigma, s;\, ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \rightarrow \mathit{error}}\ \textsc{UFor-Ivdep-Err}$$

3.3.9 Break and Continue Statements

Recall that **break**-statements exit from the innermost loop and that **continue**-statements continue with the next iteration of the innermost loop.

- In case a **break**- or **continue**-statement is encountered, all remaining statements of the innermost loop are ignored, i.e., statements are thrown away until a **<u>for</u>**-statement is encountered:

$$\frac{s \text{ is not a } \underline{\mathbf{for}}\text{-statement} \qquad s' \in \{\mathbf{break}, \mathbf{continue}\}}{\begin{array}{l} \vdash \langle \sigma, s';\, s;\, ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \\ \qquad\qquad \rightarrow \langle \sigma, s';\, ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \end{array}}\ \textsc{Break-Continue}$$

- In case a **break**-statement is encountered and the first statement in the continuation is a **<u>for</u>**-statement, the loop is exited. That is, the **<u>for</u>**-statement is removed from the continuation and the **independent** and **ivdep** annotations are checked. Moreover, the scope is cleared from n any variables from the loop body.

$$\frac{\begin{array}{c} s = a\ \underline{\mathbf{for}}(\mathbf{var}\, n = e_1;\, n\, op\, e_2;\, n \mathrel{+}= e_3)\ \{ss'\} \\ (a = \mathbf{independent}\, r) \Rightarrow \mathtt{indep}(\mathscr{L}, \mathscr{C}, \{n\} \cup \mathit{Var}(r)) \\ (a = \mathbf{ivdep}) \Rightarrow \mathtt{back\_indep}(\mathscr{L}, \mathscr{C}, \{n\}) \\ \mathscr{C}^* = \mathtt{pop}(\mathscr{C}) \qquad \mathscr{L}^* = \mathtt{pop}(\mathscr{L}) \end{array}}{\begin{array}{l} \vdash \langle \sigma, \mathbf{break};\, s;\, ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \\ \qquad\qquad \rightarrow \langle \sigma, \mathbf{clear\_scope};\, ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}^*, \mathscr{C}^*) \rangle \end{array}}\ \textsc{Break-For}$$

As the loop is exited, the top elements of $\mathscr{L}$ and $\mathscr{C}$ are removed.

If the loop is annotated with **independent** and a loop-carried dependence is detected, we transition to the error state:

$$\frac{\begin{array}{c} s = a\ \underline{\textbf{for}}(\textbf{var}\, n = e_1;\, n\, op\, e_2;\, n \mathrel{+}= e_3)\ \{ss'\} \\ (a = \textbf{independent}\, r) \wedge \neg\texttt{indep}(\mathscr{L}, \mathscr{C}, \{n\} \cup Var(r)) \end{array}}{\vdash \langle \sigma, \textbf{break}; s; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \to error}\ \textsc{Break-For-Indep-Err}$$

Similarly, if the loop is annotated with **ivdep** and a backward loop-carried dependence is detected, we transition to the error state:

$$\frac{\begin{array}{c} s = a\,\underline{\textbf{for}}(\textbf{var}\, n = e_1;\, n\, op\, e_2;\, n \mathrel{+}= e_3)\ \{ss'\} \\ (a = \textbf{ivdep}) \wedge \neg\texttt{back\_indep}(\mathscr{L}, \mathscr{C}, \{n\}) \end{array}}{\vdash \langle \sigma, \textbf{break}; s; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \to error}\ \textsc{Break-For-Ivdep-Err}$$

- In case a **continue**-statement is encountered and the first statement in the continuation is a **for**-statement, the next loop iteration is started. The annotations do not need to be checked in this case; this is taken care of by the **for**-rules. The scope is also cleared any variables from the loop body.

$$\frac{s = a\ \underline{\textbf{for}}(\textbf{var}\, n = e_1;\, n\, op\, e_2;\, n \mathrel{+}= e_3)\ \{ss'\}}{\begin{array}{l} \vdash \langle \sigma, \textbf{continue}; s; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \\ \qquad \to \langle \sigma, \textbf{clear\_scope}; s; ss, (\mathscr{R}, \mathscr{W}), (\mathscr{L}, \mathscr{C}) \rangle \end{array}}\ \textsc{Continue-For}$$

4 Towards Static Verification of PENCIL Programs

Parallelising compilers can detect loops that can be executed in parallel. However, this detection is not perfect, and therefore the PENCIL language defines pragmas to declare that loops are parallel. If a loop is declared parallel then the compiler is allowed to assume that it is indeed parallel.

This chapter addresses the problem of how to verify that loops that are declared parallel can indeed safely be implemented as parallel loops. The solution is to add specifications to the PENCIL program, that when verified guarantee that the program can be parallelised without changing its meaning. In addition, these specifications can express function properties and requirements. For example, we can require all elements of an array to be positive and/or prove that an array remains unmodified.

In order to simplify the presentation in this chapter, we limit ourselves to single loops. This is not a real restriction and at the end of the chapter, we will explain how our approach can be extended to nested loops. Below, we present some background information, and then we introduce the specification language for parallel loops. Next, we sketch how we can implement automated verification of the specifications. Finally, we conclude with future work.

4.1 Background

This section briefly discusses some background on the theory of loop dependences and separation logic.

4.1.1 Loop Dependences

For a single loop with multiple statements, several types of loop dependences can be identified. There exists a *dependence* from statement S_{src} to statement S_{sink} in the body of a loop if there exist two iterations i and j of that loop, such that:

- Iteration i is before iteration j, *i.e.*, $i \leq j$.
- If the iterations are the same ($i = j$) then S_{src} must syntactically occur before S_{sink}.
- Statement S_{src} on iteration i and statement S_{sink} on iteration j access the same memory location.
- At least one of these accesses is a write.

The *distance of a dependence* is defined as the difference between j and i.

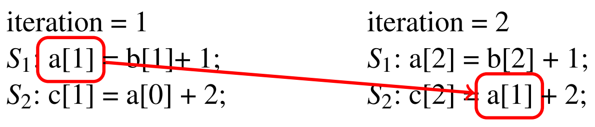
Loop dependences with distance 0, *i.e.*, when $i = j$, are called *loop independent dependences*. These dependences only have to be considered when the loop body has to be transformed, which is out of the scope of this deliverable. In this case the loop iterations may be executed in any arbitrary order.

Loop dependences with a positive distance are called *loop-carried dependences* and are classified into forward and backward dependences. When S_{src} syntactically appears before S_{sink}

(or if they are the same statement) there is a *forward loop-carried dependence* and when S_{sink} syntactically appears before S_{src} there is a *backward loop-carried dependence*. The following examples illustrate forward and backward loop-carried dependences.

Example 4.1.1 (Forward Loop Dependence)

```
for(int i=0;i<N;i++){
    S1: a[i] = b[i] + 1;
    S2: c[i] = a[i−1] + 2;
}
```

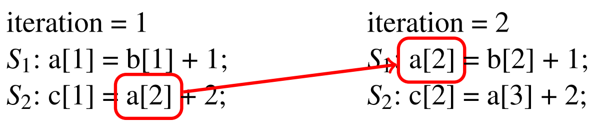


```
iteration = 1              iteration = 2
S1: a[1] = b[1] + 1;       S1: a[2] = b[2] + 1;
S2: c[1] = a[0] + 2;       S2: c[2] = a[1] + 2;
```

Here, S_1 is the source of the dependence and S_2 is the sink. The i^{th} element of the array a is shared between iteration i and $i-1$, as visualised by the first and second iteration (on the right).

Example 4.1.2 (Backward Loop Dependence)

```
for(int i=0;i<N;i++){
    S1: a[i] = b[i] + 1;
    S2: c[i] = a[i+1] + 2;
}
```



```
iteration = 1              iteration = 2
S1: a[1] = b[1] + 1;       S1: a[2] = b[2] + 1;
S2: c[1] = a[2] + 2;       S2: c[2] = a[3] + 2;
```

Here, the sink of the dependence (S_2) appears before the source (S_1) in the body of the loop. Therefore this is a backward loop-carried dependence.

The distinction between forward and backward dependences is important. Independent parallel execution of a loop with dependences is always unsafe, because it may change the result. However, only a loop with forward dependences can be vectorised, which is a special form of parallelisation, in which the sequential loop is transformed into sequential loop with less iterations, where every iteration of the new loop executes several iterations of the original loop in lock-step.

Detecting dependences in code is a difficult task. Even under the assumption that index expressions are linear[1], the task ends up being equivalent to solving a system of Diophantine equations, which is NP-complete [6]. Moreover, if the index expressions are more complicated then the problem can easily end up being undecidable (use an undecidable problem as the index expression) or simply unknown. For example, consider the loop

```
for(int i=0;i<N;i++)
{
    A[t[i]]++;
}
```

Without knowing anything about t, we cannot decide whether there are loop dependences. For safety, a compiler will typically assume there are dependences. However, sometimes the programmer has extra information, for example it might be known that all values in the array t are different, which would imply that there are no dependences at all in the program.

[1]All array index expressions are of the form A[a∗i+b], where a and b are constants and i is the loop variable.

To support a programmer to make this knowledge explicit, PENCIL provides pragmas to annotate loops that have forward dependences only and loops that have no dependences at all. However, as these annotations are written by the programmer, they might be erroneous. Therefore, we will provide a verification technique for these pragmas. Our verification technique will be based on separation logic, which is briefly described next.

4.1.2 Separation Logic.

Separation logic is described in detail in Chapter 5 of Deliverable D6.2 as a way to reason about OpenCL kernels. However, for the sake of completeness, we give a brief introduction here.

Separation logic [22] was originally developed as an extension of Hoare logic [9] to reason about programs with pointers, as it allows to reason explicitly about the heap. In classical Hoare logic, assertions are properties over the state and no distinction between variables on the heap on variables on the stack can be made, while in separation logic, the state is explicitly divided in the heap and a store related to the stack frame of the current method call. Separation logic is also suited to reason modularly about concurrent programs [18]: two threads that operate on disjoint parts of the heap do not interfere, and thus can be verified in isolation.

However, classical separation logic requires use of mutual exclusion mechanisms for all shared locations, and it forbids simultaneous reads to shared locations. To overcome this, Bornat et al. [4] extended separation logic with fractional permissions. Permissions, originally introduced by Boyland [5], denote access rights to a shared location. A full permission 1 denotes a write permission, whereas any fraction in the interval $(0,1)$ denotes a read permission. Permissions can be split and combined, thus a write permission can be split into multiple read permissions, and all of the read permissions can be joined into a write permission. In this way, data race freedom of programs using different synchronisation mechanisms can be proven. The set of permissions that a thread holds are often known as its *resources*.

We write access permissions as $\textbf{perm}(e,\pi)$, where e is an expression denoting a memory location and π is a fraction. To specify the behaviour of PENCIL programs that we are interested in for this deliverable, it is sufficient to only distinguish between read and write permissions, and the actually fraction of a read permission is irrelevant.

In separation logic there are two conjunction operators: *boolean conjunction* (&&) and *separating conjunction* ($**$). The latter is resource sensitive, the former is not. For example

$\textbf{perm}(\mathrm{x},\ \pi)\ \&\&\ \textbf{perm}(\mathrm{x},\ \pi) \equiv \textbf{perm}(\mathrm{x},\ \pi)$
$\textbf{perm}(\mathrm{x},\ \pi) ** \textbf{perm}(\mathrm{x},\ \pi) \equiv \textbf{perm}(\mathrm{x},\ 2\cdot\pi)$

To specify properties of the value stored at the location we just reference the location in our formulas. Thus, we are forced to check that every expression is *self-framed*, i.e., we need to check that only locations are accessed for which we have access permissions. This is different from traditional separation logic, which uses the PointsTo primitive that has an additional argument to denote the value stored at the location and cannot refer to the location otherwise. However, it has been proven that both logics are equivalent [20].

4.2 A Specification Language for PENCIL

Since PENCIL is an add-on to C, the starting point for the specification language is separation logic for C. This logic is known [25], therefore we will focus only on the extension needed

```
for(i=0;i<N;i+=1)
/*@
    requires perm(a[i],1) ** perm(c[i],1) ** perm(b[i],1/2);
    ensures perm(a[i],1) ** perm(c[i],1) ** perm(b[i],1/2);
@*/
{
        S1: a[i] = b[i] + 1;
        S2: c[i] = a[i] + 2;
}
```

Listing 1: Specification of independent loops

for PENCIL, *i.e.*, the ability to specify parallel loops. Finally, we will explain how to draw conclusions about the validity of PENCIL pragmas from the validity of the loop specifications.

4.2.1 Specification of Parallel Loops

The traditional way of specifying the effect of a loop is by means of an invariant that has to hold before and after the execution of each iteration in the loop. This works well for sequential loops, but offers no insight into possible parallel execution. Instead we are going to take the point of view that every iteration of the loop is executed in parallel. To be able to handle dependences, we can specify restrictions on how the execution of the statements in the iterations is scheduled. From this viewpoint, it is natural that each iteration is specified by its own contract that we call its *iteration contract*. In the iteration contract, the pre-condition specifies the permissions and resources that a particular iteration requires for execution and the post-condition specifies the permissions and resources which are released after the execution of an iteration. Listing 1 gives an example of an independent loop, specified by its iteration contract. The contract requires that at the start of iteration i, permission to write both a[i] and c[i] is available, as well as permission to read b[i]. The contract also ensures that these permissions are returned at the end of iteration i. The iteration contract implicitly requires that the separating conjunction of all iteration pre-conditions holds before the the first iteration start and that the separating conjunction of all iteration post-conditions holds after the last iteration has completed, before the program continues with the statement following the loop. In the example, the loop iterates from 0 to $N-1$, so the contract implies that before the loop, permission to write the first N elements of both a and c must be available, as well as permission to read the first N elements of b. The same permissions are ensured to be available after the loop has ended.

For specification of independent loops, the notion of iteration contract is sufficient to capture all legal execution orders. To specify dependent loops, we need the ability to specify what happens when due to a dependence the computations have the synchronise. During such a synchronisation, permissions should be transferred from the iteration containing the source of a dependence to the iteration containing the sink of that dependence. We also need the ability to state properties of the memory for which permissions are transferred, in order to prove the validity of the invariants needed for proving validity of the specification. To specify permission transfer we introduce the **send** keyword:

//@ **send** ϕ **to** L, d;

```
for(int i=0;i<N;i++)
/*@
    requires perm(a[i],1) ** perm(c[i],1) ** perm(b[i],1/2);
    ensures perm(a[i],1/2) ** perm(c[i],1) ** perm(b[i],1/2);
    ensures i>0 ==> perm(a[i−1],1/2);
    ensures i==N−1 ==> perm(a[i],1/2);
@*/
{
        S1: a[i] = b[i] +1;
        //@ send perm(a[i],1/2) to S2,1;
        //% receive perm(a[i−1],1/2) from S1,−1;
        S2: c[i] = (i?a[i−1]:0)+2;
}
```

Listing 2: Specification of a Forward Loop-Carried Dependence

This specifies a transfer of the permissions and properties denoted by the separation logic formula ϕ to the statement labelled L in the iteration $i+d$, where i is the current iteration.

Below, we will give two examples that illustrate how loops are specified with **send** clauses. The **send** clause alone completely specifies both how permissions are provided and used by the iterations. However, to make the specifications more readable, we also mark the place where the permission are used with a corresponding **receive** statement.

Listing 2 gives the annotated version of the program in Example 4.1.1, illustrating forward dependence. Iteration i starts with full permission on a[i] and c[i] and read permission on b[i]. The first statement is a write to a[i], which needs write permission. Except for the first iteration, the second statement reads a[i−1], which is not allowed unless read permission is available. Hence a **send** annotation is specified after the first assignment that transfers a read permission on a[i] to the next iteration (and in addition, keeps the other half of the permission itself). The post-condition of the iteration contract reflects this, as this ensures that the original permissions on b[i] and c[i] are released, as well as the read permission on a[i], which was not sent. Every iteration, except the first, receives a read permission on a[i−1]. The post-condition of the iteration contract also specifies that this read permission is released. Finally, since the last iteration cannot transfer a read permission on a[i], the iteration contract's post-condition also specifies that the last iteration returns this non-transferred read permission on a[i].

Listing 3 shows the annotated program for Example 4.1.2, illustrating a backward dependence.

The specifications in both listings are valid. Hence every execution order of the loop bodies that respects the order implied by the **send** annotations yields the same result as sequential execution. In the case of the forward dependence example, vectorised execution of the loop is such an order. For the backward dependence example, only sequential execution obeys the ordering requirement.

```
for( i=0;i<N;i++)
/*@
    requires i==0 ==> perm(a[0],1/2);
    requires perm(a[i],1/2) ** perm(a[i+1],1/2) ** perm(c[i],1) ** perm(b[i],1/2);
    ensures perm(a[i],1) ** perm(c[i],1) ** perm(b[i],1/2);
    ensures i==N−1 ==> perm(a[i+1],1/2);
@*/
{
        //% receive perm(a[i],1/2) from S2, −1;
        S1: a[i] = b[i] +1;
        S2: c[i] = a[i+1]+2;
        //@ send perm(a[i+1],1/2) to S1, 1;
}
```

Listing 3: Specification of a Backward Loop-Carried Dependence

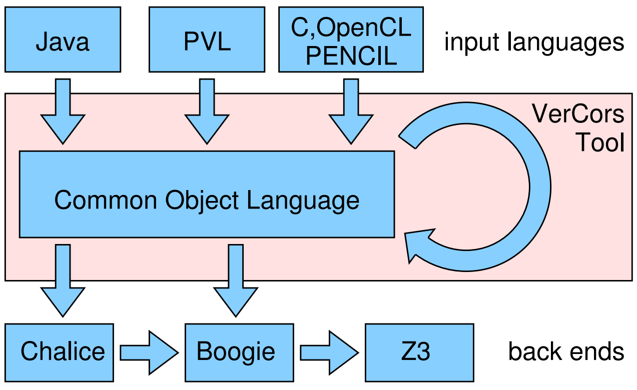


Figure 4.1: VerCors tool architecture.

4.3 Verification of Parallel Loops by means of Encoding

This section explains how we will implement automated verification of the specifications explained above in the VerCors tool set.

4.3.1 The VerCors Tool Set

The VerCors tool set implements thread-modular static verification of concurrent programs, annotated with functional properties and heap access permissions. The tool supports both generic multithreaded and vector-based programming models. In particular, it can verify multithreaded programs written in Java, specified with JML, extended with separation logic. It can also verify parallelisable programs written in a toy language that supports the characteristic features of OpenCL.

Rather than building yet another verifier, the VerCors tool set leverages existing verifiers. That is, it is designed as a compiler that translates specified programs to a simpler language. These simplified programs are then verified by a third-party verifier. If there are errors then the error messages are converted to refer to the original input code.

Figure 4.1 shows the overall architecture of the tool. Its main input language is Java. For prototyping, we use the toy language PVL, which is a very simple object-oriented language that has been extended to express GPU kernels too. Moreover, it has a built-in specification language that allows us to work on full functional verification of programs, including kernels. The C language family front-end is work-in-progress, but will support OpenCL and PENCIL in the near future. We mainly use Chalice [15], a verifier for an idealised concurrent programming language, as our back-end, but for sequential programs we also use the intermediate program verification language Boogie [2].

The main concept behind the verification of programs that have been specified with the VerCors tool set is that those parts of the program whose verification is not directly supported by the simpler language of the underlying verifier, are encoded as specified programs in the simpler programming language in such a way that this results in the same verification conditions. For example, let us assume for a moment that our back-end does not support any kind of loops and consider the following variant of the Hoare Logic rule for loops:

$$\frac{\{I \wedge b\}S\{I\}}{\{I\}\textbf{while}(b)\ \textbf{invariant}\ I\{S\}\{I \wedge \neg b\}}\ \textsc{Loop}$$

This rule states that for the program to be correct, the invariant I must be proven to hold just before the loop, and after the loop it can be assumed that both the invariant and the negation of the loop condition b can be assumed to be valid. In addition, the loop body must preserve the invariant: it must hold that when the loop body is executed in a state where the invariant and the loop condition hold, its execution ends in a state where the invariant holds.

If the underlying verifier does not support loops, a program as

S_{pre}; **while**(b) **invariant** I { S } ; S_{post} ;

can be verified by encoding the loop as a method call to loop_main:

S_{pre}; loop_main(); S_{post};

where loop_main is a method specified as follows:

```
/*@
    requires I;
    ensures I ∧ ¬b;
@*/
void loop_main();
```

This ensures that the pre- and post-condition of the loop are correctly required and assumed.

Moreover, preservation of the loop invariant by each iteration of the loop body is encoded by verifying the following specified method:

```
/*@
    requires I ∧ b;
    ensures I;
@*/
void loop_iteration(){
    S;
}
```

```
void program(){
    $S_{\text{pre}}$;
    for(int i=0;i<N;i++)
    /*@
        requires pre(i);
        ensures post(i);
    @*/
    {
        $S$;
    }
    $S_{\text{post}}$;
}
```

Listing 4: Loop with an iteration contract.

The verification conditions for this method correspond precisely to the conditions that must be checked for the loop body. As a result, if the transformed program can be verified with existing tools then the original code satisfies its specifications too. Note that while the transformation preserves logical correctness, the transformed code is no longer executable and hence unsuitable for verification techniques other than static verification.

The following sections describe how we use similar encoding processes to encode the different types of parallel loops.

4.3.2 Independent Parallel Loops

First we explain how to transform a specified program with independent parallel loops into a specified program without parallel loops, in such a way that if the transformed program satisfies its specification then the original program also satisfies its specification.

Listing 4 shows the pattern of a program with a parallel loop that has been specified with an iteration contract. Proving that this program is correct requires that following proof obligations are discharged:

- After S_{pre}, the separating conjunction of all of the pre-conditions holds.
- The loop body S satisfies the iteration contract.
- The statement S_{post} can be proven correct, assuming that the separating conjunction of the post-conditions holds.

We may encode these obligations as procedures (see Listing 5) by the following steps:

1. We replace the loop in the program with a call to a procedure loop_main, whose arguments are the free variables occurring in the loop. The contract of this procedure requires the separating conjunction of all pre-conditions and ensures the separating conjunction of all post-conditions. After this replacement, we can verify the program (host_program) with existing tools to discharge the first and last proof obligations.
2. To discharge the remaining proof obligation, we generate a procedure loop_body, whose arguments are the loop variable i plus the same arguments as loop_main. The contract of

```
/*@
    requires 0 <= i && i < N;
    requires pre(i);
    ensures post(i);
@*/
loop_body(int i,int N,free(S)))
{
    S;
}

/*@
    requires (\forall* int i;0 <= i && i < N;pre(i));
    ensures (\forall* int i;0 <= i && i < N;post(i));
@*/
loop_main(int N,free(S)));

void program(){
    S_pre;
    loop_main(N,free(S));
    S_post;
}
```

Listing 5: Encoded proof obligations for a loop with an iteration contract.

this procedure is the iteration contract of the loop body, preceded by a requirement that states that the value of the iteration variable is within the bounds of the loop.

4.3.3 Vectorisable Parallel Loops

The previous section dealt with independent loops, this section extends the verification concept to include loop dependences in general and forward loop dependences in particular. The latter are important because loops with forward dependences can be vectorised, which makes it legal to add the **ivdep** pragma in PENCIL.

In our specification language, dependences are indicated by **send** annotations, as explained above. For this section we consider **send** annotations, as they might occur in the loop body of our example (listing 4). Those annotations would have the form

//@ **send** $\phi(i)$ **to** L, $d(i)$;

If the destination label L is either the label of the **send** instruction or of a later statement then the loop has a forward dependence. Otherwise the loop has a backward dependence. If a loop specification can be proven correct then in any case sequential execution of the loop is correct. If in addition all dependences are forward dependences then vectorised execution will be correct too.

Verification of the **send** instruction is performed by replacing the **send** annotation with a procedure call send_phi(i); and by inserting a procedure call recv_phi(i); at the location of the

```
/*@
      requires $d(i) > i$;
      requires is_iteration($d(i)$) ==> $\phi(i)$;
@*/
void send_phi(int i);

/*@
      ensures is_iteration($d^{-1}(i)$) ==> $\phi(d^{-1}(i))$;
@*/
void recv_phi(int i);
```

Listing 6: Contracts for the encoding of dependences.

label L. The contracts of these methods encode the transfer of the resources specified by $\phi(i)$ from the sending iteration to the receiving iteration, subject to two conditions:

1. Permissions can only be transferred to future iterations ($d(i) > i$).
2. Transfer only happens if both the sending and the receiving iterations exist.

Listing 6 shows contract for the methods send_phi and recv_phi. These contracts assume that the inverse of the function d is expressible as a formula. The existence of iteration i is expressed with the boolean function is_iteration(i), whose definition is derived from the loop bounds. For example,

```
for(int i=0;i<N;i++)
```

gives rise to

```
boolean is_iteration(int i){return 0 <= i < N;}
```

Note that in the setting of polyhedral compilers, the functions d can be restricted to linear functions, which are easily inverted.

Also note that the requirement $d(i) > i$ only is necessary to ensure that the sequential execution of the loop yields the correct result. With and without this restriction, all parallel executions that are scheduled in such a way that the schedule obeys all ordering requirements imposed by the **send** statement are correct. Hence the techniques described in this chapter are not only applicable to the PENCIL language, but also to different parallel paradigms, such as GPU kernels with barriers.

We have shown how to verify parallel loops, even in the presence of dependences from one loop iteration to the next. We conjecture that if verification of a loop is possible without using **send** then it is correct to tag the loop as independent, where correct means that the semantics of the loop will never be error. Moreover, if **send** is used with labels occurring after the statement then it is correct to use the **ivdep** tag. In the next section, we explain how to deal with another kind of dependences: reductions.

4.4 Reductions

A reduction is an operation that aggregates data from a large data structure into a small data structure. One of the simplest examples is computing the sum of all of the elements of an array.

```
int sum;
float A[N];
  ...
for(int i=0;i<N;i++)
/*@
   requires perm(A[i],1/2);
   reduces sum, +, A[i];
   ensures perm(A[i],1/2) ** A[i]==\old(A[i]);
@*/
{
      sum += A[i];
}
```

Listing 7: Specified parallel summation.

Listing 7 shows an implementation of summing the elements of an array, complete with loop specifications. Its iteration contract uses a new keyword: **reduces**, which takes three arguments. The first argument is the name of the output variable. The second argument is the operation that is used for the reduction. The third argument is an expression that signals the contribution of this iteration to the output. This argument can be omitted if the resulting value is irrelevant. Every pair of name and operation pair in a PENCIL reduction pragma would correspond to a separate **reduces** clause in our iteration contracts.

To verify that the summation can be run as a reduction, we will verify by means of a syntactic check that the way in which the reduction variable is used is safe with respect to the operation. In our example the only use of sum is as the left-hand side of a += operator, which is permissible. Initially, this will be the only use of the reduction variable that we allow, but other patterns may be added as needed. The other proof obligations for a reducing parallel loop are checked by encoding them as method contracts (see Listing 8).

In addition to the clauses already described in the previous sections, the host code contract requires write permission for the reduction variable, and it ensures write permission on the reduction variable and the fact that it has been increased by the sum of all of the elements of the array.

The contract of the iteration method requires write permission on sum, which is also required to be 0 and ensures write permission on sum, plus the fact that afterwards it is equal to the contribution of the iteration. Note that if we omit the syntactic check, verification of the two generated contracts is not sufficient to show that the original parallel reducing loop satisfies its specification. For example, if we replace sum+=A[i] by

```
    float tmp=sum; tmp = tmp + A[i]; sum=tmp;
```

then the generated methods would all verify, but parallelised execution could yield many different results because the atomic execution of the increment of sum that is guaranteed for sum=...+ by the **reduces** tag, is not guaranteed.

```
/*@
   requires perm(sum,1);
   requires (\forall* int i; 0 <= i < N ; perm(A[i],1/2));
   ensures (\forall* int i; 0 <= i < N ; perm(A[i],1/2) ** A[i]==\old(A[i]));
   ensures perm(sum,1) ** sum==\old(sum)+(\sum int i; 0 <= i < N ; A[i]);
@*/
loop_main(float A[N],float sum,int N);

/*@
  requires 0 <= i < N ;
  requires perm(sum,1) ** sum==0;
  requires perm(A[i],1/2);
  ensures perm(A[i],1/2) ** A[i]==\old(A[i]);
  ensures sum==A[i];
@*/
loop_iteration(int i,float A[N],float sum,int N){
   sum += A[i];
}
```

Listing 8: Encoded Proof Obligations for summation.

4.5 Future Work

In this chapter, we have explained how PENCIL programs may be specified and how those specification can be checked. Going forward in the project, we will both develop the theoretical foundations for the specifications as well as implementing the verification as part of the VerCors tool set. Besides implementing a verifier for the specification language, we will also investigate how to automatically generate most and preferably all of the specifications, rather than requiring them to be provided with the code.

The theoretical foundation will be based on the semantics for PENCIL, described in chapter 3. It will also be based on a formal definition for our specification language. The semantics of that language will be based on a suitable variant of separation logic. The goal will be to prove the conjecture that for every program, whose specification has been proven correct and for every input that is valid according to that specification, the semantics of running the program on that input cannot result in an error. Or in other words, running a verified program on a legal input will not encounter any error condition.

Another goal for the verification tool is to minimise the amount of specifications that have to be handwritten. The ideal result would be that given a PENCIL program that is annotated with just pragmas, the tool will be able to fully automatically derive enough other specifications to prove that the PENCIL pragmas are justified. Complete functional specifications are much harder to derive, but we will investigate if it is possible to infer the iteration contracts from the procedure contracts of the method in which they occur.

Finally, in this chapter we have explained our approach assuming that loops are not nested. Of course in many cases loops are nested, but this need not be a problem. The PENCIL language allows only a limited form of nested loops. It allows the innermost loop to have dependences, wrapped around those can be several parallel loops and wrapped around those can be several non-parallel loops. Our approach for loops with dependences as explained, works for the innermost loop. This innermost loop can then be considered as yet another statement. Next, we have several independent loops, which are no different from a single independent loop. Thus, the techniques sketched in the chapter can reduce the problem of specifying and verifying nested parallel loops to the problem of specifying and verifying a C program. The remaining non-parallel outer loops are no different from normal programs, and the techniques for dealing with them are known.

5 OpenMP

As PENCIL has not seen any use yet outside the CARP consortium, a pertinent question is whether the ideas from the previous chapters also apply to languages related to PENCIL. In particular, we made a preliminary investigation to see whether our ideas also apply to OpenMP.

OpenMP was selected due to its similarities with PENCIL: like PENCIL, OpenMP extends C with an number of pragmas that enable running code on parallel hardware. Moreover, among other languages taking this approach (such as OpenACC and OpenHMPP), OpenMP seems to be the language that has gotten most traction.

5.1 OpenMP and its Relationship to PENCIL

The central pragma in OpenMP is the one for the *parallel for-loop*. Consider, for example, the following piece of OpenMP code:

```
#pragma omp parallel for shared(A, B, C)
for (int i = 0; i < n; ++i) {
  C[i] = A[i] + B[i];
}
```

Ignoring the pragma, it is clear that this code adds two vectors, `A` and `B`, of length `n` and stores the result in `C`. Declaring the loop as being an OpenMP parallel for-loop tells the OpenMP compiler that the iterations of this loop may be executed in parallel. This means that a number of tasks will be created—one for each loop iteration—and that these tasks will be executed by a number of threads in parallel until no task remains. Declaring the vectors as shared means that only one copy of each of the vectors exists (no separate copy is created within the context of each thread).

In the above example, executing the loop iterations in parallel yields the same result as in the sequential case: the vector elements accessed in each loop iteration are different. Unfortunately, one can write loops for which this it is not the case. Consider, for example, the following loop:

```
#pragma omp parallel for shared(A, B, C)
for (int i = 0; i < n; ++i) {
  C = A[i] + B[i];
}
```

Executing this loop sequentially would yield `C = A[n - 1] + B[n - 1]` upon loop termination. However, parallelising the loop by creating a separate task for each loop iteration does *not* guarantee the resulting value of `C` to be the same as in the sequential case: the loop iterations may be executed in an arbitrary order and the $(n-1)$th iteration is not necessarily the last iteration to be completed.

The difference between the above two loops may be explained by their loop-carried dependences: the first loop does not have any such dependences, whilst the second loop has a loop-carried dependence on `C`. In fact, to establish whether the loop iterations of a loop can correctly be executed in parallel, it suffices to check that the loop is free of any loop-carried dependences. In this respect, there is a similarity between the problem of establishing whether we can (a) annotate a for-loop with the OpenMP parallel-for pragma and (b) annotate a loop with

PENCIL's independent pragma: in both cases we need to show that there are *no* loop-carried dependences.

As an aside, observe that the OpenMP parallel for-loop is also susceptible to the OpenCL verification methods from CARP Deliverable D6.2: once we have parallelised a loop, the problem of checking whether we were 'legally' allowed to perform the parallelisation boils down to checking that the parallelised version is free of data-races. Obviously, this is not the case in the second example from above: the threads will race on `C`.

5.2 Related Work

The existing body of literature on the verification of OpenMP programs is rather small. However, both runtime verification and static verification approaches can be found in the literature.

In the case of runtime verification, the parallelised OpenMP program is executed several times with one or more test vectors. During the runs certain data are collected which allow one to determine off-line whether any data-races occurred during the runs. A disadvantage of runtime verification is that data-races may not be exhibited on certain hardware or for certain test vectors. Hence, data-race freedom of programs is not guaranteed and these techniques can mostly considered to be bug finding techniques.

In the case of static verification, one tries to prove that the OpenMP program is free from data-races or loop-carried dependences. A disadvantage of the techniques in this category is that they are prone to false-positives: a data-race might be reported where none exists due to certain abstractions being used in the methods.

Runtime Verification Each of the runtime verification methods specifically targeted towards OpenMP uses variation of Lamport vector clocks [14], which means that data-races are detected by means of a happens-before analysis. A series of papers by Jun et al. investigates the scalability [11, 19] and quality of the errors [10] reported by this approach. The Intel Thread Checker is a tool based on this approach [21, 23], an evaluation of the tool occurs in [12]. A comparison between the Intel Thread Checker and the Sun Thread Analyzer occurs in [24]. Both tools are compared with the work of Jun et al. in [7].

In recent work, Jun et al. [8, 17, 13] combine happens-before analysis with lockset analysis to achieve more scalable data race detection (without the false positives that usually occur in lockset analysis).

Static Verification Lin [16] presents a static non-concurrency analysis. This analysis determines which code fragments can never be executed in parallel (i.e. are non-concurrent). The result of the analysis is used to find potential data races in fragments that are supposed to be executed in parallel. The static analysis component of the Sun OpenMP compiler makes use of this analysis. The reported method for race detection potentially yields many false positives. Static non-concurrency analysis is also used by Yu et al. [26] to detect data races. Their data race analysis employs extended finite-state machines and is reported to be more accurate.

Basupalli et al. [3] verify that so-called Affine Control Loops (ACLs) with OpenMP directives satisfy the requirements of the given directives. A Polyhedral Reduced Dependency Graph of the loop(-nest) is built and it is checked that the graph does not have any dependences that are in violation of the given OpenMP directives.

5.3 Technical Challenges

Recall from Section 5.1 that there are at least two approaches to establishing statically whether we can rightfully annotate a loop with a parallel-for pragma: we can try to establish either (a) that the loop does not have any loop carried dependences or (b) that the compiled code is free of data races. We describe some technical challenges associated with both approaches.

In the case of (a), note that the verification requires information about the original source code: we need to know which loops are annotated. From a technical perspective this means that we need to have access to a parser that is able to parse OpenMP pragmas. In fact we would need to have access to two such parsers, as OpenMP both extends C and FORTRAN. Currently, the only freely available, industrial strength parsers that are able to parse OpenMP are the C and FORTRAN parsers from the GNU Compiler Collection (GCC). This fits badly with our current tool-chains, which are mostly Clang/LLVM-based[1].

In the case of (b), the situation is slightly better with respect to parser availability: Although, we would still need to depend on the GCC parsers, the DragonEgg plugin for GCC[2] can be exploited to compile OpenMP code to LLVM bitcode. Hence, this approach provides a better fit with our current tool-chains. Also with this approach, it is easy to detect which loops were annotated with parallel-for pragmas: these loops will be outlined as separate procedures. Unfortunately, the outlined procedures will not only contain the loop-body, but will contain the actual loop: The procedure represents the code that will be executed by a single thread and, as there may be many more loop-iterations than threads, a loop is still required in the outlined procedure. Separating the loop-body from the rest of the loop is harder at bitcode level than at source level due to the unstructured nature of the bit code.

5.4 Conclusion

As the notion of a loop-carried dependence is important in both PENCIL and OpenMP, we expect that the verification techniques developed for PENCIL will also be applicable to OpenMP (and vice versa). Adaptation of the techniques should be straightforward, however, as explained, there are some technological challenges associated with this.

[1] OpenMP support for Clang/LLVM is currently under development: `http://clang-omp.github.io/`.
[2] `http://dragonegg.llvm.org/`

6 Future Work and Open Problems

In the remaining period of the project, we plan to work on the following:

- Formally establish soundness of the static verification technique w.r.t. the operational semantics. Additionally, we will prove that if static verification succeeds, the formal semantics will never contain the error value.
- Develop tool support for PENCIL verification. We will encode verification of the loop iteration specifications that capture the loop annotations as part of the VerCors tool set [1]. Additionally, we will ensure that the sequential verification techniques of functional properties provided by the VerCors tool set are fully compatible with the PENCIL semantics, and if necessary, adaptations are made.
- Develop support to compile PENCIL specifications to OpenCL specifications. In particular, if a PENCIL program can be verified, then the OpenCL program produced by the (baseline) PENCIL compiler should be verifiable with the compiled specifications.
- We will investigate in more detail how the specification techniques for parallelisable loops can be applied to OpenMP programs as well.

Bibliography

[1] A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. The vercors project: setting up basecamp. In K. Claessen and N. Swamy, editors, *PLPV*, pages 71–82. ACM, 2012.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[3] V. Basupalli, T. Yuki, S. V. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott. `ompVerify`: Polyhedral analysis for the OpenMP programmer. In *Proceedings of the 7th International Workshop on OpenMP (IWOMP 2011)*, volume 6665 of *Lecture Notes in Computer Science*, pages 37–53. Springer, 2011.

[4] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *POPL*, pages 259–270. ACM, 2005.

[5] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

[6] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In D. S. Wise, editor, *PLDI*, pages 15–29. ACM, 1991.

[7] O.-K. Ha, Y.-J. Kim, M.-H. Kang, and Y.-K. Jun. Empirical comparison of race detection tools for OpenMP programs. In *Proceedings of the International Conference on Grid and Distributed Computing (GDC 2009)*, volume 63 of *Communications in Computer and Information Science*, pages 108–116. Springer, 2009.

[8] O.-K. Ha, I.-B. Kuh, G. M. Tchamgoue, and Y.-K. Jun. On-the-fly detection of data races in OpenMP programs. In *Proceedings of the 10th Workshop on Parallel and Distributed Systems (PADTAD 2012)*, pages 1–10. ACM, 2012.

[9] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[10] M.-H. Kang, O.-K. Ha, S.-W. Jun, and Y.-K. Jun. A tool for detecting first races in OpenMP programs. In *Proceedings of the 10th International Conference on Parallel Computing Technologies (PaCT 2009)*, volume 5698 of *Lecture Notes in Computer Science*, pages 299–303. Springer, 2009.

[11] Y.-J. Kim, M.-H. Kang, O.-K. Ha, and Y.-K. Jun. Efficient race verification for debugging programs with OpenMP directives. In *Proceedings of the 9th International Conference on Parallel Computing Technologies (PaCT 2007)*, volume 4671 of *Lecture Notes in Computer Science*, pages 230–239. Springer, 2007.

[12] Y.-J. Kim, D. Kim, and Y.-K. Jun. An empirical analysis of Intel Thread Checker for detecting races in OpenMP programs. In *Proceedings of the 7th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2008)*, pages 409–414. IEEE Computer Society, 2008.

[13] I.-B. Kuh, O.-K. Ha, and Y.-K. Jun. Tracing logical concurrency for dynamic race dectection in OpenMP programs. In *Proceedings of the 1st International Conference on Software Technology (SoftTech 2012)*, volume 5 of *Advanced Science and Technology Letters*, pages 222–224. SERSC, 2012.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[15] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *FOSAD*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer, 2009.

[16] Y. Lin. Static nonconcurrency analysis of OpenMP programs. In *Proceedings of the International Workshops on OpenMP (IWOMP 2005 and 2006)*, volume 4315 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2008.

[17] Y. Meng, O.-K. Ha, and Y.-K. Jun. Dynamic instrumentation for nested fork-join parallelism in OpenMP programs. In *Proceedings of the 4th International Conference on Future Generation Information Technology (FGIT 2012)*, volume 7709 of *Lecture Notes in Computer Science*, pages 154–158. Springer, 2012.

[18] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

[19] S.-H. Park, M.-Y. Park, and Y.-K. Jun. A comparison of scalable labeling schemes for detecting races in OpenMP programs. In *Proceedings of the International Workshop on OpenMP Applications and Tools (WOMPAT 2001)*, volume 2104 of *Lecture Notes in Computer Science*, pages 68–80. Springer, 2001.

[20] M. Parkinson and A. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3:01):1–54, 2012.

[21] P. Petersen and S. Shah. OpenMP support in the Intel Thread Checker. In *Proceedings of International Workshop on OpenMP Applications and Tools (WOMPAT 2003)*, volume 2716 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2003.

[22] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[23] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the Intel Thread Checker race detector. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID 2006)*, pages 34–41. ACM, 2006.

[24] C. Terboven. Comparing Intel Thread Checker and Sun Thread Analyzer. In *Proceedings of the International Conference of Parallel Computing (ParCo 2007)*, volume 38 of *Publication Series of the John von Neumann Institute for Computing*, pages 669–676. John von Neumann Institute for Computing, 2007.

[25] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 97–108. ACM, 2007.

[26] F. Yu, S.-C. Yang, F. Wang, G.-C. Chen, and C.-C. Chan. Symbolic consistency checking of OpenMP parallel programs. In *Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2012 (LCTES'12)*, pages 139–148. ACM, 2012.