DATA FLOW ANALYSIS OF APPLICATIVE PROGRAMS USING MINIMAL FUNCTION GRAPHS : ABRIDGED VERSION

Neil D. Jones
DIKU
University Park 1
DK-2100 Copenhagen Ø
Denmark

Alan Mycroft
Computer Laboratory
Cambridge University
Cambridge CB2 3QG
England

INTRODUCTION

Data or program flow analysis is concerned with the static analysis of programs, to obtain as much information as possible about their possible run time behavior without actually having to run the programs. Due to the unsolvability of the halting problem (and nearly any other question concerning program behavior), such analyses are necessarily only approximate whenever the analysis algorithm is guaranteed to terminate. Further, exact analysis may be impossible due to the lack of knowledge of input data values, so the analysis can at best yield information about sets of possible computations.

Consequently most research in flow analysis has been concerned with effectively finding "safe" descriptions of program behavior, yielding answers which can always be relied upon but which are sometimes too conservative in relation to the program's actual behavior. Program verification has similar goals, with the difference that flow analysis emphasizes program descriptions obtainable by finitely terminating, deterministic algorithms, while program verification uses deductive methods which may in principle yield more precise results but which are not guaranteed to terminate.

Collecting semantics for flow charts

With the emergence of sophisticated program analysis methods and a wide range of applications, it became clear that more attention had to be paid to the semantic aspects of program analysis. A central paper in this connection was (Cousot, Cousot 77a).

This begins by presenting an operational semantics for a simple flow chart language. It then develops the concept of a *collecting semantics* (there called a "static semantics"). This associates with each program point the set of all memory states which can ever obtain when program control reaches that point, when the program is run on data from a given (perhaps infinite) initial data space. The effect of the program as a whole is then a function mapping the program's initial data into a set of states at each program point. It was shown that a wide variety of flow analyses (but not all) may be realized by finding finitely computable approximations to (*abstractions of*) the collecting semantics.

The collecting semantics is expressed in terms of a lattice of sets of program states (ordered by inclusion), and the various approximations are expressed by simpler lattices, connected to the program state lattice by adjoined *abstraction* and *concretization* functions. Flow analysis may be thought of as simply executing the program, but over a domain of state descriptions in the approximate lattice instead of the precise but infinite collecting semantics lattice. A major contribution of the paper was thus to put on a firm theoretical foundation the well-known idea that program analysis can be done by *abstract interpretation* of the program over nonstandard data domains (see for example (Sintzoff 72)). Termination of the flow analysis procedure may (for example) be achieved by choosing an approximation lattice which has no infinite ascending chains.

This approach is appealing because of its generality; it unifies a large number of special analyses into a common framework. In particular, this makes questions of correctness (*i.e.* safety) much easier to formulate and answer, and sets up a general correctness framework applicable to many different semantic analyses. Another virtue is that it is solidly based in semantics, so that precise execution of the program is included as a special case (albeit an

uncomputable one). This implies that program verification may also be based on the collecting semantics, a theme which is further developed in (Cousot, Cousot 77b) and several subsequent works.

Collecting semantics for applicative programs

It is natural to try to adapt the ideas of (Cousot, Cousot 77a) to the class of applicative programs. How to do this is not entirely clear, though. What is a program point, and how (and what) does one "collect"? The most natural approach is to say that a program function should be approximated by a function on abstract domains. A collecting semantics may be defined by "lifting" the program's functions so they operate on sets of arguments and produce sets of results.

This method was proposed in (Mycroft 81) and developed further in (Mycroft, Nielson 83) and (Nielson 84). However, these works involve considerable mathematical machinery in setting up the general framework, including the use of a new power domain and tensor products in order to form the "collecting semantics" used in the proofs of program correctness. Related work has been reported in (Burn, Hankin, Abramsky 1985) concerning the typed lambda calculus.

These frameworks cannot deal with problems where we are interested in the set of values a function may be called with. We will see more examples later, but an obvious example is optimizing a function definition based on the ways in which it might be called (in essence the above framework assumes that all functions may be called at all arguments).

There is a basic problem with the method above when it comes to practical implementation, coming from the fact that a programmer-defined function is approximated by a function on abstract data values. This works well enough for strictness analysis, since a two-element approximation lattice suffices, and so even higher-order functions may be represented in the form of finite tables. On the other hand, even so simple an analysis as "constant propagation" uses a lattice with height three but infinitely many elements. Consequently functions cannot be represented as tables in a finite computer memory.

One could argue in favor of using intensional, or algorithmic representations of the flow analytic functions, so the outcome of flow analysis is not a table of data descriptions but rather a program for computing them on demand. This idea is perhaps worth working out for practical application, but there appear to be a number of problems to be solved first, for example concerning termination. Related ideas have appeared concerning incremental flow analysis in (Jazayeri 75).

2. PRELIMINARIES

In this section we give a standard semantics for functional programs, and then show how this semantics may be factored into two parts - a core, which is made abstract by leaving several sets, domains and functions undefined, and an *interpretation*, which supplies the missing details. This can be varied to yield a spectrum of more or less abstract models, including as instances the minimal function graph semantics, the standard semantics, and a model for constant propagation.

We assume some background on the reader's part, e.g (Gordon 79) or (Stoy 77). The words "domain" or "cpo" are used to mean a chain-complete partial order, whose least element will be denoted by $\perp$. If A is a partially ordered set (with limits of all non-empty chains - e.g. a set) then $A_\perp$ (sometimes called up(A) or "A lifted") is the cpo obtained by adding a new element $\perp$, with $\perp \sqsubseteq a$ for all $a \in A$ and otherwise $a \sqsubseteq a'$ only when this holds in A. If A is a set with equality, then $A_\perp$ is a flat cpo. As an extension to this common notation, for any symbol \$ we write $A_\$$ to mean $A_\perp$ but with the convention that the 'introduced' bottom symbol is named \$. This will be useful when we later wish to refer to elements of a doubly lifted set.

If not specified otherwise, we use Cartesian products and powers, for example in Φ^n below. Alternately the "smash" product may be specified, in which all tuples containing at least one $\perp$ component are identified. Products and powers are ordered componentwise, as usual.

The i-th element of a tuple is indicated by a subscript or by $\downarrow$ as an infix operator, *e.g.* v_i or $v \downarrow i$.

The function space $X \to Y$ is the cpo of continuous functions from set or cpo X to cpo Y, with order $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all $x \in X$ (if X is a set then $x \sqsubseteq x'$ iff $x = x'$).

A functional language and its semantics

We consider a simple functional language with two syntactic categories, Expression and Program. A program is a system of recursively defined functions. For notational simplicity we name all variables, program functions and base functions by X_i, F_i and A_i, respectively (constants are 0-ary base

functions, or k-ary base functions which ignore their arguments). Abstract syntax is given by:

$$\begin{aligned} \text{U: Program} &::= F_1(X_1,\ldots,X_k) = E_1 \ \& \ \ldots \ \& \\ &\quad F_n(X_1,\ldots,X_k) = E_n \\ \text{E: Expression} &::= X_i \mid E_1 \rightarrow E_2, E_3 \\ &\quad \mid F_i(E_1,\ldots,E_k) \mid A_i(E_1,\ldots,E_k) \end{aligned}$$

To simplify the notation we have assumed all functions take the same number, k, of arguments. In practice and examples we lift this restriction. Moreover, we will assume that there is an underlying *set* S of values, containing (*true*, *false* and (for the flowchart development) *undef*), as well as functions $a_i: S^k \rightarrow S$ which will form a basis for interpreting the symbols A_i. (For the flowchart case we would expect the a_i to be strict with respect to *undef*). This expectation that the base functions are explainable at the set level is common in languages like ours using call-by-value and enables us to avoid the use of power domains in later interpretations in favour of power sets. We do not care further about S and the a_i but want this basis to ensure that three example interpretations we define later are indeed related.

Following is a denotational semantics for the language. It specifies that all function calls are "by value", and uses semantic functions **E** for expressions and **U** for programs, with functionalities

$$\begin{aligned} \mathbf{E}: &\ \text{Expression} \rightarrow \ \Phi^n \rightarrow (S^k \rightarrow S_\perp) \\ \mathbf{U}: &\ \text{Program} \ \rightarrow \ \Phi^n \end{aligned}$$

The following sets and domains will be used (ranged over by the indicated symbols):

v	: S	is the set of *data values*, defined previously
v	: S^k	is the set of *variable environments*
ϕ	: $\Phi = S^k \rightarrow S_\perp$	is the domain of *function denotations*
$\boldsymbol{\phi}$	: Φ^n	is the domain of *function environments*

$$\mathbf{E}[\![X_i]\!]\,\boldsymbol{\phi}\,\mathbf{v} = \ v_i$$

$$\begin{aligned} &\mathbf{E}[\![A_i(E_1,\ldots,E_k)]\!]\,\boldsymbol{\phi}\,\mathbf{v} \ = \\ &\quad \textit{let} \quad \mathbf{v}' = (\mathbf{E}[\![E_1]\!]\,\boldsymbol{\phi}\mathbf{v},\ldots,\mathbf{E}[\![E_k]\!]\,\boldsymbol{\phi}\mathbf{v}) \ \textit{in} \\ &\qquad \perp \quad \textit{if some } v_i = \perp \\ &\qquad a_i(\mathbf{v}') \quad \textit{otherwise} \end{aligned}$$

$$\begin{aligned} &\mathbf{E}[\![F_i(E_1,\ldots,E_k)]\!]\,\boldsymbol{\phi}\,\mathbf{v} \ = \\ &\quad \textit{let} \quad \mathbf{v}' = (\mathbf{E}[\![E_1]\!]\,\boldsymbol{\phi}\mathbf{v},\ldots,\mathbf{E}[\![E_k]\!]\,\boldsymbol{\phi}\mathbf{v}) \\ &\quad \textit{in} \quad \perp \quad \textit{if some } v_i = \perp \\ &\qquad \phi_i(\mathbf{v}') \quad \textit{otherwise} \end{aligned}$$

$$\begin{aligned} &\mathbf{E}[\![E_1 \rightarrow E_2, E_3]\!]\,\boldsymbol{\phi}\mathbf{v} \ = \\ &\quad \textit{case } \mathbf{E}[\![E_1]\!]\boldsymbol{\phi}\mathbf{v} \textit{ of} \\ &\quad \textit{true} : \quad \mathbf{E}[\![E_2]\!]\boldsymbol{\phi}\mathbf{v} \\ &\quad \textit{false} : \quad \mathbf{E}[\![E_3]\!]\boldsymbol{\phi}\mathbf{v} \\ &\quad \textit{otherwise} \quad \perp \end{aligned}$$

$$\begin{aligned} &\mathbf{U}[\![\ F_1(X_1,\ldots,X_k) = E_1 \ \& \ \ldots \ \& F_n(X_1,\ldots,X_k) = E_n \ ]\!] \\ &\quad = \ \textit{fix} \ \lambda\boldsymbol{\phi}.(\mathbf{E}[\![E_1]\!]\boldsymbol{\phi},\ldots,\mathbf{E}[\![E_n]\!]\boldsymbol{\phi}) \end{aligned}$$

As is customary in denotational semantics, each program function F_i is bound to a function $\phi_i : S^k \rightarrow S_\perp$ so the meaning of the entire program is the least fixpoint of

$$\boldsymbol{\phi} = (\mathbf{E}[\![E_1]\!]\boldsymbol{\phi},\ldots,\mathbf{E}[\![E_n]\!]\boldsymbol{\phi})$$

Interpretations

Definition 2.1 An *interpretation*, **I**, consists of the following:

v:	V - a partial order of *data values* , such that $V_\perp$ is a cpo
c:	C - a set of *input data specifications* (*e.g.* $\mathbb{P}(V^k)$ if V is a set)
d:	D - a domain of *expression values*
$\boldsymbol{\phi}$:	Φ - a domain of values for *program functions*

As before we define:

v: V^k	variable environments
$\boldsymbol{\phi}$: Φ^n	function environments

In addition , **I** contains continuous auxiliary functions of the following types:

fetch_i	: $V^k \rightarrow D$	i = 1,...,k
basic_i	: $D^k \rightarrow D$	i = 1,...,k
apply_i	: $\Phi^n \times D^k \rightarrow D$	i = 1,...,n
cond	: $D \times D \times D \rightarrow D$	
init	: $C^n \times \Phi^n \rightarrow \Phi^n$	initial argument information
iterate	: $\Phi^n \times (V^k \rightarrow D)^n \rightarrow \Phi^n$	

The following intuitive explanations are relevant to the minimal function graph interpretation and its various abstractions; they can be simplified considerably for a standard semantics. Function "init" takes as input an initial argument specification $\mathbf{c} \in C^n$ and a function environment $\phi \in \Phi^n$. It yields a function environment ϕ' identical to ϕ, except that in ϕ' it has been "registered" that the value of $F_i(v)$ must be computed for each $v \in c_i$ ($i = 1,...,n$). Examples will be seen in sections 3 and 4.

Function "iterate" abstracts the fixpoint iteration process. It takes as input a current function environment ϕ together with the meaning $e_i \in V^k \to D$ of each of the equation system's right side expressions. It yields as result a new function environment, updated by binding each argument v in the domain of ϕ_i to the value of (or contained in) $e_i(v)$.

Definition 2.2 The semantics *specified by interpretation* **I** consists of two functions

E : Expression $\to \Phi^n \to V^k \to D$ and

U : Program $\to C^n \to \Phi^n$ defined by:

$$E[\![X_i]\!]\,\phi\,v = \text{fetch}_i(v)$$

$$E[\![A_i(E_1,...,E_k)]\!]\,\phi\,v = \text{basic}_i(E[\![E_1]\!]\phi v,..., E[\![E_k]\!]\phi v)$$

$$E[\![F_i(E_1,...,E_k)]\!]\,\phi\,v = \text{apply}_i(\phi, E[\![E_1]\!]\phi v,..., E[\![E_k]\!]\phi v)$$

$$E[\![E_1 \to E_2,E_3]\!]\,\phi\,v = \text{cond}(E[\![E_1]\!]\phi v,\ E[\![E_2]\!]\phi v,\ E[\![E_3]\!]\phi v)$$

$$U[\![F_1(X_1,...,X_k)= E_1\ \&...\&F_n(X_1,...,X_k)=E_n]\!]\,\mathbf{c} = \mathit{fix}\ \lambda\phi.\ \text{iterate}(\text{init}(\mathbf{c},\phi), E[\![E_1]\!]\,\text{init}(\mathbf{c},\phi), ..., E[\![E_n]\!]\,\text{init}(\mathbf{c},\phi)) \quad \square$$

Standard interpretation

The following expresses our original semantics in these terms.

Definition The *standard interpretation* $\mathbf{I_{std}}$ consists of:

V = S (ordered by equality)
C = $\{*\}$ (unused)
D = $S_\perp$
Φ = $S^k \to S_\perp$ (the cpo of continuous partial functions)

with basic functions defined by:

$$\text{fetch}_i(v) = v_i$$

$$\text{basic}_i(\mathbf{d}) = \perp \quad \text{if some } \mathbf{d}_j = \perp$$
$$= a_i(\mathbf{d}) \quad \text{otherwise}$$

$$\text{apply}_i(\phi, \mathbf{d}) = \perp \quad \text{if some } \mathbf{d}_j = \perp$$
$$= \phi_i(\mathbf{d}) \quad \text{otherwise}$$

$$\text{cond}(d_1, d_2, d_3) = \mathit{case}\ d_1\ \mathit{of}\ \perp : \perp;\ \text{true} : d_2;\ \text{false} : d_3;\ \mathit{otherwise} : \perp$$

$$\text{init}(\mathbf{c},\phi) = \phi$$

$$\text{iterate}(\phi, e_1,...,e_n) = (e_1,...,e_n)$$

We will write $\mathbf{E_{std}}$ and $\mathbf{U_{std}}$ when referring to the semantic functions induced by this interpretation. Note that $\mathbf{U_{std}}$ differs from **U** defined previously in that its functionality is Program $\to \{*\}^n \to \Phi^n$ instead of the isomorphic Program $\to \Phi^n$.

3. A SEMANTICS BASED ON MINIMAL FUNCTION GRAPHS

Why minimal function graphs?

The semantics just given is simple and mathematically elegant, but it can be argued that it is excessively general for some applications. For example, in a **module** construction one may wish to assert that only certain functions will be called from the external world, and these only with a certain range of argument values. Similarly when performing program analysis or transformation we may wish to eliminate a function call or simplify an expression, based on knowledge of the variable values on which it may be evaluated. This is exemplified by the "constant propagation" analysis used in most optimizing compilers (section 4

formalises it within our framework), and is seen in many places in the MIX system (Jones, Sestoft, Søndergaard 85).

We thus define a *minimal function graph* semantics, which given specifications C_i of possible input arguments to the various F_i, maps each F_j to the smallest set of (argument,result) pairs sufficient to contain all the uses of F_j encountered during application of any user function to any of its specified input. A bit more precisely, for each j, this is the smallest set S_j which contains $(v, F_j(v))$ for every v in C_j, and which is "closed" in the following sense: if S_i contains $(v, F_i(v))$ then S_j contains $(w, F_j(w))$ for every w such that the value of $F_j(w)$ is needed in order to compute $F_i(v)$. (Note that $F_i(v)$ may equal $\perp$ which we will write as ! in the following). For example, consider

$$F(X) = (X = 1 \rightarrow 1,\ even(X) \rightarrow F(X/2)*2,\ F(3*X+1))$$

where the values of F(0), F(2) and F(3) are desired. The smallest set of (argument,result) pairs (written "argument $\mapsto$ result") which is sufficient to apply the program to {0,2,3} is the following. "!" indicates nontermination on a specified argument (see later).

$$[\,0 \mapsto !,\ 1 \mapsto 1,\ 2 \mapsto 2,\ 3 \mapsto 32,\ 4 \mapsto 4,\ 5 \mapsto 16,\ 8 \mapsto 8,\ 10 \mapsto 3,\ 16 \mapsto 16\,]$$

Our goal will be to set up a mathematical framework which includes both the standard and the minimal function graph semantics (hereafter abbreviated to "mfg semantics") as special cases. Other cases will include a variety of nonstandard interpretations useful in program analysis and optimization.

This framework will require a different domain Φ of function values than $S^k \rightarrow S_\perp$ as used in the standard interpretation. To see this, consider the role that the object $\perp$ plays in a traditional semantics. Suppose one wishes to know the value of an expression in a given program on an argument tuple v. During the computation of the fixpoint describing the entire program's behavior, the value $\perp$ indicates "not yet terminated" - but the possibility remains that after finitely many more computation steps, a defined value will be obtained. If on the other hand, the expression evaluates to $\perp$ in the *final* fixpoint function environment ϕ, then $\perp$ indicates that the expression could not be evaluated on v - or in informal terms, the program "goes into an infinite loop" on argument v.

Further, the traditional semantics in effect applies F to all possible arguments v from S^k, and so may yield $\perp$ on irrelevant arguments, on which the program was never intended to be run. This is not satisfactory in some applications, where it is important to be able to distinguish among: "F is called on v and terminates, yielding v", "F is called on v but does not terminate" and "F is not called on v" .

Our solution will be to use $S_{!\perp}$ (recall this notation from section 2) instead of $S_\perp$ for the domain of function results. For typographic reasons we often write this as $S_{!\perp}$. The new domain $S_{!\perp}$ is "S lifted twice", and (by definition of lifting) has two new elements lying below all elements of S. In order to avoid confusion in names we will denote the "middle bottom" by !. The intended interpretation is that if $F: S^k \rightarrow S_{!\perp}$ then

F(v) = ! means that F(v) has been called but no value has (yet) been obtained, while

$F(v) = \perp$ means that F(v) has not been called.

Note that the ordering $\perp \sqsubset ! \sqsubset v$ is indeed natural from the information theoretic standpoint.

Notation: for $b_i \in B_!$, $[a_1 \mapsto b_1, a_2 \mapsto b_2, \ldots]$ denotes the function $f: A \rightarrow B_{!\perp}$ such that $f(a_1)=b_1$, $f(a_2) = b_2, \ldots$ and $f(a) = \perp$ for all $a \in A \setminus \{a_1, a_2, \ldots\}$

The *restriction operator* # is defined by

$$\#: (A \rightarrow B_\perp) \times \mathbb{P}(A) \rightarrow (A \rightarrow B_{!\perp}):$$

$$f\#c = \lambda x.\ \begin{cases} \perp & \text{if } x \notin c \\ ! & \text{if } x \in c \text{ and } f(x) = \perp \\ f(x) & \text{if } x \in c \text{ and } f(x) \neq \perp \end{cases}$$

Informal description

In the mfg interpretation each (user) function symbol will be mapped to an element of $S^k \rightarrow S_{!\perp}$. The program's input data is given by a subset c of S^k for each F_i, specifying the arguments to which F_i must be applied. An initial data specification is thus an element of $C = \mathbb{P}(S^k)$, the set of all subsets of argument tuples in S^k. In order to

trace the flow of function arguments, semantic function $\mathbf{E}$ will be extended to map an expression E to a pair:

$$\mathbf{E}[\![E]\!]\ \phi\ v\ =\ (\text{value}, (c_1,...,c_n))$$

Here "value" lies in $S_\perp$ as in the standard semantics, and each c_i is the set of argument tuples v such that the value of $F_i(v)$ is needed in order to evaluate E. The idea is that during expression evaluation one accumulates arguments which must be added to the minimal function graph. Function "iterate" ensures that these get added to the current function graphs, if not already there. We now formalize these ideas.

The minimal function graph interpretation

Definition $C = \mathbb{P}(S^k)$ is the power set of argument tuples. A *program initialization* is an element of C^n and variables c, $\mathbf{c}$ will range over C and C^n, respectively. C is ordered by subset inclusion, and C^n is ordered componentwise. The function $only_i : S^k \to C^n$, which yields a singleton argument set in the i-th component, is defined by

$$only_i(v) = (\underset{1}{\{\}},...,\underset{i}{\{v\}},...,\underset{n}{\{\}})$$

Definition The *minimal function graph* interpretation, $\mathbf{I_{mfg}}$, consists of

$$\begin{aligned} v: V &= S \\ c: C &= \mathbb{P}(S^k) \\ d: D &= S_\perp \times C^n \\ \phi: \Phi &= S^k \to S_{!\perp} \end{aligned}$$

with auxiliary functions

$$fetch_i(v) \quad = \quad (v_i, (\{\},...,\{\}))$$

$$\begin{aligned} &basic_i((v_1,c_1),...,(v_k,c_k)) = \\ &\quad let\ \ v = (v_1,...,v_k)\ in \\ &\quad if\ \text{some } v_j = \perp\ then\ (\perp,\ \bigsqcup_1^k c_j) \\ &\qquad\qquad else\ (a_i(v),\ \bigsqcup_1^k c_j) \end{aligned}$$

$$\begin{aligned} &apply_i(\Phi,(v_1,c_1),...,(v_k,c_k)) = \\ &\quad let\ \ v = (v_1,...,v_k)\ in \\ &\quad if\ \ v = \perp\ \ then\ (\perp,\ \bigsqcup_1^k c_j) \\ &\qquad\qquad else\ (\phi_i(v),\ only_i(v) \sqcup \bigsqcup_1^k c_j) \end{aligned}$$

$$\begin{aligned} &cond((v_1,c_1),(v_2,c_2),(v_3,c_3)) = \\ &\quad case\ v_1\ of \qquad true: (v_2,\ c_1 \sqcup c_2) \\ &\qquad\qquad\qquad\quad false: (v_3,\ c_1 \sqcup c_3) \\ &\qquad\quad otherwise \quad (\perp,\ c_1) \end{aligned}$$

$$init(c,\phi) = \quad ((\lambda v.\perp)\ \#\ c_1,...,(\lambda v.\perp)\ \#\ c_n) \sqcup \phi$$

$$\begin{aligned} &iterate(\phi, e_1,...,e_n) = \\ &\quad let\ \ \mathbf{args} = (\{v \mid \phi_1(v) \neq \perp\},...,\{v \mid \phi_n(v) \neq \perp\}) \\ &\quad let\ \ \ \mathbf{c} = \bigsqcup \{ e_i(v)\downarrow 2 \mid 1 \le i \le n, v \in \mathbf{args}_i \} \\ &\quad in\ \ \ ([\lambda v.\, e_1(v)\downarrow 1]\ \#\ c_1\ ,...,\ [\lambda v.e_n(v)\downarrow 1]\ \#\ c_n) \end{aligned}$$

We will write $\mathbf{E_{mfg}}$ and $\mathbf{U_{mfg}}$ when referring to this interpretation. Although we have used some operators not usually seen in denotational semantics (for example $\sqcup$ and (in iterate) comparison with $\perp$ and { x | condition }), it may be verified that the uses above yield continuous functions.

As mentioned above, a component $\mathbf{c}_i$ of $\mathbf{c}$ is a set of arguments v such that the call $F_i(v)$ needs to be evaluated. The function environment ϕ may be thought of as a "global storage" which records for each F_i the set of $(v, F_i(v))$ pairs which have been calculated, and pairs (v, !) for arguments v such that $F_i(v)$ has been demanded but not yet calculated.

First the set of all arguments on which each F_i is to be or has been calculated is obtained; the collection of all these is called **args**. Then the meaning function e_i of expression E_i is applied to every argument in $\mathbf{args}_i$. This yields a collection of pairs (value, $\mathbf{c}'$) where $\mathbf{c}'$ gives any further arguments of $F_1,...,F_n$ which are needed in order to evaluate E_i on v (in the above, $\mathbf{c}' = e_i(v)\downarrow 2$). These are joined together to yield $\mathbf{c}$, the updated set of needed arguments to $F_1,...,F_n$. The new set of

(argument,result) pairs for F_i is $[\ \lambda v.\ e_i(v)\downarrow 1\ ]\ \#\ c_i$, and is got by evaluating e_i (the meaning of E_i) on each $v \in c_i$, with the value ! being used in the case e_i is undefined (i.e. $\perp$) on v.

4. EXAMPLES OF ABSTRACT INTERPRETATION

Constant propagation

This interpretation is often used in compilers, to recognize variables whose values are nonvarying, so that computations involving them may be performed at compile time. The idea is that variables which assume only a single value are modelled precisely, and variables which may be bound to more than one value are modelled by a "don't know" value $\top$. Thus a set of values is mapped into an approximation by the following *abstraction function*. Let X be any set, and let $X_\perp^\top$ be the flat lattice with order $\perp < x < \top$ for all $x \in X$. Then abs has (polymorphic) functionality:

$$abs : \mathbb{P}(X) \to X_\perp^\top \quad \text{and is defined by:}$$

$$abs(\underline{x}) = \begin{cases} \perp, & \text{if } \underline{x} = \text{the empty set} \\ x, & \text{if } \underline{x} = \{x\} \\ \top, & \text{if } \underline{x} \text{ has two or more elements.} \end{cases}$$

Definition The *constant propagation interpretation*, $\mathbf{I_{con}}$, consists of:

$v: V = S_\perp^\top$

$c: C = V^k$ - description of a set of argument tuples (smash product)

$d: D = V \times C^n$ - expression values

$\phi: \Phi = C \Diamond V$ - function values

The use of call by value in our language leads to the use of smash product for C, since a function call is performed only when no argument value is $\perp$. It is thus natural that an uncalled function is modelled by the function value $\perp$, and that called functions are modelled by pairs (c,v) with $c \neq \perp$. The notation $C \Diamond V$ indicates a "semi-smash product ", with elements :

$$C \Diamond V = \{\perp\} \cup \{(c,v) \in C \times V \mid c \neq \perp\}$$

and order $\perp \sqsubseteq (c,v)$, and $(c,v) \sqsubseteq (c',v')$ provided $c \sqsubseteq c'$ and $v \sqsubseteq v'$. Note that $C \Diamond V$ has least upper bounds. The definitions of the auxiliary functions are omitted to save space, but may easily be inferred from the domain definitions and the requirement of safety.

Remarks

We will write E_{con} and U_{con} when referring to this interpretation. Note the close similarity to $\mathbf{I_{mfg}}$ (the semantic relation between these two interpretations will be exhibited in section 5). Differences arise from the possibility of imprecise knowledge of data values, in which case $\top$ is used, and the fact that a function value f is represented by a single pair (v,v), where v represents *all* of ϕ's arguments and v represents all of ϕ's function values.

For the conditional $E_1 \to E_2, E_3$, E_1 is first evaluated. It is handled the same way as in $\mathbf{I_{mfg}}$ if it yields $\perp$, true or false. If E_1 yields $\top$, then we cannot determine whether E_2 or E_3 will be evaluated, so for safety's sake both are evaluated and the least upper bound of the results is used. "Iterate" updates ϕ in a way closely analogous to that of $\mathbf{I_{mfg}}$.

Examples Consider the following program:

$$F(X,Y) = (X = 0 \to Y,\ G(F(X-1,Y))$$
$$\&\quad G(U) = U^2 - U$$

The progressive values of $\phi = (\phi_1, \phi_2)$ given by "iterate" are as follows:

Initial data specification $s = (11, \perp)$. For brevity we write 11 and not (1,1).

ϕ_1	ϕ_2	Remarks
$(11, \perp)$	$\perp$	$\phi_2 = \perp$ since G is not called. F(0,1) *is* called, giving argument description $\top 1$. Evaluating F's right side on $\top 1$ yields new ϕ_1, ϕ_2 :
$(\top 1, 1)$	$\perp$	F($\top$,1) and G(1) are called, yielding:
$(\top 1, 1)$	(1,0)	F's right side now evaluates to $1 \sqcup 0 = \top$

(⊤1, ⊤)	(1,0)	so G(⊤) is called, yielding at last ϕ:
(⊤1, ⊤)	(⊤, ⊤)	

Abstract interpretation in the MIX system

The MIX system is a partial evaluator for programs written in the applicative language used in this article. It accepts as input two items: a program (call it p), and known values (v say) of some (x say), but not necessarily all, of its arguments. It produces as output a so-called *residual program* r which, when run with p's remaining data as input, will yield the same results that p would, when run with all of its data. The existence and computability of r from p and the known data is simply the S-m-n theorem of recursive function theory, but the traditional construction builds a trivial r, essentially of the form:

let x = v **in** p

MIX is a program whose effect is to produce an efficient residual program r, given p and the known data values as input.

Partial evaluation may be employed in compiling. An interpreter is a function of two arguments (the interpreted program and its runtime data). Partially evaluating the interpreter with respect to the program as known data will thus yield a residual program in the language in which the interpreter is written, so the net effect of MIX in this case is to compile *from* the interpreted language *into* the interpreter's own language.

Further, an interpreter may automatically be transformed into a compiler (again from the language it interprets into its own language) by partially evaluating MIX itself with respect to the interpreter as known data. Further yet, a compiler generator may be constructed by a mind-boggling use of MIX to partially evaluate itself, given itself as known data. The paper (Jones, Sestoft, Søndergaard 85) describes the succesful construction of an efficient MIX. In the experiments we have undertaken, the MIX-generated compilers run about 20 times as fast as compilation by use of MIX to partially evaluate an interpreter.

MIX as we have implemented it is heavily dependent on abstract interpretation. There are two main interpretations, which are described in the full version of this paper.

5. SAFENESS OF ABSTRACT INTERPRETATIONS

The general framework

For practical use, it is essential that the results of abstract interpretation reliably describe the actual computations of the program being analyzed; in other words the interpretation must be *safe*. Moreover, it must be possible to compute the program and value descriptions by finitely terminating algorithms. For example, this is possible with the constant propagation interpretation because all the cpo's used there are of finite height, so the fixpoint computation involving "iterate" is guaranteed to terminate (note, however, that V has infinitely many elements).

We formulate safety by defining what it means for one interpretation to be an *abstraction* or "safe approximation" to another, and then will argue that $\mathbf{I_{con}}$ abstracts $\mathbf{I_{mfg}}$.

Definition Let **I, I'** be two interpretations, where **I** has posets and domains V, D, Φ, C and auxiliary functions fetch_i, basic_i ,... and **I'** has posets and domains V', D', Φ', C' and auxiliary functions fetch_i', basic_i' ,... . An *abstraction* h: **I** → **I'** consists of four functions (*abstraction functions*) $h_V: V \to V'$, $h_D: D \to D'$, $h_\Phi: \Phi \to \Phi'$ and $h_C: C \to C'$ which satisfy the conditions 1,2 and 3 below for all $v \in V^k$, $d \in D$, $\mathbf{d} \in D^k$, $\phi \in \Phi^n$ and $\mathbf{c} \in C^n$. The subscripts on the h's will be dropped when no confusion arises.

1. h_Φ is strict and continuous, and h_V, h_D and h_C are monotonic

2. $h(\text{fetch}_i(\mathbf{v})) \sqsubseteq \text{fetch}_i'(h(\mathbf{v}))$
   $h(\text{basic}_i(\mathbf{d})) \sqsubseteq \text{basic}_i'(h(\mathbf{d}))$
   $h(\text{apply}_i(\mathbf{f},\mathbf{d})) \sqsubseteq \text{apply}_i'(h(\mathbf{f}), h(\mathbf{d}))$
   $h(\text{cond}(d_1,d_2,d_3)) \sqsubseteq \text{cond}'(h(d_1),h(d_2),h(d_3))$
   $h(\text{init}(\mathbf{c}, \mathbf{f})) \sqsubseteq \text{init}'(h(\mathbf{c}), h(\mathbf{f}))$

3. Let $e_i : V^k \to D$ and $e_i' : (V')^k \to D'$ (i=1,...,n) satisfy $h_D(e_i(\mathbf{v})) \sqsubseteq e_i'(h_V(\mathbf{v}))$ for every $\mathbf{v} \in V^k$. Then the following must hold for all $\phi \in \Phi^n$:

   $h(\text{iterate}(\phi,e_1,...,e_n)) \sqsubseteq \text{iterate}'(h(\phi), e_1',...,e_n'))$

□

Condition 2 is essentially algebraic; if "$\sqsubseteq$" were replaced by "=", it would specify that h is a homomorphism from the algebra $(\{V,D,\Phi,C\}, \{fetch_i, apply_i,\ldots\})$ to algebra $(\{V',D',\Phi',C'\}, \{fetch_i',\ldots\})$. Condition 3 is the natural extension of the homomorphic condition to our use of higher order functions in "iterate". The following shows that $\mathbf{I}'$ is always safe with respect to $\mathbf{I}$.

Theorem Suppose h: $\mathbf{I} \to \mathbf{I}'$ is an abstraction of $\mathbf{I}$ by $\mathbf{I}'$. Then for any program pgm and $\mathbf{c} \in C^n$

$$h(\mathbf{U}_{\mathbf{I}}[\![pgm]\!]\,\mathbf{c}) \sqsubseteq \mathbf{U}_{\mathbf{I}'}[\![pgm]\!]\,(h(\mathbf{c}))$$

Proof is found in the full version of the paper.

Examples

The standard interpretation may be seen to be an abstraction of the minimal function graph interpretation by defining h: $\mathbf{I_{mfg}} \to \mathbf{I_{std}}$ as follows:

$$h_V(v) = v$$
$$h_C(c) = *$$
$$h_D(v,c) = v$$
$$h_\Phi(\phi) = \lambda v.\ \perp \text{ if } \phi(v) \in \{!, \perp\}$$
$$\phi(v) \text{ otherwise}$$

Note that last line is really a flattening operator $X_{!_\perp} \to X_\perp$ which is a (left) inverse of the restriction (#) operator given earlier.

The constant propagation interpretation also safely approximates the minimal function graph semantics, as may be verifed using the abstraction function h: $\mathbf{I_{mfg}} \to \mathbf{I_{con}}$ defined as follows (abs is as described at the beginning of section 4) :

$$h_V(v) = v$$
$$h_C(c) = abs(c)$$
$$h_D(v, (c_1,\ldots,c_n)) = (v, (abs(c_1), \ldots, abs(c_n)))$$
$$h_\Phi(\phi) = (v, v) \text{ where}$$
$$v = abs\{ v' \in V^k \mid \phi(v') \neq \perp \}$$
$$v = abs\{ \phi(v') \mid v' \in V^k, \phi(v') \notin \{!,\perp\} \}$$

Theorem If $\mathbf{I}$, $\mathbf{I}'$, $\mathbf{I}''$ are interpretations and h: $\mathbf{I} \to \mathbf{I}'$, h': $\mathbf{I}' \to \mathbf{I}''$ are abstractions then Id: $\mathbf{I} \to \mathbf{I}$ and (h' o h): $\mathbf{I} \to \mathbf{I}''$ are abstractions, where Id and (_ o _) are interpreted pointwise on the 4-tuple of abstraction functions.

This says that the structure formed with interpretations as objects and abstraction functions as morphisms forms a category.

6. RELATION TO COLLECTING SEMANTICS FOR FLOW CHARTS

In this section we will develop a "collecting interpretation", and argue that it gives an exact reproduction of Cousot's collecting (or static) semantics. Firstly, however, we recall the definition of a flow chart program and McCarthy's translation into a set of recursive functions.

A flow chart is a directed ordered graph on nodes $N_1,\ldots,N_n$ with *start node* N_1 and *stop node* N_n. All nodes except the stop node are labelled and have either one or two ordered successors, while N_n has label **stop** and no successor. In the following $X_1,\ldots,X_k$ are variables, and Exp is an expression built only from the X_i and base functions A_i. Without loss of generality (by assuming appropriate base functions) we assume that Exp_i is $A_i(X_1,\ldots,X_k)$. The possible forms of N_i $(1 \le i \le n)$ and successors are:

$X_j := Exp_i;\ N_s$
if $Exp_i;\ N_s, N_{s'}$
go; N_s

N_s and $N_{s'}$ are successor nodes, and in the second case N_s is taken if Exp evaluates to "true" and $N_{s'}$ if Exp evaluates to "false".

McCarthy showed that such a flow chart could be translated into a computationally equivalent applicative program in the following way. Letting $F_1,\ldots,F_n$ be function symbols, an applicative program of the following form is obtained:

$$F_1(X_1,\ldots,X_k) = E_1$$
$$\ldots \quad \ldots \quad \ldots$$
$$F_n(X_1,\ldots,X_k) = E_n$$

in which E_i is defined by cases on the form of N_i.

Form of N_i	Form of E_i
$X_j := Exp_i; N_s$	$F_s(X_1,...,X_{j-1},Exp_i,X_{j+1},...,X_k)$
if $Exp_i; N_s, N_{s'}$	$Exp_i \rightarrow F_s(X_1,...,X_k), F_{s'}(X_1,...,X_k)$
go; N_s	$F_s(X_1,...,X_k)$
stop	$F_n(X_1,...,X_k)$

It may seem curious that we model N_n : **stop** by $F_n(X_1,...,X_k) = F_n(X_1,...,X_k)$, but we are only interested in the values that F_n is ever called with and not its result.

In (Cousot and Cousot 77a) some terminology is developed to describe standard and collecting (there called static) semantics for flow chart programs. Unfortunately, their work is based in the rather old-fashioned use of complete lattices for all domains and domain constructors. This leads to some unresolved problems (for example the meaning of 2^{Env}). Moreover . . . In the following we assume these changes have been made and freely use our nomenclature. Their terms and our equivalents follow.

Cousot and Cousot	Our equivalent
Ident	$\{ X_1, ..., X_k \}$
Arcs	$\{ N_1, ..., N_n \}$
Value	V = S : a *set* with true, false, undef
Env = $Ident^0 \rightarrow$ Value	V^k
State = $Arcs^0 \times Env$	$\{1,...,n\} \times V^k$ (pairs (program counter, environment))
Context = 2^{Env}	$C = \mathbb{P}(V^k)$
Context-vector = $Arcs^0 \rightarrow$ Context	C^n

They develop an operational semantics for flow charts. The static or collecting semantics is defined by associating with the program a global description (Context-vector) $\underline{Cv} = (c_1,...,c_n)$, where $c_i \in \mathbb{P}(V^k)$ is the set of environments that may obtain at program node i when the program is run starting at the start node with input (undef,...,undef). They further associate with a flow chart program a function (note that, up to currying, this is the same as their n-context function)

$\underline{F\text{-cont}}$: Context-vector → Context-vector
$\underline{F\text{-cont}}(c)\downarrow i$ = . . . (omitted) . . .

which, when given a current set of environments for each program point, transforms it into the new set of environments that either result from performing one program evaluation step or which contain (undef,...,undef) for the start node N_1. They then argue that $\underline{Cv}$ equals the least fixpoint of $\underline{F\text{-cont}}$.

Recall now that the mfg interpretation has:

$V = S$ $\quad C = \mathbb{P}(S^k)$

$\Phi = S^k \rightarrow S_{!\perp}$ $\quad D = S_\perp \times C$

However, due to the special form of equation systems produced by the translation of flowchart programs (note especially the rule for **stop** and the tail recursiveness of other translated constructs) we can show that only the elements $\{!,\perp\}$ can occur in the image spaces of elements of Φ^n and hence $\mathbf{U_{mfg}}[\![pgm]\!]\,c$. This leads us to define an interpretation $\mathbf{I_{coll}}$ which is identical to $\mathbf{I_{mfg}}$ with the exception that $\Phi = C = \mathbb{P}(S^k)$ which is isomorphic to $S^k \rightarrow \{!,\perp\}$ under the isomorphism $\phi \leftrightarrow C \Leftrightarrow \forall v\,(v \in C \Leftrightarrow \phi(v) = !)$. The auxiliary functions in $\mathbf{I_{coll}}$ are identical to those in $\mathbf{I_{mfg}}$ save for the insertion of this isomorphism in the definitions of $apply_i$, iterate and init. The following lemma is straightforward; its second part follows from the translation of **stop**:

Lemma $\mathbf{I_{coll}}$ is an abstraction of $\mathbf{I_{mfg}}$ under the natural abstraction functions ($h_V = h_C = h_D = Id$, h_Φ the above isomorphism). Moreover $\mathbf{U_{mfg}}$ and $\mathbf{U_{coll}}$ agree on equation systems derived from flowcharts.

Theorem 6.1 Let pgm be the equation system corresponding to a given flow chart. Then letting $\mathbf{c_0}$ = ({(undef,...,undef)}, {}, ..., {}) the following holds:

$$\mathbf{U_{coll}}[\![pgm]\!]\,\mathbf{c_0} = \underline{Cv} = \mathit{fix}\ \underline{F\text{-cont}}$$

Proof is omitted from this version of the paper.

Embedding the Cousot framework in ours

[appears in the full version of the paper]

REFERENCES

(Aho, Ullman 77)
Aho, Alfred V. and Jeffrey D. Ullman, *Principles of Compiler Design*. Reading, MA., Addison-Wesley, 1977

(Barth 78)
Barth, J., "A Practical Interprocedural Data-Flow Analysis Algorithm", *Commun. ACM*, no. 21 (1978), pp. 724-736

(Clarke, Richardson 81)
Clarke, L. A. and D. J. Richardson, "Symbolic Program Evaluation Methods for Program Analysis", *in Program Flow Analysis: Theory and Applications*, Englewood Cliffs, NJ, Prentice-Hall, 1981

(Cousot, Cousot 77a)
Cousot, Patrick and Radhia Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints",*Conf. Rec. of 4th ACM Symp. on Principles of Programming Languages*, Los Angeles, CA (January 1977) pp. 238-252

(Cousot, Cousot 77b)
Cousot, Patrick and Radhia Cousot "Automatic Synthesis of Optimal Invariant Assertions: Mathematical Foundations", *Proc. ACM Symp. on Artificial Intelligence and Programming Languages*, Rochester, NY, SIGPLAN NOTICES, 12, no. 8 (August 1977), pp. 1-12.

(Cousot, Cousot 77c)
Cousot, Patrick and Radhia Cousot, "Static Determination of Dynamic Properties of Recursive Procedures," *IFIP Working Conference on Programming Concepts*, St. Andrews, N.B., Canada (August 1977), ed. E.J. Neuhold. New York: North-Holland, pp. 237-277.

(Donzeau-Gouge 81)
Donzeau-Gouge, Veronique,"Denotational Definitions of Properties of Program Computations", *in Program Flow Analysis: Theory and Applications*, Englewood Cliffs, NJ, Prentice-Hall, 1981

(Fosdick, Osterweil 76)
Fosdick, L.D. and L.J. Osterweil, "Data Flow Analysis in Software Reliability", *Comput. Surv.* 8, no. 3 (September 1976), pp. 305-330

(Gordon 79)
Gordon, M. J. C. *The Denotational Description of Programming Languages*, Springer-Verlag, 1979

(Hecht 77)
Hecht, Matthew S., *Flow Analysis of Computer Programs*. New York: Elsevier North-Holland, 1977.

(Hewitt, Smith)
Hewitt, C. and B. Smith, "Towards a Programming Apprentice", *Proc. of IEEE TransSoftware Eng.*, SE-1, no. 1 (March 1975), 26-45

(Jazayeri 75)
Jazayeri, Mehdi, "Live Variable Analysis, Attribute Grammars, and Program Optimization", tech. report, Univ. N. Carolina, Chapel Hill, NC, 1975

(Jones, Muchnick 81)
Jones, Neil D. and Steven S. Muchnick, "A Flexible Approach to Interprocedural Data Flow Analysis," *Conf. Rec. of 9th ACM Symp. on Principles of Programming Languages*,(1981) pp. 66-74

(Jones, Sestoft, Søndergaard 85)
Jones, Neil D., Peter Sestoft, Harald Søndergaard, "An Experiment in Partial Evaluation: the Generation of a Compiler Generator", *Proc. Conf. on Rewriting Techniques and Applications, Lecture Notes in Computer Science*, Springer-Verlag, 1985

(Kam, Ullman 76)
Kam, J. B. and Jeffrey D. Ullman, "Global Data Flow Analysis and Iterative Algorithms,"
. *ACM.*, 23, no. 1 (January 1976), pp. 158-171.

(Muchnick, Jones 81)
Muchnick, Steven S. and Neil D. Jones (eds.), *Program Flow Ananysis: Theory and Applications*, Englewood Cliffs, New Jersey, Prentice-Hall, 1981

(Mycroft 80)
Mycroft, Alan, "The Theory and Practice of Transforming Call-by-need into Call-by-value", *Proc. 4th Int. Symp. on Programming: Lecture Notes in Computer Science* no. 83, Paris, April, 1980, pp. 269-281

(Mycroft 81)
Mycroft, Alan, *Abstract Interpretation and Optimising Transformations for Applicative Programs*, Ph. D. Thesis, University of Edinburgh, Scotland, 1981

(Mycroft, Nielson 83)
Mycroft, Alan and Nielson, F. "Strong Abstract Interpretation Using Power Domains (Extended Abstract)" *Proc. 10th ICALP : Springer LNCS* 154, Diaz, J. (ed.), Barcelona, Spain, July, 1983, pp. 536-547

(Nielson 82)
Nielson, Flemming, "A Denotational Framework for Flow Analysis", *Acta Informatica, Springer-Verlag*, vol. 18, pp. 265-287, 1982

(Nielson 84)
Nielson, Flemming, *Abstract Interpretation Using Domain Theory*, thesis CST-31-84, University of Edinburgh, Scotland, 1984

(Sharir 81)
Sharir, Micha, "Data Flow Analysis of Applicative Programs", *Proceedings ICALP 1981*, Lecture Notes in Computer Science 115, pp. 98-113, 1981

(Sintzoff 72)
Sintzoff, M. "Calulating Properties of Programs by Valuations on Specific Models", *Proc. Symp. on Proving Assertions about Programs*, New Mexico, ACM, pp. 203-207, 1972.

(Stoy 77)
Stoy, Joseph E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, Cambridge, MA: MIT Press, 1977.