Equivalence of Conservative, Free, Linear Program Schemas is Decidable

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

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

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

Abstract

A program schema defines a class of programs, all of which have identical statement structures, but whose expressions may differ. We prove that given any two structured schemas which are conservative, linear and free, it is decidable whether they are equivalent.

Key words: program schemas, decidability, conservative schemas, free schemas, linear schemas

1 Introduction

A schema represents the statement structure of a program by replacing computational expressions with terms involving function and predicate symbols. For example, in Figure 1, P_1 and P_2 are programs with the same structure, represented by schema S_1. A schema, S, thus stands for 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.

This paper is concerned with the problem of finding a class of schemas for which equivalence is decidable. Two schemas are *equivalent* if and only if under all interpretations the corresponding programs are semantically equivalent. In general, the equivalence of schemas is undecidable[1]. However, in an early result about program schemas, Ianov [2] introduced a restrictive class of schemas, for which equivalence is decidable. Unfortunately, Ianov schemas,

```
s := f1();
while b1(i)
do
begin
   s := f2(s,i);
   i := f3(i);
end
```

Schema S_1

```
s := 0;
while i > 0
do
begin
   s := s + i;
   i := i - 1;
end
```

Program P_1

```
s := 1;
while i > 0
do
begin
   s := s * i;
   i := i - 1;
end
```

Program P_2

Fig. 1. A schema and two programs in its equivalence class

contain only a *single* program variable, so they form a very restricted class of schemas.

Other positive results are those of Paterson [1] and Sabelfeld [3]. Paterson 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 executable path. Sabelfeld 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 free interpretation.

In an attempt to escape the restrictions of Ianov schemas, while retaining the decidability of equivalence, much work has been undertaken to define classes of schemas which have an arbitrary number of program variables, but which have other structural and semantic restrictions placed upon them.

Three schema classes which have been widely studied are:

Free schemas [4], where any path is executable under some interpretation.
Liberal schemas [1], where no value can be computed more than once in any interpretation.
Conservative schemas [1] in which the right-hand side of every assignment contains the variable assigned to.

In all these cases and in all possible combinations thereof, decidability (or otherwise) of equivalence remains open.

A **linear schema** (or *non–repeating* schema) is one where each function and predicate name occurs at most once. We use the term CFL schemas to denote the class of schemas which are Conservative, Free and Linear.

Earlier work on program schemas considered unstructured languages in which there were no block structured constructs. In these unstructured languages, non-sequential control flow is determined solely by jump statements. In this paper, only structured schemas are considered; those which can be formed from assignment, sequencing, conditionals and while loops. The contribution of this paper is to demonstrate that equivalence is decidable for such structured CFL schemas.

2 Organisation of the Paper

This section provides both an overview of the paper's structure and a sketch of the proof which follows in the remainder of the paper.

Section 3: Preliminary Definitions

Section 3 presents the syntax and semantics of schemas. In common with other approaches to program schemas[5,6,1,3], the semantics of schemas is defined using *Herbrand Interpretations* of a schema. Schemas are equivalent if and only if they have the same semantics.

A schema S, and an interpretation i, together give rise to a *path*. A path is a possibly infinite sequence of elements each of which corresponds to the execution of an assignment or a predicate of the schema.

Section 4: Paths and Interpretations Passing Through Symbols

Given a schema S, we define what it means for a terminating interpretation, i, to pass *semantically* through either a function or predicate symbol. Importantly, if an interpretation i passes through a function or predicate symbol in S, it does so in all schemas equivalent to S. The main result of this section is that for a CFL schema S, an interpretation i passes semantically through a function symbol f if and only if f occurs on the path determined by S and i.

Section 5: Equivalence is Decidable for Predicate-Free Conservative Schemas

A predicate–free schema consists of just a sequence of assignments. In Section 5, it is shown that the equivalence of *predicate–free* CFL schemas is decidable.

Section 6: Equivalent Schemas have Identical Predicate Sets

In Section 6, it is shown that two equivalent CFL schemas have the same set of predicates and that corresponding predicates are of the same type (*if* or *while*) in each schema.

Section 7: Equivalent Schemas have Identical Symbol Sets in Loop Bodies

Having shown that equivalent CFL schemas, S and T have the same set of *while* and *if* predicates, in this section we go on to show that for each *while* predicate p the set of symbols (functions and predicates) in the body of p in S is the same as the set of symbols in the body of p in T.

Section 8: Standardised CFL schemas: A Canonical Form

The results of the previous section *almost* carry over to *if* statements. Indeed in Theorem 43, it is shown that in equivalent CFL schemas, that both the *then* and *else* parts of corresponding *if*s contain the same set of *while* predicates and function symbols.

All that is further required is that corresponding *if* statements contain the same set of *if* predicates. Unfortunately, this is not the case. Consider, for example, the equivalent CFL schemas (where λ represents the empty sequence of statements):

$$\boxed{\begin{array}{lll} \mathit{if} & p(\mathbf{x}) & \\ \mathit{then} & \mathit{if} & q(\mathbf{y}) \\ & \mathit{then} & S \\ & \mathit{else} & \lambda \\ \mathit{else} & \lambda & \end{array}} \quad \texttt{and} \quad \boxed{\begin{array}{lll} \mathit{if} & q(\mathbf{y}) & \\ \mathit{then} & \mathit{if} & p(\mathbf{x}) \\ & \mathit{then} & S \\ & \mathit{else} & \lambda \\ \mathit{else} & \lambda & \end{array}}$$

We call such statements *ambiguous*. Fortunately, Section 8 shows that these ambiguous statements have a computable canonical form, which we call 'standardised CFL schemas'. Theorem 49 states that equivalent standardised CFL schemas contain the same sets of symbols in each part of each *if* predicate.

Section 9: Interchanging Commuting Subschemas

In Section 9, the main result is proved. First we show that if two CFL schemas are equivalent then any function or predicate symbol references the same vector of variables (Proposition 50). Lemma 56 then strengthens this by proving that, if S and T are equivalent standardised schemas, and the order of two subschemas of S differs from the order of the corresponding subschemas in T, then these subschemas may be interchanged while preserving equivalence.

This interchange of subschemas may be performed finitely many times to obtain a schema which is identical to T. There are finitely many schemas equivalent to a given schema, and it is this that makes equivalence decidable.

It is shown that it will take polynomial time before S is transformed into T, or there are no such pairs left in S, in which case S and T are not equivalent.

Section 10: Relevance of Linear Schemas to Program Slicing

In Section 10, we discuss the relevance of Linear Schemas to issues, of particular interest to us, concerning program slicing.

3 Preliminary Definitions

This section presents the syntax and semantics of schemas and the schema classes of interest.

The first task is to define the class $Sch(\mathcal{F}, \mathcal{P}, \mathcal{V})$ of schemas that are to be considered. Here $\mathcal{F}$, $\mathcal{P}$ and $\mathcal{V}$ denote fixed sets of *function symbols*, of *predicate symbols* and of *variable symbols* respectively. Each function or predicate symbol $g \in \mathcal{F} \cup \mathcal{P}$ has an *arity*, that is, a non-negative integer which is the number of arguments referenced by g. Note that in the case when the arity is zero then g may be thought of as a constant. A *function expression* $f(\mathbf{x})$ is formed by a function symbol f of arity n together with its arguments $\mathbf{x}$ which is an n-tuple of variable symbols. A *predicate expression* $p(\mathbf{y})$ is similarly formed by a predicate symbol p with its arguments $\mathbf{y}$ which again denotes a tuple of variable symbols of the correct arity.

Definition 1 (Structured Schemas)
An *atomic* schema is an assignment of the form $y{:=}f(\mathbf{x})$ where $y \in \mathcal{V}$, and $f(\mathbf{x})$ is a function expression. From these all schemas in the set $Sch(\mathcal{F}, \mathcal{P}, \mathcal{V})$ of all schemas on the symbols $\mathcal{F}$, $\mathcal{P}$ and $\mathcal{V}$ may be 'built up' from the following constructs on schemas.

Sequences $S' = U_1 U_2 \ldots U_r$ may be formed provided that $U_1, \ldots, U_r$ are schemas. This includes the *empty sequence* consisting of the empty sequence of schemas. We use the symbol λ to refer to the empty sequence.

if **Schemas** $S'' = \mathit{if}\ p(\mathbf{x})\, \mathit{then}\ T_1\, \mathit{else}\ T_2$ may be formed whenever $p(\mathbf{x})$ is a predicate expression and when the schemas T_1 and T_2 are not both λ.

while **Schemas** $S''' = \mathit{while}\ q(\mathbf{y})\, \mathit{do}\ T$ may be formed whenever $q(\mathbf{y})$ is a predicate expression and T is a schema.

In the above definition S'' will be referred to as an *if schema* and S''' as a *while schema*. The predicate symbols p and q are called the *guards* of the schemas S'' and S''', respectively. The *subschemas* of a schema are defined as follows; the empty sequence λ is a subschema of every schema; the only subschemas of an atomic schema S are S itself and λ; the subschemas of $U_1 \ldots U_r$ are those of each U_j for $1 \le j \le r$ and also the schemas $U_i U_{i+1} \ldots U_j$ for $1 \le i \le j \le r$; 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'''). The set of function symbols in a schema S is defined as $\mathit{Funcs}(S) \subseteq \mathcal{F}$.

The sets of *if* and *while* predicate symbols in S are denoted by $\mathit{ifPreds}(S)$ and $\mathit{whilePreds}(S)$; their union is $\mathit{Preds}(S)$. A schema without predicates is called *predicate–free*.

Definition 2 (Linear Schemas)
If every element of $\mathcal{F} \cup \mathcal{P}$ does not appear more than once in S, then S is said to be *linear*.

If S is linear, we define $\mathit{Symbols}(S, p)$ to be the set of function and predicate symbols in the body of p in S (if p is a while predicate) or in the two parts of p (if p is an *if* predicate) in S. In the latter case we define $\mathit{Symbols}(S, p, \mathit{True})$, $\mathit{Symbols}(S, p, \mathit{False})$ to be the set of function and predicate symbols in its true and false parts, respectively. If S contains an assignment $y{:=}f(\mathbf{x})$ then we define $y = \mathrm{assign}_S(f)$, $\mathit{refvec}_S(f) = \mathbf{x}$ and the set of components of $\mathbf{x}$ is $\mathit{Refset}_S(f) \subseteq \mathcal{V}$. If $p \in \mathit{Preds}(S)$ then $\mathit{refvec}_S(p)$ and $\mathit{Refset}_S(p)$ are defined similarly.

Finally $|S|$ is defined as follows: $|S|$ is the total number of function, predicate symbols in S.

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*.

Definition 3 (State and Interpretation)
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 $p^i : D^m \to \{\mathit{True}, \mathit{False}\}$. The set of all interpretations with domain D will be denoted $\mathit{Int}(\mathcal{F}, \mathcal{P}, D)$.

An important special case, that suffices to determine the equivalence of schemas, is the domain $\mathit{Term}(\mathcal{F}, \mathcal{V})$ of all terms from $\mathcal{F}$ and $\mathcal{V}$.

Definition 4 (Terms)
The set $\mathit{Term}(\mathcal{F}, \mathcal{V})$ of terms is defined as follows:

(1) each variable is a term,
(2) 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.

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 arity of the vector term $\mathbf{t}$ is that of p. If t is a term, we define $\mathit{TermSymbols}(t) \subseteq \mathcal{F} \cup \mathcal{V}$ to be the set of function and variable symbols that it contains. We refer to a term $f(\mathbf{t})$ as an f-term. Let $F \in \mathcal{F}^*$; then we define an F-term recursively as follows; if $F = F'g$, for $g \in \mathcal{F}$ and $F' \in \mathcal{F}^*$, then t is an F-term if and only if t is a g-term, $t = g(\mathbf{t})$ say, and a component of the vector term $\mathbf{t}$ is an F'-term.

The notion of a Herbrand interpretation is now given. It is well known [7, Section 4-14] that these interpretations are the only ones that need to be considered when considering equivalence of schemas. This fact is stated more precisely in Theorem 18.

Definition 5 (Herbrand Interpretation)
An interpretation i is said to be *Herbrand* whenever the domain is $\mathit{Term}(\mathcal{F}, \mathcal{V})$ and if $f \in \mathcal{F}$ is a function symbol of arity n then

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

for all n-tuples of terms $(t_1, \ldots, t_n)$.

Definition 6 (Changing a Herbrand Interpretation)

Given a Herbrand interpretation i and $X \in \{\mathit{True}, \mathit{False}\}$ and $p \in \mathcal{P}$, the Herbrand interpretation $i(p = X)$ is given by

$$q^{i(p=X)}(\mathbf{t}) = \begin{cases} q^i(\mathbf{t}) & q \neq p \\ X & q = p \end{cases}$$

for every vector of terms $\mathbf{t}$ of the appropriate length and every $q \in \mathcal{P}$.

Definition 7 (The Natural State e)
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 $v \in \mathcal{V}$.

The execution of a program defines a finite or 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 8 (The set $\Pi^\omega(S)$ of paths through S)
If L is a finite 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. If σ is a word, or a set of words over an alphabet, then $\mathit{pre}(\sigma)$ is the set of all finite prefixes of (elements of) σ.

For each schema S the alphabet of S, written $\alpha(S)$ is defined by

$$\alpha(S) = A \cup B \cup C$$

where

$$\begin{aligned} A &= \{<y{:=}f(\mathbf{x})> \ \mid \ y{:=}f(\mathbf{x}) \text{ is an assignment in } S\} \\ B &= \{<p(\mathbf{x}) = \mathit{True}> \ \mid \ p(\mathbf{x}) \text{ is a predicate expression in } S\} \\ C &= \{<p(\mathbf{x}) = \mathit{False}> \ \mid \ p(\mathbf{x}) \text{ is a predicate expression in } S\}. \end{aligned}$$

The set, $\Pi(S) \subseteq (\alpha(S))^*$, of all finite paths of schema S is defined inductively as follows:

For assignments,

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

For sequences,

$$\Pi(\lambda) = \lambda.$$

$$\Pi(S_1 S_2 \ldots S_r) = \Pi(S_1) \ldots \Pi(S_r).$$

For *if* schemas,

$$\Pi(\mathit{if}\ \ p(\mathbf{x})\ \mathit{then}\ \ T_1\ \mathit{else}\ \ T_2)$$

is the set of all concatenations of $<p(\mathbf{x}) = \mathit{True}>$ with a word in $\Pi(T_1)$ and all concatenations of $<p(\mathbf{x}) = \mathit{False}>$ 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

$$[<q(\mathbf{y}) = \mathit{True}>\ \Pi(T)]^*\ <q(\mathbf{y}) = \mathit{False}>$$

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

$\Pi^\omega(S)$ is the set of finite and infinite paths of S. It is defined to be the set of all paths all of whose finite prefixes are prefixes of elements of $\Pi(S)$). Formally,

$$\Pi^\omega(S) = \{\sigma \in (\alpha(S))^\omega | \mathit{pre}(\sigma) \subseteq \mathit{pre}(\Pi(S))\}.$$

Elements of $\Pi^\omega(S)$ are called *paths* through S.

Definition 9 (Paths *passing through* a Function or Predicate Symbol)

We say that a path *passes* through a function symbol f (or a predicate p) if it contains an assignment with function symbol f (or $<p(\mathbf{x}) = \mathit{True}>$ or $<p(\mathbf{x}) = \mathit{False}>$).

Definition 10 (The Schema corresponding to a Path)
Given a path σ the predicate–free schema $\mathit{Atrace}(\sigma)$ consists of all the assignments along σ in the same order as in σ; and $\mathit{Atrace}(\sigma) = \lambda$ if σ has no assignments.

Lemma 11 *Let S be a schema; then if $\sigma \in \mathit{pre}(\Pi(S))$, the set $\{l \in \alpha(S) | \sigma l \in \mathit{pre}(\Pi(S))\}$ is one of the following:*

> *The empty set*
>
> *A singleton containing an assignment*
>
> *A pair $\{<p(\mathbf{x}) = \mathit{True}>, <p(\mathbf{x}) = \mathit{False}>\}$ where $p(\mathbf{x})$ is a predicate expression in S.*

Proof. This follows by induction on $|S|$.

Lemma 11 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.

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

Definition 12 (The Semantics of Predicate–free Schemas)
Let i be an interpretation and d an initial state, then the final state $\mathcal{M}[\![S]\!]_d^i$ of a schema S is defined as follows:

For assignments,

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

(where $d(\mathbf{x})$ is the tuple of terms $d(x)$ formed by applying d to each of the variable symbols x_k in $\mathbf{x}$)

for sequences,

$$\mathcal{M}[\![\lambda]\!]_d^i = d$$

and

$$\mathcal{M}[\![S_1 S_2]\!]_d^i \quad = \quad \mathcal{M}[\![S_2]\!]^i_{\mathcal{M}[\![S_1]\!]_d^i}.$$

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 13 (The Path $\pi(S, i, d)$ of a Predicate–free Schema)
Let i be an interpretation and d an initial state, then the path $\pi(S, i, d) \in \Pi^\omega(S)$ of a schema S is defined as follows:

For assignments,

$$\pi(y{:=}f(\mathbf{x}), i, d) \quad = \quad <y{:=}f(\mathbf{x})> .$$

For sequences,

$$\pi(\lambda, i, d) = \lambda.$$

and

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

This uniquely defines $\pi(S, i, d)$ if S is predicate–free.

Definition 14 ($\pi(S, i, d)$ where S is not predicate–free)
For a general schema S, we require that $\pi(S, i, d)$ to be in $\Pi^\omega(S)$ and that for every prefix σ of $\pi(S, i, d)$ ending in $<p(\mathbf{x}) = X>$, where X denotes *True* or

False and $p(\mathbf{x})$ a predicate expression in S, we have $p^i(\mathcal{M}[\![Atrace(\sigma)]\!]^i_d(\mathbf{x})) = X$. By Lemma 11, this defines the path $\pi(S,i,d) \in \Pi^\omega(S)$ uniquely.

We are now ready to define the semantics of a general schema S with respect to an initial state $d \neq \bot$ and an interpretation i.

Definition 15 (The Semantics of Schemas with Predicates)
If $\pi(S,i,d)$ is finite, we define

$$\mathcal{M}[\![S]\!]^i_d = \mathcal{M}[\![Atrace(\pi(S,i,d))]\!]^i_d$$

(which is already defined, since $Atrace(\pi(S,i,d))$ is predicate–free) otherwise $\pi(S,i,d)$ is infinite and we define $\mathcal{M}[\![S]\!]^i_d = \bot$.

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 16 (Terminating Interpretations)
If $\mathcal{M}[\![S]\!]^i_e \neq \bot$, then we say that i is a *terminating* interpretation for S.

Definition 17 (Equivalence of Schemas)
We say that schemas $S, T \in Sch(\mathcal{F},\mathcal{P},\mathcal{V})$ are *equivalent,* written $S \cong T$, if for every domain D and state $d : \mathcal{V} \to D$ and every $i \in Int(\mathcal{F},\mathcal{P},D)$ we have $\mathcal{M}[\![S]\!]^i_d = \mathcal{M}[\![T]\!]^i_d$ (including the case that $\mathcal{M}[\![S]\!]^i_d$ or $\mathcal{M}[\![T]\!]^i_d = \bot$).

The following theorem, which is a restatement of [7, Theorem 4-1], ensures that we only need to consider Herbrand interpretations and the natural state e.

Theorem 18 *Let $S \in Sch(\mathcal{F},\mathcal{P},\mathcal{V})$, and let D be a domain. Then for all states $d : \mathcal{V} \to D$ with $d \neq \bot$ and interpretations $i \in Int(\mathcal{F},\mathcal{P},D)$ the following holds.*

(1) We have $\pi(S,j,e) = \pi(S,i,d)$ for some Herbrand interpretation $j \in Int(\mathcal{F},\mathcal{P},Term(\mathcal{F},\mathcal{V}))$.
(2) If $T \in Sch(\mathcal{F},\mathcal{P},\mathcal{V})$ and for all Herbrand interpretations j we have $\mathcal{M}[\![S]\!]^j_e = \mathcal{M}[\![T]\!]^j_e$, then $S \cong T$.

Throughout the remainder of the paper, all interpretations will be assumed to be Herbrand. For convenience, if i is a Herbrand interpretation we define $\pi(S,i) = \pi(S,i,e)$. Also, if S is predicate–free and $d : \mathcal{V} \to Term(\mathcal{F},\mathcal{V})$ is a state then we define unambiguously $\mathcal{M}[\![S]\!]_d = \mathcal{M}[\![S]\!]^i_d$.

Definition 19 (Free Schemas)

Let $S \in Sch(\mathcal{F}, \mathcal{P}, \mathcal{V})$. If for every path $\sigma \in \Pi^\omega(S)$ there exists a domain D, an interpretation $i \in Int(\mathcal{F}, \mathcal{P}, D)$ and a state $d \in \mathrm{State}(\mathcal{V}, D)$, such that $\sigma = \pi(S, i, d)$, then S is said to be *free.*

Definition 20 (Conservative Schemas)
Let $S \in Sch(\mathcal{F}, \mathcal{P}, \mathcal{V})$. If in every assignment $y{:=}f(\mathbf{x})$ of S, the variable y is one of the components of $\mathbf{x}$, then S is said to be *conservative.* We refer to a schema which is conservative, free and linear as a CFL schema.

Proposition 21 *If $S\, z{:=}h(\mathbf{c})$ is any conservative predicate–free schema, $x \in \mathcal{V}$ and $\mathcal{M}[\![S]\!]_e(x)$ is an F-term for some $F \in \mathcal{F}^*$, then $\mathcal{M}[\![S\, z{:=}h(\mathbf{c})]\!]_e(x) = \mathcal{M}[\![S]\!]_e(x)$ if $z \neq x$; if $z = x$ then $\mathcal{M}[\![S\, z{:=}h(\mathbf{c})]\!]_e(x)$ is an Fh-term.*

Proof. This follows from Definition 12 and the fact that z is a component of $\mathbf{c}$, since $S\, z{:=}h(\mathbf{c})$ is conservative.

Proposition 22 *Let S be a free linear schema containing a subschema*

$$\textit{while } p(\mathbf{x})\, \textit{do } T.$$

Then every finite path $\sigma \in \Pi(T)$ passes through a function symbol $f \in \mathcal{F}$ with $\mathrm{assign}_T(f) \in Refset_S(p)$.

Proof. Suppose $\sigma \in \Pi(T)$ does not satisfy this condition. Since S is free, there is a path ρ $<p(\mathbf{x}) = True>$ σ $<p(\mathbf{x}) = False>$ ρ' $\in \pi(S, i)$ for some interpretation i. Thus $p^i(\mathcal{M}[\![Atrace(\rho)]\!]_e(\mathbf{x})) = True$ and $p^i(\mathcal{M}[\![Atrace(\rho\sigma)]\!]_e(\mathbf{x})) = False$. But if the path σ does not contain an assignment to any component of $\mathbf{x}$, then $\mathcal{M}[\![Atrace(\rho)]\!]_e(\mathbf{x}) = \mathcal{M}[\![Atrace(\rho\sigma)]\!]_e(\mathbf{x})$, giving a contradiction.

Proposition 23 *If S is a free schema, then each of its subschemas is either λ or contains a function symbol.*

Proof. This follows by induction on $|S|$, using the fact that the body of a free while schema in S contains at least one assignment, by Proposition 22.

4 Paths and Interpretations Passing Through Symbols

This section defines what it means for a schema to pass semantically through a predicate.

Definition 24 (*PathSymbols*)
Let $S \in Sch(\mathcal{F}, \mathcal{P}, \mathcal{V})$ be a schema and let $\mu \in \Pi^\omega(S)$ be a path through S. We define $PathSymbols(\mu) \subseteq \mathcal{F} \cup \mathcal{P}$ to be the set of all elements of $\mathcal{F} \cup \mathcal{P}$ through which μ passes.

Definition 25 (*Passpred*)
An interpretation i passes through a predicate symbol p in a schema S if i terminates for S and there exists an interpretation j, differing from i only at p such that $\mathcal{M}[\![S]\!]_e^i \neq \mathcal{M}[\![S]\!]_e^j$. We write $Passpred(S,p)$ for the set of interpretations passing through p in S.

Definition 26 (*Passfunc*)
A terminating interpretation i passes through a *function* symbol $f \in \mathcal{F}$ in a schema S if $f \in TermSymbols(\mathcal{M}[\![S]\!]_e^i(x))$ for some variable $x \in \mathcal{V}$. We write $Passfunc(S,f)$ for the set of interpretations passing through f in S.

Observe that the preceding two definitions are given in terms of $\mathcal{M}[\![S]\!]_e^i$, and hence depend only on the equivalence class of S.

Lemma 27 *Let S be a linear schema and let $p \in Preds(S)$. Let the interpretation i be terminating for S.*

(1) If $i \in Passpred(S,p)$, then $p \in PathSymbols(\pi(S,i))$.
(2) If $p \in whilePreds(S) \cap PathSymbols(\pi(S,i))$ then $i \in Passpred(S,p)$ and $\mathcal{M}[\![S]\!]_e^{i(p=True)} = \perp$.

Proof.

(1) If $p \notin PathSymbols(\pi(S,i))$ then changing the interpretation i only at p will not change the path $\pi(S,i)$, and hence will not change $\mathcal{M}[\![S]\!]_e^i$; thus $i \notin Passpred(S,p)$, giving a contradiction.
(2) Now assume that $p \in whilePreds(S) \cap PathSymbols(\pi(S,i))$; thus the path $\pi(S,i)$ contains a letter $<p(\mathbf{x}) = X>$ and so we may write $\sigma' <p(\mathbf{x}) = X> \in pre(\pi(S,i))$ for some word $\sigma' \in (\alpha(S))^*$ not containing a letter $<p(\mathbf{x}) = Y>$. Thus $\sigma' <p(\mathbf{x}) = True> \in pre(\pi(S,i(p = True)))$. Thus $\pi(S,i(p = True))$ contains the letter $<p(\mathbf{x}) = True>$ but not the letter $<p(\mathbf{x}) = False>$, hence $\pi(S,i(p = True))$ is not finite and so $\mathcal{M}[\![S]\!]_e^{i(p=True)} = \perp \neq \mathcal{M}[\![S]\!]_e^i$ and thus $i \in Passpred(S,p)$.

Part (2) of Lemma 29 gives the corresponding statement for function symbols.

Remark 28 If S is a linear schema and i is an interpretation, then $p \in ifPreds(S) \cap PathSymbols(\pi(S,i))$ does not imply $i \in Passpred(S,p)$. To see this, let S be the schema

$$
\begin{array}{l}
\textit{if } \; p(x) \\
\textit{then } \; \textit{if } \; q(x) \\
\qquad \quad \textit{then } \; x{:=}f(x) \\
\qquad \quad \textit{else } \; \lambda \\
\textit{else } \; \textit{if } \; r(x) \\
\qquad \quad \textit{then } \; y{:=}g(y) \\
\qquad \quad \textit{else } \; \lambda
\end{array}
$$

and let i be an interpretation such that q^i, r^i always map to *False*. Then for all interpretations j that differ from i only at p, the final state $\mathcal{M}[\![S]\!]^j_e = \mathcal{M}[\![S]\!]^i_e = e$. Consequently, by definition, i does not pass through p in S.

Lemma 29 *Let S be a conservative schema.*

(1) If S is predicate–free, $S = S'\,x{:=}f(\mathbf{a})\,S''\,y{:=}g(\mathbf{b})\,S'''$ and x is a component of $\mathbf{b}$ then there are words $F, G \in \mathcal{F}^*$ such that $\mathcal{M}[\![S]\!]_e(y)$ is an $fFgG$-term.

(2) Let i be a terminating interpretation and let $f \in \mathcal{F}$. Then

$$i \in \mathit{Passfunc}(S, f) \iff f \in \mathit{PathSymbols}(\pi(S, i)).$$

Proof.

(1) By induction on $|S''|$, and using Proposition 21, it follows that there is a word $F \in \mathcal{F}^*$ such that $\mathcal{M}[\![S'\,x{:=}f(\mathbf{a})\,S'']\!]_e(x)$ is an fF-term. Thus $\mathcal{M}[\![S'\,x{:=}f(\mathbf{a})\,S''\,y{:=}g(\mathbf{b})]\!]_e(y)$ is an fFg-term. Hence there exists $G \in \mathcal{F}^*$ such that $\mathcal{M}[\![S]\!]_e(y)$ is an $fFgG$-term, by Proposition 21 and using induction on $|S'''|$.

(2) Observe that we have to prove that $\mathcal{F} \cap \mathit{TermSymbols}(\mathcal{M}[\![S]\!]^i_e(x)) = \mathcal{F} \cap \mathit{PathSymbols}(\pi(S, i))$. If S is predicate–free then this follows from Part (1) of this Lemma. For the general case, let $\mu = \pi(S, i)$ and observe that

$$\mathcal{F} \cap \mathit{PathSymbols}(\mu) = \mathcal{F} \cap \mathit{PathSymbols}(\pi(\mathit{Atrace}(\mu), i))$$

and $\mathit{TermSymbols}(\mathcal{M}[\![S]\!]^i_e(v)) = \mathit{TermSymbols}(\mathcal{M}[\![\mathit{Atrace}(\mu)]\!]^i_e(v))$ and hence the result follows from the restricted case applied to $\mathit{Atrace}(\mu)$.

5 Equivalence is Decidable for Predicate-Free Conservative Schemas

In this section we consider conservative predicate–free schemas. This is important because deciding equivalence for the more general classes of schemas depends on establishing conditions which characterize equivalence of schemas from this restricted class of schemas.

Definition 30 (Commuting Assignments)
We say that two assignments $x{:=}f(\mathbf{a})$ and $y{:=}g(\mathbf{b})$ *commute* if and only if x is not a component of $\mathbf{b}$, y is not a component of $\mathbf{a}$, and $x \neq y$.

Observe that if $x{:=}f(\mathbf{a})$ and $y{:=}g(\mathbf{b})$ are both conservative then the last condition $x \neq y$ is redundant in the preceding definition.

Lemma 31 *If $x{:=}f(\mathbf{a})$ and $y{:=}g(\mathbf{b})$ commute then*

$$x{:=}f(\mathbf{a})\, y{:=}g(\mathbf{b}) \quad \cong \quad y{:=}g(\mathbf{b})\, x{:=}f(\mathbf{a})\,.$$

Proposition 32 *Let S be a linear predicate–free schema. If $S = S_1 S_2$ and $\mathcal{M}[\![S_1]\!]_e(x) = \mathcal{M}[\![S]\!]_e(y)$, then $x = y$ and there are no assignments to y in S_2.*

Proof. Suppose $\mathcal{M}[\![S_1]\!]_e(x)$ is an f-term; then $x = \text{assign}_{S_1}(f)$ and so S_1 contains an f-assignment. Since S is linear, there is no f-assignment in S_2. Since $\mathcal{M}[\![S]\!]_e(y)$ is also an f-term, $y = \text{assign}_S(f) = \text{assign}_{S_1}(f) = x$ and there is no assignment to y in S after the f-assignments, hence S_2 has no assignments to y.

The following result is in fact true without the linearity hypothesis, but we do not need this stronger form of the Theorem.

Theorem 33 *Let S and T be conservative, linear and predicate–free schemas and assume that $S \cong T$. Then S can be obtained from T by finitely many interchanges of two adjacent commuting assignments.*

Proof. The result follows by induction on $|S|$. If $S = \lambda$ then also $T = \lambda$, and the result is immediate. Thus we may assume that

$$S = S'\, x{:=}f(\mathbf{a}).$$

Hence there is a vector term $\mathbf{t}$ such that $\mathcal{M}[\![T]\!]_e(x) = \mathcal{M}[\![S]\!]_e(x) = f(\mathbf{t})$ and so we may write

$$T = T'\, x{:=}f(\mathbf{b})\, T''$$

where T'' does not contain an assignment to x. We will show that the assignment $x{:=}f(\mathbf{b})$ commutes with every assignment in T''; thus $x{:=}f(\mathbf{b})\, T'' \cong T'' x{:=}f(\mathbf{b})$ from Lemma 31 and hence $T \cong T'\, T''\, x{:=}f(\mathbf{b})$. We will also show that $\mathbf{a} = \mathbf{b}$ and $S' \cong T'T''$ and then the result follows from the inductive hypothesis applied to S'.

No assignment in T'' references x. To see this, assume that there is an assignment $y{:=}g(\mathbf{c})$ in T'' which references x. Then $\mathcal{M}[\![S]\!]_e(y) = \mathcal{M}[\![T]\!]_e(y)$ is an $fFgG$-term for $F, G \in \mathcal{F}^*$, by Lemma 29, Part (1) and so $\mathcal{M}[\![S']\!]_e(y)$ is an $fFgG$-term using Proposition 21, since $x \neq y$. But this is impossible since S is linear and so $f \notin \mathit{Funcs}(S')$.

Let a_i, b_i, t_i be the ith components of the vectors $\mathbf{a}, \mathbf{b}, \mathbf{t}$. Since T is conservative, there is some $n \leq \mathit{arity}(f)$ such that $x = b_n$.

We now show that $a_i = b_i$ for all i, and hence $a_n = x$. Let $i \leq \mathit{arity}(f)$. Observe that

$$\mathcal{M}[\![S']\!]_e(a_i) = t_i = \mathcal{M}[\![T']\!]_e(b_i). \tag{1}$$

Suppose $a_i \neq x$. Then by Proposition 21 $\mathcal{M}[\![S']\!]_e(a_i) = \mathcal{M}[\![S]\!]_e(a_i) = \mathcal{M}[\![T]\!]_e(a_i)$ and so by (1) and Proposition 32 applied to T, it follows that $a_i = b_i$ and there is no assignment to a_i in T''. On the other hand, suppose $a_i = x = a_n$. Then from (1) we get $t_i = t_n$ and so $\mathcal{M}[\![T']\!]_e(b_i) = \mathcal{M}[\![T']\!]_e(b_n)$. By Proposition 32 $b_i = b_n = x$.

Thus we have shown that $\mathbf{a} = \mathbf{b}$ and $x{:=}f(\mathbf{b})$ commutes with every assignment in T''.

It remains to show that $S' \cong T'T''$. If $y \neq x$ then by Proposition 21

$$\mathcal{M}[\![S']\!]_e(y) = \mathcal{M}[\![S]\!]_e(y) = \mathcal{M}[\![T]\!]_e(y) = \mathcal{M}[\![T'\,T'']\!]_e(y),$$

and

$$\mathcal{M}[\![S']\!]_e(x) = t_m = \mathcal{M}[\![T'\,T'']\!]_e(x),$$

giving the result.

6 Equivalent Schemas have Identical Predicate Sets

In this section, it is shown that two equivalent CFL schemas have the same set of predicates and that corresponding predicates are of the same type (*if* or *while*) in each schema. A summary of the proof is now given:-

Summary

A path through S is *unitary* if it does not enter the body of any *while* schema more than once. Similarly we define an interpretation i to be unitary with respect to a set of predicate symbols Q if and only if for all q in Q, there is at most one predicate term for which outermost symbol is q that gets mapped by i to *True*.

Let $whilePreds(S)$ be the set of *while* predicate symbols occurring in S. In Lemma 36 we show

(1) If an interpretation i is unitary with respect to $whilePreds(S)$ then it terminates and the path $\pi(S, i)$ is unitary.
(2) If a path $\sigma \in \Pi(S)$ is unitary then there exists a unitary interpretation i with respect to $\mathcal{P}$ satisfying $\sigma = \pi(S, i, e)$

where S is a free, linear schema.

From this result and other results in Section 4 we prove Proposition 37, which states that for all predicate symbols p in S,

(1) $p \in whilePreds(S)$ if and only if for every interpretation $i \in Passpred(S, p)$, the interpretation $i(p = True)$ does not terminate.
(2) For all $p \in Preds(S)$ there exists an interpretation in $Passpred(S, p)$.

where S is a CFL Schema.

From this it immediately follows that equivalent CFL schemas, S and T have the same set of *while* and *if* predicates, since (1) shows that equivalent CFL schemas have the same set of *while* predicates and (2) shows that equivalent CFL schemas have the same set of predicates (*if*s and *while*s together).

The main result of this section is Theorem 38. This Theorem is the first strong statement concerning the similarity in the structure of two equivalent schemas.

Definition 34 (Unitary Paths)
If a schema S is linear, we say that a path through S is *unitary* if it does not enter the body of any *while* schema more than once (that is, for all *while* predicates p, no letter $<p(\mathbf{x}) = True>$ occurs more than once in the path).

If a path through a linear schema is unitary, then no if predicate and no assignment occurs more than once in the path.

Definition 35 (Unitary Interpretations with respect to Predicates)

Let $Q \subseteq \mathcal{P}$ be a set of predicate symbols and let i be an interpretation. We say that i is *Unitary* with respect to Q if for every $q \in Q$ we have $q^i(\mathbf{t}) = True$ for at most one vector term $\mathbf{t}$.

Lemma 36 *Let S be a free, linear schema.*

(1) If an interpretation i is unitary with respect to $whilePreds(S)$ then it

terminates and the path $\pi(S,i)$ is unitary.

(2) If a path $\sigma \in \Pi(S)$ is unitary then there exists a unitary interpretation i with respect to $\mathcal{P}$ satisfying $\sigma = \pi(S,i)$.

Proof.

(1) Let $\sigma = \pi(S,i)$ and suppose that the path σ is not unitary; thus we may write
$$\sigma' <p(\mathbf{x}) = \mathit{True}> \sigma'' <p(\mathbf{x}) = \mathit{True}> \in \mathit{pre}(\sigma)$$
for some $p \in \mathit{whilePreds}(S)$. Hence
$$p^i(\mathcal{M}[\![\mathit{Atrace}(\sigma')]\!]_e(\mathbf{x})) = p^i(\mathcal{M}[\![\mathit{Atrace}(\sigma' <p(\mathbf{x}) = \mathit{True}> \sigma'')]\!]_e(\mathbf{x})) = \mathit{True}.$$
Since i is unitary with respect to $\mathit{whilePreds}(S)$, we have
$$\mathcal{M}[\![\mathit{Atrace}(\sigma')]\!]_e(\mathbf{x}) = \mathcal{M}[\![\mathit{Atrace}(\sigma' <p(\mathbf{x}) = \mathit{True}> \sigma'')]\!]_e(\mathbf{x}).$$
Thus there is no interpretation j for which
$$\sigma' <p(\mathbf{x}) = \mathit{True}> \sigma'' <p(\mathbf{x}) = \mathit{False}> \in \mathit{pre}(\pi(S,j)),$$
contradicting freeness since $\sigma' <p(\mathbf{x}) = \mathit{True}> \sigma'' <p(\mathbf{x}) = \mathit{False}> \in \mathit{pre}(\Pi(S))$ by Lemma 11 applied to $\sigma' <p(\mathbf{x}) = \mathit{True}> \sigma''$. Thus σ is unitary and hence i terminates.

(2) Given such a path σ, let i be an interpretation satisfying $\sigma = \pi(S,i)$. For any predicate symbol p, the path σ does not contain more than one occurrence of $<p(\mathbf{x}) = \mathit{True}>$, and so a maximum of one vector term needs to map to True under p^i, and so i may be assumed to be unitary with respect to $\mathcal{P}$.

The following result is in fact true without the linearity hypothesis, but then the proof is somewhat more complicated.

Proposition 37 *Let S be a CFL schema and let $p \in \mathit{Preds}(S)$.*

(1) If for every interpretation i in $\mathit{Passpred}(S,p)$, the interpretation $i(p = \mathit{True})$ does not terminate, then $p \in \mathit{whilePreds}(S)$.

(2) There exists an interpretation in $\mathit{Passpred}(S,p)$.

Proof.

(1) Assume that $p \in \mathit{ifPreds}(S)$. Thus p has a function symbol f in one of its parts; its X-part, say. Let $\{X,Y\} = \{\mathit{True},\mathit{False}\}$. Let $\sigma \in \Pi(S)$ be a unitary path passing through f. By Lemma 36, there exists an interpretation i, unitary with respect to $\mathit{whilePreds}(S)$, satisfying $\pi(S,i) = \sigma$. By Part (2) of Lemma 29, $i \in \mathit{Passfunc}(S,f)$. Clearly $i(p =$

True) is unitary with respect to $whilePreds(S)$ and hence terminates and $i(p = Y) \notin Passfunc(S, f)$ and so $i \in Passpred(S, p)$, as required.

(2) If $p \in ifPreds(S)$ this follows from Part (1) of the Proposition. If $p \in whilePreds(S)$ this follows from the freeness of S and Part (2) of Lemma 27.

We can use Proposition 37 and Lemma 27 to show that equivalent CFL schemas have the same set of while and if predicates.

Theorem 38 *Let S, T be equivalent CFL schemas. Then*

$$ifPreds(S) = ifPreds(T)$$

and

$$whilePreds(S) = whilePreds(T).$$

Proof.

If $p \in Preds(S)$, then by Part (2) of Proposition 37, there exists an interpretation passing through p in S, and hence in T, since $S \cong T$; thus $Preds(S) \subseteq Preds(T)$ and equality similarly holds.

If $p \in ifPreds(S)$, then by Part (1) of Proposition 37, there exists an interpretation i passing through p in S such that $i(p = True)$ terminates for S. But these statements hold with T replacing S, since $S \cong T$, and so by both parts of Lemma 27, $p \notin whilePreds(T)$.

Thus $ifPreds(S) \subseteq ifPreds(T)$ and equality similarly holds. Since

$$whilePreds(S) = Preds(S) - ifPreds(S)$$

and

$$whilePreds(T) = Preds(S) - ifPreds(T),$$

the results follow.

7 Equivalent Schemas have Identical Symbol Sets in Loop Bodies

The main result of this section, Theorem 41, extends Theorem 38 in that it gives information about the symbols lying in the body of a *while* predicate of two equivalent schemas. The result is first sketched in outline and then presented in detail.

Summary

Let S be a linear schema, let $p \in \mathit{whilePreds}(S)$ and let i be an interpretation which passes through p. We say that i passes *truly* through p (in S) if the meanings of S with respect to i and $i(p = \mathit{False})$ are different.

In Lemma 40, the main result of this section, it is proved that if x is a function or predicate symbol then x is in the body of *while* predicate, p, of a CFL schema if and only if

(1) there exists an interpretation passing through x, and
(2) any interpretation passing through x passes truly through p.

From this it immediately follows that corresponding *while* loops of equivalent CFL schemas contain the same symbols.

Definition 39 (True Passing through a While Predicate Symbol) Let S be a linear schema, let $p \in \mathit{whilePreds}(S)$ and let i be an interpretation which passes through p. We say that i passes *truly* through p (in S) if $\mathcal{M}[\![S]\!]_e^{i(p=\mathit{False})} \neq \mathcal{M}[\![S]\!]_e^i$.

Clearly, if $S \cong T$ and i passes truly through p in S, then i passes truly through p in T.

Lemma 40 *Let $x \in \mathit{Funcs}(S) \cup \mathit{Preds}(S)$ be a symbol in a CFL schema S and let $p \in \mathit{whilePreds}(S)$. Then $x \in \mathit{Symbols}(S,p)$ if and only if there exists an interpretation passing through x, and any interpretation passing through x passes truly through p.*

Proof.

($\Rightarrow$) If $x \in \mathcal{F}$, by Lemma 29, Part (2), and the freeness of S, there exists an interpretation passing through x. If $x \in \mathcal{P}$, the same conclusion follows from Part 2 of Proposition 37.

To prove the second assertion, assume that an interpretation i passes through $x \in \mathit{Symbols}(S,p)$. Hence $x \in \mathit{PathSymbols}(\pi(S,i))$. Thus we may write

$$\pi(S,i) = \sigma' <p(\mathbf{x}) = \mathit{True}> \sigma'' <p(\mathbf{x}) = \mathit{False}> \sigma''' \in \Pi(S),$$

where the word σ'' does not contain a letter $<p(\mathbf{x}) = \mathit{True}>$, and hence is a path through the body of p. Since S is free, by Proposition 22 there is an assignment $a{:=}f(\mathbf{b})$ in σ''. Hence $f \in \bigcup_{v \in \mathcal{V}} \mathit{TermSymbols}(\mathcal{M}[\![S]\!]_e^i(v))$ by Lemma 29, Part (2). But $f \in \mathit{Symbols}(S,p)$ and so

$$f \notin \bigcup_{v \in \mathcal{V}} \mathit{TermSymbols}(\mathcal{M}[\![S]\!]_e^{i(p=\mathit{False})}(v)),$$

so i passes truly through p.

($\Leftarrow$) Clearly the schema S contains x. Assume that $x \notin \mathit{Symbols}(S,p)$. We will find an interpretation $i = i(p = \mathit{False})$ passing through x, giving a contradiction.

If $x \in \mathcal{F}$ or x is a *while* predicate then by the freeness of S there exists a finite path $\sigma \in \Pi(S)$ passing through x and not containing a letter $<p(\mathbf{x}) = \mathit{True}>$, and so there exists an interpretation, i with $\sigma = \pi(S,i)$ and $i = i(p = \mathit{False})$. If $x \in \mathcal{F}$ then i passes through x by Lemma 29, Part (2); if $x \in \mathit{whilePreds}(S)$, then this follows from Part (2) of Lemma 27.

If $x \in \mathit{ifPreds}(S)$ and $p \notin \mathit{Symbols}(S,x)$, then the same argument is valid if σ is chosen to be a path which passes through a function symbol in one of the parts of x (by Proposition 23, there must be one) and does not contain the letter $<p(\mathbf{x}) = \mathit{True}>$.

Lastly, if x is an *if* predicate and (say) $p \in \mathit{Symbols}(S,x,\mathit{True})$, then the interpretation i is found as follows:

Let $\sigma \in \Pi(S)$ be a unitary path passing through p and let j be a unitary interpretation with respect to $\mathit{whilePreds}(S)$ and satisfying $\pi(S,j) = \sigma$. Then $i = j(p = \mathit{True})(x = \mathit{False})$ satisfies $\perp \neq \mathcal{M}[\![S]\!]_e^{i(p=\mathit{False})} = \mathcal{M}[\![S]\!]_e^{i}$ since $i(p = \mathit{False})$ is unitary with respect to $\mathit{whilePreds}(S)$ and $p \notin \mathit{PathSymbols}(\pi(S,i))$; and i passes through x since it terminates and $i(x = \mathit{True}) = j(p = \mathit{True})(x = \mathit{True})$ does not terminate, by Lemma 27, Part (1), again giving a contradiction.

A similar argument is valid if $p \in \mathit{Symbols}(S,x,\mathit{False})$.

Theorem 41 *Let S, T be equivalent CFL schemas and let $p \in \mathit{whilePreds}(S)$. Then*

$$\mathit{Symbols}(S,p) = \mathit{Symbols}(T,p).$$

Proof.
This follows from Lemma 40.

8 Standardised CFL schemas: A Canonical Form

The main result of this section, Theorem 49, is the counterpart of Theorem 41 for if predicates.

Lemma 42 *Let S be a CFL schema with $p \in \mathit{ifPreds}(S)$ and let $\{X,Y\} = \{\mathit{True},\mathit{False}\}$. A symbol $x \in \mathcal{F} \cup \mathit{whilePreds}(S)$ lies in $\mathit{Symbols}(S,p,X)$ if and only if there exists an interpretation i which is unitary with respect to $\mathit{whilePreds}(S)$ and passes through x, and if an interpretation k satisfies $k = k(p = Y)$ then k does not pass through x.*

Proof.

($\Rightarrow$) Let $\sigma \in \Pi(S)$ be a unitary path with $x \in \mathit{PathSymbols}(\sigma)$. By Lemma 36, Part 2, there exists an interpretation i, unitary with respect to $\mathit{whilePreds}(S)$ and satisfying $\sigma = \pi(S, j)$. We may assume that $i = i(p = X)$ since the path σ has exactly one occurrence of a letter $<p(\mathbf{x}) = X>$, and none of $<p(\mathbf{x}) = Y>$. By Lemma 29, Part (2) (if $x \in \mathcal{F}$) or Part (2) of Lemma 27 (if $x \in \mathit{whilePreds}(S)$), the interpretation i passes through x.

On the other hand, clearly, any interpretation k such that $k = k(p = Y)$ does not pass through x.

($\Leftarrow$) Clearly S contains the symbol x. If $x \notin \mathit{Symbols}(S, p, X)$, then there exists a unitary path $\sigma \in \Pi(S)$ passing through x and not entering the X-part of p. Let k be an interpretation which is unitary with respect to $\mathit{whilePreds}(S)$ and satisfies $\sigma = \pi(S, k)$. We may assume $k = k(p = Y)$ since the path σ has no occurrences of the form $<p(\mathbf{x}) = X>$, giving a contradiction.

Theorem 43 *Let S, T be equivalent CFL schemas. Then*

$$p \in \mathit{ifPreds}(S) \text{ and } X \in \{\mathit{True}, \mathit{False}\}$$

$$\Longrightarrow$$

$$\begin{gathered} \mathcal{F} \cap \mathit{Symbols}(S, p, X) = \mathcal{F} \cap \mathit{Symbols}(T, p, X) \\ \textit{and} \\ \mathit{whilePreds}(S) \cap \mathit{Symbols}(S, p, X) = \mathit{whilePreds}(S) \cap \mathit{Symbols}(T, p, X). \end{gathered}$$

Proof.
This follows from Lemma 42.

Before attempting to extend Lemma 42 to allow $x \in \mathit{ifPreds}(S)$, we have to deal with the fact that this Lemma is false under this hypothesis; indeed, the schemas:-

$$\begin{array}{lll} \textit{if} \;\; p(\mathbf{x}) & & \\ \textit{then} & \textit{if} \;\; q(\mathbf{y}) & \\ & \textit{then} & S' \\ & \textit{else} & \lambda \\ \textit{else} \;\; \lambda & & \end{array}$$

and

$$\begin{array}{lll} \textit{if} \;\; q(\mathbf{y}) & & \\ \textit{then} & \textit{if} \;\; p(\mathbf{x}) & \\ & \textit{then} & S' \\ & \textit{else} & \lambda \\ \textit{else} \;\; \lambda & & \end{array}$$

are equivalent so $S \cong T$ does not imply

$$\mathit{Symbols}(S, p, \mathit{True}) = \mathit{Symbols}(T, p, \mathit{True})$$

for schemas, S, T, containing the two above respectively.

Definition 44 (Ambiguous Predicates)
If S is a schema and $p, q \in \mathit{ifPreds}(S)$ and $q \in \mathit{Symbols}(S, p)$ and

$$\mathcal{F} \cap \mathit{Symbols}(S, p) = \mathcal{F} \cap \mathit{Symbols}(S, q)$$

and one part of q is λ and every predicate p' satisfying

$$p' \in \mathit{Symbols}(S, p) \text{ and } q \in \mathit{Symbols}(S, p')$$

is an *if* predicate, then p is called an ambiguous predicate.

We call q a *non–initial* predicate of the ambiguous predicate p.

This motivates the following definition, which eliminates this problem.

Definition 45 (Standardised Schemas)
We assume a total ordering $\triangleleft$ on the set $\mathcal{P}$. A schema S is *standardised* if it satisfies the following condition; if S has an if predicate p, and one part of p is λ and the other is an if schema guarded by q, one of whose parts is λ, then $p \triangleleft q$.

Every schema can be replaced by an equivalent standardised schema in finitely many steps each of which consists of interchanging predicates and possibly interchanging true and false parts of predicates.

Definition 46 (Indecomposable Schemas)
A schema S is said to be *indecomposable* if it cannot be expressed as $S = S_1 S_2$ unless S_1 or $S_2 = \lambda$; that is, S is either atomic, an *if* schema or a *while* schema.

Lemma 47 (Nesting of If Predicates)
If p is an ambiguous predicate of a CFL schema S and q is a non–initial predicate of p then there exists a sequence of if *predicates $p_0 = p, p_1, \ldots, p_n = q \in \mathit{ifPreds}(S)$ satisfying $\{p_1, \ldots, p_n\} = \mathit{Symbols}(S, p) - \mathit{Symbols}(S, q)$ and one part of each p_j for $j < n$ is λ and the other is the* if *schema of S guarded by p_{j+1}. Also, if S is standardised then $p_j \triangleleft p_{j+1}$ for all $j < n$.*

Proof.
For every predicate r of S, let S_r be the subschema of which it is the guard. Note that every subschema of S is either λ or contains an element of $\mathcal{F}$. Thus $\mathcal{F} \cap \mathit{Symbols}(S, p) = \mathcal{F} \cap \mathit{Symbols}(S, q)$ implies that one part of p is λ. Let S' be the other part, which clearly contains q.

The result follows by induction on $|Symbols(S,p)|$. If $S' = S_q$, the result is immediate, so we assume that this is false. Necessarily S' is indecomposable, since if $S' = S_1 S_2$ with S_1 containing q, say, then S_2 would contain an element of $\mathcal{F} \cap Symbols(S,p) - \mathcal{F} \cap Symbols(S,q) = \emptyset$. Thus $S' = S_{p_1}$ for a predicate p_1 which according to the hypotheses, must be an *if* predicate. Also, the part of p_1 not containing q must be λ, since $\mathcal{F} \cap Symbols(S,p) = \mathcal{F} \cap Symbols(S,q)$, and so $p \lhd p_1$ if S is standardised. The result now follows from the inductive hypotheses applied to p_1 and q.

We now extend Theorem 43 to all predicates for equivalent standardised CFL schemas (Theorem 49).

Lemma 48 *Let S, T be equivalent standardised CFL schemas. Then*

$$p \in ifPreds(S) \text{ and } X \in \{True, False\}$$

$$\Longrightarrow$$

$$ifPreds(S) \cap Symbols(S,p,X) = ifPreds(S) \cap Symbols(T,p,X).$$

Proof.

The equality

$$\mathcal{F} \cap Symbols(S,p,X) = \mathcal{F} \cap Symbols(T,p,X) \qquad (1)$$

follows from Theorem 43. We now use (1) to prove the corresponding result for intersections with $ifPreds(S)$.

Now assume that $q \in ifPreds(S)$ and $q \neq p$. Write $\{X,Y\} = \{True, False\}$. Assume that

$$q \in Symbols(S,p,X) - Symbols(T,p,X); \qquad (2)$$

thus

$$\mathcal{F} \cap Symbols(S,q) \subseteq \mathcal{F} \cap Symbols(S,p,X)$$

and so

$$\mathcal{F} \cap Symbols(T,q) \subseteq \mathcal{F} \cap Symbols(T,p,X) \qquad (3)$$

using (1) and Theorem 43.

Recall that

$$\mathcal{F} \cap Symbols(T,q) \neq \emptyset,$$

since the parts of q are not both λ.

Observe that

$$Symbols(T,p) \cap Symbols(T,q) \neq \emptyset \Rightarrow p \in Symbols(T,q) \text{ or } q \in Symbols(T,p)$$

from the geometry of a linear schema.

If $q \in \mathit{Symbols}(T, p)$ then $q \in \mathit{Symbols}(T, p, Y)$ by (2).

Thus

$$\emptyset \neq \mathcal{F} \cap \mathit{Symbols}(T, q) \subseteq \mathcal{F} \cap \mathit{Symbols}(T, p, Y),$$

contradicting (3) and linearity of T and so

$$p \in \mathit{Symbols}(T, q) \tag{4}$$

by the observation above.

Thus the inclusion in (3) is not strict and so

$$\mathcal{F} \cap \mathit{Symbols}(T, q) = \mathcal{F} \cap \mathit{Symbols}(T, p, X). \tag{5}$$

From (4) and (5), and using Proposition 23, the part of q not containing p and the Y-part of p must be λ in T (similarly, the Y–part of p in S is λ and one part of q in S is λ, by (1) applied to p and q). If there were a *while* predicate p' satisfying $p' \in \mathit{Symbols}(S, p)$, $q \in \mathit{Symbols}(S, p')$, then $q \in \mathit{Symbols}(T, p, X)$, follows from Theorem 43 and Theorem 41 applied to p'. This is a contradiction, since we are assuming that $q \notin \mathit{Symbols}(T, p, X)$. There is thus, no such *while* predicate p' and therefore p is an ambiguous predicate in S and q is a non–initial predicate in p. Thus by Lemma 47, we have $p \lhd q$.

Similarly for T, if there were a *while* predicate p' satisfying $p' \in \mathit{Symbols}(T, q)$, and $p \in \mathit{Symbols}(T, p')$, then by Theorem 41 applied to p' and Theorem 43 applied to q, (3) would be false; Therefore q is an ambiguous predicate in T and p is a non–initial predicate in q and hence by Lemma 47, we have $q \lhd p$, giving a contradiction.

We have shown that $q \in \mathit{Symbols}(S, p, X) \Rightarrow q \in \mathit{Symbols}(T, p, X)$ and clearly the converse holds. This proves the Lemma.

Combining Lemma 48 with Theorem 41 and Theorem 43 gives the following result.

Theorem 49 *Let S, T be equivalent standardised CFL schemas and let p be a predicate symbol in S and $X \in \{\mathit{True}, \mathit{False}\}$. Then (where defined)*

$$\mathit{Symbols}(S, p) = \mathit{Symbols}(T, p)$$

and

$$\mathit{Symbols}(S, p, X) = \mathit{Symbols}(T, p, X).$$

9 Interchanging Commuting Subschemas

First, Theorem 49 is strengthened to show that two equivalent standardised CFL schemas S and T are *almost identical* except for differences in the ordering of subschemas assembled sequentially (Proposition 50). Lemma 56 then strengthens this further by proving that if the order of two subschemas of S differs from the order of the corresponding subschemas in T, these subschemas may be interchanged while preserving equivalence.

This interchange of subschemas may be performed finitely many times to obtain a schema which is identical to T. Thus there are finitely many schemas equivalent to a given schema, and it is this that makes equivalence decidable.

The following result restricts still further the ways in which two equivalent schemas may differ.

Proposition 50 *If S, T are equivalent CFL schemas, then*

$$Funcs(S) = Funcs(T)$$

and $\mathrm{assign}_S(f) = \mathrm{assign}_T(f)$ *and* $refvec_S(f) = refvec_T(f)$ *for each* $f \in Funcs(S)$. *Also* $refvec_S(p) = refvec_T(p)$ *for each* $p \in Preds(S)$.

Proof.
For assignments the results follow from the fact that for every assignment there exists an interpretation i that passes through it which is unitary with respect to $whilePreds(S)$ and Theorem 33 applied to $Atrace(\pi(S,i)) \cong Atrace(\pi(T,i))$.

We now consider the case of a predicate. Let $p \in Preds(S) = Preds(T)$ and let $n \leq arity(p)$. Suppose p contains a function symbol f in its body or true part and has the variable x as its nth argument in S, whereas in T the corresponding variable is $y \neq x$. There is an interpretation i passing through f which is unitary with respect to $whilePreds(S) = whilePreds(T)$. Write $\pi(S,i) = \sigma <p(\mathbf{x}) = True> \sigma'$ and $\pi(T,i) = \theta <p(\mathbf{y}) = True> \theta'$, where p does not occur in σ or θ, and σ and θ' do not contain the letters $<p(\mathbf{x}) = True>$ and $<p(\mathbf{y}) = True>$ respectively. Thus $\mathcal{M}[\![Atrace(\sigma)]\!]_e(x)$ is a g_x-term for some $g_x \in Funcs(S)$ with $\mathrm{assign}_S(g_x) = x$. Let j be the interpretation which differs from i only in that p^j maps to $False$ if the nth argument of p is a g_x-term. Then j is unitary with respect to $whilePreds(S) = whilePreds(T)$ and the path $\pi(S,j)$ does not pass through f, whereas $\pi(T,j) = \pi(T,i)$, since $\mathcal{M}[\![Atrace(\theta)]\!]_e(y)$ is not a g_x-term, and so $\mathcal{M}[\![T]\!]^j_e = \mathcal{M}[\![T]\!]^i_e$, contradicting Lemma 29, Part (2) and $S \cong T$. A similar argument holds if $f \in Symbols(T, p, False) = Symbols(T', p, False)$.

Thus, two equivalent schemas may only differ in the way that their symbols are ordered. This motivates the following definition of the ordering of symbols in a schema.

Definition 51 (Ordering of Functions Symbols in a Schema)
We write $f <_S g$ if $f \neq g$ and f occurs before g on at least one unitary path through S.

Lemma 52 *Let S be a schema with distinct function symbols f, g, h.*

(1) If $f <_S g$ then f occurs before g on every unitary path passing through both symbols.
(2) There exists a unitary path passing through both f and g if and only if f and g do not lie in different parts of the same if *predicate.*
(3) If f, $f' \in Symbols(S,p)$ and $g \notin Symbols(S,p)$ for some predicate p, then $f <_S g \iff f' <_S g$ and $g <_S f \iff g <_S f'$.
(4) If $f <_S g$ and $g <_S h$ then $f <_S h$.

Proof. (1)
This follows by induction on $|S|$. Clearly S does not just consist of one assignment. If S is indecomposable, then either S is a *while* schema (in which case the result follows by the inductive hypothesis) or an *if* schema. If f and g lie in the same part of this schema, again the result follows by the inductive hypothesis; if they lie in different parts, then $f <_S g$ is impossible.
Alternatively, S is not indecomposable; so let $S = S_1 S_2$ nontrivially. If S_1 contains f and S_2 contains g then the result is obvious; otherwise the result follows from the inductive hypothesis.
(2), (3), (4), (5). These again follow by induction on $|S|$.

Lemma 53 *Let T_1, T_2 be schemas with $whilePreds(T_1) = whilePreds(T_2)$, $ifPreds(T_1) = ifPreds(T_2)$, $Symbols(T_1,p) = Symbols(T_2,p)$ and (where defined)*

$$Symbols(T_1, p, X) = Symbols(T_2, p, X)$$

for all $p \in Preds(T_1)$ and $X \in \{True, False\}$ and $<_{T_1} = <_{T_2}$. Then T_1 and T_2 are identical except for the variables referenced by elements of $\mathcal{F} \cup \mathcal{P}$ and the variables assigned by elements of $\mathcal{F}$.

Proof.
This follows by induction on $|T_1| = |T_2|$.

Lemma 54

(1) Let $d_1 \neq \perp, d_2 \neq \perp$ be states, let S be any schema and let i be an interpretation and assume that $d_1(x) = d_2(x)$ for every variable $x \in \mathcal{V}$ referenced in S. Then $\mathcal{M}[\![S]\!]^i_{d_1} = \perp \iff \mathcal{M}[\![S]\!]^i_{d_2} = \perp$ and $\mathcal{M}[\![S]\!]^i_{d_1}(y) = \mathcal{M}[\![S]\!]^i_{d_2}(y)$ for every variable y assigned in S.
(2) Let S_1, S_2 be schemas and assume that every assignment in S_1 commutes

with every assignment in S_2 and that no predicate in S_2 references a variable assigned in S_1, or vice versa. Then $S_1\, S_2 \cong S_2\, S_1$.

Proof.

(1) This follows by induction on $|S|$.
(2) Let i be an interpretation. We first prove

$$\mathcal{M}[\![S_1 S_2]\!]^i_e = \perp \iff \mathcal{M}[\![S_2 S_1]\!]^i_e = \perp. \tag{1}$$

We have

$$\mathcal{M}[\![S_2]\!]^i_e \neq \perp \Rightarrow (\mathcal{M}[\![S_1]\!]^i_e = \perp \iff \mathcal{M}[\![S_2 S_1]\!]^i_e = \perp) \tag{2}$$

by the previous result applied to the states e and $\mathcal{M}[\![S_2]\!]^i_e$. Similarly

$$\mathcal{M}[\![S_1]\!]^i_e \neq \perp \Rightarrow (\mathcal{M}[\![S_2]\!]^i_e = \perp \iff \mathcal{M}[\![S_1 S_2]\!]^i_e = \perp). \tag{3}$$

Also clearly

$$\mathcal{M}[\![S_1]\!]^i_e = \perp \Rightarrow \mathcal{M}[\![S_1 S_2]\!]^i_e = \perp. \tag{4}$$

If $\mathcal{M}[\![S_1]\!]^i_e = \mathcal{M}[\![S_2]\!]^i_e \neq \perp$ then (1) follows from (2) and (3). If $\mathcal{M}[\![S_1]\!]^i_e = \perp \neq \mathcal{M}[\![S_2]\!]^i_e$ then (1) follows from (2) and (4). (1) follows similarly if $\mathcal{M}[\![S_2]\!]^i_e = \perp \neq \mathcal{M}[\![S_1]\!]^i_e$. Lastly, if $\mathcal{M}[\![S_1]\!]^i_e = \perp = \mathcal{M}[\![S_2]\!]^i_e$, then (1) is clear.

Thus we may assume that i terminates for $S_1\, S_2$ and $S_2\, S_1$. Let y be a variable assigned in S_1. Since the assignments of S_1 commute with those of S_2, this means that y is not assigned in S_2. Thus $\mathcal{M}[\![S_1\, S_2]\!]^i_e(y) = \mathcal{M}[\![S_1]\!]^i_e(y)$. Also $\mathcal{M}[\![S_1]\!]^i_e(y) = \mathcal{M}[\![S_2\, S_1]\!]^i_e(y)$ by the previous result, since S_2 clearly terminates. Thus

$$\mathcal{M}[\![S_1\, S_2]\!]^i_e(y) = \mathcal{M}[\![S_2\, S_1]\!]^i_e(y).$$

Interchanging the schemas shows that this holds if instead y is assigned by S_2; and it clearly holds if y is not assigned by either schema. Thus $S_1\, S_2 \cong S_2\, S_1$.

Lemma 55 *Let S, T be equivalent CFL schemas and assume that $f <_S g$ and $g <_T f$. Then the f-assignment commutes with the g-assignment.*

Proof. Let i be an interpretation which is unitary with respect to $\mathit{whilePreds}(S) = \mathit{whilePreds}(T)$ and such that $\pi(S, i)$ passes through f and g. By Part (1) of Lemma 52, f occurs before g on $\pi(S, i)$, and by Part (1) of this Lemma and Part (2) of Lemma 29, g occurs before f on $\pi(T, i)$. The result follows from Theorem 33 applied to the equivalent schemas $\mathit{Atrace}(\pi(S, i))$ and $\mathit{Atrace}(\pi(T, i))$.

Lemma 56 *Let S, T be distinct equivalent standardised CFL schemas.*

(1) *There is a pair $f, g \in$ Funcs(S) satisfying $f <_S g$ and $g <_T f$, such that there is no $h \in \mathcal{F}$ satisfying $f <_S h <_S g$.*
(2) *Given $f, g \in$ Funcs(S) satisfying the conditions in Part (1) of this Lemma, let $\bar{S}$ be the minimal subschema of S (with respect to $|\,|$) containing f and g. Then $\bar{S}$ has the form S_1S_2 with $f \in$ Funcs(S_1) and $g \in$ Funcs(S_2), and no variable referenced in S_1 (including variables referenced by predicates) is assigned in S_2, or vice versa. Also, if $f' \in$ Funcs(S_1) and $g' \in$ Funcs(S_2), then $f' <_S g'$ and $g' <_T f'$.*
(3) *Lastly, S can be transformed into T by finitely many transformations of the following form; finding $f, g \in$ Funcs(S) satisfying the conditions in Part (1) of this Lemma, finding $\bar{S} = S_1S_2$ satisfying the conditions in Part (2), and replacing S_1S_2 in S by S_2S_1.*

Proof.

(1) If $<_S = <_T$ then $S = T$ follows from Lemma 53, Theorem 33, and Proposition 50, contradicting the hypotheses of the Lemma. Assume, thus, that $<_S \neq <_T$ holds. If a pair of function symbols are incomparable with respect to $<_S$, then by part (2) of Lemma 52 and using Lemma 49, they are incomparable with respect to $<_T$. Thus by the transitivity of these relations there exists $f, g \in F$ such that $g <_T f$ and f immediately precedes g in S (that is, $f <_S g$, and there does not exist $h \in \mathcal{F}$ satisfying $f <_S h <_S g$).
(2) If $\bar{S}$ were a *while* schema, this would contradict the minimality of $|\bar{S}|$, since the body of $\bar{S}$ would also contain f and g; and if S' were an *if* schema, then f and g would have to lie in different parts of $\bar{S}$, contradicting Part (2) of Lemma 52; thus we can write $S' = S_1S_2$ with $f \in \mathit{Funcs}(S_1)$ and $g \in \mathit{Funcs}(S_2)$. By the immediacy and minimality conditions, each S_r is indecomposable. Thus either $\mathit{Funcs}(S_1) = \{f\}$ or $\mathit{Funcs}(S_1) = \mathcal{F} \cap \mathit{Symbols}(S, p) = \mathcal{F} \cap \mathit{Symbols}(T, p)$ for some $p \in \mathit{Preds}(S) = \mathit{Preds}(T)$; and a similar statement holds for S_2 and g.

We will show that S_1 and S_2 satisfy the hypotheses of Lemma 54. By Lemma 52, Part (3), we get

$$f' \in \mathit{Funcs}(S_1), g' \in \mathit{Funcs}(S_2) \Rightarrow f' <_S g'. \tag{1}$$

Since also $g <_T f$, by Lemma 52, Part (3), and Theorem 49, if necessary,

$$f' \in \mathit{Funcs}(S_1), g' \in \mathit{Funcs}(S_2) \Rightarrow g' <_T f'. \tag{2}$$

By (1), (2) and Lemma 55, if $f' \in \mathit{Funcs}(S_1)$, $g' \in \mathit{Funcs}(S_2)$ then the f'-assignment commutes with the g'-assignment.

We now show that S_2 does not contain a predicate referencing a variable assigned by a function symbol in S_1. Thus suppose that S_2 contains a predicate q with $a \in \mathit{Refset}_{S_2}(q)$ and that S_1 contains an assignment

$a{:=}h(\mathbf{b})$. (Thus S_2 is an *if* or *while* schema with guard $r \in \mathcal{P}$, say.) Let

$$g_q \in \mathcal{F} \cap \mathit{Symbols}(S,q) \subseteq \mathcal{F} \cap \mathit{Symbols}(S,r).$$

By (1) we get $h <_S g_q$. Let μ be a unitary path passing through h and g_q; since μ clearly passes through q, and $h \notin \mathit{Symbols}(S,q)$, we may write

$$\mu = \mu'(q(\mathbf{x}) = X)\mu'',$$

where h occurs in μ' and q does not. Let $\{X, Y\} = \{\mathit{True}, \mathit{False}\}$ and let i be an interpretation satisfying $\pi(S,i) = \mu$ which is unitary with respect to $\mathit{whilePreds}(S) = \mathit{whilePreds}(T)$; then $q^i(\mathcal{M}[\![\mathit{Atrace}(\mu')]\!]_e(\mathbf{x})) = X$. Let the interpretation j be identical to i except that $q^i(\mathcal{M}[\![\mathit{Atrace}(\mu')]\!]_e(\mathbf{x})) = Y$. Then j is also unitary with respect to $\mathit{whilePreds}(S)$ (since if $q \in \mathit{whilePreds}(S)$, then $X = \mathit{True}$) and $\mu'(q(\mathbf{x}) = Y) \in \mathit{pre}(\pi(S,j))$. Hence j does not pass through g_q and so

$$\mathcal{M}[\![S]\!]_e^i \neq \mathcal{M}[\![S]\!]_e^j. \tag{3}$$

We now show that $g_q <_T h$. Observe that

$$g_q \in \mathcal{F} \cap \mathit{Symbols}(S,q) \subseteq \mathcal{F} \cap \mathit{Symbols}(S,r) = \mathcal{F} \cap \mathit{Symbols}(T,r) \ni g$$

and recall that $g <_T f$. If $h = f$ then $g_q <_T h$ follows from Part (3) of Lemma 52; otherwise S_1 is an *if* or *while* schema with guard s, say, and $f, h \in \mathit{Symbols}(S,s)$ and $\mathit{Symbols}(S,s) \cap \mathit{Symbols}(S,r) = \emptyset$. These statements also hold in T, so again Lemma 52 may be used.

Thus the unitary path $\pi(T,i)$ meets the predicate q before it meets the function symbol h and since $q \in \mathit{whilePreds}(T)$ implies $X = \mathit{True}$, the path $\pi(T,i)$ does not contain $(q(\mathbf{x}) = X)$ after meeting h, so $\mathcal{M}[\![T]\!]_e^i = \mathcal{M}[\![T]\!]_e^j$, contradicting $S \cong T$ and (3).

A similar argument disposes of the possibility that S_1 has a predicate referencing a variable assigned in S_2. Thus by Lemma 54, Part (2), the subschemas S_1 and S_2 may be interchanged while preserving equivalence of S with the new schema. This schema can easily be shown to be CFL.

(3) This transformation reduces the number of 'disagreeing pairs' of function symbols, and so finitely many such transformations are sufficient to obtain an equivalent schema T' from S with $<_{T'} = <_T$.

By Lemma 53, Theorem 33, and Proposition 50, $T = T'$.

Our main Theorem follows.

Theorem 57 *Let S, T be CFL schemas. It is possible to decide whether they are equivalent.*

Proof.
We may assume that S and T are standardised. By Lemma 56, there are

finitely many schemas from our class which are equivalent to S, and it is possible to construct all of them, so it suffices to check whether T is one of them.

Complexity

In this section we will show that the problem of determining equivalence of CFL schemas S and T has time complexity that is polynomial in $\max(|S|, |T|)$, where $|S|$ is the number of function and predicate symbols in S.

Theorem 58 *Let S, T be CFL schemas. There is an algorithm for deciding their equivalence which has polynomial time complexity with respect to $max(|S|, |T|)$.*

Proof.
The first step is to outline an algorithm for deciding whether T and S are equivalent. The following algorithm suffices.

(1) If $\mathit{ifPreds}(S) \cup \mathit{Funcs}(S) \neq \mathit{ifPreds}(T) \cup \mathit{Funcs}(T)$ then terminate with the result that S and T are not equivalent.
(2) Choose an ordering $\triangleleft$ on the *if* predicate symbols of S in such a way that S is standardised with respect to $\triangleleft$. S defines such an ordering: if one part of both p and q in S is λ and the other part of p is an *if* schema of which q is the guard then $p \triangleleft q$.
(3) Standardise T with respect to $\triangleleft$ to form T'.
(4) Set $i = 0$ and $S^0 = S$.
(5) Repeat the following steps:
(6) If $S^i = T'$ then terminate with the result that S and T are equivalent.
(7) Find $f, g \in \mathit{Funcs}(S)$ satisfying $f <_{S^i} g$ and $g <_{T'} f$, and there is no h with $f <_{S^i} h$ and $h <_{S^i} g$. By Lemma 56, if S and T are equivalent then there are such f and g. If there is no such pair f, g then terminate with the result that S and T are not equivalent.
(8) Find the minimal subschema $S_1 S_2$, of S^i, containing f and g.
(9) Interchange S_1 and S_2 in S^i to obtain a new schema S^{i+1} equivalent to S^i (as described in Lemma 56).
(10) Set $i = i + 1$.

Let $n = \max(|S|, |T|)$. It is now sufficient to prove that the above algorithm has time complexity that is polynomial in n.

Clearly the first step can be performed in polynomial time: it simply involves collecting and comparing the sets of function and *if* predicate symbols of S and T. Note that, if the algorithm moves past the first step then $n = |S| = |T|$.

The time taken to create and store the ordering in the second step is polynomial in n since the number of pairs of predicate symbols of S is bounded above by n^2. (Checking whether a predicate is ambiguous can be done in constant time.)

In order to standardise T with respect to $\lhd$ it is sufficient to check each pair p and q of *if* predicates of T, whether p is an ambiguous predicate one part of which is an *if* schema guarded by q. This can be done in $O(n)$ since it is sufficient to check each predicate of T. (Checking whether p is an ambiguous predicate, one part of which is an *if* schema guarded by q, can be done in constant time.) To check whether it is necessary to reorder p and q (as described in Section 8) it is sufficient to look up the pair (p, q) in the list representing $\lhd$. This gives a polynomial time worst case complexity for each pair (p, q) of predicates. Thus, since the number of pairs of predicates is polynomial in n, the time taken to standardise T is polynomial in n.

Now consider the loop. Since each iteration of the loop reduces the differences between the orderings $<_{T'}$ and $<_{S^i}$, and there are $O(n^2)$ such differences, the number of iterations of the loop is of $O(n^2)$. Thus, it is sufficient to prove that each step in the body of the loop may be completed in time that is polynomial in n.

It is possible to determining whether $S^i = T'$ in linear time.

Consider the problem of finding $f, g \in \mathit{Funcs}(S)$ satisfying $f <_{S^i} g$ and $g <_{T'} f$ and there does not exist h with $s <_{S^i} h$ and $h <_{S^i} g$, or determining that there is no such pair. The number of pairs $f, g \in \mathit{Funcs}(S)$ satisfying $f <_{S^i} g$ such that there does not exist h with $s <_{S^i} h$ and $h <_{S^i} g$, is quadratic in n. Each of these can be checked against $<_{T'}$ in polynomial time and thus this step may be performed in polynomial time.

Note that a $w$hile schema has only one more subschema than its body and an if schema only has one more subschema than the sum of the number of subschemas of its $t$hen and $e$lse parts. Thus, of all the linear schemas S' with $|S'| = n$, the one with the most subschemas consist of a sequence of n assignment statements. In this case S' has $\binom{n}{2} = \frac{n(n-1)}{2}$ subschemas. Thus, the total number of subschemas, of S, is of $O(n^2)$. The problem of finding the minimal subschema $S_1 S_2$ thus reduces to searching through the $O(n^2)$ subschemas that contain f and g and may thus be solved in polynomial time.

Finally, S_1 and S_2 may be interchanged in constant time. This completes the proof that there exists an algorithm, which has complexity polynomial in $\max(|S|, |T|)$, for checking equivalence of CFL schemas S and T.

10 The Relevance of Linear Schemas to Program Slicing

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 safely be applied to programs. Surveys on the theory of program schemas can be found in the works of Ershov [8], Greibach [9] and Manna [7].

Our interest in the theory of program schemas, linear (or non-repeating) schemas in particular, is motivated by certain theoretical questions concerning program slicing [10]. In program slicing, statements are deleted from a program, leaving a resulting program called a *slice*. The slice must preserve *the effect* of the original program on a set of variables of interest, called the *slicing criterion*. Like program optimisation, slicing can be thought of as a transformation that preserves certain semantic properties. The equivalence preserved by the common slicing algorithms [10–13] turns out to be based on a lazy semantics [14,15] of programs. Slicing has many applications including program comprehension [16,17], software maintenance [18–21], debugging [22–25], testing [26–30], re–engineering [31,32], component re–use [33,34], program integration [35,36], and software metrics [37–39]. There are several surveys of slicing techniques, applications and variations [40–43]. All applications of slicing rely upon the fact that a slice is faithful to a projection of the original program's semantics, yet it is typically a smaller program. A major aim in program slicing is to produce small slices.

The most widely used slicing algorithms, Weiser's [10,11] and the Program Dependence Graph approach [12], essentially produce the same slices [1]. Importantly, for this discussion, they both operate at a level of abstraction where, in a program, the only information that can be utilised about each expression, e, is the set of variables referenced by e. Weiser termed this approach *Dataflow Analysis*, but we call it *DefRef abstraction* [2], as the term *Dataflow Analysis* now has more general connotations. Figure 1 shows two distinct programs which are identical to each other under DefRef abstraction. Algorithms that use DefRef abstraction are limited in the sense that they cannot take advantage of situations where expressions in the program are equal, nor can any form of expression simplification be used. All the information required to do such things has been 'abstracted away'. For example, after DefRef abstraction of $\{y:=x+1;\ \ z:=x+1\}$ the only remaining information is that the variable

[1] They both, in essence, compute the transitive closure of the union of *control dependence* and *data dependence* [48].

[2] Other approaches to program slicing exist [44–46,18,19] which do not use DefRef abstraction, but we are concerned with the theoretical properties of traditional slicing.

y is assigned an expression which references x and the variable z is assigned an expression which references x, and the assignments happen in that order.

Analysing a program, P, after doing DefRef abstraction, is identical to first converting P to a corresponding linear schema, S, and then analysing S. A linear schema (see Definition 2), is one where each function and predicate name is only allowed to occur once in the schema. The schema $\{y{:=}f(x);\ z{:=}f(x)\}$, for example, is not linear but $\{y{:=}f(x);\ z{:=}g(x)\}$ is. A linear schema corresponding to program P has the same structure as P but much of the detail of each expression has been removed. To produce a linear schema corresponding to P, every expression, e, in P is replaced by a symbolic expression of the form $f(v_1, \ldots, v_n)$ where $\{v_1, \ldots, v_n\}$ is the set of variables referenced by e. To guarantee linearity, simply ensure that all function and predicate names are distinct. In analysing a program via a corresponding linear schema, apart from the program's structure being preserved, the only information about the program that is available is the name of the variable assigned to in each assignment statement and the set of variables names referenced by each expression [3]. Importantly, with linear schemas we can never infer that different variables or expressions have the same value, nor may we exploit other properties which can be derived from knowing that the same function or predicate has been applied in more than one place.

The connection between DefRef analysis and linear schemas just described motivates us to use the theory of schemas in order both to give definitions of, and to ask questions about, different forms of slice. For example, we can define a *strong slice* [24] with respect to a set of variables V of a schema S as a schema T obtained from S by deleting statements where in all Herbrand interpretations, i, starting in the natural state, either S and T both fail to terminate or S and T both terminate in i with the same values for all variables in V. A *weak slice* [11] with respect to a set of variables V of a schema S, on the other hand, is a schema T obtained from S by deleting statements where in all Herbrand interpretations, i, starting in the natural state, either S does not terminate in i or S and T both terminate in i with the same values for all variables in V. One problem of particular interest to us, the *Dataflow Minimality Problem* [10,49], concerns the existence of algorithms for computing minimal slices at this level of abstraction. We now describe the problem.

A *statement minimal* slice [4] of program, P, is a slice of P where deleting further statements yields a non-slice of P. Statement minimal slices are not

[3] Expressions occur both as predicates in `if`s and `while`s and on the right hand side of assignment statements

[4] To be precise, we should really talk about, 'statement minimal *strong* slices' or statement minimal *weak* slices, or, in general, 'statement minimal $\mathcal{M}$-slices' where $\mathcal{M}$ is meaning intended to be preserved by the form of slicing of interest [44].

Program P_2	Schema S_2	Dataflow Minimal Slice

while $i < 0$
do
  begin
    if $c = 3$
    then
      begin
        $c := 4$;
        $x := 5$
      end ;
    $i := i + 1$
  end
	while $p_1(i)$
	do
	  begin
	    if $p_2(c)$
	    then
	      begin
	        $c := f_3()$;
	        $x := f_4()$
	      end ;
	    $i := f_5(i)$
	  end
		while $i < 0$
		do
		  begin
		    if $c = 3$
		    then
		      begin
		
		        $x := 5$
		      end ;
		    $i := i + 1$
		  end

Fig. 2.

computable [10]. Weiser noticed, further that his algorithm did not even produce *dataflow minimal*[5] slices. That is, it sometimes fails to delete statements which, even after DefRef abstraction, can be shown to have no effect on the slicing criterion. An example of this is now given.

Using Weiser's definition [11], an end–slice of P, with respect to the variable x is any program, P', obtained from P by statement deletion such that P' terminates whenever P does, with the same final value for x. In attempting to slice P_2, in Figure 2, Weiser's algorithm returns P_2; it fails to delete any statements at all. This is acceptable since, by definition, every program is a valid end–slice of itself. The smaller program on the right hand side of Figure 2, however, is also a valid end-slice. To justify this, it turns out that we do not need to consider program P_2 itself; analysis of S_2, a linear schema corresponding to P_2, is sufficient. We observe that the constant assignment $c{:=}f_3()$ is executed if and only if the constant assignment $x{:=}f_4()$ is executed. Having been assigned a constant value, the value of x cannot be further changed by the body of the loop. The initial value of c is important, but the later assignment to c cannot affect the final value of x. The assignment $c{:=}f_3()$, therefore, need not be included in the slice. The reason that Weiser's algorithm includes $c{:=}f_3()$ is that the assignment $x{:=}f_4()$ is controlled by the predicate $p_2(c)$, which, in turn, is data dependent on $c{:=}f_3()$ and so, since Weiser's algorithm computes the transitive closure, it infers that $x{:=}f_4()$ depends on $c{:=}f_3()$[6]. The program, on the right hand side of Figure 2 is, in fact, a dataflow minimal slice of P_2.

A schema T is a *dataflow minimal* slice of S with respect to a set of variables V, if and only if T is a slice of S with respect to V, and no schema T'

[5] Weiser called it 'dataflow consistent'[10].

[6] It turns out that Weiser's Algorithm includes unnecessary nodes, in this case, because S_2 is not liberal [1].

obtained from T by deleting further statements is a slice of S. A program P' is a dataflow minimal slice of P if and only if S' is a dataflow minimal slice of S where P and P' correspond to schemas S and S', respectively, via the same interpretation.

Since Weiser first raised the question in his 1979 thesis [10], the question remains open as to whether dataflow minimal slices are computable. A more general question, of interest is: for what classes of schemas are dataflow minimal slices computable? For linear schemas, decidability of equivalence implies computability of dataflow minimal strong slices. A trivial algorithm for producing dataflow minimal strong slices would first add a sequence of 'killing assignments', K, to the variables not in the slicing criterion to the end of the program, P, being sliced, and next, convert PK to a linear schema $P'K'$, and finally, for all possible schemas S obtained from P' by statement deletion, test $P'K'$ and SK' for equivalence. The 'smallest' such S correspond to dataflow minimal slices of P. Unfortunately, this algorithm does not work for CFL schemas since statement deletion does not preserve freeness. Future work, therefore, will consider decidability of equivalence for more general classes of linear schemas and also investigate conditions on schemas under which Weiser's algorithm is guaranteed to produce dataflow minimal slices.

11 Conclusion

We have proved that it is decidable whether two CFL schemas are equivalent. Our method of proof was to show that equivalent schemas in the class considered have almost identical structure:- two standardised equivalent structured CFL schemas can only differ in the ordering of sequentially combined commuting sub–schemas. Clearly, there are only finitely many pairs of such subschemas and trivially an algorithm for finding all of them exists. Since standardising a schema is also computable, it follows that the equivalence of structured CFL schemas is decidable.

References

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

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

[3] 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.

[4] D. C. Luckham, D. M. R. Park, M. S. Paterson, On formalised computer programs, Journal of Computer and System Sciences 4 (1970) 220–249.

[5] E. J. Weyuker, Modifications of the program scheme model, Journal of Computer and System Sciences 18 (3) (1979) 281–293.

[6] E. J. Weyuker, Translatability and decidability questions for restricted classes of program schemas, SIAM Journal on Computing 8 (4) (1979) 587–598.

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

[8] A. P. Ershov, Theory of program schemata, Information Processing 1971 .

[9] 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.

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

[11] M. Weiser, Program slicing, IEEE Transactions on Software Engineering 10 (4) (1984) 352–357.

[12] S. Horwitz, T. Reps, D. W. Binkley, Interprocedural slicing using dependence graphs, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, Georgia, 1988, pp. 25–46, proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[13] S. Danicic, M. Harman, Y. Sivagurunathan, A parallel algorithm for static program slicing, Information Processing Letters 56 (6) (1995) 307–313.

[14] R. Cartwright, M. Felleisen, The semantics of program dependence, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, 1989, pp. 13–27.

[15] M. Harman, D. Simpson, S. Danicic, Slicing programs in the presence of errors, Formal Aspects of Computing 8 (4) (1996) 490–497.

[16] 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.

[17] M. Harman, R. M. Hierons, S. Danicic, J. Howroyd, C. Fox, Pre/post conditioned slicing, in: IEEE International Conference on Software Maintenance (ICSM'01), IEEE Computer Society Press, Los Alamitos, California, USA, Florence, Italy, 2001, pp. 138–147.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

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

[24] 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).

[25] 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.

[26] 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.

[27] 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.

[28] M. Harman, S. Danicic, Using program slicing to simplify testing, Software Testing, Verification and Reliability 5 (3) (1995) 143–162.

[29] R. M. Hierons, M. Harman, S. Danicic, Using program slicing to assist in the detection of equivalent mutants, Software Testing, Verification and Reliability 9 (4) (1999) 233–262.

[30] R. M. Hierons, M. Harman, C. Fox, L. Ouarbya, M. Daoudi, Conditioned slicing supports partition testing, Software Testing, Verification and Reliability To appear.

[31] 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.

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

[33] 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.

[34] 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.

[35] D. W. Binkley, S. Horwitz, T. Reps, Program integration for languages with procedure calls, ACM Transactions on Software Engineering and Methodology 4 (1) (1995) 3–35.

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

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

[38] L. M. Ott, J. J. Thuss, Slice based metrics for estimating cohesion, in: Proceedings of the IEEE-CS International Metrics Symposium, IEEE Computer Society Press, Los Alamitos, California, USA, Baltimore, Maryland, USA, 1993, pp. 71–81.

[39] 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.

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

[41] A. De Lucia, Program slicing: Methods and applications, in: 1^{st} IEEE International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society Press, Los Alamitos, California, USA, Florence, Italy, 2001, pp. 142–149.

[42] M. Harman, R. M. Hierons, An overview of program slicing, Software Focus 2 (3) (2001) 85–92.

[43] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3 (3) (1995) 121–189.

[44] M. Harman, S. Danicic, Amorphous program slicing, in: 5^{th} IEEE International Workshop on Program Comprenhesion (IWPC'97), IEEE Computer Society Press, Los Alamitos, California, USA, Dearborn, Michigan, USA, 1997, pp. 70–79.

[45] J. Field, G. Ramalingam, F. Tip, Parametric program slicing, in: 22^{nd} ACM Symposium on Principles of Programming Languages, San Francisco, CA, 1995, pp. 379–392.

[46] G. Snelting, Combining slicing and constraint solving for validation of measurement software, in: Static Analysis Symposium (SAS'96), LNCS 1145, 1996, pp. 332–348.

[47] S. Horwitz, T. Reps, D. W. Binkley, Interprocedural slicing using dependence graphs, ACM Transactions on Programming Languages and Systems 12 (1) (1990) 26–61.

[48] M. S. Hecht, Flow Analysis of Computer Programs, Elsevier, 1977.

[49] S. Danicic, Dataflow minimal slicing, PhD thesis, University of North London, UK, School of Informatics (Apr. 1999).