LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

Finite Constants: Characterizations of a New Decidable Set of Constants

by

Bernhard Steffen & Jens Knoop

ECS-LFCS-89-79
(also published as CSR-297-89)

LFCS Report Series

June 1989

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Finite Constants: Characterizations of a New Decidable Set of Constants

Bernhard Steffen * Jens Knoop †

Abstract

Constant propagation – the replacement of program terms which represent a unique value at run time by their values – is a classical program optimization method. In spite of being treated for years, constant propagation still has been in the unsatisfactory phase of heuristics. We enhance the known constant propagation techniques to obtain an algorithm which is *optimal* for programs without loops. Fundamental is the introduction of a new decidable set of constants, the *finite constants*. This set has two different characterizations: a denotational one, which directly specifies our iterative algorithm and an operational one, which delivers the completeness or optimality of this algorithm for programs without loops. The algorithm is implemented in a commercial compiler project [1].

1 Motivation

Constant propagation – the replacement of program terms which represent a unique value at run time by their values – is a classical program optimization method. In spite of being treated for years, constant propagation still has been in the unsatisfactory phase of heuristics. This is mainly because it is in general undecidable whether a term can be replaced by a constant or not [13, 14]. The proof of undecidability constructs for each polynomial a complex term which is constant iff the polynomial has no natural root. Undecidability then follows from the undecidability of the existence of such a root (cf. Hilbert's 10^{th} problem). But there is a huge gap between this theoretical result and algorithmic reality. For example current algorithms fail already for loop-free programs even if the term language contains only + as an arithmetic operator (Presburger Arithmetic) although this situation is perfectly decidable.

*Laboratory for Foundations of Computer Science, University of Edinburgh – The author is supported by the Science and Engineering Research Council grant GC/D69464

†Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität, D-2300 Kiel 1 – The author is supported by the Deutsche Forschungsgemeinschaft grant La 426/9-1

[1] This paper is an extended version of [20].

In this paper we enhance the known constant propagation techniques to obtain an algorithm which is optimal for programs without loops. Fundamental is the introduction of a new decidable set of constants, the *finite constants*. This set has two different characterizations: an operational one, in which it can be regarded as a natural generalization of a set arising from an algorithm of Kam and Ullman [7], and a denotational one, which is suitable for implementation. Indeed, the denotational characterization of the finite constants specifies our iterative algorithm, which is actually implemented in a commercial compiler project, and the operational characterization delivers the completeness or optimality of this algorithm for programs without loops. This can be regarded as a first completeness or optimality result for constant propagation.

2 Background

The essence of data flow analysis is to determine information which is valid for every program state, which might result from a program execution reaching a particular program point.

Completeness therefore is specified by the *meet over all paths* (MOP) strategy in the sense of Kam and Ullman [7], which "intersects" ("meets") all informations which are valid for the execution of some program path from the starting point of a program to a particular program point. This execution directed strategy is closely related to the *operational semantics* of a program. Unfortunately, it is in general not computable because the number of program paths might be infinite.

Efficient algorithms are often specified by the *maximum fixed point* (MFP) strategy in the sense of Kam and Ullman [7]. This strategy iteratively approximates the greatest solution of a set of simultaneous equations. A pure MFP–strategy represents control flow by computing the entry information for every statement as the "intersection" ("meet") of all exit informations of those statements which might dynamically precede, and it represents the semantics by means of a (local) semantic functional which is used to derive the exit information of a statement from its entry information (see Equation System 5.4). Therefore the MFP–strategy has a *denotational* character: it compositionally constructs the global semantics from the semantics of the *primitives* (statements), which is given in a domain theoretic way (see section 5).

Coincidence of the operational and the denotational approach can be proved for a wide range of analysis situations with practical relevance [10, 11, 15, 16, 17]. Here the following theorem is central (cf. [7, 10, 11]):

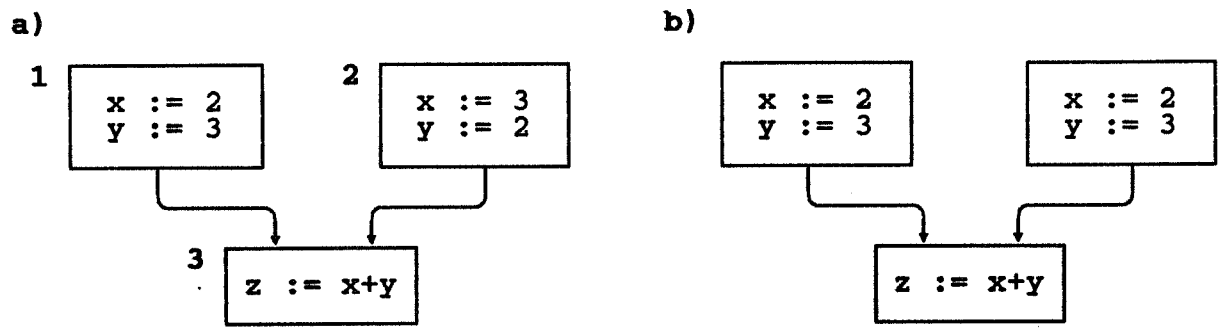
Theorem 2.1 (Coincidence Theorem)

The solutions of the MFP–strategy and the MOP–strategy coincide, if the abstract semantics of all primitives (or statements) is given by a distributive function (i.e.

by a function $f : C \to D$ *with:* $\forall C' \subseteq C.\ f(\sqcap C') = \sqcap \{f(c) \mid c \in C'\}$ *for some complete semi–lattices* C *and* D*).*

This coincidence is the basis for proving *completeness* of data flow analysis algorithms (cf. [15, 16, 17]). However, current constant propagation algorithms (for example [1, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 22, 23, 24]) violate the distributivity condition and fail to detect constancy even in situations in which it is obvious from the operational point of view (i.e. for the MOP–strategy). Kam and Ullman [7] gave a simple example (see Figure 1(a)[2]) for illustration:

Figure 1



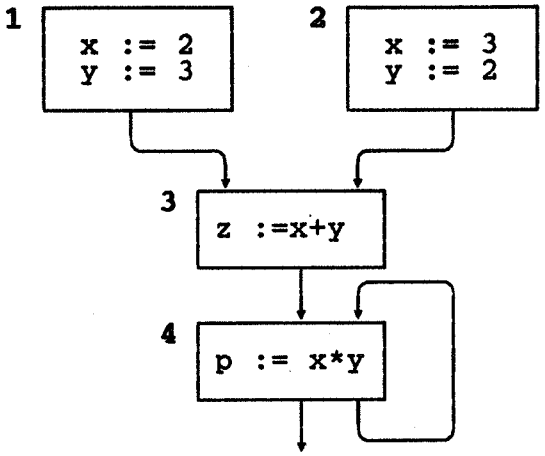
The point of the example in Figure 1(a) is that $x+y$ has a unique value in node **3** even though its operands, x and y, are not constant. This phenomenon cannot be handled by pure MFP–algorithms which inductively determine constant terms on the basis of data flow informations which, as usual, express constancy of variables only. They detect at most *simple constants* [13, 14], i.e. program terms which only possess constant subterms. An example is the term $x + y$ in Figure 1(b).

Kam and Ullman [7] gave a heuristics which extends the pure MFP–approach to cover the situation of Figure 1(a). They determine constancy at a node (here **3**) by separate investigation of the exit informations of all preceding nodes (here **1** and **2**), instead of dealing with the entry information of the considered node. Thus they obtain $x + y = 5$ in node **3** because $x + y = 5$ at the end of both preceding nodes.

Kam and Ullman's heuristics adopts operational ideas for the MFP–approach: instead of applying the semantic functions locally on the entry information of the considered statement, they involve backward paths (of length one) and try to derive constancy on the basis of the entry informations to these paths. Clearly, this operationally based "one-step-look-back" heuristics fails as soon as we enlarge the context, see Figure 2:

[2]Formally, this figure does not display flow graphs in the sense of section 3. The definition there would require to split the upper nodes into two nodes, which we avoided here for clarity.

Figure 2



Here, Kam and Ullman's algorithm detects the constancy of $x + y$, but it fails to detect the constancy of $x * y$. The detection of $x * y = 6$ would require a "look-back" of two nodes. This observation suggests two pure approaches to cover the phenomenon of constant terms with non-constant subterms. First, a purely *operational approach*: it realizes the idea of an arbitrary (finite) "look-back" and characterizes the *(finite) operational constants* (see section 4). Second, a purely *denotational approach*: it realizes the idea of associating values not only with variables, but also with terms, and characterizes the *(finite) denotational constants* (see section 5). Indeed, it will turn out that the *operational approach* and the *denotational approach* deliver the same set of *finite constants*. Moreover, this set is optimal for programs without loops.

3 Preliminaries

We consider terms $t \in \mathbf{T}$ which are inductively built from variables $x \in \mathbf{V}$ and operators $op \in \mathbf{Op}$. The *semantics* of terms of $\mathbf{T}$ is induced by an *interpretation* $I = (\mathbf{D} \cup \{\bot\}, I_0)$, where $\mathbf{D}$ denotes a non empty data domain, $\bot \notin \mathbf{D}$ a new datum, and I_0 a function which associates with each 0–ary operator $c \in \mathbf{Op}$ a datum $I_0(c) \in \mathbf{D}$ and with each n–ary operator $op \in \mathbf{Op}$, $n \geq 1$, a total function $I_0(op) : (\mathbf{D} \cup \{\bot\})^n \to (\mathbf{D} \cup \{\bot\})$, which is assumed to be strict (i.e. $I_0(op)(d_1, .., d_n) = \bot$, whenever there exists a $j \in \{1, .., n\}$ with $d_j = \bot$). $\Sigma = \{\sigma \mid \sigma : \mathbf{V} \to (\mathbf{D} \cup \{\bot\})\}$ denotes the set of all *states* and $\sigma_\bot$ the distinct *start state* which assigns $\bot$ to all variables $x \in \mathbf{V}$ (this choice of $\sigma_\bot$ reflects the fact that we do not assume anything about the context of the program being optimized). The *semantics* of terms $t \in \mathbf{T}$ is given by the evaluation function $\mathbf{Eval} : \mathbf{T} \to (\Sigma \to \mathbf{D} \cup \{\bot\})$, which is inductively defined by: $\forall \sigma \in \Sigma \; \forall t \in \mathbf{T}$.

$$\mathbf{Eval}(t)(\sigma) =_{df} \begin{cases} \sigma(x) & \text{if } t = x \in \mathbf{V} \\ I_0(c) & \text{if } t = c \text{ is a 0-ary operator} \\ I_0(op)(\mathbf{Eval}(t_1)(\sigma), ., \mathbf{Eval}(t_n)(\sigma)) & \text{if } t = op(t_1, ., t_n) \end{cases}$$

In the following we assume $\mathbf{D} \subseteq \mathbf{T}$, i.e. data $d \in \mathbf{D}$ are considered as 0-ary operators that evaluate to d.

As usual, we represent imperative programs as *flow graphs* $G = (N, E, \mathbf{s})$ with node set N, edge set E and a distinguished node $\mathbf{s} \in N$. (Flow graphs are obtainable for example by the algorithm of [2]). Nodes $n \in N$ represent assignments of the form $x := t$ and edges $(n, m) \in E$ represent the nondeterministic branching structure of G. (As usual in data flow analysis we do not consider deterministic branching here to avoid undecidability. However, it is possible to heuristically cover some aspects of deterministic branching (see section 9).) $\mathbf{s}$ denotes the unique start node of N, which is assumed to possess no predecessors. For simplicity we assume that every node $n \in N$ is reachable from $\mathbf{s}$. Additionally, for each node $n = (x := t) \in N$ we define two functions

$$\delta_n : \mathbf{T} \to \mathbf{T} \text{ by } \delta_n(s) =_{df} s[t/x] \text{ for all } s \in \mathbf{T}$$

where $[t/x]$ stands for the simultaneous replacement of all occurrences of x by t, and $\theta_n : \Sigma \to \Sigma$, defined by: $\forall \sigma \in \Sigma \; \forall y \in \mathbf{V}$.

$$\theta_n(\sigma)(y) =_{df} \begin{cases} \mathbf{Eval}(t)(\sigma) & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

δ_n realizes the backward substitution and θ_n the state transformation caused by the assignment of node n. The following relationship between δ_n and θ_n follows immediately by induction on the structure of the term $t \in \mathbf{T}$:

Lemma 3.1 $\forall t \in \mathbf{T} \; \forall \sigma \in \Sigma \; \forall n \in N. \; \mathbf{Eval}(\delta_n(t))(\sigma) = \mathbf{Eval}(t)(\theta_n(\sigma))$

A *finite path* of G is a sequence $(n_1, .., n_q)$ of nodes such that $(n_j, n_{j+1}) \in E$ for $j \in \{1, ., q-1\}$. $\mathbf{P}(n_1, n_q)$ denotes the set of all finite paths from n_1 to n_q. Now the state transformations $\theta_n : \Sigma \to \Sigma$ can be extended to cover finite paths as well. For each path $p = (n_1, ..., n_q) \in \mathbf{P}(n_1, n_q)$ we define $\Theta_p : \Sigma \to \Sigma$ by $\Theta_p =_{df} \theta_{n_1}$ if $q = 1$ and $\Theta_{(n_2, ., n_q)} \circ \theta_{n_1}$ otherwise.

Let now $\Sigma_n =_{df} \{\sigma \mid \exists p = (\mathbf{s}, ., n) \in \mathbf{P}(\mathbf{s}, n) : \Theta_p(\sigma_\perp)\} \subseteq \Sigma$ denote the set of all *possible states* at a node $n \in N$. Then the set of all terms which represent the unique value $d \in \mathbf{D}$ at a node $n \in N$ is given by

$$\mathit{Constants}(n, d) =_{df} \{t \in \mathbf{T} \mid \forall \sigma \in \Sigma_n. \; \mathbf{Eval}(t)(\sigma) = d\}.$$

Indeed, $\mathit{Constants}(n, d)$ specifies the ideal (but in general not decidable) solution of the constant propagation problem.

In the following we generally assume that $G=(N,E,\mathbf{s})$ is an arbitrary but fixed flow graph, m and n, possibly indexed, are nodes of N, t, possibly indexed, is an element of $\mathbf{T}$ and d is an element of $\mathbf{D}$. In particular, all statements are formulated wrt this choice of G. For example, the set of finite constants is meant to be the set of finite constants wrt G (see Definition 6.4).

4 The Operational Approach

For each path $p=(n_1,...,n_q) \in \mathbf{P}(n_1,n_q)$ and each term $t \in \mathbf{T}$ we define the *t-associated* path to p as $p_t=((n_1,t_1),...,(n_q,t_q))$ with $t_q=t$ and $t_j=\delta_{n_j}(t_{j+1})$ for all $1 \leq j < q$. Additionally, let $\mathbf{P}(n_1,n_q,t)=\{p_t \mid p \in \mathbf{P}(n_1,n_q)\}$. Then we obtain as an inductive extension of Lemma 3.1:

Lemma 4.1 (Substitution Lemma)

$\forall t \in \mathbf{T}\ \forall n \in N\ \forall p_t=((\mathbf{s},t_\mathbf{s}),.,(n,t)) \in \mathbf{P}(\mathbf{s},n,t).$

$\mathbf{Eval}(t_\mathbf{s})(\sigma_\perp) = \mathbf{Eval}(t)(\Theta_p(\sigma_\perp))$

Moreover, we call $p_t=((n_1,t_1),.,(n_q,t_q))$ a *relevant path* of length k wrt t for n, $p_t \in \mathbf{R}_k(n,t)$, iff

1. $n_q=n$ and $t_q=t$
2. $q=k$ or ($q<k$ and $n_1=\mathbf{s}$)
3. $\forall i,j \in \{1,...,q\}.\ (n_i,t_i)=(n_j,t_j) \Rightarrow i=j$

Note that $\mathbf{R}_\omega(n,t)$ denotes the set of all relevant paths from $\mathbf{s}$ to n. The central definition of this section, however, is:

Definition 4.2 *For $k \in \omega \cup \{\omega\}$ we define:*

1. *t is a k-*constant *of value d at node n, $t \in C_k(n,d)$, iff*

$$\forall p_t=((n_1,t_1),.,(n_q=n,t_q=t)) \in \mathbf{R}_k(n,t).\ \mathbf{Eval}(t_1)(\sigma_\perp) = d$$

2. *the set of* finite operational constants *of value d at node n, $C_{fop}(n,d)$, by*

$$C_{fop}(n,d) =_{df} \bigcup\{C_k(n,d) \mid k \in \omega\}$$

3. *the set of* operational constants *of value d at node n, $C_{op}(n,d)$, by*

$$C_{op}(n,d) =_{df} \bigcup\{C_k(n,d) \mid k \in \omega \cup \{\omega\}\}$$

Finite operational constants realize the idea of an arbitrary but finite "look-back" in a pure operational approach (see section 2). They generalize the set of constants which is characterized by the algorithm of Kam and Ullman [7]. This is in general not true for subsets $C_k(.,.)$ of $C_{fop}(.,.)$ because Kam and Ullman's algorithm is enhanced by the denotational mechanism it is based on. Indeed, for every $k \in \omega$ there exists a program with a Kam/Ullman–constant which is not a k–constant. On the other hand, there exist already 3–constants which are not Kam/Ullman–constants. An example is $x * y$ in Figure 2. We choose the pure operational (and later denotational) approach for our generalization because mixed approaches are difficult to classify. For example, try to characterize the set of Kam/Ullman–constants in an intuitive way.

Finally let us state the main result of this section:

Theorem 4.3

1. $C_{op}(n,d) = \mathit{Constants}(n,d)$
2. $\exists k \in \omega.\ C_{fop}(n,d) = C_k(n,d)$

Proof: Applying the Substitution Lemma 4.1, we obtain that $\mathit{Constants}(n,d)$ coincides with

$$\{t \in \mathbf{T} \mid \forall p_t = ((\mathbf{s}, t_\mathbf{s}), ., (n,t)) \in \mathbf{P}(\mathbf{s},n,t).\ \mathbf{Eval}(t_\mathbf{s})(\sigma_\perp) = d\}$$

which can easily be proved to coincide with $C_{op}(n,d)$. This proves (1).

To show the nontrivial inclusion "$\subseteq$" of (2), let $\mathit{Var}(G)$ and $\mathit{Var}(t)$ denote the set of all the variables that occur in G and in the term t respectively. Furthermore, let $EQ_\mathbf{V}(\mathbf{T})$ be the equivalence relation on $\mathbf{T}$ defined by:

$$\forall\, t_1, t_2 \in \mathbf{T}.\ \ (t_1,t_2) \in EQ_\mathbf{V}(\mathbf{T}) \iff \mathit{Var}(t_1) = \mathit{Var}(t_2)$$

and $\mathit{Var}(cl) = \bigcup\{\mathit{Var}(t) \mid t \in cl\}$ for $cl \in EQ_\mathbf{V}(\mathbf{T})$. Then the strictness of the evaluation function **Eval** yields:

$$\forall cl \in EQ_\mathbf{V}(\mathbf{T}).\ \ \mathit{Var}(cl) \backslash \mathit{Var}(G) \neq \emptyset \Rightarrow C_{fop}(n,d) \cap cl = \emptyset$$

Hence, $C_{fop}(n,d)$ is contained in the union of the obviously finite set of classes

$$CL =_{df} \{\, cl \in EQ_\mathbf{V}(\mathbf{T}) \mid \mathit{Var}(cl) \backslash \mathit{Var}(G) = \emptyset \ \wedge\ cl \cap C_{fop}(n,d) \neq \emptyset \,\}$$

Now let $cl \in CL$ and $t \in cl$ with $t \in C_{fop}(n,d)$. Then Definition 4.2(2) implies the existence of a $k \in \omega$ with $t \in C_k(n,d)$. Thus we have:

$$\forall p_t = ((n_1,t_1), ..., (n_q = n, t_q = t)) \in \mathbf{R}_k(n,t).\ \ \mathit{Var}(t_1) = \emptyset$$

Furthermore, given an arbitrary $s \in cl$ the equality $\mathit{Var}(s) = \mathit{Var}(t)$ delivers:

$$\forall p_s = ((n_1,t_1), ., (n_q = n, t_q = s)) \in \mathbf{R}_k(n,s).\ \ \mathit{Var}(t_1) = \emptyset$$

Together this yields:

$$\forall s \in cl.\ s \in C_{fop}(n,d) \iff s \in C_k(n,d)$$

Thus there exists for every class $cl \in CL$ a constant $k_{cl} \in \omega$ with

$$\forall t \in cl.\ \ t \in C_{fop}(n,d) \iff t \in C_{k_{cl}}(n,d)$$

This shows that $max\{k_{cl} \mid cl \in CL\}$ (which is finite because of the finiteness of CL) satisfies (2). □

To simplify the proof of the equivalence between our operational and our denotational approach (section 6), we introduce an equivalent characterization of finite operational constants:

Lemma 4.4 $\forall t \in \mathbf{T}.\ t \in C_{fop}(n,d) \iff$
$\exists k \in \omega\ \forall p_t = ((n_1,t_1),..,(n_q = n, t_q = t)) \in \mathbf{R}_k(n,t).\ n_1 = \mathbf{s} \wedge \mathbf{Eval}(t_1)(\sigma_\perp) = d$

While "$\Leftarrow$" is obvious, "$\Rightarrow$" follows from Theorem 4.3(2) and the observation that the new characterization only delays the evaluation of t until each relevant path has reached the start node **s**. This is definitely the case after at most $|N|$ more steps.

5 The Denotational Approach

Let T be an arbitrary subset of the set of all terms **T**. Then $Part(T)$ denotes the set of all partitions of T. Moreover, $\mathbf{Part} =_{df} \bigcup \{Part(T) \mid T \subseteq \mathbf{T}\}$ and $CS(p) =_{df} \{t \mid t$ lies in a class of $p\}$ for $p \in \mathbf{Part}$. We call $CS(p)$ the *carrier set* of p. Partitions can alternatively be viewed as equivalence relations on their carrier sets. This view enables us to define the meet–operation $\sqcap$ on **Part** as the realization of the set theoretical intersection of their corresponding equivalence relations. Note, with this definition $(\mathbf{Part}, \sqcap)$ is a complete semi–lattice (this is essential for the algorithmic characterization of the denotational approach).

The evaluation function $\mathbf{Eval} : \mathbf{T} \to (\Sigma \to \mathbf{D} \cup \{\perp\})$ of section 3 induces a unique partition $Part(\mathbf{Eval})_T$ on every carrier set $T \subseteq \mathbf{T}$:

Definition 5.1

1. $\forall t_1, t_2 \in T \subseteq \mathbf{T}.\ (t_1,t_2) \in Part(\mathbf{Eval})_T \iff \mathbf{Eval}(t_1)(\sigma_\perp) = \mathbf{Eval}(t_2)(\sigma_\perp)$
2. $Part(\mathbf{Eval}) =_{df} \{\ Part(\mathbf{Eval})_T \mid T \subseteq \mathbf{T}\ \}$

The denotational approach uses a semantic functional $[\![\,.\,]\!] : N \to (\mathbf{Part} \to \mathbf{Part})$ which is defined by:

$$\forall n \in N\ \forall p \in \mathbf{Part}.\ \ [\![\, n \,]\!](p) =_{df} \{(r,s) \mid (\delta_n(r), \delta_n(s)) \in p\}$$

Remark 5.2 $[\![\ ]\!]$ is defined for arbitrary partitions. If the carrier set T is closed under all δ_n, $n \in N$, Algorithm A (see below) is complete (i.e. it computes $Constants(n,d)$ for all $n \in N$ and $d \in \mathbf{D}$). Unfortunately, the closure under all the δ_n, $n \in N$, would usually lead to an infinite carrier set. This is responsible for the case where $C_{fden} \subset C_{den}$ (see Definition 5.5).

It is easy to show that $[\![\ ]\!]$ satisfies the condition for the Coincidence Theorem 2.1, i.e. it is easy to prove that:

Lemma 5.3 *The semantic functions* $[\![n]\!]$, $n \in N$, *are distributive.*

Given an arbitrary *start partition* $sp \in Part(\mathbf{Eval})$, the denotational approach is characterized by Equation System 5.4. It labels every node n of G with a pre–partition $\mathbf{pre_{sp}}(n)$ and a post–partition $\mathbf{post_{sp}}(n)$, which are the greatest solutions of this equation system.

Equation System 5.4

$$\begin{aligned} \mathbf{pre}(n) &= \begin{cases} sp & \text{if } n = \mathbf{s} \\ \sqcap \{ \mathbf{post}(m) \mid (m,n) \in E \} & \text{otherwise} \end{cases} \\ \mathbf{post}(n) &= [\![n]\!](\mathbf{pre}(n)) \end{aligned}$$

In analogy to section 4 we invent:

Definition 5.5 *For* $sp \in Part(\mathbf{Eval})$ *we define:*

1. *t is a sp–*constant *of value d at node n,* $t \in C_{sp}(n,d)$, *iff* $(t,d) \in \mathbf{pre_{sp}}(n)$
2. *the set of* finite denotational constants *of value d at node n,* $C_{fden}(n,d)$, *by*

$$C_{fden}(n,d) =_{df} \bigcup \{ C_{sp}(n,d) \mid sp \in Part(\mathbf{Eval}) \wedge |CS(sp)| \in \omega \}$$

3. *the set of* denotational constants *of value d at node n,* $C_{den}(n,d)$, *by*

$$C_{den}(n,d) =_{df} \bigcup \{ C_{sp}(n,d) \mid sp \in Part(\mathbf{Eval}) \}$$

Continuing the analogy with section 4 we have:

Theorem 5.6

1. $C_{den}(n,d) = Constants(n,d)$

2. $(\forall T \subseteq \mathbf{T}.\ |T| \in \omega)\ \exists sp \in Part(\mathbf{Eval}).$

$$|CS(sp)| \in \omega \wedge C_{fden}(n,d) \cap T \subseteq C_{sp}(n,d)$$

The proof of (1) follows by means of the Substitution Lemma 4.1, Lemma 5.3 and the Coincidence Theorem 2.1. For (2) let $S =_{df} C_{fden}(n,d) \cap T$. Then Definition 5.5(2) delivers:

$$\forall s \in S\ \exists\, sp_s \in Part(\mathbf{Eval}).\ |CS(sp_s)| \in \omega \wedge s \in C_{sp_s}(n,d)$$

Thus $\bigsqcup \{sp_s \mid s \in S\}$ (which is finite because of the finiteness of S) satisfies (2).

Remark 5.7 Theorem 5.6(2) is weaker than Theorem 4.3(2). This is because operational constants satisfy the formula

$$(s \in C_k(n,d) \wedge k \in \omega)$$
$$\Rightarrow\ (\forall t \in \mathbf{T}.\ (Var(s) = Var(t) \wedge t \in C_{fop}(n,d)) \Rightarrow t \in C_k(n,d))$$

whereas the corresponding formula for denotational constants

$$(s \in C_{sp}(n,d) \wedge |CS(sp)| \in \omega)$$
$$\Rightarrow (\forall t \in \mathbf{T}.\ (Var(s) = Var(t) \wedge t \in C_{fden}(n,d)) \Rightarrow t \in C_{sp}(n,d))$$

is false (remember, $Var(s)$ and $Var(t)$ denote the sets of all variables contained in s and t respectively).

The practical relevance of the denotational approach comes from its characterization in terms of an iterative algorithm:

Algorithm A:

Input: An arbitrary flow graph $G = (G, E, \mathbf{s})$ and a partition $sp \in Part(\mathbf{Eval})$ with finite carrier set $CS(sp)$.

Output: A designation of G with pre-partitions (stored in *pre*) and post-partitions (stored in *post*) characterizing valid equivalence information.

Remark: $\top$ denotes the "universal" data flow information, which is assumed to "contain" every data flow information. The variable *workset* controls the iterative process.

(Initialization of the designation arrays *pre* and *post* and the variable *workset*)

```
FOR all nodes n∈N DO
  IF n = s THEN (pre[n], post[n]) := (sp, [[n]](sp))
  ELSE (pre[n], post[n]) := (⊤, ⊤) FI
OD;
workset := N;
```

(Iterative fixed point computation)

```
WHILE  workset ≠ ∅  DO
  LET  n ∈ workset
    BEGIN
      workset := workset \ {n};
      IF  pre[n] ⊐ pre[n] ⊓ ⨅ { post[m] | (m,n) ∈ E }
        THEN
          pre[n] := pre[n] ⊓ ⨅ { post[m] | (m,n) ∈ E };
          post[n] := [[n]](pre[n]);
          workset := workset ∪ { m | (n,m) ∈ E }
      FI
    END
OD.
```

This algorithm, which matches the classical pattern of Kildall's data flow analysis algorithm [10, 11], discovers all *sp*-constants:

Theorem 5.8 *Given a finite start partition* $sp \in Part(\mathbf{Eval})$, *Algorithm A computes* $\mathbf{pre}_{sp}(n)$ *and* $\mathbf{post}_{sp}(n)$, *for all nodes* $n \in N$. *In particular, it determines* $C_{sp}(n,d)$, *for all* $n \in N$ *and* $d \in \mathbf{D}$.

6 Equivalence and Optimality

In this section we show that the operational approach and the denotational approach characterize exactly the same set of constants, which we call *finite constants* (see Equivalence Theorem 6.1). Indeed, this set of constants is algorithmically decidable (see Decidability Theorem 7.5) and it is optimal for DAGs, i.e. given an acyclic flow graph the set of finite constants and the set of all terms which represent a unique value at run time, coincide (see Optimality Theorem 6.6).

Theorem 6.1 (Equivalence Theorem)

1. $C_{op}(n,d) = Constants(n,d) = C_{den}(n,d)$
2. $C_{fop}(n,d) = C_{fden}(n,d)$

While (1) is an immediate consequence of Theorem 4.3 (1) and Theorem 5.6 (1), the proof of (2) is based on the notion of finite MOP–constants:

Definition 6.2

1. *Given* $sp \in Part(\mathbf{Eval})$, *we call* t *to be a* sp–MOP–constant *of value* d *at* n, $t \in C_{sp,MOP}(n,d)$, *iff*

$$\forall p_t = ((n_1 = \mathbf{s}, t_1), .., (n_q = n, t_q = t)) \in \mathbf{P}(\mathbf{s}, n, t).\ (t_1, d) \in sp$$

2. *The set of* finite MOP–constants *of value* d *at* n, $C_{fMOP}(n,d)$, *is given by:*

$$C_{fMOP}(n,d) =_{df} \bigcup \{C_{sp,MOP}(n,d) \mid sp \in Part(\mathbf{Eval}) \wedge |\, CS(sp) \,| \in \omega\}$$

Applying Lemma 5.3, the Coincidence Theorem 2.1 yields:

Lemma 6.3 $C_{fden}(n,d) = C_{fMOP}(n,d)$

Thus, in order to complete the proof of the Equivalence Theorem 6.1, it remains to show:

$$C_{fMOP}(n,d) = C_{fop}(n,d)$$

The first inclusion, $C_{fMOP}(n,d) \subseteq C_{fop}(n,d)$, is obvious by contraposition. For the second inclusion, $C_{fMOP}(n,d) \supseteq C_{fop}(n,d)$, let $t \in C_{fop}(n,d)$. Then Lemma 4.4 and the definition of relevant paths deliver the existence of a $k \in \omega$, such that $T_{\mathbf{R},t}$ and $T_{\mathbf{P},t}$ coincide, where

- $T_{\mathbf{R},t} =_{df} \{t_1 \mid p_t = ((n_1 = \mathbf{s}, t_1), ., (n_q = n, t_q = t)) \in \mathbf{R}_k(n,t)\}$ and
- $T_{\mathbf{P},t} =_{df} \{t_1 \mid p_t = ((n_1 = \mathbf{s}, t_1), ., (n_q = n, t_q = t)) \in \mathbf{P}(\mathbf{s}, n, t)\}$

On the other hand, we have $|T_{\mathbf{R},t}| \leq |\mathbf{R}_k(n,t)| < \omega$. Together with Definition 5.1(1) we therefore conclude for $sp = Part(\mathbf{Eval})_{T_{\mathbf{R},t} \cup \{d\}}$:

$$t \in C_{sp,MOP}(n,d) \subseteq C_{fMOP}(n,d)$$

which completes the proof of the Equivalence Theorem 6.1.

The Equivalence Theorem 6.1(2) gives rise to:

Definition 6.4 *The set of* finite constants *of value* d *at node* n, $C_{fin}(n,d)$, *is defined by:*

$$C_{fin}(n,d) \ =_{df} \ C_{fop}(n,d) \ = \ C_{fden}(n,d)$$

Remark 6.5

Due to the Equivalence Theorem 6.1, the operational constants $C_{op}(n,d)$ and the denotational constants $C_{den}(n,d)$ coincide with the set of all constant terms $Constants(n,d)$. The undecidability result of [13, 14] is reflected by the necessity of considering infinite sets of paths in the operational approach and infinite start partitions in the denotational approach.

If G is an acyclic flow graph, the number of nodes $|N|$ is an upper bound for the length of paths in G. This and Theorem 6.1 imply the sequence of equalities

$$C_{fop}(n,d) = C_{op}(n,d) = Constants(n,d) = C_{den}(n,d) = C_{fden}(n,d)$$

which directly proves the optimality of finite constants for DAGs:

Theorem 6.6 (Optimality Theorem)
Let $G=(N,E,\mathbf{s})$ be an acyclic flow graph. Then we have:

$$C_{fop}(n,d) = Constants(n,d) = C_{fden}(n,d)$$

7 Decidability

In this section we develop a uniform algorithm that, given an arbitrary flow graph $G=(N,E,\mathbf{s})$, a node $n \in N$ and an arbitrary term $t \in \mathbf{T}$, decides, whether t is a finite constant of G at the node n or not. Additionally, in the positive case, it determines the corresponding value as well. This decision algorithm, which we will refer to as **Algorithm B**, is based upon the denotational characterization of the finite constants (see section 5) and consists of two parts:

- Computing a subset $\mathcal{S}(G,n,t) \subseteq \mathbf{T}$ such that all finite subsets $T \subseteq \mathbf{T}$ satisfy:

$$\forall\, d \in \mathbf{D}.\ \ t \in C_{Part(\mathbf{Eval})_T}(n,d) \Rightarrow t \in C_{Part(\mathbf{Eval})_{\mathcal{S}(G,n,t)}}(n,d)$$

- Determining $C_{Part(\mathbf{Eval})_{\mathcal{S}(G,n,t)}}(n,d)$ for all values $d \in \mathbf{D}$.

According to Theorem 5.8, it is sufficient to find an algorithm that computes such a subset $\mathcal{S}(G,n,t)$. The following five step procedure meets this requirement:

1. Transform the flow graph G by adding a new node n' to N, such that
   - n' represents the same assignment as n
   - n' has the same set of predecessors as n
   - n' has no successors
2. Construct a regular expression ρ over N that represents $\mathbf{P}(\mathbf{s},n')$. Here, "+" stands for *nondeterministic branching*, " ; " for *sequential composition* and " * " for *indefinite looping*. An algorithm for this construction is given in [21].

3. Replace indefinite looping, $*$, by bounded looping, $^{(k)}$, where k is the number of variables which occur on the left hand side of an assignment of the corresponding subexpression of ρ, to arrive at the $(*-\text{free})$ regular expression ρ_r.

4. Evaluate the functional $\Delta_{\rho_r} : \mathcal{P}(\mathbf{T}) \to \mathcal{P}(\mathbf{T})$, which is inductively defined by:

$$\Delta_\rho(T) =_{df} \begin{cases} \{\delta_\rho(s) \mid s \in T\} & \text{if } \rho \in N \\ \Delta_{\rho_1}(\Delta_{\rho_2}(T)) & \text{if } \rho = \rho_1 \,;\, \rho_2 \\ \Delta_{\rho_1}(T) \cup \Delta_{\rho_2}(T) & \text{if } \rho = \rho_1 + \rho_2 \\ \bigcup\{\Delta^j_{\rho_1}(T) \mid j \in \{0,...,k\}\} & \text{if } \rho = \rho_1^{(k)} \\ \bigcup\{\Delta^j_{\rho_1}(T) \mid j \in \omega\}\} & \text{if } \rho = \rho_1^* \end{cases}$$

(Here, $\mathcal{P}(\mathbf{T})$ denotes the powerset of $\mathbf{T}$ and Δ^j_ρ the j–times repeated application of Δ_ρ. In particular, Δ^0_ρ is the identity on $\mathcal{P}(\mathbf{T})$).

5. Finally set:

$$\mathcal{S}(G,n,t) =_{df} \begin{cases} \Delta_{\rho_r}(\{t\}) \cup \{\, d \,\} & \text{if } \forall\, s \in \Delta_{\rho_r}(\{t\}).\ \mathbf{Eval}(s)(\sigma_\perp) = d \\ \emptyset & \text{otherwise} \end{cases}$$

To establish the main result of this section, the Decidability Theorem 7.5, let us now collect some basic properties of Δ_ρ. First, we obtain by an easy structural induction:

Lemma 7.1 *Given a regular expression ρ, we have:*

$$\forall\, T \subseteq \mathbf{T}.\ \ |T| = \omega \ \Rightarrow\ |\Delta_\rho(T)| = \omega$$

The second property gives us a handle on how to deal with loops. Its rather technical proof can be found in [18].

Lemma 7.2 *Let ρ be a regular expression and k be the number of variables which occur on the left hand side of an assignment of ρ. Then we have:*

$$\forall\, T \subseteq \mathbf{T}.\ \Delta^{k+1}_\rho(T) \not\subseteq \Delta^{(k)}_\rho(T) \ \Rightarrow\ |\,\Delta_{\rho^*}(T)\,| = \omega$$

This and Lemma 7.1 are enough to justify the reduction step (3) of the first part of Algorithm B.

Theorem 7.3 *Given a regular expression ρ and a term $t \in \mathbf{T}$, we have:*

$$|\Delta_\rho(\{t\})| \in \omega \iff \Delta_\rho(\{t\}) = \Delta_{\rho_r}(\{t\})$$

Finally, we have as a consequence of Lemma 6.3:

Lemma 7.4 *Let $G=(N,E,\mathbf{s})$ be a flow graph, $n \in N$, $sp \in Part(\mathbf{Eval})$, $t \in \mathbf{T}$, $d \in \mathbf{D}$ and ρ be a regular expression representing the set of all paths from $\mathbf{s}$ to n* [3]*. Then we have for $t \in C_{den}(n,d)$:*

$$t \in C_{sp}(n,d) \iff \Delta_\rho(\{t\}) \cup \{d\} \subseteq CS(sp)$$

Now, the main result of this section follows easily by means of Theorem 7.3 and Lemma 7.4:

Theorem 7.5 (Decidability Theorem)

Algorithm B decides for every flow graph $G=(N,E,\mathbf{s})$, $n \in N$ and $t \in \mathbf{T}$ whether t is a finite constant at node n, i.e. whether $t \in \bigcup\{C_{fin}(n,d) \mid d \in \mathbf{D}\}$. In the positive case, it additionally determines the value of t at n.

Further details and slightly stronger results can be found in [18].

It is actually possible to decide on the fly whether $t \in C_{fin}(n,d)$. One must only take care during the reduction of the $*$-operator. If the reduction does not cover the full effect of the original regular expression, Theorem 7.3 and Lemma 7.4 justify to stop the algorithm because t can no longer be a finite constant. With this extra check, we succeed in computing a nontrivial set $\mathcal{S}(G,n,t)$ iff $t \in C_{fin}(n,d)$. However, the two step structure of Algorithm B allows us to check for constancy of terms in parallel. For example, a slight modification in the definition of $\mathcal{S}$ leads to an algorithm that covers the set of all program terms in one go (cf. [18]).

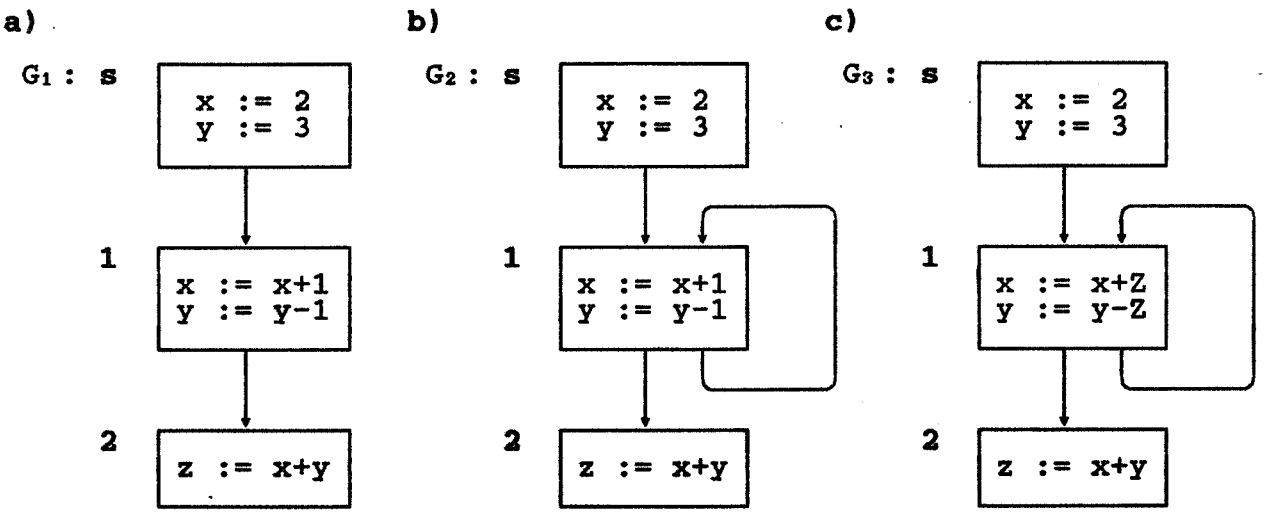
8 Extensions

In the following discussion we assume the strict integer arithmetic, i.e. the value of t is $\mathbf{Eval}(t)(\sigma_\perp)$ whenever t contains no variables, and it is $\perp$ otherwise. In this setting the term $x+y$ is a finite constant at the entrance of node **2** in the flow graph of Figure 3(a), but not in the flow graph of Figure 3(b): for Figure 3(a) it is easy to prove that $(x+y) \in C_k(\mathbf{2},5)$ for all $k \geq 3$ and $(x+y) \in C_{sp}(\mathbf{2},5)$ for every start partition sp with $|CS(sp)| \in \omega$ and $(((2+1)+(3-1)),5) \in sp$. On the other hand, $x+y$ is not a finite constant in the example of Figure 3(b): it is not finite in the operational sense because G_2 contains relevant paths of arbitrary length. This implies that there is a relevant path $p=((n_1=\mathbf{1},t_1),..,(n_q=\mathbf{2},x+y))$ in $\mathbf{R}_k(\mathbf{2},x+y)$ for each $k \in \omega$, where t_1 contains variables. Thus we have $\mathbf{Eval}(t_1)(\sigma_\perp)=\perp$. Also, it is not finite in the denotational sense because the

[3] Note that ρ might contain the $*$-operator.

constancy of $x + y$ would only be detected if the start partition contained all equivalences between 5 and the infinite sequence of terms $((2 + 1) + (3 - 1))$, $((2 + 1) + 1) + ((3 - 1) - 1)), \ldots$.

Figure 3



This observation might suggest that the notion of finite constants is relatively restrictive, an impression caused by the syntactic nature of the backward substitution function δ, which has simplified the development of the theory. However, all the properties shown remain true if we extend δ to $\delta_{\mathbf{Rew}}$, which reduces the backward substituted terms by means of a rewriting function **Rew** which preserves distributivity, i.e. which induces distributive semantic functions $[\![n]\!]$: **Part** $\rightarrow$ **Part**, $n \in N$, by:

$$[\![n]\!](p) = \{(r, s) \mid (\delta_{\mathbf{Rew},n}(r), \delta_{\mathbf{Rew},n}(s)) \in p\}.$$

Concerning the example of Figure 3(b), a generalization by means of a distributivity preserving rewriting function **Rew** with $\mathbf{Rew}((x + 1) + (y - 1)) = x + y$ would classify $x + y$ as a finite constant. However, it could fail in the example of Figure 3(c). A generalization by means of a rewriting function **Rew'** with $\mathbf{Rew}'((x + Z) + (y - Z)) = x + y$, whenever Z denotes a variable or a 0-ary operator, would even succeed in this case. Thus the generalization of the pure backward substitution function δ to functions $\delta_{\mathbf{Rew}}$ leads to a hierarchy of sets of finite constants, each one wrt the concrete choice of **Rew**. Indeed, the size of a set of finite constants strongly depends on this choice, but the principle of finiteness always remains the same.

9 Conclusions

We have presented an enhanced algorithm for constant propagation, which is optimal for acyclic flow graphs. Fundamental for achieving this optimal algorithm was the introduction of a new decidable set of constants, the finite constants. We have proved that this set has two characterizations: a denotational one, which specifies our algorithm and an operational one, which delivers the optimality of this algorithm. Our algorithm has been implemented in a commercial compiler project [19]. Currently, the implementation determines an approximation of the set of terms which is necessary to detect all finite constants. This approximation is efficiently computable and leads to results which exceed all optimizations of [7, 10, 11, 13, 14]. Additionally, our algorithm is a sound basis for further extensions, for example for taking advantage of path informations in the sense of [3, 23]. This can be done by means of the collecting semantics of [12]. We have exhibited a way for generalizing the notion of finite constants by the transition from the pure backward substitution function δ to functions $\delta_{\mathbf{Rew}}$, which reduce the backward substituted terms by means of distributivity preserving rewriting functions **Rew**. This generalization leads to a hierarchy of sets of finite constants, each one wrt the concrete choice of **Rew**. While the size of a set of finite constants strongly depends on this choice the principle of finiteness always remains the same. Therefore we believe that the notion of finite constants gives new insight in the nature of the constant propagation problem.

10 Acknowledgements

We would like to thank Michael Mendler, Oliver Rüthing and Martina Will for their careful proof reading and their valuable comments.

References

[1] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

[2] F. E. Allen. Control flow analysis. *ACM Sigplan Notices*, July 1970.

[3] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *15th POPL*, pages 1–11, San Diego, 1988.

[4] J. Cocke and J. T. Schwartz. Programming languages and their compilers. Preliminary notes, Courant Institute of Mathematical Sciences, New York University, April 1970.

[5] V. Donzeau-Gouge. Denotational definition of properties of program computations. In St. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice Hall, New Jersey, 1981.

[6] J. Ferrante and K. J. Ottenstein. A program form based on data dependency in predicate regions. In *10th POPL*, pages 217–236, Austin, Texas, 1983.

[7] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309–317, 1975.

[8] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[9] K. W. Kennedy. Variable subsumption with constant folding. SETL Newsletter 112, Courant Institute of Mathematical Sciences, New York University, August 1973.

[10] G. A. Kildall. Global expression optimization during compilation. Technical Report 72-06-02, University of Washington, Computer Science Group, Seattle, Washington, 1972.

[11] G. A. Kildall. A unified approach to global program optimization. In *1st POPL*, pages 194–206, Boston, Massachusetts, 1973.

[12] F. Nielson. A bibliography on abstract interpretations. *ACM Sigplan Notices*, 21:31–38, 1986.

[13] J. H. Reif and R. Lewis. Symbolic evaluation and the global value graph. In *4th POPL*, pages 104–118, Los Angeles, California, 1977.

[14] J. H. Reif and R. Lewis. Efficient symbolic analysis of programs. Technical Report 37-82, Harvard University, Aiken Computation Laboratory, 1982.

[15] B. Steffen. *Abstrakte Interpretationen beim Optimieren von Programmlaufzeiten. Ein Optimalitätskonzept und seine Anwendung.* PhD thesis, Christian-Albrechts-Universität Kiel, 1987.

[16] B. Steffen. Optimal run time optimization - proved by a new look at abstract interpretations. In *TAPSOFT '87*, pages 52–68. LNCS 249, 1987.

[17] B. Steffen. Optimal data flow analysis via observational equivalence. In *MFCS '89*, Lecture Notes in Computer Science. Springer Verlag, 1989.

[18] B. Steffen and J. Knoop. Term closures for optimal program optimization. To appear.

[19] B. Steffen and J. Knoop. Ein optimales interpretationsunabhängiges Datenflußanalyseverfahren. Technical Report 8711: Programmspezifikation – Midlum/Föhr, 1987, Christian-Albrechts-Universität Kiel, Institut für Informatik, West-Germany, 1987.

[20] B. Steffen and J. Knoop. Finite constants: Characterizations of a new decidable set of constants. In *MFCS '89*, Lecture Notes in Computer Science. Springer Verlag, 1989.

[21] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.

[22] B. Wegbreit. Property extraction in well–founded property sets. *IEEE Transactions on Software Engineering*, 1(3):270–285, September 1975.

[23] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. In *12th POPL*, pages 291–299, New Orleans, Lousiana, 1985.

[24] R. Wilhelm. Global flow analysis and optimization in the MUG2 compiler generating system. In St. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications.* Prentice Hall, New Jersey, 1981.