New Generation Computing, 9 (1991) 305-333
OHMSHA, LTD. and Springer-Verlag

The Derivation of an Algorithm for Program Specialisation*

John GALLAGHER
Department of Computer Science,
University of Bristol,
Bristol BS8 1TR, U. K.
Maurice BRUYNOOGHE
Department of Computer Science,
Katholieke Universiteit Leuven,
Celestijnenlaan 200A, B-3001 Leuven, Belgium.

Received 27 November 1990
Revised manuscript received 9 August 1991

Abstract In this paper we develop an algorithm, based on abstract interpretation, for source specialisation of logic programs. This approach is more general than partial evaluation, another technique for source specialisation, and can perform some source specialisations that cannot be done by partial evaluation; examples are specialisations that use information from infinite computations. Our algorithm for program specialisation uses *minimal function graphs* as a basis. Previous work on minimal function graphs is extended by describing a scheme for constructing a minimal function graph for a simple functional language, and then using that to define a minimal function graph constructor for Prolog. We show how to compute a more precise approximation to the minimal function graph than was obtained in previous work. The efficient computation of minimal function graphs is also discussed. An abstract interpretation based on unfolding paths is then developed for Prolog program specialisation.

Keywords: Program Specialisation, Partial Evaluation, Obstruct Interpretation

§1 Introduction

An algorithm for Prolog program specialisation will be derived, based on general principles of program analysis. The aim is to improve on and generalise

* Work performed in ESPRIT Basic Research Action project COMPULOG (3012).

methods for partial evaluation.

Our approach is to develop a general method of specialising a function to a given input or set of inputs. We develop the notion of *minimal function graphs* or mfgs in a general way. An mfg can be constructed from any function by defining a set of function *activations* generated from an initial goal. This set can be computed simultaneously with a function restricted to the set of activations.

The application of this technique to Prolog program analysis is developed by giving an evaluation function for Prolog goals. From this an mfg is constructed, using the general construction for deriving an mfg. An mfg for Prolog describes the part of the total meaning of a Prolog program that is used to compute some initial goals. This restricted meaning can then be used for specialising the program for the initial goals. For our application we use an abstract interpretation based on *characteristic paths*, which describe the unfolding properties of atoms. Finally a procedure for constructing a specialised program from the results of the analysis is described.

§2 Minimal Function Graphs

Minimal function graphs were introduced in Ref. 8) for program analysis, and the idea was applied to logic programs in Refs. 19), 20) and 4). Minimal function graphs (or mfgs) appear to bridge the gap between program analysis frameworks based on operational semantics,[3] and those based on denotational semantics.[7,12] With suitably-chosen semantic functions, a minimal function graph can be defined with a direct correspondence to the states in a computation reachable from some given initial state. Marriott and Søndergaard noted in Ref. 12) that a "query-directed" semantics can be defined in a general way from standard semantic functions. In this paper we examine this idea in more detail and introduce a method of constructing a minimal function graph for a given function with respect to given initial function calls. We claim that our construction clarifies the concept of mfg, and allows us to discuss various semantic and practical aspects of mfgs in a more general context, without involving issues of language semantics or abstract interpretation.

The idea of a minimal function graph can be informally illustrated.

Example 1

The factorial function *fact* over the natural numbers is defined by the expression:

$$fact(x) = \text{if } x = 0 \text{ then } 1 \text{ else } fact(x - 1) * x$$

fact denotes an infinite set of pairs, sometimes called its function graph:

$$\{0 \mapsto 1,\ 1 \mapsto 1,\ 2 \mapsto 2,\ \ldots\}$$

Suppose that we are given a particular argument v of *fact* to compute.

Some subset of the function graph is "used" to compute $fact(v)$ with the given definition. In the case of, say, $v = 3$ the subset is:

$$\{0 \mapsto 1,\ 1 \mapsto 1,\ 2 \mapsto 2,\ 3 \mapsto 6\}$$

This set characterises the part of the factorial function that is activated or called during the computation of $fact(3)$. The term minimal function graph or mfg was used in Ref. 8); the above function graph is called the mfg of $fact(3)$.

Clearly for any v the mfg of $fact(v)$ is the restriction of $fact$ to some subset S_v, written $fact|_{S_v}$. How to compute S_v, given v and any function, is the subject of this section.

The relevance of this concept to program analysis can be stated straight away. A program's meaning is represented by a function from input to output, and its mfg with respect to some given input represents that part of the total meaning of the program that is needed for computing a result for the given input. This information can often be used to optimise the program for the given input. The construction of mfgs is next discussed; the aim is to derive an algorithm for constructing an mfg, which can be used for building program analysers for specific languages, given a meaning function for the language. The second part of the paper illustrates the procedure for Prolog.

2.1 A Procedure for Constructing mfgs

Jones and Mycroft[8] developed the mfg for a simple functional language. An mfg construction for Prolog (or other languages) can be obtained by writing an evaluation function for Prolog in the functional language, and then constructing the mfg by Jones and Mycroft's method. This approach has the advantage of completely separating the computation of the mfg from the definition of the semantics of the programming language.

Our construction broadly follows that of Jones and Mycroft,[8] but is designed to lead more directly to a practical algorithm. The main difference from Jones and Mycroft's presentation is that we use a separate function for generating function activations, and we keep activations explicitly as a set. We do not employ the double-bottomed lattice used by them to distinguish "$f(x)$ not defined" from "$f(x)$ not called".

Winsborough defines an mfg for Prolog,[19] and discusses some basic difficulties in the direct computation of the mfg. He defines another object called the partial function graph, which is greater than the mfg but can be calculated directly. Like Winsborough, we find that we cannot in general compute the mfg directly, but we suggest a different way around the problem, and obtain a more precise result.

[1] Functions and activations

Let X be a lattice with bottom element $\perp$ and partial order $\sqsubseteq$. The

cartesian product X^k is ordered componentwise. Let $f_1, ..., f_n$: $X^k \to X$, be n continuous (possibly mutually recursive) functions. Assume that X includes the values *true* and *false*. (All functions are k-ary to simplify the notation).

Definition 2.1 a functional language
A function f_i, $1 \leq i \leq n$, has a definition of the form $f(x_1, ..., x_k) = \mathcal{E}$, where $\mathcal{E}$ is either

- a variable x_j, $1 \leq j \leq k$, or
- a constant c_1, c_2, ..., or an expression of the form
- $f_j(\mathcal{E}_1, ..., \mathcal{E}_k)$, $1 \leq j \leq n$, or
- $a_j(\mathcal{E}_1, ..., \mathcal{E}_k)$, $j > 0$, where a_j: $X^k \to X$ is some function that does not depend on $f_1, ..., f_n$, or
- if $\mathcal{E}_1$ then $\mathcal{E}_2$ else $\mathcal{E}_3$,

where $\mathcal{E}_1$, $\mathcal{E}_2$, ... are also expressions.

Definition 2.2 partial order on functions
Let f and g be functions $X \to Y$, where Y is partially ordered by $\sqsubseteq$. The partial order $\leq$ on functions is defined as $f \leq g$ iff $\forall x \in X.\ f(x) \sqsubseteq g(x)$. The least function yields $\perp$ for all arguments.

Definition 2.3
A function environment is an n-tuple of functions $X^k \to X$. If ϕ is a function environment, its components are denoted $\phi_1, ..., \phi_n$.

Definition 2.4 function evaluation in a function environment
Given a function environment ϕ, the value of an expression is computed by the following evaluation function **E**. Let $\mathbf{v} \in X^k$, with components $v_1, ..., v_k$.

$$\mathbf{E}[\![x_i]\!]\phi\mathbf{v} = v_i$$
$$\mathbf{E}[\![c_i]\!]\phi\mathbf{v} = c_i$$
$$\mathbf{E}[\![a_j(\mathcal{E}_1, ..., \mathcal{E}_k)]\!]\phi\mathbf{v} = a_j(\mathbf{E}[\![\mathcal{E}_1]\!]\phi\mathbf{v}, ..., \mathbf{E}[\![\mathcal{E}_k]\!]\phi\mathbf{v})$$
$$\mathbf{E}[\![f_j(\mathcal{E}_1, ..., \mathcal{E}_k)]\!]\phi\mathbf{v} = \phi_j(\mathbf{E}[\![\mathcal{E}_1]\!]\phi\mathbf{v}, ..., \mathbf{E}[\![\mathcal{E}_k]\!]\phi\mathbf{v})$$
$$\mathbf{E}[\![\text{if } \mathcal{E}_1 \text{ then } \mathcal{E}_2 \text{ else } \mathcal{E}_3]\!]\phi\mathbf{v} =$$
$$\quad \text{if } \mathbf{E}[\![\mathcal{E}_1]\!]\phi\mathbf{v} = \textit{true} \text{ then } \mathbf{E}[\![\mathcal{E}_2]\!]\phi\mathbf{v}$$
$$\quad \text{else if } \mathbf{E}[\![\mathcal{E}_1]\!]\phi\mathbf{v} = \textit{false} \text{ then } \mathbf{E}[\![\mathcal{E}_3]\!]\phi\mathbf{v}$$
$$\quad \text{else } \perp$$

E is continuous if the functions $f_1, ..., f_n, a_1, a_2, ...$ are all continuous. The function **E** evaluates an expression using the value of ϕ_j in the environment to represent f_j. We now use **E** to define the denotations of the functions $f_1, ..., f_n$.

Definition 2.5 fixpoint computation of function denotation
The denotation of the functions $f_1, ..., f_n$ defined respectively by expressions $\mathcal{E}^1, ..., \mathcal{E}^n$ is given by the least function environment ϕ satisfying the fixpoint equation:

$$\phi = (\mathbf{E}[\![\mathcal{E}^1]\!]\phi, ..., \mathbf{E}[\![\mathcal{E}^n]\!]\phi)$$

If $\mathcal{E}$ is continuous, the solution exists and can be computed as the limit of the Kleene sequence

$$\phi^0, \phi^1, \phi^2, ...$$

where ϕ^0 is the least function environment, and

$$\phi^{i+1} = (\mathbf{E}[\![\mathcal{E}^1]\!]\phi^i, ..., \mathbf{E}[\![\mathcal{E}^n]\!]\phi^i).$$

Definition 2.6 activations
An activation of a function f_i is an argument k-tuple for f_i; an activation environment of $f_1, ..., f_n$ is an n-tuple of sets of activations of the respective functions. Let $\mathbf{W} = \mathbf{2}^{(X^k)}$; the set of activation environments for $f_1, ..., f_n$ is thus $\mathbf{W}^n$. In the following, we denote by $\emptyset^n$ the n-tuple $(\emptyset, ..., \emptyset)$. Elements of $\mathbf{W}^n$ are partially ordered by $\sqsubseteq$, defined by $(\mathbf{a}_1, ..., \mathbf{a}_n) \sqsubseteq (\mathbf{b}_1, ..., \mathbf{b}_n)$ iff $\mathbf{a}_1 \subseteq \mathbf{b}_1 \wedge ... \wedge \mathbf{a}_n \subseteq \mathbf{b}_{n_1}$ and if $\mathbf{a}, \mathbf{b} \in \mathbf{W}^n$ then $\mathbf{a} \sqcup \mathbf{b}$ is defined as $(\mathbf{a}_1 \sqcup \mathbf{b}_1, ..., \mathbf{a}_n \sqcup \mathbf{b}_n)$.

Definition 2.7
Let $\mathbf{v} \in X^k$, with components $v_1, ..., v_k$; let $\mathbf{W}^n$ be the set of activation environments. We define $only_i$: $X^k \to \mathbf{W}^n (1 \leq i \leq n)$ as follows (slightly modifying Jones and Mycroft's definition[8]):

$$only_i(\mathbf{v}) = \text{if } v_j = \perp \text{ for some } j(1 \leq j \leq k) \text{ then } \emptyset^n$$
else $(\emptyset, ..., \{\mathbf{v}\}, ..., \emptyset) \in \mathbf{W}^n$, where $\{\mathbf{v}\}$ is the i-th component

The evaluation of an activation gives rise to other activations. For example, the evaluation of *fact*(4) gives rise to the call *fact*(3), according to the expression defining *fact*. This in turn activates *fact*(2) and so on. Given some function definitions and an initial activation environment **init**, we would like to collect all of the activations that arise during the evaluation of **init**. We now formalise the propagation of activations.

Definition 2.8
Let $\mathbf{w} \in \mathbf{W}$ be a set of activations and let ϕ be a function environment. The activation function **Act** returns an activation environment and is defined as follows:

$$\mathbf{Act}[\![x_i]\!]\phi\mathbf{w} = \emptyset^n$$
$$\mathbf{Act}[\![c_i]\!]\phi\mathbf{w} = \emptyset^n$$
$$\mathbf{Act}[\![a_j(\mathcal{E}_1, ..., \mathcal{E}_k)]\!]\phi\mathbf{w} = \sqcup\{\mathbf{Act}[\![\mathcal{E}_i]\!]\phi\mathbf{w} \mid 1 \leq i \leq k\}$$
$$\mathbf{Act}[\![f_j(\mathcal{E}_1, ..., \mathcal{E}_k)]\!]\phi\mathbf{w} = \sqcup\{\mathbf{Act}[\![\mathcal{E}_i]\!]\phi\mathbf{w} \mid 1 \leq i \leq k\} \sqcup \mathbf{b}$$
where $\mathbf{b} = \sqcup\{only_j((\mathbf{E}[\![\mathcal{E}_1]\!]\phi\mathbf{x}, ..., \mathbf{E}[\![\mathcal{E}_k]\!]\phi\mathbf{x})) \mid \mathbf{x} \in \mathbf{w}\}$
$$\mathbf{Act}[\![\text{if } \mathcal{E}_1 \text{ then } \mathcal{E}_2 \text{ else } \mathcal{E}_3]\!]\phi\mathbf{w} = \sqcup\{\mathbf{b} \mid \mathbf{x} \in \mathbf{w}\}$$
where $\mathbf{b} =$ if $\mathbf{E}[\![\mathcal{E}_1]\!]\phi\mathbf{x} = \mathit{true}$ then $\mathbf{Act}[\![\mathcal{E}_1]\!]\phi\{\mathbf{x}\} \sqcup$
$\mathbf{Act}[\![\mathcal{E}_2]\!]\phi\{\mathbf{x}\}$

else if $\mathbf{E}[\![\mathcal{E}_1]\!]\phi\mathbf{x} = \mathit{false}$ then $\mathbf{Act}[\![\mathcal{E}_1]\!]\phi\{\mathbf{x}\} \sqcup$
$\mathbf{Act}[\![\mathcal{E}_3]\!]\phi\{\mathbf{x}\}$
else $\mathbf{Act}[\![\mathcal{E}_1]\!]\phi\{\mathbf{x}\}$

The function $\mathbf{Act}[\![\mathcal{E}]\!]\phi\mathbf{w}$ collects all the activations that arise in subexpressions of $\mathcal{E}$ needed to evaluate arguments in $\mathbf{w}$. The most important case is where an expression $f_j(\mathcal{E}_1, ..., \mathcal{E}_k)$ has to be evaluated. The activations from each of the arguments $\mathcal{E}_1, ..., \mathcal{E}_k$ are computed first; then the values of these arguments are computed (using $\mathbf{E}$), and the resulting tuple is added to the activations of f_j (using $only_j$, which ensures that no activations including $\perp$ are added).

Definition 2.9 complete activation environment
Let ϕ be the denotation of the functions $f_1, ..., f_n$, as given by the fixpoint equation in Definition 2.5. Then the complete activation environment generated by an initial activation environment **init** is the least solution for $\mathbf{a} = (\mathbf{a}_1, ..., \mathbf{a}_n)$ of the equation:

$$\mathbf{a} = \mathbf{init} \sqcup \mathbf{Act}[\![\mathcal{E}^1]\!]\psi\mathbf{a}_1 \sqcup ... \sqcup \mathbf{Act}[\![\mathcal{E}^n]\!]\psi\mathbf{a}_n$$

Since **Act** can be shown to be continuous in its last argument, the solution to this equation is the limit of the sequence of activation environments:

$$\mathbf{a}^0, \mathbf{a}^1, \mathbf{a}^2, ...$$

where $\mathbf{a}^0 = \emptyset^n$ and

$$\mathbf{a}^{i+1} = \mathbf{init} \sqcup \mathbf{Act}[\![\mathcal{E}^1]\!]\psi\mathbf{a}_1^i \sqcup ... \sqcup \mathbf{Act}[\![\mathcal{E}^n]\!]\psi\mathbf{a}_n^i.$$

Definition 2.10 restriction of a function environment
Let ϕ be a function environment and let $\mathbf{a}$ be an activation environment. The restriction of ϕ to $\mathbf{a}$, written $\phi|_{\mathbf{a}}$, is the function environment $(\phi_1|_{\mathbf{a}_1}, ..., \phi_n|_{\mathbf{a}_n})$, where $\phi_i|_{\mathbf{a}_i}(\mathbf{v}) =$ if $\mathbf{v} \in \mathbf{a}_i$ then $\phi_i(\mathbf{v})$ else $\perp$.

Definition 2.11 mfg
Let $f_1, ..., f_n$ be functions defined by expressions $\mathcal{E}^1, ..., \mathcal{E}^n$; let **init** be an activation environment, and let $\mathbf{a}$ be the complete activation environment generated from **init**. Let ϕ be the denotation of the functions $f_1, ..., f_n$, as given by the fixpoint equation in Definition 2.5. Then the mfg with respect to **init** is the function environment $\phi|_{\mathbf{a}}$.

That is, the mfg is the restriction of the functions to those values that are activated during the evaluation of **init**. (We could if desired use the restriction operator employed by Jones and Mycroft to distinguish the results of undefined values from uncalled values).

〔2〕 Computation of the mfg

We would like if possible to compute a converging sequence of pairs

$$(\mathbf{b}^0, \phi^0), (\mathbf{b}^1, \phi^1), (\mathbf{b}^2, \phi^2), \ldots$$

where $\mathbf{b}_0 = \emptyset^n$, ϕ^0 is the function enviroment that is everywhere undefined, and the limit yields the complete activation environment and the mfg. We might think of trying the following sequence:

$$\mathbf{b}^{i+1} = \mathbf{init} \sqcup \mathbf{Act}[\![\mathcal{E}^1]\!]\phi^i\mathbf{b}_1^i \sqcup \ldots \sqcup \mathbf{Act}[\![\mathcal{E}^n]\!]\phi^i\mathbf{b}_n^i, \text{ and}$$
$$\phi^{i+1} = (\mathbf{E}[\![\mathcal{E}^1]\!]\phi^i, \ldots, \mathbf{E}[\![\mathcal{E}^n]\!]\phi^i)|_{\mathbf{b}^i}.$$

but there are fundamental difficulties with this idea, as pointed out by Winsborough.[19] The function $\mathbf{Act}[\![\mathcal{E}]\!]\phi\mathbf{a}$ is not monotonic with respect to the argument ϕ; that is, $\phi_1 \preceq \phi_2 \not\Rightarrow \mathbf{Act}[\![\mathcal{E}]\!]\phi_1\mathbf{a} \sqsubseteq \mathbf{Act}[\![\mathcal{E}]\!]\phi_2\mathbf{a}$. This is shown in the following example.

Example 2
Let $\mathcal{E} = f_1(f_1(x))$. Suppose $\phi = \{a \mapsto b\}$ and $\psi = \{a \mapsto c\}$ where $b \sqsubseteq c$. Hence $\phi \preceq \psi$. Then $\mathbf{Act}[\![\mathcal{E}]\!]\phi\{a\} = \{a, b\}$, but $\mathbf{Act}[\![\mathcal{E}]\!]\psi\{a\} = \{a, c\}$. Hence the set of activations has not been increased by increasing the function environment.

Apart from the difficulty in showing that a fixed point exists, the example shows that some activations generated might be spurious, arising from partially defined results.

In Ref. 8), the mfg is computed by such a sequence, but the monotonicity problem does not appear because flat domains are used in the mfg intepretation, and thus there are no partial results except $\perp$, which is not an activation. However flat domains are not enough to define the Prolog semantics in the next section.

Winsborough approached this problem by showing how to compute another converging sequence whose limit yields an activation environment slightly larger than the required one (the *partial function graph* or *pfg*). Our approach is similar, but both the method and the result differ. A function $\mathbf{Act}'$ is introduced, which yields the *downward closure* of **Act**. The required monotonicity properties hold for $\mathbf{Act}'$. However we will not have to compute $\mathbf{Act}'$ explicitly.

Definition 2.12 downward closure
Let X be a set partially ordered by $\sqsubseteq$, and let $A \subseteq X$. A is downward closed if $x \in A$ implies $y \in A$ for all $y \sqsubseteq x$. The downward closure of a set A is the set $\{y \mid y \sqsubseteq x, x \in A\}$. Denote by $\lfloor A \rfloor$ *the* downward closure of A.

Definition 2.13 downward closure of activation environments
Let $\mathbf{a} \in \mathbf{W}^n$ be an activation environment. The downward closure of $\mathbf{a}$, $\lfloor \mathbf{a} \rfloor$, is $(\lfloor \mathbf{a}_1 \rfloor, \ldots, \lfloor \mathbf{a}_n \rfloor)$.

Definition 2.14 $\mathbf{Act}'$

$$\mathbf{Act}'[\![\mathcal{E}]\!]\phi\mathbf{a} = \lfloor \mathbf{Act}[\![\mathcal{E}]\!]\phi\mathbf{a} \rfloor$$

Lemma 2.15
$\mathbf{Act}'[\![\mathcal{E}]\!]\phi\mathbf{w}$ is monotonic in both its last and second last arguments.

Proof informal
Act is monotonic in its last argument, so it follows that **Act**′ is too.

For the second last argument, let ϕ_1, ϕ_2 be function environments, where $\phi_1 \leq \phi_2$: Let $\mathbf{w} \in \mathbf{W}$. We must show that $(\lfloor \mathbf{a}_1 \rfloor, ..., \lfloor \mathbf{a}_n \rfloor) \sqsubseteq (\lfloor \mathbf{b}_1 \rfloor, ..., \lfloor \mathbf{b}_n \rfloor)$, where $(\mathbf{a}_1, ..., \mathbf{a}_n) = \mathbf{Act}[\![\mathcal{E}]\!]\phi_1\mathbf{w}$ and $(\mathbf{b}_1, ..., \mathbf{b}_n) = \mathbf{Act}[\![\mathcal{E}]\!]\phi_2\mathbf{w}$, which is true if for each j, $1 \leq j \leq n$, $\lfloor \mathbf{a}_j \rfloor \subseteq \lfloor \mathbf{b}_j \rfloor$.

This is true if for each element $\mathbf{x} \in \mathbf{a}_j$, there exists some $\mathbf{y} \in \mathbf{b}_j$ such that $\mathbf{x} \sqsubseteq \mathbf{y}$, where $\sqsubseteq$ is the partial order on activations. The proof reduces to showing this by examining the cases in the definition of **Act**. □

Lemma 2.16
Let X be a set partially ordered by $\sqsubseteq$ and let $A, B \subseteq X$.
Then $\lfloor A \cup B \rfloor = \lfloor A \rfloor \cup \lfloor B \rfloor$.

Proof

$$\begin{aligned} x \in \lfloor A \cup B \rfloor &\equiv \exists y.\ x \sqsubseteq y \wedge (y \in A \vee y \in B) \\ &\equiv \exists y.\ (x \subseteq y \wedge y \in A) \vee (x \sqsubseteq y \wedge y \in B) \\ &\equiv \exists y.\ (x \subseteq y \wedge y \in A) \wedge \exists y.\ (x \sqsubseteq y \wedge y \in B) \\ &\equiv x \in \lfloor A \rfloor \vee x \in \lfloor B \rfloor \\ &\equiv x \in \lfloor A \rfloor \vee \lfloor B \rfloor \quad \square \end{aligned}$$

Lemma 2.17
Let $\mathbf{a}, \mathbf{b} \in \mathbf{W}^n$. Then $\lfloor \mathbf{a} \sqcup \mathbf{b} \rfloor = \lfloor \mathbf{a} \rfloor \sqcup \lfloor \mathbf{b} \rfloor$.

Proof

$$\begin{aligned} \lfloor \mathbf{a} \sqcup \mathbf{b} \rfloor &= \lfloor (\mathbf{a}_1 \cup \mathbf{b}_1, ..., \mathbf{a}_n \cup \mathbf{b}_n) \rfloor \\ &= (\lfloor \mathbf{a}_1 \cup \mathbf{b}_1 \rfloor, ..., \lfloor \mathbf{a}_n \cup \mathbf{b}_n \rfloor) \text{ by Definition 2.13} \\ &= (\lfloor \mathbf{a}_1 \rfloor \cup \lfloor \mathbf{b}_1 \rfloor, ..., \lfloor \mathbf{a}_n \rfloor \cup \lfloor \mathbf{b}_n \rfloor) \text{ by Lemma 2.16} \\ &= \lfloor \mathbf{a} \rfloor \sqcup \lfloor \mathbf{b} \rfloor \text{ by Definitions 2.6 and 2.13} \quad \square \end{aligned}$$

Now we show that downward closures need not always be explicitly represented when computing **Act**′.

Lemma 2.18

$$\mathbf{Act}'[\![\mathcal{E}]\!]\phi\lfloor \mathbf{a} \rfloor = \mathbf{Act}'[\![\mathcal{E}]\!]\phi\mathbf{a}$$

Proof
The proof is by induction on the depth of the expression $\mathcal{E}$, called $depth(\mathcal{E})$. $depth(\mathcal{E})$ is defined as

- 0 if $\mathcal{E}$ is a variable or a constant,
- $1 + max\{depth(\mathcal{E}_1), ..., depth(\mathcal{E}_k)\}$ if $\mathcal{E} = f_j(\mathcal{E}_1, ..., \mathcal{E}_k)$ or $\mathcal{E} = a_j(\mathcal{E}_1, ..., \mathcal{E}_k)$,

- $1 + max\{depth(\mathcal{E}_1), depth(\mathcal{E}_2), depth(\mathcal{E}_3)\}$ if $\mathcal{E} = if\ \mathcal{E}_1$ then $\mathcal{E}_2$ else $\mathcal{E}_3$.

If $depth(\mathcal{E}) = 0$ then the result holds since $\lfloor \emptyset^n \rfloor = \emptyset^n$.

Suppose the result holds for all $\mathcal{E}$ such that $depth(\mathcal{E}) \leq n$. Now show that it holds where $depth(\mathcal{E}) \leq n + 1$. There are three cases, $\mathcal{E} = a_j(\mathcal{E}_1, ..., \mathcal{E}_k)$, $\mathcal{E} = f_j(\mathcal{E}_1, ..., \mathcal{E}_k)$ and $\mathcal{E} =$ if $\mathcal{E}_1$ then $\mathcal{E}_2$ else $\mathcal{E}_3$. We prove it just for $\mathcal{E} = f_j(\mathcal{E}_1, ..., \mathcal{E}_k)$; the other two cases follow a similar pattern.

$$
\begin{aligned}
&\mathbf{Act}'[\![f_j(\mathcal{E}_1, ..., \mathcal{E}_k)]\!]\phi\lfloor \mathbf{w} \rfloor \\
&= \left\lfloor \begin{array}{c} \bigsqcup\{\mathbf{Act}[\![\mathcal{E}_i]\!]\phi\lfloor \mathbf{w} \rfloor \mid 1 \leq i \leq \mathrm{k}\} \\ \bigsqcup\bigsqcup\{only_j((\mathbf{E}[\![\mathcal{E}_1]\!]\phi\mathbf{x}, ..., \mathbf{E}[\![\mathcal{E}_k]\!]\phi\mathbf{x})) \mid \mathbf{x} \in \lfloor \mathbf{w} \rfloor\} \end{array} \right\rfloor \\
&= \bigsqcup\{\lfloor \mathbf{Act}[\![\mathcal{E}_i]\!]\phi\lfloor \mathbf{w} \rfloor \rfloor \mid 1 \leq i \leq k\} \\
&\quad \bigsqcup\bigsqcup\{\lfloor only_j((\mathbf{E}[\![\mathcal{E}_1]\!]\phi\mathbf{x}, ..., \mathbf{E}[\![\mathcal{E}_k]\!]\phi\mathbf{x}))\rfloor \mid \mathbf{x} \in \lfloor \mathbf{w} \rfloor\} \\
&\quad \text{(by Lemma 2.17)} \\
&= \bigsqcup\{\lfloor \mathbf{Act}[\![\mathcal{E}_i]\!]\phi\mathbf{w} \rfloor \mid 1 \leq i \leq k\} \\
&\quad \bigsqcup\bigsqcup\{\lfloor only_j((\mathbf{E}[\![\mathcal{E}_1]\!]\phi\mathbf{x}, ..., \mathbf{E}[\![\mathcal{E}_k]\!]\phi\mathbf{x}))\rfloor \mid \mathbf{x} \in \lfloor \mathbf{w} \rfloor\} \\
&\quad \text{(by induction hypothesis)} \\
&= \bigsqcup\{\lfloor \mathbf{Act}[\![\mathcal{E}_i]\!]\phi\mathbf{w} \rfloor \mid 1 \leq i \leq k\} \\
&\quad \bigsqcup\bigsqcup\{\lfloor only_j((\mathbf{E}[\![\mathcal{E}_1]\!]\phi\mathbf{x}, ..., \mathbf{E}[\![\mathcal{E}_k]\!]\phi\mathbf{x}))\rfloor \mid \mathbf{x} \in \mathbf{w}\} \\
&\quad \text{(since every element of } \lfloor \mathbf{w} \rfloor \text{ has an upper bound in } \mathbf{w}, \text{ and } \mathbf{E} \text{ is} \\
&\quad \text{monotonic in } \mathbf{x}) \\
&= \left\lfloor \begin{array}{c} \bigsqcup\{\mathbf{Act}[\![\mathcal{E}_i]\!]\phi\mathbf{w} \mid 1 \leq i \leq k\} \\ \bigsqcup\bigsqcup\{only_j((\mathbf{E}[\![\mathcal{E}_1]\!]\phi\mathbf{x}, ..., \mathbf{E}[\![\mathcal{E}_k]\!]\phi\mathbf{x})) \mid \mathbf{x} \in \mathbf{w}\} \end{array} \right\rfloor \\
&\quad \text{(by Lemma 2.17)} \\
&= \mathbf{Act}'[\![f_j(\mathcal{E}_1, ..., \mathcal{E}_k)]\!]\phi\mathbf{w} \quad \square
\end{aligned}
$$

Lemma 2.19

Let $\mathbf{a}$ be an activation environment, and let

$$
\begin{aligned}
\mathbf{b} &= \mathbf{init} \sqcup \mathbf{Act}[\![\mathcal{E}^1]\!]\psi\mathbf{a}_1 \sqcup ... \sqcup \mathbf{Act}[\![\mathcal{E}^n]\!]\psi\mathbf{a}_n. \\
\mathbf{c} &= \lfloor \mathbf{init} \rfloor \sqcup \mathbf{Act}'[\![\mathcal{E}^1]\!]\phi\lfloor \mathbf{a}_1 \rfloor \sqcup ... \sqcup \mathbf{Act}'[\![\mathcal{E}^n]\!]\phi\lfloor \mathbf{a}_n \rfloor.
\end{aligned}
$$

Then $\lfloor \mathbf{b} \rfloor = \mathbf{c}$.

Proof

$$
\begin{aligned}
\lfloor \mathbf{b} \rfloor &= \lfloor \mathbf{init} \sqcup \mathbf{Act}[\![\mathcal{E}^1]\!]\psi\mathbf{a}_1 \sqcup ... \sqcup \mathbf{Act}[\![\mathcal{E}^n]\!]\psi\mathbf{a}_n \rfloor \\
&= \lfloor \mathbf{init} \rfloor \sqcup \lfloor \mathbf{Act}[\![\mathcal{E}^1]\!]\psi\mathbf{a}_1 \rfloor \sqcup ... \sqcup \lfloor \mathbf{Act}[\![\mathcal{E}^n]\!]\psi\mathbf{a}_n \rfloor \\
&= \lfloor \mathbf{init} \rfloor \sqcup \lfloor \mathbf{Act}[\![\mathcal{E}^1]\!]\phi\lfloor \mathbf{a}_1 \rfloor \rfloor \sqcup ... \sqcup \lfloor \mathbf{Act}[\![\mathcal{E}^n]\!]\phi\lfloor \mathbf{a}_n \rfloor \rfloor \\
&= \mathbf{c} \quad \square
\end{aligned}
$$

(3) Algorithms for computing the mfg

Consider the sequence (referred to as S_1)

$$(\mathbf{b}^0, \phi^0), (\mathbf{b}^1, \phi^1), (\mathbf{b}^2, \phi^2), ...$$

where $\mathbf{b}^0 = \emptyset^n$, ϕ^0 is the everywhere-undefined function environment, and

$$\mathbf{b}^{i+1} = \lfloor \mathbf{init} \rfloor \sqcup \mathbf{Act}'[\![\mathcal{E}^1]\!] \phi^i \mathbf{b}_1^i \sqcup \ldots \sqcup \mathbf{Act}'[\![\mathcal{E}^n]\!] \phi^i \mathbf{b}_n^i, \text{ and}$$
$$\phi^{i+1} = (\mathbf{E}[\![\mathcal{E}^1]\!] \phi^i, \ldots, \mathbf{E}[\![\mathcal{E}^n]\!] \phi^i)|_{\mathbf{b}^i}.$$

S_1 is monotonically increasing and converges due to the continuity of **Act′** in its last two arguments, and the continuity of **E**. We will assume finite convergence, (that is, that there are no infinite ascending chains in the domain X).

The sequence S_2 is defined as

$$(\mathbf{a}^0, \phi^0), (\mathbf{a}^1, \phi^1), (\mathbf{a}^2, \phi^2), \ldots$$

with $\mathbf{a}^0 = \emptyset^n$, ϕ^0 the everywhere-undefined function environment, and

$$\mathbf{a}^{i+1} = \mathbf{init} \sqcup \mathbf{Act}[\![\mathcal{E}^1]\!] \phi^i \mathbf{a}_1^i \sqcup \ldots \sqcup \mathbf{Act}[\![\mathcal{E}^n]\!] \phi^i \mathbf{a}_n^i, \text{ and}$$
$$\phi^{i+1} = (\mathbf{E}[\![\mathcal{E}^1]\!] \phi^i, \ldots, \mathbf{E}[\![\mathcal{E}^n]\!] \phi^i)|_{\lfloor \mathbf{a}^i \rfloor}.$$

Note that in S_2 the function environment ϕ^{i+1} is restricted by the downward closure of $\mathbf{a}^i$.

Lemma 2.20

$$S_1 = (\lfloor \mathbf{a}^0 \rfloor, \phi^0), (\lfloor \mathbf{a}^1 \rfloor, \phi^1), (\lfloor \mathbf{a}^2 \rfloor, \phi^2), \ldots$$

Proof
Using Lemma 2.19 and simple induction on the length of the sequence. □

Lemma 2.21
The sequence S_2 converges.

Proof
Let $S_1 = (\mathbf{b}^0, \phi^0), (\mathbf{b}^1, \phi^1), (\mathbf{b}^2, \phi^2), \ldots$, $S_2 = (\mathbf{a}^0, \phi^0), (\mathbf{a}^1, \phi^1), (\mathbf{a}^2, \phi^2), \ldots$, as defined above.

The sequence S_1 converges: suppose $(\mathbf{b}^{i+m}, \phi^{i+m}) = (\mathbf{b}^i, \phi^i)$, ($m = 1, 2, \ldots$). Then by Lemma 2.20, $(\mathbf{b}^i, \phi^i) = (\lfloor \mathbf{a}^i \rfloor, \phi^i)$, and hence $\phi^i = \phi^{i+m}$, ($m = 1, 2, \ldots$).

Act is monotonic in its last argument, hence (with the assumption of no infinite ascending chains in $\mathbf{W}^n$) there exists $k \geq i$ such that $(\mathbf{a}^{k+1}, \phi^{k+1}) = (\mathbf{a}^k, \phi^k)$. □

Example 3
Take the factorial function and compute sequence S_2 with initial activation *fact*(2). The set of natural numbers is augmented by $\perp$ and the "flat" partial order $n \sqsubseteq m$ iff $n = \perp$ is used. Represent a partial factorial function by the set of argument-value pairs $n \mapsto fact(n)$, $fact(n) \neq \perp$. In the computation this represents the values of the factorial function that have been computed "so far". Thus $\perp_{fact}$ is represented by $\emptyset$. The computation of the sequence S_2 gives rise to the following sequence of values:

$\phi^0 = \emptyset$, $\mathbf{a}^0 = \emptyset$ $\quad$ $\phi^1 = \emptyset$, $\mathbf{a}^1 = \{2\}$
$\phi^2 = \emptyset$, $\mathbf{a}^2 = \{2, 1\}$ $\quad$ $\phi^3 = \emptyset$, $\mathbf{a}^3 = \{2, 1, 0\}$
$\phi^4 = \{0 \mapsto 1\}$, $\mathbf{a}^4 = \{2, 1, 0\}$ $\quad$ $\phi^5 = \{0 \mapsto 1, 1 \mapsto 1\}$, $\mathbf{a}^5 = \{2, 1, 0\}$
$\phi^6 = \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2\}$, $\mathbf{a}^6 = \{2, 1, 0\}$ $\quad$ $\phi^7 = \phi^6$, $\mathbf{a}^7 = \mathbf{a}^6$

Note that in this small example the set of activations is independent from the function environment; in general, activations and results are interdependent, as will be seen later in the mfg for Prolog.

Computation of the sequence S_2 is the basis of algorithms for program analysis. During the computation of S_2, the sequence of activation environments does not necessarily monotonically increase, but the corresponding downward closed activation environments do increase, as we can see by looking at the respective elements in S_1. The result is different from Winsborough's *pfg*, where the set of activations only increases. The pfg would result if we replaced the computation of $\mathbf{a}^{i+1}$ in S_2 by:

$$\mathbf{a}^{i+1} = \mathbf{a}^i \sqcup \mathbf{init} \sqcup \mathbf{Act}[\![\ \mathcal{E}^1\ ]\!]\phi^i\mathbf{a}_1^i \sqcup \ldots \sqcup \mathbf{Act}[\![\ \mathcal{E}^n\ ]\!]\phi^i\mathbf{a}_n^i$$

to ensure that $\mathbf{a}^i \sqsubseteq \mathbf{a}^{i+1}$. The point of convergence of S_2 is even closer to the mfg than the pfg; in the pfg, any spurious activations arising from partial function environments accumulate in the activation environment and appear in the fixed point, but in S_2, such spurious activations do get eliminated, unless they are self-perpetuating. (Self-perpetuating activations are caused by circular definitions such as $f(x) = \ldots f(x) \ldots$; clearly any activation of f will give rise to another identical one). But we have not managed to get a direct computation of the mfg, due to the problem of self-perpetuating activations.

The correctness of algorithms based on computing the sequence S_2 is expressed in the following proposition, whose proof is sketched. The proposition states in effect that given an initial activation **init**, the function computed by S_2 is greater than the mfg associated with **init**.

Proposition 2.22
Let **init** be an activation environment and let:

- $\phi = (\mathbf{E}[\![\ \mathcal{E}^1\ ]\!]\phi, \ldots, \mathbf{E}[\![\ \mathcal{E}^n\ ]\!]\phi)$
- $\mathbf{b} = \mathbf{init} \sqcup \mathbf{Act}[\![\ \mathcal{E}^1\ ]\!]\phi\mathbf{b}_1 \sqcup \ldots \sqcup \mathbf{Act}[\![\ \mathcal{E}^n\ ]\!]\phi\mathbf{b}_n$
- $\psi = (\mathbf{E}[\![\ \mathcal{E}^1\ ]\!]\psi, \ldots, \mathbf{E}[\![\ \mathcal{E}^n\ ]\!]\psi)|_{\mathbf{a}}$
- $\mathbf{a} = \mathbf{init} \sqcup \mathbf{Act}[\![\ \mathcal{E}^1\ ]\!]\psi\mathbf{a}_1 \sqcup \ldots \sqcup \mathbf{Act}[\![\ \mathcal{E}^n\ ]\!]\psi\mathbf{a}_n$

Then $\mathbf{v} \in \mathbf{b}_j \Rightarrow \phi_j(\mathbf{v}) = \psi_j(\mathbf{v})$.

Proof (outline)
The result is proved by induction on the *recursion depth* associated with evaluating the activations in **init**. The recursion depth of an activation **v**, in an environment ϕ, is defined by **R**:

$$\mathbf{R}[\![\ x_i\ ]\!]\phi\mathbf{v} = 0$$

$$\mathbf{R}[\![c_i]\!]\phi\mathbf{v} = 0$$
$$\mathbf{R}[\![a_j(\mathcal{E}_1, ..., \mathcal{E}_k)]\!]\phi\mathbf{v} = 1 + max\{\mathbf{R}[\![\mathcal{E}_1]\!]\phi\mathbf{v}, ..., \mathbf{R}[\![\mathcal{E}_k]\!]\phi\mathbf{v}\}$$
$$\mathbf{R}[\![f_j(\mathcal{E}_1, ..., \mathcal{E}_k)]\!]\phi\mathbf{v} = 1 + max\{\mathbf{R}[\![\mathcal{E}_1]\!]\phi\mathbf{v}, ..., \mathbf{R}[\![\mathcal{E}_k]\!]\phi\mathbf{v}\}$$
$$\mathbf{E}[\![\text{if } \mathcal{E}_1 \textit{ then } \mathcal{E}_2 \text{ else } \mathcal{E}_3]\!]\phi\mathbf{v} =$$
$$\quad \text{if } \phi(\mathbf{v}) = \textit{true} \text{ then } 1 + max\{\mathbf{R}[\![\mathcal{E}_1]\!]\phi\mathbf{v}, \mathbf{R}[\![\mathcal{E}_2]\!]\phi\mathbf{v}\}$$
$$\quad \text{else } 1 + max\{\mathbf{R}[\![\mathcal{E}_1]\!]\phi\mathbf{v}, \mathbf{R}[\![\mathcal{E}_3]\!]\phi\mathbf{v}\}$$

Basis: Suppose that for all $\mathbf{x} \in \mathbf{init}_j$ $(1 \leq j \leq n)$, $\mathbf{R}[\![\mathcal{E}^j]\!]\phi\mathbf{x} = 0$; then $\mathbf{init} = \mathbf{b} = \mathbf{a}$ and $\phi_j(\mathbf{x}) = \psi_j(\mathbf{x})$.

Induction: Assume that the result holds for all **init** such that for all $\mathbf{x} \in \mathbf{init}_j$ $(1 \leq j \leq n)$, $\mathbf{R}[\![\mathcal{E}^j]\!]\phi\mathbf{x} \leq m$.

Now take **init** with some $\mathbf{v} \in \mathbf{init}_j$, and suppose $\mathbf{R}[\![\mathcal{E}^j]\!]\phi\mathbf{v} = m + 1$. $\phi_j(\mathbf{v}) = \mathbf{E}[\![\mathcal{E}^j]\!]\phi\mathbf{v}$; by examining the structure of $\mathcal{E}^j$, we can construct another set of activations, **init**′, replacing **v** by "subcalls" in $\mathcal{E}^j$. Using the induction hypothesis we show that $\mathbf{E}[\![\mathcal{E}^j]\!]\phi\mathbf{v} = \mathbf{E}[\![\mathcal{E}^j]\!]\psi\mathbf{v}$, and that the activations of the subcalls are in **a**. Hence $\phi_j(\mathbf{v}) = \psi_j(\mathbf{v})$.

Hence we can conclude that the result holds for all **init**. □

[4] Efficient algorithms

We now consider how to implement the computation of the sequence S_2 efficiently. We can examine important aspects of efficiency without knowing the details of the functions $f_1, ..., f_n$, or the details of the data structures used for storing the activations and the functions, though further optimisation may be possible if these are known (as we will see for in the case of the Prolog functions). The naive enumeration of the elements of S_2, using the equations above, gives the following algorithm. A function environment is represented by that part of the functions that are defined.

begin
$\quad \mathbf{a}^0 \leftarrow \emptyset^n$
$\quad \psi^0 \leftarrow \emptyset^n$
$\quad i \leftarrow 0$
$\quad$ **repeat**
$\quad\quad \mathbf{a}^{i+1} \leftarrow \mathbf{init} \sqcup \mathbf{Act}[\![\mathcal{E}^1]\!]\psi^i\mathbf{a}_1^i \sqcup ... \sqcup \mathbf{Act}[\![\mathcal{E}^n]\!]\psi^i\mathbf{a}_n^i$
$\quad\quad \psi^{i+1} \leftarrow (\mathbf{E}[\![\mathcal{E}^1]\!]\psi^i, ..., \mathbf{E}[\![\mathcal{E}^n]\!]\psi^i)|_{\lfloor \mathbf{a}^i \rfloor}$
$\quad\quad i \leftarrow i + 1$
$\quad$ **unitl** $\mathbf{a}^{i+1} = \mathbf{a}^i$ and $\psi^{i+1} = \psi^i$
end

This is not very efficient since the calculation of $(\mathbf{a}^{i+1}, \psi^{i+1})$ repeats much of the work done in computing $(\mathbf{a}^i, \psi^i)$, and the checking of the termination condition is expensive. An improvement can be made by making use of the *new* function values and activations computed on each iteration.

Each ψ^i computed is expressible as $\psi^{i-1} \sqcup \psi^{new}$, for some ψ^{new}, since the

function environment monotonically increases. Thus the expression computing ψ^{i+1} can be rewritten (using continuity of **E** and $|$) as follows:

$$\begin{aligned}\psi^{i+1} &= (\mathbf{E}[\![\ \mathcal{E}^1]\!](\psi^{i-1} \sqcup \psi^{new}), \ldots, \mathbf{E}[\![\ \mathcal{E}^n]\!](\psi^{i-1} \sqcup \psi^{new}))|_{\lfloor \mathbf{a}^i \rfloor} \\ &= (\mathbf{E}[\![\ \mathcal{E}^1]\!]\psi^{i-1}, \ldots, \mathbf{E}[\![\ \mathcal{E}^n]\!](\psi^{i-1})|_{\lfloor \mathbf{a}^i \rfloor} \\ &\quad \sqcup (\mathbf{E}[\![\mathcal{E}^1]\!]\psi^{new}, \ldots, \mathbf{E}[\![\ \mathcal{E}^1]\!]\psi^{new})|_{\lfloor \mathbf{a}^i \rfloor}\end{aligned}$$

Similarly, since the (downward closed) activation sets are monotonically increasing, we can write $\lfloor \mathbf{a}^i \rfloor$ as $\lfloor \mathbf{a}_{i-1} \rfloor \sqcup \mathbf{a}^{new}$, for some activation environment $\mathbf{a}^{new}$. Thus again using continuity we can rewrite the above expression for ψ^{i+1} into four subexpressions:

$$\begin{aligned}(\mathbf{E}[\![\ \mathcal{E}^1]\!]\psi^i, \ldots, \mathbf{E}[\![\ \mathcal{E}^1]\!]\psi^i)|_{\lfloor \mathbf{a}^i \rfloor} = {} & (\mathbf{E}[\![\ \mathcal{E}^1]\!]\psi^{i-1}, \ldots, \mathbf{E}[\![\ \mathcal{E}^1]\!]\psi^{i-1})|_{\lfloor \mathbf{a}^{i-1} \rfloor} \\ & \sqcup (\mathbf{E}[\![\ \mathcal{E}^1]\!]\psi^{i-1}, \ldots, \mathbf{E}[\![\ \mathcal{E}^1]\!]\psi^{i-1})|_{\mathbf{a}^{new}} \\ & \sqcup (\mathbf{E}[\![\ \mathcal{E}^1]\!]\psi^{new}, \ldots, \mathbf{E}[\![\ \mathcal{E}^1]\!]\psi^{new})|_{\lfloor \mathbf{a}^{i-1} \rfloor} \\ & \sqcup (\mathbf{E}[\![\ \mathcal{E}^1]\!]\psi^{new}, \ldots, \mathbf{E}[\![\ \mathcal{E}^1]\!]\psi^{new})|_{\mathbf{a}^{new}}\end{aligned}$$

The first of these subexpressions equals ψ^i, which has already been computed on the previous iteration. The other three represent the use of the information in the new part of ψ^i and $\mathbf{a}^i$.

We can use this analysis to give a more efficient algorithm. Termination occurs when some iteration adds nothing new. This algorithm can be compared to algorithms that build a computation tree or similar structures; the function and activation environments are incrementally extended on each iteration. New activations occur at the "leaves" of the computation tree rooted at some activation in **init**; the tree may be extended either by using the new activations, or by using new function values on previous activations. The previous algorithm, on the other hand, corresponded to a recomputation of the tree from its root on each iteration.

The *new* parts ψ^{new} and $\mathbf{a}^{new}$ can be computed on each iteration, and then used in the following iteration. In the following algorithm, ϕ^1, ϕ^2 and ϕ^3 compute the subexpressions identified above. The operator $\phi_j - \psi_j$ on functions computes the function $\phi_j|_D$ where $D = \{\mathbf{x} \mid \phi_j(\mathbf{x}) \sqsubseteq \phi_j(\mathbf{x})\}$, and the operator is extended componentwise to function environments.

begin

$\mathbf{a}^0 \leftarrow \emptyset^n$; $\mathbf{a}^{new} \leftarrow \emptyset^n$

$\psi^0 \leftarrow \emptyset^n$; $\psi^{new} \leftarrow \emptyset^n$

$i \leftarrow 0$

repeat

$\mathbf{a}^{i+1} \leftarrow \mathbf{init} \sqcup \mathbf{Act}[\![\ \mathcal{E}^1]\!]\psi^i\mathbf{a}^i_1 \sqcup \ldots \sqcup \mathbf{Act}[\![\ \mathcal{E}^n]\!]\psi^i\mathbf{a}^i_n$

$\phi^1 \leftarrow (\mathbf{E}[\![\ \mathcal{E}^1]\!]\psi^{new}, \ldots, \mathbf{E}[\![\ \mathcal{E}^n]\!]\psi^{new})|_{\lfloor \mathbf{a}^{i-1} \rfloor}$

$\phi^2 \leftarrow (\mathbf{E}[\![\ \mathcal{E}^1]\!]\psi^{i-1}, \ldots, \mathbf{E}[\![\ \mathcal{E}^n]\!]\psi^{i-1})|_{\mathbf{a}^{new}}$

$\phi^3 \leftarrow (\mathbf{E}[\![\ \mathcal{E}^1]\!]\psi^{new}, \ldots, \mathbf{E}[\![\ \mathcal{E}^n]\!]\psi^{new})|_{\mathbf{a}^{new}}$

$\psi^{i+1} \leftarrow \psi^i \sqcup \phi^1 \sqcup \phi^2 \sqcup \phi^3$

$$\psi^{new} \leftarrow \psi^{i+1} - \psi^i$$
$$\mathbf{a}^{new} \leftarrow \mathbf{a}^{i+1} - \mathbf{a}^i$$
$$i \leftarrow i + 1$$
until $\mathbf{a}^{new} = \emptyset$ and $\psi^{new} = \emptyset$

end

Since we expect the number of new values on some iterations to be relatively small compared with the total number of accumulated values, this algorithm is much more efficient, and the termination check is also much more efficient. The extra work of computing $\mathbf{a}^{new}$ and ψ^{new} could be reduced by combining their computation with that of $\mathbf{a}^{i+1}$ and ψ^{i+1} respectively.

Note that $\mathbf{a}^{i+1}$ is not necessarily greater than $\mathbf{a}^i$. This means that we cannot optimise the computation of $\mathbf{a}^{i+1}$ in the same way as ψ^{i+1}. A further adjustment to the algorithm could be to compute essentially Winsborough's pfg, replacing the assignment to $\mathbf{a}^{i+1}$ by the assignments:

$$\mathbf{b}_1 \leftarrow \mathbf{init} \sqcup \mathbf{Act}[\![\ \mathcal{E}^1\ ]\!]\psi^{new}\mathbf{a}_1^{i-1} \sqcup \ldots \sqcup \mathbf{Act}[\![\ \mathcal{E}^n\ ]\!]\psi^{new}\mathbf{a}_n^{i-1}$$
$$\mathbf{b}_2 \leftarrow \mathbf{Act}[\![\ \mathcal{E}^1\ ]\!]\psi^{i-1}\mathbf{a}_1^{new} \sqcup \ldots \sqcup \mathbf{Act}[\![\ \mathcal{E}^n\ ]\!]\psi^{i-1}\mathbf{a}_n^{new}$$
$$\mathbf{b}_3 \leftarrow \mathbf{Act}[\![\ \mathcal{E}^1\ ]\!]\psi^{new}\mathbf{a}_1^{new} \sqcup \ldots \sqcup \mathbf{Act}[\![\ \mathcal{E}^n\ ]\!]\psi^{new}\mathbf{a}_n^{new}$$
$$\mathbf{a}^{i+1} \leftarrow \mathbf{a}_i \sqcup \mathbf{b}_1 \sqcup \mathbf{b}_2 \sqcup \mathbf{b}_3$$

This would give a strictly increasing activation environment, and it suggests that the pfg may be more efficiently computable than the more precise result.

2.2 Computing the mfg: Discussion

We have defined the mfg and shown that the sequence S_2 converges to something close to the mfg. We followed the line of previous work, but we were able to achieve a theoretical improvement over previous attempts, and get closer to a direct computation of the mfg. However the question of whether the mfg can be computed directly in this framework with one converging sequence is still open. Our method used downward closures to construct an increasing set of activations, and then showed how to avoid the explicit computation of downward closures. The mfg itself could be computed with a second fixpoint computation, using the first fixpoint result, as Winsborough points out.

The use of a *basis* of the set of activations instead of downward closures would also have been possible; in the case of Prolog, this could mean defining activations in terms of singleton sets of substitutions instead of arbitrary sets of substitutions (as used by Jones and Søndergaard[7]).

We outlined the development of an efficient algorithm for computing the sequence S_2. The main idea was the incremental extension of the sets of activations and function values on each iteration.

§3 A Minimal Function Graph for Prolog

Using the general scheme of Section 2.1 we can now define an mfg constructor for Prolog from the functions giving operational semantics of (definite clause) Prolog. Program analyses are based on safe approximations of program behaviour. The semantic functions below give a so-called *core* semantics of Prolog, which can be interpreted over different domains to give different approximations, including the precise, concrete semantics. We will not go further into aspects of safe approximation or abstract interpretations here; the explanations in Refs. 7) and 3) and elsewhere can be consulted. The computation of a goal is defined in terms of meanings of program parts, including clause bodies and atoms. Thus the minimal function graph for a given goal will identify the activations of program parts—information that can be directly related to the text of the program.

3.1 Semantic Function for Prolog

The core semantic functions given by Jones and Søndergaard are taken as a basis. Please refer to Ref. 7) for further details. These functions can be *interpreted* by giving definitions to the undefined domains and functions. The usual semantics can also be obtained in this way. The renaming index used in Ref. 7) is omitted. Renamings can be handled in the concrete semantics by the technique of canonical terms and substitutions as used, for example, in Refs. 13, 20) and 12).

Definition 3.1
Let a program consist of a set of definite clauses $\{h_1 \leftarrow b_1, ..., h_n \leftarrow b_n\}$. Let **Asub** be a partially-ordered set consisting of descriptions of sets of substitutions. An operation $\sqcup$ computes the least upper bound of a set of elements of **Asub**. *Body* denotes the set of conjunctions, and *Atom* the set of atoms. The empty conjunction is denoted $[]$, and non-empty conjunctions as $[a_1, ..., a_j]$.

The syntax of the function definitions differs slightly from that used in the previous section; conditional expressions $b_1 \rightarrow \mathcal{E}$ are used instead of if-then-else expressions, and the use of the set in **A** may be considered simply as shorthand since it contains a fixed finite number of elements.

- **Core Semantic Functions**:

  B: *Body* × **Asub** → **Asub**
  A: *Atom* × **Asub** → **Asub**

- **Domain-dependent Functions**:

  call: *Atom* × **Asub** × *Atom* → **Asub**
  return: **Asub** × *Atom* × **Asub** × *Atom* → **Asub**
  δ: *Atom* × **Asub** → **Asub**

- **Definitions**:

$$\mathbf{B}(b, \Theta) = b = [] \rightarrow \Theta;$$
$$b = [a_1, ..., a_i] \rightarrow \mathbf{A}(a_i, \delta(a_i, \mathbf{B}([a_1, ..., a_{i-1}], \Theta)))$$
$$\mathbf{A}(a, \Theta) = \bigcup\{return(\mathbf{B}(b_j, call(h_j, \Theta, a)), h_j, \Theta, a) \mid 1 \leq j \leq n\}$$

Let $a_0 \in Atom$ and $\Theta_0 \in \mathbf{Asub}$. Then the answer to a_0 with Θ_0 is defined to be $\mathbf{A}(a_0, \Theta_0)$.

The function **B** gives the meaning of a conjunction with an abstract substitution. It computes the conjunction left to right, calling the function **A** on each atom in the conjunction; the activation of atom a_j in a conjunction $[a_1, ..., a_j]$ depends on the result of evaluating $[a_1, ..., a_{j-1}]$. The function **A** computes the meaning of an atom a by computing the set of meanings of the bodies of clauses whose heads unify with a, and finding the least upper bound of these results. The functions *call* and *return* perform similar roles as their counterparts defined in Ref. 7), namely, calling a clause and returning the result. In the concrete semantics *call* performs renaming and unification, and *return* composes the body meaning with the calling substitution.

We add one domain-dependent function, δ, not contained in Ref. 7). Its purpose is to make an approximation immediately before each atom is called. Say that a conjunction $[a_1, ..., a_j]$ is being evaluated, and that $[a_1, ..., a_{j-1}]$ has a result Θ. Instead of activating a_j with Θ, as in the concrete semantics, some generalisation of Θ is made and used to activate a_j. For safe interpretations it is required that $\Theta \subseteq \delta(a, \Theta)$. This does not enrich the framework, but we find it more convenient for our application. In the concrete semantics, $\delta(a, \Theta) = \Theta$.

The *call*, *return* and δ functions, together with definitions for **Asub** and its lub operator, define an interpretation of these equations.

3.2 Mfg Construction for Prolog

The algorithm for computing the mfg for Prolog can be constructed according to the patten in Section 2.1[3], by constructing the sequence S_2. The elements of the construction are summarised as follows:

(1) The functions $f_1, ..., f_n$ of the general construction consist of the two functions **A**, **B**. The functions $a_1, a_2, ...$ consist of *call*, *return*, $\delta, \sqcup$.
(2) An activation environment for **A** and **B** is a pair of sets (A, B), where $A \subseteq Atom \times \mathbf{Asub}$ and $B \subseteq Body \times \mathbf{Asub}$.
(3) A function environment for **A** and B is a pair (ϕ, ψ), where ϕ is of type $Atom \times \mathbf{Asub} \rightarrow \mathbf{Asub}$ and ψ is of type $Body \times \mathbf{Asub} \rightarrow \mathbf{Asub}$.
(4) An initial activation will be taken to be a call to **A**, that is, a pair $(a_0, \Theta_0) \in (Atom \times \mathbf{Asub})$.

Assuming the continuity of the functions, a converging sequence of sets of activations and a sequence of answers for activated values are computed, as

shown in Section 2.1[3]; we have

- $(A^0, B^0) = (\emptyset, \emptyset)$,
- ϕ^0 and ψ^0 are everywhere-undefined functions,
- (A^i, B^i) are the sets of activations of **A** and **B** in the i-th element of the sequence S_2,
- (ϕ^i, ψ^i) are the i-th function environment for **A** and **B** in the sequence S_2.

Thus the following outline algorithm can be obtained directly from the one at the start of Section 2.1[4], where $\mathcal{E}^A$ and $\mathcal{E}^B$ are the expressions defining **A** and **B** repectively. $\sqcup$ stands for the pairwise union of a pair of sets.

begin
$(A^0, B^0) \leftarrow (\emptyset, \emptyset)$
$(\phi^0, \psi^0) \leftarrow (\emptyset, \emptyset)$
$i \leftarrow 0$
repeat
$(A^{i+1}, B^{i+1}) \leftarrow (\{(a_0, \Theta_0)\}, \emptyset)$
$\sqcup$ **Act**$[\![\mathcal{E}^A]\!](\phi^i, \psi^i)A^i$
$\sqcup$ **Act**$[\![\mathcal{E}^B]\!](\phi^i, \psi^i)B^i$
$\phi^{i+1} \leftarrow$ **E**$[\![\mathcal{E}^A]\!](\phi^i, \psi^i)|_{\lfloor A_i \rfloor}$
$\psi^{i+1} \leftarrow$ **E**$[\![\mathcal{E}^B]\!](\phi^i, \psi^i)|_{\lfloor B_i \rfloor}$
$i \leftarrow i + 1$
until $A^{i+1} = A^i \wedge B^{i+1} = B^i \wedge \phi^{i+1} = \phi^i \wedge \psi^{i+1} = \psi^i$
end

The occurrences of **E**, **Act**, $\sqcup$ and the function restriction operator can now be unfolded and eliminated, since they are called on known expressions. This process results in the following algorithm. (Note: $\sqcup$ here stands for the lub operator on **Asub**).

begin
$\phi^0, \psi^0, A^0, B^0 \leftarrow \emptyset$

repeat
$\phi^{i+1}(a, \Theta) \leftarrow$ if $(a, \Theta) \in \lfloor A^i \rfloor$ then
$\sqcup\{return\ (\psi^i(b_j, call(h_j, \Theta, a)), h_j, \Theta, a) \mid 1 \leq j \leq n\}$
else $\perp$

$\psi^{i+1}(b, \Theta) \leftarrow$ if $(b, \Theta) \in \lfloor B^i \rfloor$ then
if $b = []$ then Θ else
if $b = [a_1, \ldots, a_i]$ then $\phi^i(a_i, \delta(a_i, \psi^i([a_1, \ldots, a_{i-1}], \Theta)))$
fi
else $\perp$

$A^{i+1} \leftarrow \{(a_0, \Theta_0)\}$

$$
\begin{aligned}
& \cup \\
& \{(a_i, \delta(a_i, \psi^i([a_1, ..., a_{i-1}], \Theta)))|([a_1, ..., a_i], \Theta) \in B^i\} \\
B^{i+1} \leftarrow \quad & \{([a_1, ..., a_{i-1}], \Theta)|([a_1, ..., a_i], \Theta) \in B^i\} \\
& \cup \\
& \left\{(b_j, call(h_j, \Theta, a)) \middle| \begin{matrix}(a, \Theta) \in A^i, \\ 1 \leq j \leq n\end{matrix}\right\}
\end{aligned}
$$

until $A^{i+1} = A^i \wedge B^{i+1} = B^i \wedge \phi^{i+1} = \phi^i \wedge \psi^{i+1} = \psi^i$

It can be seen that given an activation of **A**, say (a, Θ), the call $\mathbf{B}(b_j, call(h_j, \Theta, a))$ is activated for $1 \leq j \leq n$, where $h_1 \leftarrow b_1, ..., h_n \leftarrow b_n$ are the clauses in P. An activation for **B**, say $([a_1, ..., a_i], \Theta)$ gives rise to the activations $\mathbf{A}(a_i, \delta(a_i, \mathbf{B}([a_1, ..., a_{i-1}], \Theta))$ and $\mathbf{B}([a_1, ..., a_{i-1}], \Theta)$.

This algorithm is the naive, inefficient version, and can be converted into a practical algorithm that incrementally extends the activations and function environments on each iteration, as explained in Section 2.1[4].

There are some other improvements that can be made, which depend on the structure of the **A** and **B** functions. For instance, in the computation for B^{i+1}, the expression

$$\{([a_1, ..., a_{i-1}], \Theta)|([a_1, ..., a_i], \Theta) \in B^i\}$$

could be replaced by

$$\{([a_1, ..., a_j], \Theta)|([a_1, ..., a_i], \Theta) \in B^i, 1 \leq j < i\}$$

thereby generating activations for **B** faster.

Due to its length, the resulting algorithm after these improvements is not displayed here. The behaviour of the optimised algorithm can be compared to OLDT resolution, or other methods based on tabulating activations and results. The mfg construction intuitively captures the "top-down" or "query-directed" nature of these methods.

§4 Program Specialisation

Given a definite program P and a goal G, the aim of program specialisation is to derive a more efficient program P' that gives exactly the same answers for G as P does. The question of whether P' should also preserve finite failure of calls to G is a separate question which we do not address here. Thus specialisation includes topics such as partial evaluation as well as other (compiler-based) techniques for optimising a program to compute certain given goals more efficiently. Partial evaluation was introduced into logic programming by Komorowski[9] and is a topic widely researched in a variety of formalisms.[17] The achievement of better techniques for partial evaluation is a major motivation for the present work, since the potential uses of partial evaluation are very wide-ranging.

The computation of an mfg for P and G yields a description of the set of atoms, and their results, activated during computations of $P \cup \{G\}$. The procedures in P can then be specialised to the set of activations. In this section we construct an interpretation of the abstract mfg given in the previous section, suitable for program specialisation. There are many useful variations on the main idea presented below, which is to develop an abstract interpretation based on the unfolding properties of atoms. The idea is to take key ideas from partial evaluation such as unfolding and determinacy, and combine them with ideas derived from global analysis such as approximation of results. The result is a method that is more powerful than partial evaluation in logic programming as defined in Ref. 11). In Section 6 the relationship between this work and partial evaluation is discussed.

4.1 Characteristic Paths

The idea of a *characteristic path* of an atomic goal will now be developed. This is the central notion in the interpretation, which ensures termination of the analysis and gives the connection to partial evaluation methods.

Definition 4.1
Assume that each clause in a program is assigned a unique identifier, say a number. Let G be a goal. We define $tree(G)$ to be the SLD-tree of $P \cup \{G\}$, with the Prolog computation rule, decorated with labels on the edges. That is, each node in $tree(G)$ is labelled by a resolvent; the root is labelled by G. The edge connecting node r to s is labelled by the identifier of a clause used to resolve the leftmost atom at r. We call such trees labelled SLD-trees.

Definition 4.2
A path in a labelled SLD-tree is a sequence of clause identifiers labelling a connected sequence of edges starting at the root. The set of paths in a tree T is called $paths(T)$. Paths are written as lists of clause identifiers, such as $\langle c_1, ..., c_k \rangle$.

Definition 4.3
A path in a labelled SLD-tree is determinate if each node through which it passes (except possibly the last one if it is finite) is connected to exactly one other node. It means that the atom at the left of the resolvent at each node can resolve with only one clause.

In performing program source specialisation we are usually interested in unfolding determinate sections of the computation. The importance of determinacy in partially evaluating Prolog was identified by Komorowski in Ref. 9). Here we identify determinate computation paths rather than individual determinate computation steps. Unfolding determinate paths saves run time resolutions, does not alter the backtracking behaviour of the program and does not increase the size of the program, since no choices are unfolded. The next definition associates with every atomic goal the longest determinate path in its labelled

SLD-tree.

Definition 4.4
Let A be an atomic goal. The maximal determinate path for A, called *maxdet*(A) is the longest determinate path in *paths*(*tree*(A)). If *tree*(A) contains only one branch, which terminates in failure, then *maxdet*(A) is not defined.

Finally, the *characteristic path* of an atom is defined.

Definition 4.5
Let A be an atomic goal. Then *chpath*(A), the characteristic path of A, is defined as:

$$chpath(A) = (maxdet(A), S) \text{ where}$$
$$S = \left\{ c_{l+1} \,\middle|\, \begin{array}{l} maxdet(A) = \langle c_1, \ldots, c_l \rangle, \\ \langle c_1, \ldots, c_{l+1} \rangle \in tree(A) \end{array} \right\}$$

Note that if *maxdet*(A) = *tree*(A) (that is, the computation of A is completely determinate) then $S = \emptyset$. Otherwise S is the set of next choices, which has more than one element since *maxdet*(A) is the maximal determinate path of A. If *maxdet*(A) is not defined (i.e. A fails determinately) then *chpath*(A) is not defined.

Example 4
Let P be the following program and let A = append (Us, Vs, [a, b]).

(1) append (nil, Ys, Ys).
(2) append ([X|Xs], Ys, [X|Zs]):- append (Xs, Ys, Zs).

$$paths(tree(A)) = \{\langle\rangle, \langle 1\rangle, \langle 2\rangle, \langle 2, 1\rangle, \langle 2, 2\rangle, \langle 2, 2, 1\rangle\}$$
$$maxdet(A) = \langle\rangle$$
$$chpath(A) = (\langle\rangle, \{1, 2\})$$

Let A = append ([a, b, c|Us], Vs, Ws). Then

$$paths(tree(A)) = \{\langle\rangle, \langle 2\rangle, \langle 2, 2\rangle, \langle 2, 2, 2\rangle, \langle 2, 2, 2, 1\rangle, \langle 2, 2, 2, 2\rangle, \ldots\}$$
$$maxdet(A) = \langle 2, 2, 2\rangle$$
$$chpath(A) = (\langle 2, 2, 2\rangle, \{1, 2\})$$

We now present an adaptation of the definition of *resultant* from Ref. 11).

Definition 4.6
Let $K = \langle c_1, \ldots, c_n \rangle$ be a finite path in a labelled SLD-tree, where c_i, $1 \leq i \leq n$, are the identifiers of clauses in a program. Let G be a goal whose leftmost atom can be resolved with clause c_1. Suppose $G = G_0$ and $G_0, G_1, \ldots, G_n$ is an SLD derivation in which G_i, $1 \leq i \leq n$, is the result of resolving the leftmost atom in G_{i-1} with clause c_i. (Such a derivation does not exist for every G and

K.) Then the resultant associated with K and G is the formula $G\theta \leftarrow G_n$, where θ is the composition of the substitutions $\theta_1 \ldots \theta_n$ associated with the derivation.

Let $K = c_1, \ldots, c_n$ be a finite path, and suppose G is the most general atomic goal resolving with c_1. Then $G\theta \leftarrow G_n$ (if it exists) is the most general resultant associated with K.

4.2 Substitutions

Let **Sub** be the set of idempotent substitutions. Let θ be a substitution; its component bindings are written $x \mapsto t$. $dom(\theta)$ stands for the set of variables $\{x \mid x \neq x\theta\}$, and if S is a set of variables, $\theta|_S$ stands for the substitution obtained by restricting θ to S. For any θ, $\phi \in$ **Sub**, $\theta \leq \phi$ iff $\exists \rho . \theta = \phi\rho|_{dom(\theta) \cup dom(\phi)}$. $\leq$ is a pre-order relation on **Sub**, and it induces an equivalence relation $\sim$ where $\theta \sim \phi$ iff $\theta \leq \phi$ and $\phi \leq \theta$. Let $[\theta]$ stand for the equivalence class containing θ, and **Sub**$_\sim$ for the set of equivalence classes of **Sub**. The induced order $\leq$ on **Sub**$_\sim$ is a partial order. Assuming **Sub**$_\sim$ is extended with a bottom element $\perp$, **Sub**$_\sim$ forms a complete lattice.

Lemma 4.7
Let $\theta, \phi \in$ **Sub**. Let $\{x_1, \ldots, x_n\} = dom(\theta) \cup dom(\phi)$. Let f be some n-ary function symbol. Then $\theta \leq \phi$ iff $f(x_1, \ldots, x_n)\ \theta$ is an instance of $f(x_1, \ldots, x_n)\phi$.

Proof

$$\begin{aligned} \theta \leq \phi &\equiv \exists \rho .\ \theta = \phi\rho|_{\{x_1, \ldots, x_n\}} \\ &\equiv \exists \rho .\ f(x_1, \ldots, x_n)\theta = f(x_1, \ldots, x_n)\phi\rho \\ &\equiv f(x_1, \ldots, x_n)\theta \text{ is an instance of } f(x_1, \ldots, x_n)\phi \end{aligned}$$ □

A important tool for the interpretation is the *most specific generalisation* (or *msg*) of a set of terms, and the msg of a set of substitutions.

Definition 4.8
Let T be a non-empty set of terms. A generalisation of T is a term s such that for all $t \in T$, t is an instance of s. A most specific generalisation (or *msg*) of T is a generalisation u such that for all other generalisations s of T, u is an instance of s. An msg of a non-empty set of terms always exists and is unique modulo variable renaming. We write $msg(T) = u$ although msg is not a function. Efficient algorithms for computing a most specific generalisation of a set of terms were given in Ref. 14) and 15).

An msg of a set of substitutions is defined using the msg of a set of terms.

Definition 4.9
Let Φ be a non-empty set of substitutions. Assume there is a procedure for computing the msg of a set of terms. We define $msg(\Phi)$ as:

$$msg(\Phi) = \{x_1/t_1, \ldots, x_n/t_n\}$$
where $\{x_1, \ldots, x_n\} = \cup\{dom(\theta) \mid \theta \in \Phi\}$

and $f(t_1, ..., t_n)$ is an msg of $\{f(x_1, ..., x_n)\theta \mid \theta \in \Phi\}$, (for some arbitrary function symbol f).

We have $[\theta] \sqcup [\phi] = [\rho]$ iff $msg\{\theta, \phi\} = \rho$ where $\theta, \phi \in \mathbf{Sub}$. This provides us with a useful way of computing the least upper bound of two elements of $\mathbf{Sub}_\sim$.

Definition 4.10
Let A be an atom and $chpath(A) = (\langle c_1, ..., c_n\rangle, S)$ its characteristic path. Define $chpaths(A) = \{\langle c_1, ..., c_n, s\rangle \mid s \in S, S \neq \emptyset\}$ or $\{\langle c_1, ..., c_n\rangle\}$ if $S = \emptyset$. Let $\{G_1 \leftarrow R_1, ..., G_k \leftarrow R_k\}$ be the set of most general resultants of the paths in $chpaths(A)$. We define $chcall(A)$, the characteristic call of A as:

$$chcall(A) = msg\{G_1, ..., G_k\}.$$

Example 5
Let $A =$ append ([a, b|Us], [c], Ws) with the same program as the previous example. Then

$$chpath(A) = (\langle 2, 2\rangle, \{1, 2\})$$

The normal resultants have heads append ([A, B], Vs, [A, B|Vs]) and append ([A, B, C|Us], Vs, [A, B, C|Ws]).

$$chcall(A) = \text{append ([A, B|Us], Vs, [A, B|Ws])}$$

$$msg\{A, chcall(A)\} = \text{append ([A, B|Us], Vs, Ws)}$$

Lemma 4.11
Let A be an atom. Then $chpath(chcall(A)) = chpath(A)$, and $chpath(msg\{A, chcall(A)\}) = chpath(A)$.

4.3 An Abstract Interpretation for Program Specialisation

We are now in a position to interpret the mfg for Prolog by giving definitions to the undefined symbols in the core semantics of Section 3.1. *call* and *return* are not unusual. They perform essentially the standard operations of unification and composition respectively on elements of $\mathbf{Sub}_\sim$. Care is taken to avoid variable name clashes. The operator δ is the interesting one from the point of view of the abstract interpretation. The idea is to abstract an atom call by computing the characteristic path of the atom, and then finding a more general call that has the same characteristic path.

- **Asub** = $\mathbf{Sub}_\sim$, with partial order $\leq$ as defined above. The least upper bound operator corresponds to computing the equivalence class of the msg of elements of **Sub**, as seen above 4.9.
- The function *call*: $Atom \times \mathbf{Sub}_\sim \times Atom \rightarrow \mathbf{Sub}_\sim$ is defined as follows. Let $\Theta \in \mathbf{Sub}_\sim$ and $h, a \in Atom$. Let P be the program. h is the head of a suitably renamed clause in P, and a is an atom occurring in a clause

body or the initial goal.

$$call\ h\ \Theta\ a = [\phi] \text{ where } \phi = mgu(h, a\theta)|_{vars(h)}$$
$$\text{where } \theta \in \Theta \text{ and } vars(a\theta) \cap vars(P) = \emptyset$$

ϕ is restricted to $vars(h)$ since it is used to compute the meaning for the body of the clause whose head is h.

- The function *return*: $\mathbf{Sub}_{\sim} \times Atom \times \mathbf{Sub}_{\sim} \times Atom \rightarrow \mathbf{Sub}_{\sim}$ is defined as follows: Let $\Theta, \Phi \in \mathbf{Sub}_{\sim}$ and $a \in Atom$. a is an atom occurring in a clause body or the initial goal. h is the head of a suitably renamed clause that unified with a.

$$return\ \Phi\ h\ \Theta\ a = [\theta\rho]$$
$$\text{where } \theta \in \Theta,\ \rho = mgu(a\theta, h\phi)|_{vars(a\theta)}$$
$$\text{where } \phi \in \Phi \text{ and } vars(a\theta) \cap vars(h\phi) = \emptyset$$

ϕ represents the answer computed for the body of the clause with head h. Thus $h\phi$ is the "answer" for the call $a\theta$, and ρ is the answer substitution for $a\theta$.

- The operator δ: $Atom \times \mathbf{Sub}_{\sim} \rightarrow \mathbf{Sub}_{\sim}$ makes use of the characteristic path.

$$\delta(a, \Theta) = [\phi]$$
$$\text{where } \phi = mgu(a, a')|_{vars(a)}$$
$$\text{where } a' = msg(a\theta, d)$$
$$\text{where } \theta \in \Theta,\ d = chcall(a\theta) \text{ and } vars(a\theta) \cap vars(d) = \emptyset$$

4.4 Termination

The lattice $\mathbf{Sub}_{\sim}$ has infinite height; however it has the property that there are no infinite strictly increasing chains (ascending chains) starting from a given (non-$\perp$) element in the lattice; (for a proof of a similar result see Ref. 6)).

The computation of the mfg consists of the generation of a sequence of pairs of activations and partial functions. By the definition of δ each activation of an atom in our interpretation is a substitution that is greater than some substitution determined by a characteristic path. In the partial functions each activation has at most one answer. Therefore if we can ensure that there is an upper bound on the number of characteristic paths encountered in a computation, then the absence of ascending chains in $\mathbf{Sub}_{\sim}$ ensures that there is no infinite ascending sequence of sets of activations, and no infinite ascending sequence of partial functions defined on the activations. Thus the computation of the mfg terminates.

To ensure that there is an upper bound on the number of characteristic paths, we can place an upper bound on the lengths of characteristic paths. That is, we can modify $chpath(A)$ to return (K, S) where K has maximum length d; in this case S is a singleton if the length of $maxdet\ (A)$ is greater than d.

In practice with a wide variety of programs we have found that, though large values for d can be used as a fail-safe, most programs have a naturally finite number of characteristic paths without using an artificial depth bound. This property, together with the abstractions placed on anwsers, ensures termination without the need for an artificial loss of precision.

The characteristic of programs that require a depth bound to ensure termination are that they give rise to indefinitely long computations that neither branch nor terminate.

§5 Constructing a Specialised Program

The mfg construction yields a set of activations of atoms and bodies, with corresponding results. In our domain of interpretation, the activation of an atom A is a pair (A, ϕ), and the result is a substitution θ. We can represent this information as a pair of atoms $(A\phi, A\phi\theta)$. Thus the output of the analysis described above is a finite set of pairs of atoms $(A_j, A_j\theta_j)$ representing the activations A_j and their answer θ_j, plus possibly some activations with no answer; some activations fail or diverge. The mfg also gives the activations and results of bodies, which we can represent in a similar way.

The construction of the specialised program takes place in two stages:

(1) Construct a specialised procedure for each A_j in the set of activations.
(2) Unfold calls to procedures containing only one clause.

5.1 Specialised Clauses

Given an activation A, we first define a renaming.

Definition 5.1
Let A be an atom, and $vars(A) = \{x_1, ..., x_n\}$. Let q be some new predicate symbol. Define $rename(A) = q(x_1, ..., x_n)$.

Renaming ensures independence of the specialised procedures, in case there are two activations, one an instance of the other. Secondly, renaming specialises the unification operations in the computation. A renaming method somewhat similar to that described here was given in Ref. 16), and by others, and a detailed description of this renaming method and the specialisation of unifications is given in Ref. 6).

Let $\{A_1, ..., A_k\}$ be the set of activated atoms in the mfg. Define a set of renamed atoms $\{rename(A_1), ..., rename(A_k)\}$.

Definition 5.2 renaming of a conjunction
Let $B_1, ..., B_m$ be a conjunction. Let $\{A_1, ..., A_q\}$ be the activated atoms returned by the mfg analysis. Define $rename(B_1, ..., B_m)$ as $R_1, ..., R_m$, where

$R_1 = rename(A_i)\rho$, where $\rho = mgu(A_i, B_1)$,
where A_i is the activation in the mfg such that $chpath(A_i) =$

$chpath(B_1)$;

R_j $(j > 1)$ = $rename(A_i)\rho$, where $\rho = mgu(A_i, B_j\sigma)$,
where A_i is the abstract atom such that $chpath(A_i) = chpath(B_j\sigma)$,
where $\sigma = \delta(B_j, \phi)$,
where ϕ is the abstract answer given by the mfg for $B_1, ..., B_{j-1}$.

This procedure simulates the generation of the call $B_j\sigma$ in the abstract analysis, and then finds the recorded activation having the same characteristic path.

Definition 5.3 specialised procedures
Let A be an atom. Let $A_1 \leftarrow B_1, ..., A_k \leftarrow B_k$ be the clauses whose head unifies with A (variables standardised apart from A). The specialised procedure for A is the set of clauses of form:

$rename(A)\theta_i \leftarrow rename(B_i\theta_i)$
where $mgu(A, A_i) = \theta_i$

The specialised program is the set of specialised procedures for the activations.

Having produced the specialised program, calls to procedures with only one clause are unfolded immediately if desired.

5.2 Correctness

Assuming that an abstract interpretation meets general soundness conditions, it returns descriptions of at least all the possible states in which each program part is activated. We refer again to the literature on abstract interpretation.[7,3] In our case, we will be guaranteed that every atom selected in the computation of $P \cup \{G\}$ will be an instance of an atom returned in the mfg generated for $P \cup \{G\}$. Since there is a specialised procedure for each activated atom in the mfg, the specialised procedures will give the same results as the original procedures for at least all of the atoms selected in computation in $P \cup \{G\}$.

We note that the *closedness* condition identified as essential for correctness in Ref. 11) is also a global condition similar to the safety condition of abstract intepretation.

5.3 Further Remarks

The specialisation procedure described here forms a basic method on which many variations are possible.

- We could use different kinds of characteristic path. Instead of the maximum determinate path, any finite set of paths for an atom could be used. In Ref. 6) an abstract interpretation using paths of length 0 is used, with useful results. Recently we have performed experiments using longer characteristic paths which go beyond the first choice point.

- A variation of the mfg construction algorithm is closely related to the partial evaluation procedure in Ref. 2), restricted to definite clauses. That procedure is based on the analysis in Ref.11), which shows that a global condition on partial evaluations is needed to ensure correctness. The closed set of atoms computed in that procedure corresponds to a set of activations computed by our global analysis algorithm. In addition, the mfg procedure computes an abstract result for each atom, which is not explicitly done in the partial evaluation algorithm; this allows specialisations of two kinds that are not possible using partial evaluation:

  (1) Abstract answers can be collected from an infinite number of branches. For example, take the program

  ```
  eqlist([], []).
  eqlist([X|Xs], [X|Ys]):- eqlist(Xs, Ys).
  ```

  The query ← *eqlist*(U, V) gives $U = V$ for all answers. This could only be detected by a global analysis, but not by partial evaluation.

  (2) Infinitely failed trees can be detected, and eliminated if desired. With a proof procedure based on negation by failure such as SLDNF, this would be an unsound transformation, since a diverging computation could be replaced by a finitely failing one. However, such transformations are clearly worth exploring in the context of definite programs, as well as with proof procedures not using negation by failure.

Example 6

Let P be the program:

```
reverse([], Ys-Ys).
reverse([X|Xs], Ys-Zs):- reverse(Xs, Ys-[X|Zs]).
```

Let A = reverse([a, b|Xs], Ys-[]). The set of activations recorded is
{reverse([A, B|Xs], Ys-Zs), reverse([A|Xs], Ys-Zs), reverse(Xs, Ys-Zs)}.

The specialised program, after unfolding determinate clauses, is:

```
reverse1([A, B|Xs], Ys, Zs):- reverse2(Xs, Ys,[B, A|Zs]).
reverse2([], Ys, Ys).
reverse2([X|Xs], Ys, Zs):- reverse2(Xs, Ys,[X|Zs]).
```

Note that this example might cause termination problems for some partial evaluators, because A generates an infinite number of calls not subsumed by any other call. The abstraction induced by the characteristic path prevents the argument Zs *of* reverse(Xs, Ys-Zs) from growing, since it is not used to select any clause head in a characteristic path. Note also that the difference list functor is removed from the procedures by the renaming process. Further examples of this

kind of transformation can be found in Ref. 6). Removing redundant functors reduces the storage requirements of programs.

§6 Discussion

We have previously proposed abstract interpretation as a basis for program specialisation.[5] The main justification was to solve the problem of termination of specialisation procedures, for which some abstraction of the concrete behaviour of the program is necessary. Subsequent experience has shown that decisions about when and how much to abstract are crucial to the performance of the specialised program.

The idea of minimal function graphs for Prolog was first developed by Winsborough.[19,20] The present work makes a contribution to techniques for the efficient computation of mfgs and develops a construction for Prolog which separates clearly the general consideration of the mfg from the semantics of Prolog. It is clear that an mfg for any language (with a continuous evaluation function) could be constructed directly in a similar way. We hope to use this method to construct mfgs for normal programs, and for concurrent logic programming languages.

The implementation suggested is similar to abstract OLDT resolution,[18,5] the procedure described in Ref. 3), and the procedure suggested in Ref. 20). Bruynooghe's procedure[3] constructs a computation tree, which may be pruned during its construction to remove spurious activations that become superseded by larger ones. Such pruning is beyond the scope of framework presented here, but it is conceivable that the information required to perform pruning could be added to our algorithm. This is a subject for future study.

Our approach to source program specialisation can be contrasted with the important method of *partial evaluation* of logic programs. Partial evaluation was introduced into logic programming by Komorowski[9] and is a topic widely researched in a variety of formalisms.[17] The achievement of better techniques for partial evaluation is a major motivation for the present work, since the potential uses of partial evaluation are very wide-ranging. Researchers into partial evaluation have followed in the main an operational approach using techniques based on unfolding the computation tree starting from a given initial goal. This approach is defined precisely in Ref. 11) and an algorithm based on this is given in Ref. 2). Partial evaluation based on unfolding, though intuitively appealing, has been problematic from the start. Apart from the termination problem there is the problem of controlling the size of the specialised program; unfolding too much can result in huge programs, or introduce unwanted backtracking; unfolding too little can miss possibilities for specialisation. Our technique considers partial evaluation in the more general context of program specialisation, and is based on a global analysis of the computation of the program with the given initial goal. Specialisations not possible in partial evaluation alone can be done, and there is the potential of combining more powerful analyses such as mode,

type and variable sharing analysis to get further improvements. Some remarks on the relation to partial evaluation were made in Section 5.3.

Finally, our abstract domain, based on the notion of the characteristic path, offers an abstraction that ensures a natural termination for a large class of programs. Termination for all programs can also be ensured by restricting the lengths of characteristic paths. The topic of termination does require further analysis, but for the time being the approach offered here offers a useful heuristic.

Future research on the method will concern experiments with different kinds of characteristic path and refinements of the implementation of the fixed point finding algorithm.

Acknowledgements

The authors would like to thank Will Winsborough for discussions on minimal function graphs, and the referees for making corrections and suggestions. The second author was supported by the Belgian National Fund for Scientific Research.

References

1) Abramsky, S. and Hankin, C., eds.: *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.
2) Benkerimi, K. and Lloyd, J. W., "A Procedure for Partial Evaluation of Logic Programs," *Proceedings of the North American Conference on Logic Programming*, November 1990, Austin, Texas (S. Debray and M. Hermenegildo, eds.), MIT Press, 1990.
3) Bruynooghe, M., "A Practical Framework for the Abstract Interpretation of Logic Programs," to appear in *Journal of Logic Programming*.
4) Codish, M., Gallagher, J. and Shapiro, E., "Using Safe Approximations of Fixed Points for Analysis of Logic Programs," in *Meta-Programming in Logic Programming* (H. Abramson and M. Rogers, eds.), MIT Press, 1989.
5) Gallagher, J., Codish, M. and Shapiro, E., "Specialisation of Prolog and FCP Programs Using Abstract Interpretation," *New Generation Computing*, *6*, pp. 159-186, 1988.
6) Gallagher, J. and Bruynooghe, M., "Some Low-Level Source Transformations for Logic Programs," *Proceedings of Meta90 Workshop on Meta Programming in Logic*, Leuven, Belgium, April 1990.
7) Jones, N. D. and Søndergaard, H., "A Semantics-Based Framework for the Abstract Interpretation of Prolog," *Interpretation of Declarative Language Abstruct* (S. Abramsky and C. Hankin, eds.), Ellis Horwood, 1987.
8) Jones, N. D. and Mycroft, A., "Dataflow Analysis of Applicative Programs Using Minimal Function Graphs," *Proceedings of Principles of Programming Languages*, 1986.
9) Komorowski, H. J., "Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog," in *9th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, pp. 255-267, 1982.

10) Lloyd, J. W., *Foundations of Logic Programming, 2nd Edition*, Springer-Verlag, 1987.
11) Lloyd, J. W. and Shepherdson, J. C., "Partial Evaluation in Logic Programming," *Technical Report, TR-87-09*, Department of Computer Science, University of Bristol, 1987 (revised 1989), to appear in *Journal of Logic Programming*.
12) Marriott, K. and Søndergaard, H., "Semantics-Based Dataflow Analysis of Logic Programs," in *Information Processing 89* (G. Ritter, ed.), North-Holland, 1989.
13) Mellish, C. S., "Abstract Interpretation of Prolog Programs," *Proc. 3rd ICLP, LNCS 225*, Springer-Verlag, 1986; also Chapter 8 in 1).
14) Plotkin, G., "A Note on Inductive Generalisation," in *Machine Intelligence, Vol. 5* (B. Meltzer and D. Michie, eds.), Edinburgh University Press, 1974.
15) Reynolds, J. C., "Transformational Systems and the Algebraic Structure of Atomic Formulas," in *Machine Intelligence, Vol. 5* (B. Meltzer and D. Michie, eds.), Edinburgh University Press, 1974.
16) Safra, S., " Partial Evaluation of Concurrent Prolog and Its Implications," *Masters thesis, Technical Report, CS86-24*, Dept. of Computer Science, Weizmann Institute, 1986.
17) Sestoft, P., "A Bibliography on Partial Evaluation and Mixed Computation," in *Proceedings of Workshop on Partial Evaluation and Mixed Computation*, Denmark, Oct. 1987.
18) Tamaki, H. and Sato, T., "OLD Resolution with Tabulation," *Proc. 3rd ICLP, LNCS 225*, Springer-Verlag, 1986.
19) Winsborough, W., "A Minimal Function Graph Semantics for Logic Programs," *Technical Report, 711*, Computer Sciences Dept., University of Wisconsin-Madison, Aug. 1987.
20) Winsborough, W., "Path-Dependent Reachability Analysis for Multiple Specialization," *Proceedings of the North American Conference on Logic Programming*, Cleveland, MIT Press, Oct. 1989.