Equivalence of Linear, Free, Liberal, Structured Program Schemas is Decidable in Polynomial Time.

Sebastian Danicic [a] Mark Harman [d] Rob Hierons [b]
John Howroyd [c] Michael R.Laurence [a]

[a] *Department of Computing Sciences, Goldsmiths College, University of London, New Cross, London SE14 6NW, UK.*

[b] *Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex, UB8 3PH, UK.*

[c] *@UK PLC 5 Jupiter House Calleva Park Aldermaston Berks RG7 8NN*

[d] *Department of Computer Science, King's College London, Strand, London WC2R 2LS, United Kingdom*

Abstract

A program schema defines a class of programs, all of which have identical statement structure, but whose expressions may differ. We define a class of syntactic *similarity* binary relations between linear structured schemas, which characterise schema equivalence for structured schemas that are linear, free and liberal. In this paper we report that similarity implies equivalence for linear schemas, and that a near-converse holds for schemas that are linear, free and liberal. We also show that the similarity of two linear schemas is polynomial-time decidable. Our main result considerably extends the class of program schemas for which equivalence is known to be decidable, and suggests that linearity is a constraint worthy of further investigation.

Key words: structured program schemas, conservative schemas, liberal schemas, free schemas, linear schemas, schema equivalence, static analysis, program slicing

1 Introduction

A program schema represents the statement structure of a program by replacing real functions and predicates with function and predicate symbols taken from sets $\mathcal{F}$ and $\mathcal{P}$ respectively. A schema S thus defines a whole class $[S]$ of programs all of the same structure. Each program in $[S]$ can be obtained from S via a mapping called an *interpretation* which gives meanings to the function and predicate symbols in S. As an example, Figure 1 gives a schema S; and the program P of Figure 2 is in the class $[S]$.

The primary application of the theory of program schemas was as a framework for investigating program transformations; in particular those used by compilers during optimisation. If it could be proved that a certain transformation on schemas preserved equivalence, then this transformation could certainly be safely applied to programs. Surveys on the theory of program schemas can be found in the works of Greibach [1] and Manna [2].

```
u := h();
if p(w)    then  v := f(u);
           else  v := g();
```

Fig. 1. Schema S

```
u := 1;
if w > 1   then  v := u + 1;
           else  v := 2;
```

Fig. 2. Program P

This paper gives a class of schemas for which equivalence is decidable. Equivalence is defined as follows. Given any variable v in a variable set $\mathcal{V}$, we say that schemas S, T are *v-equivalent*[1], written $S \cong_v T$, if given any interpretation and an initial state (that is, a mapping from the set of variables into some fixed domain) the programs defined by S and T give the same final value to the variable v, provided they both terminate. We also define $S \cong_\omega T$ to mean that given any interpretation and any initial state, the programs defined by S and T either both terminate or both fail to terminate. Thus the schema T of Figure 3 satisfies $S \cong_v T$, with S as in Figure 1; but $S \cong_\omega T$ does not

[1] For the class of *all* schemas the relation $\cong_v$ is not transitive, as an example in Section 3 shows, but it *is* an equivalence relation for the class of free, structured schemas (Proposition 19).

```
while q(v)  do    v := k(v);
if p(w)     then
                  {
                   u := h();
                   v := f(u);
                  }
            else  v := g();
```

Fig. 3. Schema T

hold. The relation $\cong_V$ for $V \subseteq \mathcal{V} \cup \{\omega\}$ means the conjunction of the relations $\cong_u$ for all $u \in V$. We write $\cong$ ('equivalence') to mean $\cong_{\mathcal{V}\cup\{\omega\}}$. Some researchers use the phrase 'functional equivalence' to refer to the relation $\cong_{\mathcal{V}\cup\{\omega\}}$ and 'weak equivalence' for $\cong_{\mathcal{V}}$.

This definition of equivalence takes no account of relations between the symbols, or requirements that a function or predicate symbol have a certain meaning, although definitions of equivalence for which interpretations are defined in this more restricted way have been considered [3–5].

Traditionally schemas were defined using a set of labelled statements or equivalently a flow diagram. All new results stated in this paper only concern *structured* schemas,[2] in which *goto* statements are forbidden, and predicate symbols are only used to build if statements, of the form *if* $q(\mathbf{u})$ *then* T_1 *else* T_2, or while statements, of the form *while* $p(\mathbf{u})$ *do* T; where in both cases $\mathbf{u}$ is a finite tuple of variables.

It has been shown that it is decidable whether two structured schemas which are Conservative, Free and Linear are equivalent [6]. The main result reported in this paper is a strengthening of this result; that it can be decided in polynomial time whether two structured schemas which are Liberal, Free and Linear (abbreviated LFL in this paper), are equivalent.

The full statement of our main theorem involves the definition of a binary relation $simil_V$ on linear schemas for $V \subseteq \mathcal{V} \cup \{\omega\}$. We report in this paper that $S\, simil_u\, T \Rightarrow S \cong_u T$ holds for linear schemas S, T and $u \in \mathcal{V} \cup \{\omega\}$. There is a near-converse; $S \cong_{\{v,\omega\}} T \Rightarrow S\, simil_{\{v,\omega\}}\, T$ holds for every $v \in \mathcal{V}$ and LFL schemas S, T. The proofs of both these results are given in the Technical Report [7, Theorem 148], on account of their length. Since it can be decided in polynomial time whether $S\, simil_u\, T$ holds (Theorem 31), our main theorem follows.

[2] Some authors, for example Manna [2] use the phrase *while* schema for what we call a *structured* schema (except that Manna allows statements like *while* $\neg p(\mathbf{u})$ *do* T); in this paper a *while* schema means a structured schema consisting of a while loop (Definition 3).

1.1 Organisation of the paper

In Section 2 we give some background to the theory of schemas. In Section 3 we give the basic schema definitions. We also give the formal definitions of free and liberal schemas, and prove that variable equivalence is in fact an equivalence relation for the class of free schemas. We also prove that it is decidable whether a schema is both free and liberal. We then define syntactic relations between the symbols in a linear schema which are required in the statement of the definition of similarity of linear schemas. We then give this definition, and prove that it is decidable in polynomial time whether two linear schemas are u-similar, given any $u \in \mathcal{V} \cup \{\omega\}$. In Section 4 we give the definition of the *slice* of a schema, given by deleting statements from a schema, and discuss conditions under which slicing preserves equivalence. In Section 5 we give the main theorem and discuss further possibilities for research.

2 Background to schema theory

2.1 Different classes of schemas

Many subclasses of schemas have been defined:

Linear schemas (Definition 4) in which each function and predicate symbol occurs at most once. [3]

Conservative schemas, in which every assignment is of the form $v := f(v_1, \ldots, v_r)$ where $v \in \{v_1, \ldots, v_r\}$.

Free schemas, (Definition 17) where all paths are executable under some interpretation.

Liberal schemas (Definition 17) in which two assignments along any executable path can always be made to assign distinct values to their respective variables.

The last three of these classes were first introduced by Paterson [8]. Of these conditions, the first two can clearly be decided for the class of all schemas. Paterson [8] also proved, using a reduction from the Post Correspondence Problem, that it is not decidable whether a schema is free. He also showed however that it *is* decidable whether a schema is both liberal and free; and since he also gave an algorithm for transforming a schema S into a schema T such that T is both liberal and free if and only if S is liberal, it is clearly decidable whether a schema is liberal. It is an open problem whether freeness is decidable for the class of linear schemas.

[3] Some authors use the phrase 'non-repeating schemas' to refer to what we call linear schemas.

All results on the decidability of equivalence of schemas are either negative or confined to very restrictive classes of schemas. In particular Paterson [8] proved, in effect, that equivalence is undecidable for the class of all (unstructured) schemas. He proved this by showing that the halting problem for Turing machines (which is, of course, undecidable) is reducible to the equivalence problem for the class of all schemas. Ashcroft and Manna showed [9] that an arbitrary schema can be effectively transformed into an equivalent structured schema, provided that statements such as $\mathit{while}\, \neg p(\mathbf{u})\, \mathit{do}\, T$ are permitted; hence Paterson's result shows that any class of schemas for which equivalence can be decided must not contain this class of schemas. Thus in order to get positive results on this problem, it is clearly necessary to define the relevant classes of schema with great care.

Although the class of linear structured schemas considered in this paper is a highly restrictive one, it has the merit that schemas in this class are the main objects studied in the field of Program Slicing (which is discussed in Section 2.3), and that this is therefore a particularly important class.

2.2 Positive results on the decidability of schema equivalence

Besides the result of [6] mentioned above, positive results on the decidability of equivalence of schemas include the following; in an early result in schema theory, Ianov [10] introduced a restrictive class of schemas, the Ianov schemas, for which equivalence is decidable. Ianov schemas are monadic (that is, they contain only a *single* variable) and all function symbols are unary; hence Ianov schemas are conservative.

Paterson [8] proved that equivalence is decidable for a class of schemas called *progressive schemas*, in which every assignment references the variable assigned by the previous assignment along every legal path.

Sabelfeld [11] proved that equivalence is decidable for another class of schemas called *through schemas.* A through schema satisfies two conditions: firstly, that on every path from an accessible predicate p to a predicate q which does not pass through another predicate, and every variable x referenced by p, there is a variable referenced by q which defines a term containing the term defined by x, and secondly, distinct variables referenced by a predicate define distinct terms under any Herbrand interpretation (Definition 9).

2.3 Relevance of schema theory to program slicing

Our interest in the theory of program schemas is motivated in part by applications in program slicing. Slicing has many applications including program comprehension [12], software maintenance [13], [14], [15], [16], debugging [17], [18], [19], [20], testing [21],

[22], [23], re-engineering [24], [25], component reuse [26], [27], program integration [28], and software metrics [29], [30], [31]. There are several surveys of slicing techniques, applications and variations [32], [33], [34]. All applications of slicing rely on the fact that a slice is faithful to a projection of the original program's semantics, yet it is typically a smaller program.

The field of (static) program slicing is largely concerned with the design of algorithms which given a program and a variable v, eliminate as much code as possible from the program, such that the program (slice) consisting of the remaining code, when executed from the same initial state, will still give the same final value for v as the original program, and preserve termination. One algorithm is thus better than another if it constructs a smaller slice.

Slicing algorithms do not normally take account of the meanings of the functions and predicates occurring in a program, nor do they 'know' when the same function or predicate occurs in more than one place in a program. In effect, therefore, they work with a linear schema defined by the program, and the semantic properties which slices of programs are required to preserve are defined in terms of schema semantics. This motivates the study of schemas, which represent large classes of programs.

Weiser [35] showed that given a program and a variable v, there was a particular set of functions and predicates (corresponding to our set $\mathcal{N}_S(v)$ for schemas in Definition 29) which may affect the final value of v; the symbols not lying in this set may simply be deleted without affecting the final value of v. In Theorem 33 we generalise this by considering ω-equivalence as a slicing criterion. In [36] it was shown that if S is LFL then none of the symbols in $\mathcal{N}_S(u)$ (for $u \in \mathcal{V} \cup \{\omega\}$) can be deleted from S without giving a u-inequivalent schema. This is however false for the class of schemas which are merely linear and free; a counterexample is given in Figure 6 in Section 5.1.

3 Basic definitions

Definition 1 (symbol sets) Throughout this paper, $\mathcal{F}$, $\mathcal{P}$ and $\mathcal{V}$ denote fixed infinite sets of *function symbols*, of *predicate symbols* and of *variables* respectively. We assume a function

$$\mathit{arity} : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}.$$

The arity of a symbol x is the number of arguments referenced by x. We assume that for each $n \in \mathbb{N}$ there are infinitely many elements of $\mathcal{F}$ and $\mathcal{P}$ of arity n, so we never run out of symbols of any required arity. Note that in the case when the arity of a function symbol g is zero, g may be thought of as a constant.

Definition 2 (terms) The set $\mathit{Term}(\mathcal{F}, \mathcal{V})$ of *terms* is defined as follows:

- each variable is a term,

- if $f \in \mathcal{F}$ is of arity n and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term.

If each term t_i is a variable, then $f(t_1, \ldots, t_n)$ is called a function expression.
We refer to a tuple $\mathbf{t} = (t_1, \ldots, t_n)$, where each t_i is a term, as a vector term. We call $p(\mathbf{t})$ a predicate term if $p \in \mathcal{P}$ and the number of components of the vector term $\mathbf{t}$ is $\mathit{arity}(p)$. If each component of $\mathbf{t}$ is a variable, then $p(\mathbf{t})$ is called a predicate expression.

Definition 3 (structured schemas) We define the set $\mathit{Sch}(\mathcal{F}, \mathcal{P}, \mathcal{V})$ of all *structured schemas* recursively as follows. The empty schema $\Lambda \in \mathit{Sch}(\mathcal{F}, \mathcal{P}, \mathcal{V})$. An assignment $y := f(\mathbf{x})$; where $y \in \mathcal{V}$, and $f(\mathbf{x})$ is a function expression, lies in $\mathit{Sch}(\mathcal{F}, \mathcal{P}, \mathcal{V})$. From these all schemas in $\mathit{Sch}(\mathcal{F}, \mathcal{P}, \mathcal{V})$ may be 'built up' from the following constructs on schemas.

- Sequences; $S' = U_1 U_2 \ldots U_r \in \mathit{Sch}(\mathcal{F}, \mathcal{P}, \mathcal{V})$ provided that each schema

$$U_1, \ldots, U_r \in \mathit{Sch}(\mathcal{F}, \mathcal{P}, \mathcal{V}).$$

  We define $S\Lambda = \Lambda S = S$ for all schemas S.
- *If* schemas; $S'' =$ *if* $p(\mathbf{x})$ *then* $\{T_1\}$ *else* $\{T_2\}$ lies in $\mathit{Sch}(\mathcal{F}, \mathcal{P}, \mathcal{V})$ whenever $p(\mathbf{x})$ is a predicate expression and $T_1, T_2 \in \mathit{Sch}(\mathcal{F}, \mathcal{P}, \mathcal{V})$.
- *While* schemas; $S''' =$ *while* $q(\mathbf{y})$ *do* $\{T\}$ lies in $\mathit{Sch}(\mathcal{F}, \mathcal{P}, \mathcal{V})$ whenever $q(\mathbf{y})$ is a predicate expression and T is a schema.

The predicate symbols p and q are called the *guards* of the schemas S'' and S''', respectively.
Finally, $|S|$ will denote the total number of function and predicate symbols in S, with n distinct occurrences of the same symbol counting n times.

Thus a schema is a word in a language over an infinite alphabet, for which Λ is the empty word. We normally omit the braces $\{$ and $\}$ if this causes no ambiguity. Also, we may write *if* $p(\mathbf{x})$ *then* $\{T_1\}$ instead of
if $p(\mathbf{x})$ *then* $\{T_1\}$ *else* $\{T_2\}$ if $T_2 = \Lambda$.

Observe that $f(\mathbf{x})$ and $p(\mathbf{x})$ in Definition 3 are always function and predicate *expressions*; that is, the components of the vector term $\mathbf{x}$ are variables.

For the remainder of this paper, the word 'schema' is intended to mean 'structured schema'.

The sets of *if* and *while* predicate symbols occurring in a schema S are denoted by $\mathit{ifPreds}(S)$ and $\mathit{whilePreds}(S)$; their union is $\mathit{Preds}(S)$. We define $\mathit{Funcs}(S) \subseteq \mathcal{F}$ to be the set of function symbols in S and define $\mathit{Symbols}(S) = \mathit{Funcs}(S) \cup \mathit{Preds}(S)$. A schema without predicates is called *predicate-free*; a schema without while predicates is called *while-free.*

Definition 4 (linear schemas) If no element of $\mathcal{F} \cup \mathcal{P}$ appears more than once in a schema S, then S is said to be *linear*. If a linear schema S contains an assignment

$y := f(\mathbf{x})$; then we define $\mathit{assign}_S(f) = y$ and $\mathbf{refvec}_S(f) = \mathbf{x}$. If $p \in \mathit{Preds}(S)$ then $\mathbf{refvec}_S(p)$ is defined similarly.

3.1 Paths through a schema

The execution of a program defines a (possibly infinite) sequence of assignments and predicates. Each such sequence will correspond to a *path* through the associated schema. The set $\Pi^\omega(S)$ of paths through S is now given.

Definition 5 (the set $\mathit{alphabet}(S)$ and the set $\Pi^\omega(S)$ of paths through S) If σ is a word, or a set of words over an alphabet, then $\mathit{pre}(\sigma)$ is the set of all prefixes of (elements of) σ. If L is any set, then we write L^* for the set of finite words over L and L^ω for the set containing both finite and infinite words over L, and we write Λ to refer to the empty word; recall that Λ is also a particular schema.
For each schema S the alphabet of S, written $\mathit{alphabet}(S)$ is defined by

$$\mathit{alphabet}(S) = A \cup B$$

where

$$A = \{y := f(\mathbf{x}) \mid\ y := f(\mathbf{x});\ \text{is an assignment in } S\},$$
$$B = \{<p(\mathbf{x}) = Z> \mid\ p(\mathbf{x}) \text{ is a predicate expression in } S,\ Z \in \{\mathsf{T}, \mathsf{F}\}\}.$$

For any letter $l \in \mathit{alphabet}(S)$, we define $\mathit{symbol}(l) \in \mathit{Symbols}(S)$ to be f if l is an assignment with function symbol f, and p if l is $<p(\mathbf{x}) = Z>$ for $Z \in \{\mathsf{T}, \mathsf{F}\}$. The words in $\Pi(S) \subseteq (\mathit{alphabet}(S))^*$ are formed by concatenation from the words of subschemas as follows:

For Λ,

$$\Pi(\Lambda) = \{\Lambda\}.$$

For assignments,

$$\Pi(y := f(\mathbf{x});) = \{y := f(\mathbf{x})\}.$$

For sequences, $\Pi(S_1 S_2 \ldots S_r) = \Pi(S_1) \ldots \Pi(S_r)$.

For *if* schemas, $\Pi(\ \mathit{if}\ p(\mathbf{x})\ \mathit{then}\ \{T_1\}\ \mathit{else}\ \{T_2\})$ is the set of all concatenations of $<p(\mathbf{x}) = \mathsf{T}>$ with a word in $\Pi(T_1)$ and all concatenations of $<p(\mathbf{x}) = \mathsf{F}>$ with a word in $\Pi(T_2)$.

For *while* schemas, $\Pi(\ \mathit{while}\ q(\mathbf{y})\ \mathit{do}\ \{T\})$ is the set of all words of the form

$$[<q(\mathbf{y}) = \mathsf{T}>\ \Pi(T)]^*\ <q(\mathbf{y}) = \mathsf{F}>$$

where $[<q(\mathbf{y}) = \mathsf{T}> \Pi(T)]^*$ denotes a finite sequence of words which are the concatenation of $\lessdot q(\mathbf{y}) = \mathsf{T}>$ with a word from $\Pi(T)$.

We define the set $\Pi^\omega(S)$ of *paths* through S as

$$\Pi^\omega(S) = \Pi(S) \cup \{\sigma \in (\mathit{alphabet}(S))^\omega - (\mathit{alphabet}(S))^* |\, \mathit{pre}(\sigma) - \{\sigma\} \subseteq \mathit{pre}(\Pi(S))\}.$$

When referring to a linear schema S, we will sometimes omit the reference to $\mathbf{refvec}_S(p)$ for $p \in \mathit{Preds}(S)$ when denoting elements of $\mathit{alphabet}(S)$; that is, we will write $\lessdot p = Z>$ to refer to $\lessdot p(\mathbf{x}) = Z>$. Since the schema S is linear, this is unambiguous.

Lemma 6 *Let S be a schema.*

(1) *If $\sigma \in \mathit{pre}(\Pi(S))$, the set $\{l \in \mathit{alphabet}(S) |\, \sigma l \in \mathit{pre}(\Pi(S))\}$ is one of the following; the empty set, a singleton containing an assignment, or a pair $\{\lessdot p(\mathbf{x}) = \mathsf{T}>, \lessdot p(\mathbf{x}) = \mathsf{F}>\}$ where $p \in \mathit{Preds}(S)$.*
(2) *An element of $\Pi(S)$ cannot be a strict prefix of another.*

Proof. Both assertions follow by induction on $|S|$. □

Lemma 6 reflects the fact that at any point in the execution of a program, there is never more than one 'next step' which may be taken.

Definition 7 (paths *passing through* a symbol) We say that a path $\sigma \in \Pi^\omega(S)$ *passes* through a function symbol f (or a predicate p) if it contains an assignment with function symbol f (or $\lessdot p(\mathbf{x}) = Z>$ for $Z \in \{\mathsf{T}, \mathsf{F}\}$). We may strengthen this by saying that σ passes through an element $l \in \mathit{alphabet}(S)$ if l occurs in σ.

Definition 8 (segments of a schema and of segments) Let S be a schema and let $\mu \in \mathit{alphabet}(S)^*$. We say that μ is a
segment (in S) if there are words μ_1, μ_2 such that $\mu_1\mu\mu_2 \in \Pi(S)$. If μ, σ are segments in S, then we say that μ is a segment of σ in S if we can write $\sigma = \mu_1\mu\mu_2$.
We say that a segment μ starts (ends) at $x \in \mathit{Symbols}(S)$ if $\tilde{x} \in \mathit{alphabet}(S)$ is the first (last) letter of μ, with $x = \mathit{symbol}(\tilde{x})$.

3.2 Semantics of structured schemas

The symbols upon which schemas are built are given meaning by defining the notions of a state and of an interpretation. It will be assumed that 'values' are given in a single set D, which will be called the *domain*. We are mainly interested in the case in which $D = \mathit{Term}(\mathcal{F}, \mathcal{V})$ (the Herbrand domain) and the function symbols represent the 'natural' functions with respect to $\mathit{Term}(\mathcal{F}, \mathcal{V})$.

Definition 9 (states, (Herbrand) interpretations and the natural state e) Given a domain D, a *state* is either $\perp$ (in the case of non-terminating programs) or a function $\mathcal{V} \to D$. The set of all such states will be denoted by $\mathrm{State}(\mathcal{V}, D)$. An interpretation i defines, for each function symbol $f \in \mathcal{F}$ of arity n, a function $f^i : D^n \to D$, and for each predicate symbol $p \in \mathcal{P}$ of arity m, a function $p^i : D^m \to \{\mathsf{T}, \mathsf{F}\}$. The set of all interpretations with domain D will be denoted $\mathit{Int}(\mathcal{F}, \mathcal{P}, D)$. When the domain used is $\mathit{Term}(\mathcal{F}, \mathcal{V})$, an interpretation i is said to be *Herbrand* if the functions $f^i : \mathit{Term}(\mathcal{F}, \mathcal{V}) \to \mathit{Term}(\mathcal{F}, \mathcal{V})$ for each $f \in \mathcal{F}$ are defined as

$$f^i(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$$

for all n-tuples of terms $(t_1, \ldots, t_n)$.
In the case when the domain is $\mathit{Term}(\mathcal{F}, \mathcal{V})$, the *natural state* $e : \mathcal{V} \to \mathit{Term}(\mathcal{F}, \mathcal{V})$ is defined by $e(v) = v$ for all $v \in \mathcal{V}$.

Note that an interpretation i being Herbrand places no restriction on the mappings $p^i : (\mathit{Term}(\mathcal{F}, \mathcal{V}))^m \to \{\mathsf{T}, \mathsf{F}\}$ defined by i for each $p \in \mathcal{P}$.
It is well known [2, Section 4-14] that Herbrand interpretations, on the domain of terms, are the only ones that need to be considered when considering equivalence of schemas. This fact is stated more precisely in Theorem 16.

A program is obtained from a schema S and an interpretation i by replacing all symbols $f \in \mathcal{F}$ and $p \in \mathcal{P}$ in S by f^i and p^i; and given an initial state $d \in \mathrm{State}(\mathcal{V}, D)$, this program defines a final state

$$\mathcal{M}[\![S]\!]^i_d \in \mathrm{State}(\mathcal{V}, D)$$

in the obvious way, which will be given formally in Definition 13. (If the program fails to terminate for an initial state d, or if $d = \perp$, then we define $\mathcal{M}[\![S]\!]^i_d = \perp$.)

Given a schema $S \in \mathit{Sch}(\mathcal{F}, \mathcal{P}, \mathcal{V})$ and a domain D, an initial state $d \in \mathrm{State}(\mathcal{V}, D)$ with $d \neq \perp$ and an interpretation $i \in \mathit{Int}(\mathcal{F}, \mathcal{P}, D)$ we now define the final state $\mathcal{M}[\![S]\!]^i_d \in \mathrm{State}(\mathcal{V}, D)$ and the associated path $\pi_S(i, d) \in \Pi^\omega(S)$.

Definition 10 (the schema $\mathit{schema}(\sigma)$) Given a word $\sigma \in (\mathit{alphabet}(S))^*$, the predicate-free schema $\mathit{schema}(\sigma)$ consists of all the assignments along σ in the same order as in σ; and $\mathit{schema}(\sigma) = \Lambda$ if σ has no assignments.

Definition 11 (semantics of predicate-free schemas) Given a state $d \neq \perp$, the final state $\mathcal{M}[\![S]\!]^i_d$ and associated path $\pi_S(i, d) \in \Pi^\omega(S)$ of a schema S are defined as follows:

For Λ,

$$\mathcal{M}[\![\Lambda]\!]^i_d = d$$

and

$$\pi_\Lambda(i,d) = \Lambda.$$

For assignments,

$$\mathcal{M}[\![y := f(\mathbf{x});]\!]^i_d(v) \quad = \quad \begin{cases} d(v) & \text{if } v \neq y, \\ f^i(d(\mathbf{x})) & \text{if } v = y \end{cases}$$

(where $d(x_1, \ldots, x_r)$ is defined to be the tuple $(d(x_1), \ldots, d(x_r))$)
and

$$\pi_{y := f(\mathbf{x});}(i,d) \quad = \quad y := f(\mathbf{x}),$$

and for sequences S_1S_2 of predicate-free schemas,

$$\mathcal{M}[\![S_1S_2]\!]^i_d \quad = \quad \mathcal{M}[\![S_2]\!]^i_{\mathcal{M}[\![S_1]\!]^i_d}$$

and

$$\pi_{S_1S_2}(i,d) \quad = \quad \pi_{S_1}(i,d)\pi_{S_2}(i, \mathcal{M}[\![S_1]\!]^i_d).$$

This uniquely defines $\mathcal{M}[\![S]\!]^i_d$ and $\pi_S(i,d)$ if S is predicate-free.

In order to give the semantics of a general schema S, first the path, $\pi_S(i,d)$, of S with respect to interpretation, i, and initial state d is defined.

Definition 12 (the path $\pi_S(i,d)$) Given a schema S, an interpretation i, and a state, $d \neq \bot$, the path $\pi_S(i,d) \in \Pi^\omega(S)$ is defined by the following condition; for all $\sigma \lessdot p(\mathbf{x}) = X \gtrdot \in \mathit{pre}(\pi_S(i,d))$, the equality $p^i(\mathcal{M}[\![\mathit{schema}(\sigma)]\!]^i_d(\mathbf{x})) = X$ holds.

In other words, the path $\pi_S(i,d)$ has the following property; if a predicate expression $p(\mathbf{x})$ along $\pi_S(i,d)$ is evaluated with respect to the predicate-free schema consisting of the sequence of assignments preceding that predicate in $\pi_S(i,d)$, then the value of the resulting predicate term given by i 'agrees' with the value given in $\pi_S(i,d)$.

By Lemma 6, this defines the path $\pi_S(i,d) \in \Pi^\omega(S)$ uniquely.

Definition 13 (semantics of arbitrary schemas) If $\pi_S(i,d)$ is finite, we define

$$\mathcal{M}[\![S]\!]^i_d = \mathcal{M}[\![\mathit{schema}(\pi_S(i,d))]\!]^i_d$$

(which is already defined, since $\mathit{schema}(\pi_S(i,d))$ is predicate-free) otherwise $\pi_S(i,d)$ is infinite and we define $\mathcal{M}[\![S]\!]^i_d = \bot$. In this last case we may say that $\mathcal{M}[\![S]\!]^i_d$ is not terminating. For convenience, if S is predicate-free and $d : \mathcal{V} \to \mathit{Term}(\mathcal{F}, \mathcal{V})$ is a state then we define unambiguously $\mathcal{M}[\![S]\!]_d = \mathcal{M}[\![S]\!]^i_d$. Also, for schemas S, T and interpretations i and j we write $\mathcal{M}[\![S]\!]^i_d(\omega) = \mathcal{M}[\![T]\!]^j_d(\omega)$ to mean $\mathcal{M}[\![S]\!]^i_d = \bot \iff \mathcal{M}[\![T]\!]^j_d = \bot$.

Observe that $\mathcal{M}[\![S_1S_2]\!]^i_d = \mathcal{M}[\![S_2]\!]^i_{\mathcal{M}[\![S_1]\!]^i_d}$ and

$$\pi_{S_1S_2}(i,d) = \pi_{S_1}(i,d)\pi_{S_2}(i, \mathcal{M}[\![S_1]\!]^i_d)$$

hold for all schemas (not just predicate-free ones).

Definition 14 (termination from the natural state e) If $\mathcal{M}[\![S]\!]^i_e \neq \perp$, then we say that i is a *terminating* interpretation for S.

Definition 15 (u-equivalence of schemas) Given any $u \in \mathcal{V} \cup \{\omega\}$, we say that schemas $S, T \in Sch(\mathcal{F}, \mathcal{P}, \mathcal{V})$ are *u-equivalent,* written $S \cong_u T$, if for every domain D and state $d : \mathcal{V} \to D$ and every $i \in Int(\mathcal{F}, \mathcal{P}, D)$, the following holds; either $u \in \mathcal{V} \wedge \perp \in \{\mathcal{M}[\![S]\!]^i_d, \mathcal{M}[\![T]\!]^i_d\}$, or

$$\mathcal{M}[\![S]\!]^i_d(u) = \mathcal{M}[\![T]\!]^i_d(u).$$

If $V \subseteq \mathcal{V} \cup \{\omega\}$, we write $S \cong_V T$ to mean $S \cong_u T \;\; \forall u \in V$ and we write $S \cong T$ to mean $S \cong_{\mathcal{V} \cup \{\omega\}} T$.

Theorem 16, which is a restatement of [2, Theorem 4-1], ensures that we may assume that D is always the Herbrand domain and $d = e$ in Definition 15; hence we only need to consider Herbrand interpretations.

Theorem 16 *Let Ω be a set of schemas in $Sch(\mathcal{F}, \mathcal{P}, \mathcal{V})$, let D be a domain, let $d \in D$ and let $i \in Int(\mathcal{F}, \mathcal{P}, D)$. Then there is a Herbrand interpretation j such that for all $S \in \Omega$, $\pi_S(j, e) = \pi_S(i, d)$ holds.*

Throughout the remainder of the paper, all interpretations will be assumed to be Herbrand.

3.3 Free and liberal schemas

Definition 17 Let $S \in Sch(\mathcal{F}, \mathcal{P}, \mathcal{V})$.

- If for every $\sigma \in pre(\Pi(S))$ there is a Herbrand interpretation i such that $\sigma \in pre(\pi_S(i, e))$, then S is said to be *free.*
- If for every prefix $\sigma = \mu <y := f(\mathbf{a})> \nu <z := g(\mathbf{b})> \in pre(\Pi(S))$ such that there is a Herbrand interpretation i such that $\sigma \in pre(\pi_S(i, e))$, we have

  $$\mathcal{M}[\![schema(\mu)]\!]_e(f(\mathbf{a})) \neq \mathcal{M}[\![schema(\mu <y := f(\mathbf{a})> \nu)]\!]_e(g(\mathbf{b})),$$

  then S is said to be *liberal.* (If $f \neq g$ then of course this condition is trivially satisfied.)

Thus a schema S is said to be free if for every path through S, there is a Herbrand interpretation which follows it with the natural state e as the initial state, and a schema S is said to be liberal if given any path through S passing through two assignments and a Herbrand interpretation which follows it with e as the initial state, the assignments give distinct values to the variables to which they assign.

Observe that if a schema S is free, and

$$\mu \lessdot p(\mathbf{x}) = X \gtrdot \mu' \lessdot p(\mathbf{y}) = \neg X \gtrdot \in \mathit{pre}(\pi_S(i,e))$$

for some Herbrand interpretation i, then

$$\mathcal{M}[\![\mathit{schema}(\mu)]\!]_e(\mathbf{x}) \neq \mathcal{M}[\![\mathit{schema}(\mu\mu')]\!]_e(\mathbf{y})$$

holds, since otherwise there would be no Herbrand interpretation whose path (for e) has the prefix $\mu \lessdot p(\mathbf{x}) = X \gtrdot \mu' \lessdot p(\mathbf{y}) = \neg X \gtrdot$. Thus a path through a free schema cannot pass twice (for initial state e) through the same predicate term.

As mentioned in the introduction, it was proved in [8] that it is decidable whether a schema is liberal, or liberal and free. Theorem 18 gives the essential result for linear schemas.

Theorem 18 (syntactic condition for being liberal and free)
Let S be a linear schema. Then S is both liberal and free if and only if for every segment $\tilde{x}\mu\tilde{y}$ in S with $\tilde{x},\tilde{y} \in \mathit{alphabet}(S)$, $\mathit{symbol}(\tilde{x}) = \mathit{symbol}(\tilde{y})$ and such that the same symbol does not occur more than once in $\tilde{x}\mu$ or $\mu\tilde{y}$, then the segment $\tilde{x}\mu$ contains an assignment to a variable referenced by $\tilde{y}$.
In particular, it is decidable whether a linear schema is both liberal and free.

Proof [8]. Assume that S is both liberal and free. Then for any segment $\tilde{x}\mu\tilde{y}$ satisfying the conditions given, there is a prefix Θ and an interpretation i such that $\Theta\tilde{x}\mu\tilde{y} \in \mathit{pre}(\pi_S(i,e))$, and distinct (predicate) terms are defined when $\tilde{x}$ and $\tilde{y}$ are reached, thus proving the condition.
To prove sufficiency, first observe that the 'non-repeating' condition on the letters of the segments $\tilde{x}\mu$ and $\mu\tilde{y}$ may be ignored, since segments that begin and end with letters having the same symbol can be removed from within $\tilde{x}\mu$ or $\mu\tilde{y}$ until it is satisfied. Consider the set of prefixes of $\Pi(S)$ of the form $\Theta\tilde{x}\mu\tilde{y}$ with $\mathit{symbol}(\tilde{x}) = \mathit{symbol}(\tilde{y})$ such that $\tilde{x}\mu\tilde{y}$ satisfies the condition given. By induction on the length of such prefixes, it can be shown that every assignment encountered along such a prefix defines a different term (for initial state e), and the result follows immediately from this.
Since there are finitely many segments in S which contain no repeated symbols except at the endpoints, and these can be enumerated, the decidability of liberality and freeness for the set of linear schemas follows easily. $\Box$

Theorem 18 can easily be generalised to apply to arbitrary unstructured schemas; we state it in restricted form in order to simplify the notation used.

Clearly the relation $\cong_\omega$ is an equivalence relation. For the relation $\cong_v$ with $v \in \mathcal{V}$ we have the following result.

Proposition 19 (transitivity of $\cong_v$ for free schemas) *Let $v \in \mathcal{V}$; then the relation $\cong_v$ is an equivalence relation when restricted to the class of free schemas.*

Proof. Only transitivity is at issue. Suppose $S' \cong_v S''$ and $S'' \cong_v S'''$ hold for free schemas S', S'', S'''. Let i be an interpretation and assume that

$$\bot \notin \{\mathcal{M}[\![S']\!]^i_e, \mathcal{M}[\![S''']\!]^i_e\}$$

holds. Let the interpretation j map every predicate term $p(\mathbf{t})$ to F unless $\pi_{S'}(i,e)$ or $\pi_{S'''}(i,e)$ passes through $p(\mathbf{t})$, in which case let $p^j(\mathbf{t}) = p^i(\mathbf{t})$. Thus $\mathcal{M}[\![S']\!]^i_e = \mathcal{M}[\![S']\!]^j_e$ and $\mathcal{M}[\![S''']\!]^i_e = \mathcal{M}[\![S''']\!]^j_e$ hold and j maps finitely many predicate terms to T, hence $\mathcal{M}[\![S'']\!]^j_e \neq \bot$ holds. Thus

$$\mathcal{M}[\![S']\!]^j_e(v) = \mathcal{M}[\![S'']\!]^j_e(v) = \mathcal{M}[\![S''']\!]^j_e(v)$$

holds, giving the result. □

Proposition 19 is false for the set of all linear schemas. To see this, consider the three linear schemas

```
S′ =  if p(u)  then  v := f₁();
               else  v := g();

S″ =  while p(u) do Λ;
      v := g();

S‴ =  if p(u)  then  v := f₂();
               else  v := g();
```

of which S'' is not free. Clearly $S' \cong_v S''$ and $S'' \cong_v S'''$ hold, but not $S' \cong_v S'''$.

We will henceforth refer to a schema which is liberal, free and linear as an LFL schema.

3.4 Subschemas of linear schemas

The *subschemas* of a schema are defined as follows; the empty sequence Λ is a subschema of every schema; if $S \in Sch(\mathcal{F}, \mathcal{P}, \mathcal{V})$ is an assignment or Λ then the only subschemas of S are S itself and Λ; the subschemas of the schema $U_1 \ldots U_r$ are those of each U_j for $1 \leq j \leq r$ and also the schemas $U_i U_{i+1} \ldots U_j$ for $i \leq j$; the subschemas of $S'' = \mathit{if}\, p(\mathbf{x})\ \mathit{then}\ \{T_1\}\ \mathit{else}\ \{T_2\}$ are S itself and those of T_1 and T_2; the subschemas of $S''' = \mathit{while}\, q(\mathbf{y})\ \mathit{do}\ \{T\}$ are S''' itself and those of T. The subschemas T_1 and T_2 of S'' are called the *true* and *false* parts of p (or of S''). In the *while* schema the subschema T is called the *body* of q (or of S''').

Definition 20 (the subschemas $S(p)$, $\mathit{part}^X_S(p)$ and $\mathit{body}_S(p)$) Let S be a linear schema. If $p \in \mathit{Preds}(S)$ then we sometimes write $S(p)$ for the while or if subschema of S of which p is the guard.

Also, if $p \in \mathit{ifPreds}(S)$ and $X \in \{\mathsf{T}, \mathsf{F}\}$ then we may write $\mathit{part}^X_S(p)$ for the X-part of p in S.
If $p \in \mathit{whilePreds}(S)$ then $\mathit{body}_S(p)$ is the body of p in S.

Definition 21 (the $\searrow_S$ 'lying below' relation, 'immediately below') Let S be a linear schema. If $p \in \mathit{Preds}(S)$, we write $p \searrow_S x$ to mean $x \in \mathit{Symbols}(\mathit{body}_S(p))$ if $p \in \mathit{whilePreds}(S)$ and $x \in \mathit{Symbols}(\mathit{part}^{\mathsf{T}}_S(p)) \cup \mathit{Symbols}(\mathit{part}^{\mathsf{F}}_S(p))$ if $p \in \mathit{ifPreds}(S)$. We may strengthen this to $p \searrow_S x\,(X)$ to mean that either $x \in \mathit{Symbols}(\mathit{part}^X_S(p))$ (if $p \in \mathit{ifPreds}(S)$), or $x \in \mathit{Symbols}(\mathit{body}_S(p))$ (if $X = \mathsf{T}$ and $p \in \mathit{whilePreds}(S)$).
Also, if $A \subseteq \mathit{Symbols}(S)$, then we say that A lies *immediately* below S (or equivalently, S lies immediately above A) if $A \subseteq \mathit{Symbols}(S)$ and there is no $p \in \mathit{whilePreds}(S)$ such that $A \subseteq \mathit{Symbols}(\mathit{body}_S(p))$. In this case, if $S = \mathit{body}_T(q)$ for some linear schema T and $q \in \mathit{whilePreds}(T)$, we may also say that A lies immediately below q in T.

Definition 22 (main subschemas of a linear schema) Let S be a linear schema. The set of *main* subschemas of S contains S itself and the bodies of all while subschemas of S.

Observe that there is exactly one main subschema of a linear schema S lying immediately above a set $A \subseteq \mathit{Symbols}(S)$.

3.5 Data dependence relations

Definition 23 (the $\rightsquigarrow_S$ 'data dependence' relation) Let S be a linear schema. We write $f \rightsquigarrow_S x$ for $f \in \mathit{Funcs}(S)$, $x \in \mathit{Symbols}(S)$ if there is a segment $\tilde{f}\sigma\tilde{x}$ in S such that $\tilde{f}$ is an assignment to f and $\tilde{x} \in \mathit{alphabet}(S)$ satisfies $\mathit{symbol}(\tilde{x}) = x$, and there is no assignment to the variable $\mathit{assign}_S(f)$ along σ. We call $\tilde{f}\sigma\tilde{x}$ an fx-segment in this case. We generalise this by defining $f \rightsquigarrow_S v$ for $f \in \mathit{Funcs}(S)$, $v \in \mathcal{V}$ if $f \rightsquigarrow_{S\,w:=g(v);}\, g$ holds for any linear schema $S\,w:=g(v);$, in which case we define an fv-segment in S to be any segment σ of S such that $\sigma\,w:=g(v)$ is an fg-segment in the schema $S\,w:=g(v);$. Lastly, we write $v \rightsquigarrow_S x$ for $v \in \mathcal{V}$, $x \in \mathit{Symbols}(S)$ if $h \rightsquigarrow_{v:=h();\,S} x$ holds for any linear schema $v:=h();\ S$, in which case we define a vx-segment in S to be any $\sigma \in \mathit{pre}(\Pi(S))$ such that $v:=h()\,\sigma$ is an hx-segment in the schema $v:=h();\ S$.
In all cases, we may strengthen the relation $x \rightsquigarrow_S y$ by writing $x \rightsquigarrow_S y\,(n)$ for $n \in \mathbb{N}$ if either $y \in \mathcal{V}$ or the nth component of $\mathbf{refvec}_S(y)$ is x or $\mathit{assign}_S(x)$.

Thus $f \rightsquigarrow_S x$ holds for $f \in \mathit{Funcs}(S)$, $x \in \mathit{Symbols}(S)$ if and only if there exists a path in S along which a (predicate) term $x(\mathbf{t})$ such that $\mathbf{t}$ has a component $f(\mathbf{t}')$ is created; and we may define an fx-segment to be any segment in S which 'witnesses' such a creation. Similar characterisations can be given for the statements $f \rightsquigarrow_S v$ and $v \rightsquigarrow_S x$ for $v \in \mathcal{V}$.
As an example, if T is the linear schema of Figure 3, the relations $v \rightsquigarrow_T q$, $k \rightsquigarrow_T q$, $v \rightsquigarrow_T k$, $k \rightsquigarrow_T k$ (but not $k \rightsquigarrow_T v$), $w \rightsquigarrow_T p$, $h \rightsquigarrow_T f$, $h \rightsquigarrow_T u$, $f \rightsquigarrow_T v$, and $g \rightsquigarrow_T v$

```
while p(v) do
        {
        u := g(v);
        v := f();
        }
```

Fig. 4. $back_S(p, f, g)$ holds here

hold.
Note that the relation $\rightsquigarrow_S$ denotes a purely syntactic property of a linear schema S; $f \rightsquigarrow_S x$ may hold even if there is no interpretation defining a path passing through the fx-segment whose existence is asserted.

3.6 Other relations between schema symbols

Definition 24 gives three relations which strengthen the data dependence relation.

Definition 24 (the *outif, thru* and *back* relations) Let S be a linear schema and let $x \in \mathcal{F} \cup \mathcal{V}$ and $y \in \mathcal{F} \cup \mathcal{P} \cup \mathcal{V}$. Let $p \in \mathcal{P}$. Assume that $x \rightsquigarrow_S y$ holds. Then we make the following definitions.

- If $p \in \mathit{whilePreds}(S)$ and both x and y are symbols in $\mathit{body}_S(p)$ but $\neg(x \rightsquigarrow_{\mathit{body}_S(p)} y)$ holds (a backward data dependence) then we write $\mathit{back}_S(p, x, y)$.
- If $Y \in \{\mathsf{T}, \mathsf{F}\}$ and $p \in \mathit{ifPreds}(S)$ and $x \in \mathit{Funcs}(\mathit{part}_S^Y(p))$ and $\neg(x \rightsquigarrow_{\mathit{part}_S^Y(p)} y) \vee (y \in \mathcal{V})$ holds, then we write $\mathit{outif}_S(p, Y, x, y)$. If $Y \in \{\mathsf{T}, \mathsf{F}\}$ and $p \in \mathit{ifPreds}(S)$ and neither x nor y is a symbol in either of the schemas $\mathit{part}_S^{\mathsf{T}}(p)$ or $\mathit{part}_S^{\mathsf{F}}(p)$ and every xy-segment contains the letter $<p = Y>$, then we write $\mathit{thru}_S(p, Y, x, y)$. (Note that $\mathit{thru}_S(p, Y, x, p)$ is always false.)

As an example, $\mathit{back}_S(p, f, g)$ holds if S is the linear schema in Figure 4.

Definition 25 (q-competing function symbols and variables) Let S be a linear schema and assume that $f \rightsquigarrow_S x\,(n)$ and $g \rightsquigarrow_S x\,(n)$ for $f, g, x \in \mathit{Symbols}(S) \cup \mathcal{V}$ and $n \in \mathbb{N}$. Let $q \in \mathit{ifPreds}(S)$. We say that f and g are q-competing for x in S if for $\{X, Y\} = \{\mathsf{T}, \mathsf{F}\}$, we have both $\mathit{outif}_S(q, X, f, x) \vee \mathit{thru}_S(q, X, f, x)$ and $\mathit{outif}_S(q, Y, g, x) \vee \mathit{thru}_S(q, Y, g, x)$.

Thus f and g are p-competing for v in the schemas of Figures 1 and 5. Proposition 26 shows that if $\mathit{thru}_S(p, Y, x, y)$ holds for suitable p, Y, x, y then $\mathit{outif}_S(p, \neg Y, f', y)$ holds for some function symbol f'.

Proposition 26 (connection between outif_S and thru_S) *Let S be a linear* *schema and assume that $thru_S(p, Y, x, y)$ holds for some $p \in ifPreds(S)$, $Y \in \{\mathsf{T}, \mathsf{F}\}$ and $x, y \in Symbols(S) \cup \mathcal{V}$. Assume that $x \rightsquigarrow_S y\,(n)$ holds for $n \in \mathbb{N}$. Then every path in $\Pi(part_S^{\neg Y}(p))$ passes through some $f' \in \mathcal{F}$ satisfying $outif_S(p, \neg Y, f', y)$ and $f' \rightsquigarrow_S y\,(n)$.*

Proof. We may assume that $x \in \mathcal{F}$ holds, otherwise we may replace S with a linear schema $x := f();\ S$. Since $thru_S(p, Y, x, y)$ holds, there is an xy-segment $\gamma = \mu <\!p = Y\!> \mu'\mu''$ in S with $\mu' \in \Pi(part_S^Y(p)))$ and $\mu'' \neq \Lambda$. We may assume that p occurs only once in the segment γ; otherwise we could delete a segment from within γ. Let $\sigma \in \Pi(part_S^{\neg Y}(p))$. The segment $\mu <\!p = \neg Y\!> \sigma\mu''$ does not enter the Y-part of p and so is not an xy-segment, by the definition of $thru_S(p, Y, x, y)$. Thus the variable assigned by x is 'killed' along σ, giving the result. $\square$

Definition 27 (the $above_S$ function) Let S be a linear schema and let x be a symbol in S. If x lies immediately below S, then we define $above_S(x) = x$; otherwise we define $above_S(x)$ to be the while predicate lying immediately below S and containing x in its body.

Definition 28 (the $\ll_S$ relation) Let S be a linear schema and let

$$\{x, y\} \subseteq Symbols(S).$$

Assume that S lies immediately above the set $\{x, y\}$. We define $x \ll_S y$ if $above_S(x) \neq above_S(y)$ and there is a segment in S which begins at $above_S(x)$ and ends at $above_S(y)$.

Observe the following; if $x \ll_S y$ then every segment in S which begins at x and ends at y passes through every occurrence of x before any occurrence of y, and x and y do not lie in opposite parts of any if predicate. Also, $\ll_S$ is transitive; and $x \ll_S y \wedge y \ll_S x$ never holds, since otherwise S would contain a while predicate containing both $above_S(x)$ and $above_S(y)$ in its body.
It can be shown (see [7, Lemma 134]) that if $back_S(q, f, x)$ holds for $q \in whilePreds(S)$, then $\neg(f \ll_{body_S(q)} x)$ holds.

```
$u := h();$
$v := f(u);$
$if\ p(w)$    $then\ \Lambda$
              $else\ \ v := g();$
```

Fig. 5. $thru_S(p, \mathsf{T}, f, v) \wedge outif_S(p, \mathsf{F}, g, v)$ holds here

3.7 The $\mathcal{N}_S$ and Inv_S sets

The symbol and variable sets of Definition 29 are purely syntactically defined, and contain all the symbols and (initial) variables which can influence the final value of a variable. This is stated precisely in Theorem 33.

Definition 29 (symbols *needed* by variables) Let S be a linear schema and let $u \in \{\omega\} \cup \mathcal{V}$. Then we define the set $\mathcal{N}_S(u)$ to be the minimal subset of $\mathit{Symbols}(S)$ satisfying the following closure conditions; if $f \in \mathcal{F}$, $x \in (\mathcal{V} \cap \{u\}) \cup \mathcal{N}_S(u)$ and $f \rightsquigarrow_S x$ then $f \in \mathcal{N}_S(u)$; and if $u = \omega$ then $\mathit{whilePreds}(S) \subseteq \mathcal{N}_S(u)$; and if $p \searrow_S x$ for $x \in \mathcal{N}_S(u)$ then $p \in \mathcal{N}_S(u)$.
We also define $Inv_S(u) \subseteq \mathcal{V}$ to contain all variables v satisfying $v \rightsquigarrow_S v$ if $v = u \in \mathcal{V}$ or $v \rightsquigarrow_S y$ for some $y \in \mathcal{N}_S(u)$.
We generalise this by defining $\mathcal{N}_S(V) = \cup_{u \in V} \mathcal{N}_S(u)$ for a set V, and similarly with Inv_S.

The functions $\mathcal{N}_S$, Inv_S have more restricted domains in Definition 29 above than in [7, Definition 35], in which $\mathcal{N}_S(x)$ and $Inv_S(x)$ for $x \in \mathit{Symbols}(S)$ are also defined.

Note that $\mathcal{N}_S(y)$ is a set of symbols of S, whereas $Inv_S(y)$ is a subset of $\mathcal{V}$.

It can easily be proved that if $v \in \mathcal{V}$ and a linear schema $S = AB$, then $Inv_S(v) = Inv_A(Inv_B(v))$.

Observe that if any of the relations given in Definition 24 hold, and $y \in \mathcal{N}_S(u)$ for some $u \in \mathcal{V} \cup \{\omega\}$, then $x \in \mathcal{N}_S(u)$ holds; in the case that $thru_S(p, Y, x, y)$ holds, this follows from Proposition 26.

3.8 Definition of u-similar and u-congruent linear schemas

Definition 30 (u-similar and u-congruent linear schemas) Let S, T be linear schemas and let $u \in \{\omega\} \cup \mathcal{V}$. Then $S \; \mathit{simil}_u \; T$ (S is u-similar to T) if and only if the following hold:

(1) $\mathcal{N}_S(u) = \mathcal{N}_T(u)$;
(2) $\mathcal{N}_S(u) \cap \mathit{ifPreds}(S) = \mathcal{N}_T(u) \cap \mathit{ifPreds}(T)$;
(3) $\mathcal{N}_S(u) \cap \mathit{whilePreds}(S) = \mathcal{N}_T(u) \cap \mathit{whilePreds}(T)$;
(4) $f \rightsquigarrow_S x\,(n) \wedge x \in \mathcal{N}_S(u) \iff f \rightsquigarrow_T x\,(n) \wedge x \in \mathcal{N}_T(u)$, for all $f \in \mathcal{F}$ and $n \geq 1$;
(5) $f \rightsquigarrow_S u \iff f \rightsquigarrow_T u$ if $u \in \mathcal{V}$ and $f \in \mathcal{F}$;
(6) $v \rightsquigarrow_S x\,(n) \iff v \rightsquigarrow_T x\,(n)$ for all $v \in \mathcal{V}$ and $x \in \mathcal{N}_S(u)$ and $n \geq 1$;
(7) $q \searrow_S p\,(Z) \iff q \searrow_T p\,(Z)$ if $u = \omega$ and $p \in \mathit{whilePreds}(S)$ and q is any predicate and $Z \in \{\mathsf{T}, \mathsf{F}\}$;
(8) $\mathit{Symbols}(\mathit{body}_S(p)) \cap \mathcal{N}_S(u) = \mathit{Symbols}(\mathit{body}_T(p)) \cap \mathcal{N}_T(u)$ if $p \in \mathit{whilePreds}(S)$;

(9) $back_S(p,f,x) \wedge x \in \mathcal{N}_S(u) \iff back_T(p,f,x) \wedge x \in \mathcal{N}_T(u)$;

(10) If $q \in ifPreds(S)$ and $Z \in \{\mathsf{T},\mathsf{F}\}$ and $f \in \mathcal{F}$ and $x \in \mathcal{N}_S(u) \cup (\mathcal{V} \cap \{u\})$ then

$$outif_S(q,Z,f,x) \vee thru_S(q,Z,f,x) \iff outif_T(q,Z,f,x) \vee thru_T(q,Z,f,x);$$

(11) If $f, f' \in \mathcal{F}$ and $f, f' \rightsquigarrow_S x\ (r)$ for $x \in \mathcal{N}_S(u) \cup (\{u\} \cap \mathcal{V})$, and $r \in \mathbb{N}$, and $\bar{S}, \bar{T}$ are the main subschemas of S and T respectively lying immediately above $\{f, f'\}$, then either $\neg(f \ll_{\bar{S}} f' \wedge f' \ll_{\bar{T}} f)$ holds, or there exists $q \in ifPreds(S)$ such that f and f' are q-competing for x in S;

(12) If $p \in whilePreds(S)$, $f \in \mathcal{F}$ and $f \rightsquigarrow_S x \wedge x \in \mathcal{N}_S(u)$ and $v = assign_S(f)$ and $w = assign_T(f)$, then

$$f \rightsquigarrow_{body_S(p)} v \wedge v \rightsquigarrow_{body_S(p)} x$$

$$\iff$$

$$f \rightsquigarrow_{body_T(p)} w \wedge w \rightsquigarrow_{body_T(p)} x$$

holds.

(13) If $p \in whilePreds(S)$, $q \in ifPreds(S)$, $f \in Funcs(S)$, $x \in \mathcal{N}_S(u)$, $Z \in \{\mathsf{T},\mathsf{F}\}$ and $f \rightsquigarrow_S x$, with $v = assign_S(f)$ and $w = assign_T(f)$ and $v \rightsquigarrow_{body_S(p)} x$, then

$$outif_{body_S(p)}(q,Z,f,v) \vee thru_{body_S(p)}(q,Z,f,v)$$

$$\iff$$

$$outif_{body_T(p)}(q,Z,f,w) \vee thru_{body_T(p)}(q,Z,f,w)$$

holds.

If $S\ simil_u\ T$ and also $\mathbf{refvec}_S(x) = \mathbf{refvec}_T(x)$ for all $x \in \mathcal{N}_S(u)$ and $assign_S(f) = assign_T(f)$ for all $f \in \mathcal{N}_S(u) \cap \mathcal{F}$, then we say that S and T are u-congruent, written $S\ cong_u\ T$.

We also write $S\ simil_V\ T$ to mean that $S\ simil_u\ T$ for all $u \in V$, and $S\ simil\ T$ to mean that $S\ simil_{\mathcal{V} \cup \{\omega\}}\ T$ holds. Also $S\ cong_V\ T$ has a similar meaning.

Observe that the two linear predicate-free schemas

$$u := f();$$
$$v := g(u);$$

and

$$u' := f();$$
$$v := g(u');$$

are v-similar but not v-congruent if $u \neq u'$; thus congruence is a stronger condition than similarity.

Informally, for two linear structured schemas S, T to satisfy $S\, \mathit{simil}_u\, T$, the following must hold;

- S and T have the same set of u-needed function symbols, if predicate symbols and while predicate symbols. (Conditions (1), (2), (3) of $S\, \mathit{simil}_u\, T$).
- S and T have the same *data dependence* relations among those symbols in $\mathcal{N}_S(u)$. (Conditions (4), (5),(6) of $S\, \mathit{simil}_u\, T$).
- S and T have the same set of u-needed symbols lying in the body of each while predicate (Condition (8) of $S\, \mathit{simil}_u\, T$). If $u = \omega$ a weaker statement also holds for while predicates lying under if predicates (Condition (7) of $S\, \mathit{simil}_u\, T$).
- Also, the bodies of while predicates in S and T satisfy the same data dependence conditions between symbols lying in $\mathcal{N}_S(u)$ (Conditions (9), (12) of $S\, \mathit{simil}_u\, T$).
- Conditions (10), (11) and (13) of $S\, \mathit{simil}_u\, T$ are a kind of counterpart for function symbols lying under if predicates to Condition (8) for symbols lying under while predicates, showing that change of ordering with respect to $\ll_S$ of function symbols (as with f, g in the v-equivalent schemas given in Figures 1 and 5) can only occur in connection with an if predicate.

Theorem 31 ($S\, \mathit{simil}_u\, T$ is decidable in polynomial time) *Given linear schemas S and T and $u \in \mathcal{V} \cup \{\omega\}$, it is decidable in polynomial time whether $S\, \mathit{simil}_u\, T$ holds.*

Proof. Given a linear schema S, encoded as indicated in Definition 3, with the braces $\{\ \}$, the truth of the relations $p \searrow_S x\,(Z)$ for each $p \in \mathit{Preds}(S)$, $x \in \mathit{Symbols}(S)$, $Z \in \{\mathsf{T}, \mathsf{F}\}$ can be established in polynomial time. Given two elements $v, w \in \mathit{alphabet}(S)$, with symbols v', w', we can decide in polynomial time whether w occurs immediately after v in any word in $\Pi(S)$, since this holds if and only if either w' occurs after v' in S without there being any other symbol between them, and $p \searrow_S v' \iff p \searrow_S w'$ for all $p \in \mathit{whilePreds}(S)$, or $v' \in \mathit{whilePreds}(S)$ and v' lies immediately above w' and there are no symbols occurring after v' in S before the closing brace $\}$ defined by v'. Thus we can construct in polynomial time a directed graph G_S, whose vertices are the elements of $\mathit{alphabet}(S)$ and such that there is an edge from vertex v to w in the graph G_S if and only if w occurs immediately after v in a word in $\Pi(S)$. Given $f \in \mathit{Funcs}(S)$ and $x \in \mathit{Symbols}(S)$, we can establish whether $f \rightsquigarrow_S x$ holds by deleting all vertices in G_S that are assignments to $\mathit{assign}_S(f)$ except the one with function symbol f or x, if $x \in \mathcal{F}$, and edges adjacent to deleted vertices, and establishing whether the letter containing x is reachable from the f-assignment in the resulting directed graph. This latter problem is well-known to be polynomial-time decidable in the size of G_S. The values of n for which $f \rightsquigarrow_S x\,(n)$ also holds can also be easily established, as can the truth of the assertions $v \rightsquigarrow_S x\,(n)$ for $v \in \mathcal{V}$ and $f \rightsquigarrow_S u$. Also, the truth of the relations above_S and $\ll_S$ for appropriate arguments can be decided in polynomial time by studying S. Having obtained this information, we can test the truth of the relations back_S, outif_S, thru_S (and hence the q-competing condition) for appropriate arguments. By comparing this information with that obtained from T and the graph G_T, it can be decided in polynomial time whether S and T satisfy $S\, \mathit{simil}_u\, T$. $\Box$

4 Slices of schemas

An important special case of the equivalence problem for schemas S, T is that in which T is a *slice* of S.

Definition 32 A *slice* of a structured schema S may be obtained recursively by the following rules;

- if $S = S_1S_2S_3$ then $S_1S_3,\ S_1S_2$ and S_2S_3 are slices of S;
- if T' is a slice of T then $\textit{while}\, p(\mathbf{u})\ \textit{do}\, T'$ is a slice of $\textit{while}\, p(\mathbf{u})\ \textit{do}\, T$;
- if T' is a slice of T then the if schema $\textit{if}\ q(\mathbf{u})\ \textit{then}\, S\ \textit{else}\, T'$ is a slice of $\textit{if}\ q(\mathbf{u})\ \textit{then}\, S\ \textit{else}\, T$ (the true and false parts may be interchanged in this example);
- a slice of a slice of S is itself a slice of S.

The following facts are easily proved. All slices of a linear schema are also linear. If a set $\Sigma \subseteq \textit{Symbols}(S)$ (for linear S) satisfies $(x \in \Sigma \wedge p \searrow_S x) \Rightarrow p \in \Sigma$, then there is a unique slice T of S satisfying $\textit{Symbols}(T) = \Sigma$; the slice T can be obtained from S by successively removing all assignments whose function symbols do not lie in Σ, and every if and while subschema of S whose guard does not lie in Σ.
A special case is given by $\Sigma = \mathcal{N}_S(V)$ for $V \subseteq \mathcal{V} \cup \{\omega\}$. In this case every slice T of S containing all symbols in $\mathcal{N}_S(V)$ satisfies $\textit{Inv}_T(V) = \textit{Inv}_S(V)$ and $S\,\textit{cong}_V\,T$, since deletion from S of symbols not lying in $\mathcal{N}_S(V)$ does not affect the schema properties defining these statements. We will show in Part (2) of Theorem 33 that $S \cong_V T$ also holds.

A slice of an LFL schema need not be free or liberal; for example, the schema $\textit{while}\, p(v)\ \textit{do}\ \Lambda$, which is not free, is a slice of the LFL schema below;

```
while p(v) do
            {
            u := h(u);
            w := k(u);
            v := g(v);
            }
```

Also, deleting the assignment $u := h(u)$; gives a schema which is free but not liberal. However the slice of an LFL schema S which contains precisely the symbols in $\mathcal{N}_S(V)$ for any $V \subseteq \mathcal{V}$ is itself LFL; this follows from Theorem 18 and the 'backward data dependence' property of $\mathcal{N}_S(V)$.

Theorem 33 was proved by Weiser in [35] for the case $u \in \mathcal{V}$, using different terminology.

Theorem 33 *Let S be a (not necessarily free or liberal) linear schema and let T be a slice of S. Let $u \in \mathcal{V} \cup \{\omega\}$, let i, j be interpretations differing only on predicates not lying in $\mathcal{N}_S(u)$, and let c, d be states such that $c(v) = d(v)$ for all $v \in Inv_S(u)$. Assume that T contains every symbol of $\mathcal{N}_S(u)$.*

(1) *If $Symbols(T) = \mathcal{N}_S(u)$, then $\mathcal{M}[\![S]\!]^i_c \neq \bot \Rightarrow \mathcal{M}[\![T]\!]^j_d \neq \bot$.*
(2) *If $u \in \mathcal{V}$ and $\mathcal{M}[\![S]\!]^i_c$ and $\mathcal{M}[\![T]\!]^j_d$ both terminate, then $\mathcal{M}[\![S]\!]^i_c(u) = \mathcal{M}[\![T]\!]^j_d(u)$; and if $u = \omega$ then $\mathcal{M}[\![S]\!]^i_c \neq \bot \iff \mathcal{M}[\![T]\!]^j_d \neq \bot$.*

In particular, $S \cong_u T$ holds.

Proof. This is proved in [7, Theorem 42]. □

Part (1) of Theorem 33 may fail for a slice T whose symbol set strictly contains $\mathcal{N}_S(u)$; for example, if S is

$v = f();$
while $p(v)$ *do* Λ

and T is the slice

while $p(v)$ *do* Λ

for a variable $v \neq u$. If the interpretation i maps every predicate term $p(t)$ to T unless $t = f()$ then $\mathcal{M}[\![S]\!]^i_e$ terminates whereas $\mathcal{M}[\![T]\!]^i_e$ does not.

5 The Main Theorems and Further Directions

Our main result consists of the following two Theorems.

Theorem 34 *Let $u \in \mathcal{V} \cup \{\omega\}$ and let S and T be u-similar linear schemas. Then S and T are u-equivalent.*

Proof. This is proved in [7, Theorem 55]. □

Theorem 35 *Let S, T be LFL schemas. Then*

$$S \cong T \iff S \; simil \; T$$

holds. If $V \subseteq \mathcal{V} \cup \{\omega\}$ and $\omega \in V$ then

$$S \cong_V T \iff S \; simil_V \; T$$

holds. In particular, it is decidable in polynomial time whether S and T are equivalent.

Proof. The first assertion is a special case of the second (where V is the set containing all variables assigned in either S or T, plus ω).
The statement $S\,\mathit{simil}_u\,T \Rightarrow S \cong_u T$ for any $u \in \mathcal{V} \cup \{\omega\}$, is Theorem 34; $S\,\mathit{simil}_V\,T \Rightarrow S \cong_V T$ for any set $V \subseteq \mathcal{V} \cup \{\omega\}$ follows immediately from this result. The proof of the converse statement for sets V containing ω is given as part of [7, Theorem 148]. The polynomial time bound follows from Theorem 31. $\Box$

An overview of the full proof of Theorem 35 is given in [7, Section 1.2].

Of the various related problems which seem worth studying (besides the 'missing'result $S \cong_v T \Rightarrow S\,\mathit{simil}_v\,T$ for $v \in \mathcal{V}$, which we have failed to prove), two strike us as being particularly promising.

5.1 Computing minimal slices of schemas

For the purpose of program slicing, given a schema S and variable u, it is of interest to be able to compute those minimal slices of S (with minimality defined by symbol sets) which are v-equivalent to S and which preserve termination. By Part (1) of Theorem 33 and [7, Theorem 76], it follows that for any $u \in \mathcal{V}$, the minimal slice T of an LFL schema S such that $S \cong_u T$ and $\mathcal{M}[\![S]\!]^j_d \neq \bot \Rightarrow \mathcal{M}[\![T]\!]^j_d \neq \bot$ always holds is precisely the slice of S such that $\mathit{Symbols}(T) = \mathcal{N}_S(u)$ holds. The first author has proved in [37] that this also holds if the linearity hypothesis is replaced by function-linearity (a schema is function-linear if it does not contain more than one occurrence of the same function symbol), provided that the definition of $\mathcal{N}_S(u)$ is generalised to allow for multiple occurrences of predicate symbols.

If S is merely free and linear then $S \cong_u T$ need not imply $\mathit{Symbols}(T) \supseteq \mathcal{N}_S(u)$, as the example of Figure 6 shows. Owing to the constant g-assignment, S is not liberal, though it is free. Clearly $f \in \mathcal{N}_S(u)$ holds, but the slice of S obtained by deleting the f-assignment, which is also free, is u-equivalent to S. It is also ω-equivalent to S, and hence satisfies the termination requirement for slices.

It would be of interest to find a method of computing the minimal slice of S satisfying these conditions under weaker hypotheses than the assumption that S is liberal, free and function-linear.

5.2 Using schema transformations to construct equivalent schemas

Given a linear schema S and $u \in \mathcal{V} \cup \{\omega\}$, it can be shown using Theorem 34 that the following transformations of S preserve u-equivalence.

- Changing the variables mentioned in S in any way that preserves u-similarity.

```
while $q(v)$ do
        {
        $v := k(v);$
        $w := h(w);$
        if $p(w)$   then
                        {
                        $u := g();$
                        $w := f(w);$
                        }
                   else  $\Lambda$
        }
```

Fig. 6. Deleting the f-assignment gives a u-equivalent slice of this schema

- Replacing S by a slice T of S, such that T contains every element of $\mathcal{N}_S(u)$.
- Pulling out a subschema from an if subschema of S; that is, replacing a subschema

```
if $p(\mathbf{v})$  then  $S_1 S_2$
            else  $S_3$
```

of S by the schema

```
$S_1$
if $p(\mathbf{v})$  then  $S_2$
            else  $S_3$
```

provided that this does not create a new fx-segment μ for $f \in \mathit{Funcs}(S_1)$ and $x \in \mathcal{N}_S(u) \cup \{u\}$ such that either $x = p$ or μ passes through $\lessdot p = \mathsf{F} \gtrdot$. Also, if $u = \omega$ then S_1 must not contain a while predicate, otherwise Condition (7) of simil_u is violated. Clearly the true and false parts of p may be interchanged.
- Changing the order of 'towers' of if predicates; that is, interchanging $p(\mathbf{u})$ and $q(\mathbf{v})$ in a subschema

```
if $p(\mathbf{u})$ then
        {
        if $q(\mathbf{v})$ then $T$
                  else $\Lambda$
        }
    else $\Lambda$
```

of S. Again, the true and false parts of p or q may be interchanged.

- Replacing a subschema S_1S_2 of S by S_2S_1 to give a schema T, provided that no variable is assigned in both S_1 and S_2, and S_1S_2 contains no fx-segment with $f \in \mathit{Funcs}(S_1)$ and $x \in \mathit{Symbols}(S_2)$, and the same statement holds with (S, 1, 2) replaced by (T, 2, 1).

We conjecture that given any LFL schema S and $u \in \mathcal{V} \cup \{\omega\}$, all u-similar LFL schemas can be obtained from S by a sequence of these transformations and their inverses.
It may also be possible to prove that given an LFL schema S, any u-equivalent LFL schema may be reached from S by a finite sequence of such transformations without using Theorem 35, thus giving an alternative (and possibly shorter) way of proving this theorem than the one we have given in the Technical Report [7].

6 Acknowlegements

This work is funded by EPSRC grant number EP/E002919/1.

References

[1] S. Greibach, Theory of program structures: schemes, semantics, verification, Vol. 36 of Lecture Notes in Computer Science, Springer-Verlag Inc., New York, NY, USA, 1975.

[2] Z. Manna, Mathematical Theory of Computation, McGraw–Hill, 1974.

[3] A. K. Chandra, On the decision problems of program schemas with commutative and invertable functions, in: Conference Record of the ACM Symposium on Principles of Programming Languages, ACM SIGACT-SIGPLAN, Boston, Massachusetts, 1973, pp. 235–242.

[4] A. K. Chandra, Z. Manna, Program schemas with equality, in: Conference Record, Fourth Annual ACM Symposium on Theory of Computing, Denver, Colorado, 1972, pp. 52–64.

[5] A. K. Chandra, Z. Manna, On the power of programming features, Computer Languages 1 (3) (1975) 219–232.

[6] M. R. Laurence, S. Danicic, M. Harman, R. M. Hierons, J. D. Howroyd, Equivalence of conservative, free, linear schemas is decidable, Theoretical Computer Science 290 (1) (2002) 831–862.

[7] M. R. Laurence, S. Danicic, M. Harman, R. Hierons, J. Howroyd, Equivalence of linear, free, liberal, structured program schemas is decidable in polynomial time, Tech. Rep. ULCS-04-014, University of Liverpool, electronically available at `http://www.csc.liv.ac.uk/research/techreports/` (2004).

[8] M. S. Paterson, Equivalence problems in a model of computation, PhD thesis, University of Cambridge, UK (1967).

[9] E. Ashcroft, Z. Manna, Translating program schemas to while-schemas, SIAM Journal on Computing 4 (2) (1975) 125–146.

[10] Y. I. Ianov, The logical schemes of algorithms, in: Problems of Cybernetics, Vol. 1, Pergamon Press, New York, 1960, pp. 82–140.

[11] V. K. Sabelfeld, An algorithm for deciding functional equivalence in a new class of program schemes, Journal of Theoretical Computer Science 71 (1990) 265–279.

[12] A. De Lucia, A. R. Fasolino, M. Munro, Understanding function behaviours through program slicing, in: 4^{th} IEEE Workshop on Program Comprehension, IEEE Computer Society Press, Los Alamitos, California, USA, Berlin, Germany, 1996, pp. 9–18.

[13] G. Canfora, A. Cimitile, A. De Lucia, G. A. D. Lucca, Software salvaging based on conditions, in: International Conference on Software Maintenance (ICSM'96), IEEE Computer Society Press, Los Alamitos, California, USA, Victoria, Canada, 1994, pp. 424–433.

[14] A. Cimitile, A. De Lucia, M. Munro, A specification driven slicing process for identifying reusable functions, Software maintenance: Research and Practice 8 (1996) 145–178.

[15] K. B. Gallagher, J. R. Lyle, Using program slicing in software maintenance, IEEE Transactions on Software Engineering 17 (8) (1991) 751–761.

[16] K. B. Gallagher, Evaluating the surgeon's assistant: Results of a pilot study, in: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, California, USA, 1992, pp. 236–244.

[17] M. Weiser, J. R. Lyle, Experiments on slicing–based debugging aids, Empirical studies of programmers, Soloway and Iyengar (eds.), Molex, 1985, Ch. 12, pp. 187–197.

[18] H. Agrawal, R. A. DeMillo, E. H. Spafford, Debugging with dynamic slicing and backtracking, Software – Practice and Experience 23 (6) (1993) 589–616.

[19] M. Kamkar, Interprocedural dynamic slicing with applications to debugging and testing, PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, available as Linköping Studies in Science and Technology, Dissertations, Number 297 (1993).

[20] J. R. Lyle, M. Weiser, Automatic program bug location by program slicing, in: 2^{nd} International Conference on Computers and Applications, IEEE Computer Society Press, Los Alamitos, California, USA, Peking, 1987, pp. 877–882.

[21] D. W. Binkley, The application of program slicing to regression testing, in: M. Harman, K. Gallagher (Eds.), Information and Software Technology Special Issue on Program Slicing, Vol. 40, Elsevier, 1998, pp. 583–594.

[22] R. Gupta, M. J. Harrold, M. L. Soffa, An approach to regression testing using slicing, in: Proceedings of the IEEE Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, California, USA, Orlando, Florida, USA, 1992, pp. 299–308.

[23] M. Harman, S. Danicic, Using program slicing to simplify testing, in: 2^{nd} EuroSTAR conference on Software Testing Analysis and Review, Brussels, 1994.

[24] G. Canfora, A. Cimitile, M. Munro, RE^2: Reverse engineering and reuse re-engineering, Journal of Software Maintenance : Research and Practice 6 (2) (1994) 53–72.

[25] D. Simpson, S. H. Valentine, R. Mitchell, L. Liu, R. Ellis, Recoup – Maintaining Fortran, ACM Fortran forum 12 (3) (1993) 26–32.

[26] J. Beck, D. Eichmann, Program and interface slicing for reverse engineering, in: IEEE/ACM 15^{th} Conference on Software Engineering (ICSE'93), IEEE Computer Society Press, Los Alamitos, California, USA, 1993, pp. 509–518.

[27] A. Cimitile, A. De Lucia, M. Munro, Identifying reusable functions using specification driven program slicing: a case study, in: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'95), IEEE Computer Society Press, Los Alamitos, California, USA, Nice, France, 1995, pp. 124–133.

[28] S. Horwitz, J. Prins, T. Reps, Integrating non–interfering versions of programs, ACM Transactions on Programming Languages and Systems 11 (3) (1989) 345–387.

[29] J. M. Bieman, L. M. Ott, Measuring functional cohesion, IEEE Transactions on Software Engineering 20 (8) (1994) 644–657.

[30] J. J. Thuss, An investigation into slice–based cohesion metrics, Master's thesis, Michigan Technological University (1988).

[31] A. Lakhotia, Rule–based approach to computing module cohesion, in: Proceedings of the 15^{th} Conference on Software Engineering (ICSE-15), 1993, pp. 34–44.

[32] D. W. Binkley, K. B. Gallagher, Program slicing, in: M. Zelkowitz (Ed.), Advances of Computing, Volume 43, Academic Press, 1996, pp. 1–50.

[33] A. De Lucia, Program slicing: methods and applications, in: 1st IEEE International workshop on Source Code Analysis and Manipulation, IEEE Computer Society Press, Los Alamitos, California, USA, Florence, Italy, 2001, pp 142-149.

[34] F. Tip, A survey of program slicing techniques, Tech. Rep. CS-R9438, Centrum voor Wiskunde en Informatica, Amsterdam (1994).

[35] M. Weiser, Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method, PhD thesis, University of Michigan, Ann Arbor, MI (1979).

[36] S. Danicic, M. Harman, R. M. Hierons, J. howroyd, M. R. Laurence, Static program slicing algorithms are minimal for free liberal program schemas, The Computer Journal 48 (6) (2006) 737–748.

[37] M. R. Laurence, Characterising minimal semantics-preserving slices of function-linear, free, liberal program schemas, submitted to Journal of Logic and Algebraic Programming in July 2005 .