Interprocedural Data Flow Analysis Based on Temporal Specifications*

Arnd Poetzsch-Heffter*

TR 93-1397
October 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* Supported by DFG grant Po 432/2-1. Author moving to Munich.

Interprocedural Data Flow Analysis Based on Temporal Specifications

Arnd Poetzsch–Heffter [1] [2]

Cornell University
4112 Upson Hall
Ithaca, NY 14853
poetzsch@cs.cornell.edu

Institut für Informatik
Technische Universität
D–80290 München
poetzsch@informatik.tu-muenchen.de

Abstract

This paper investigates the specification of data flow problems by temporal logic formulas and proves fixpoint analyses correct. Temporal formulas are interpreted w.r.t. programming language semantics given in the framework of evolving algbras. This enables very high–level specifications, in particular for history sensitive problems. E.g. the classical bit vector analyses can be refined by using information about conditions in branches without having to change their specifications. The general semantics framework makes the approach directly applicable to realistic programming languages.

We use the specifications to prove fixpoint implementations of data flow analyses correct. As an example, we develop a powerful interprocedural deadness analysis that uses constant information depending on the context where the active procedure was called. By proving such a combination of backward and forward analyses correct, we illustrate the use of specifications in correctness proofs.

1 Introduction

Motivation Data flow analysis (DFA) is a well–established method to provide static infomation about programs. Such information is typically used in optimizing compilers ([AU77]). The quality of DFAs usually has a major impact on the code quality. Therefore, a systematic study of sophisticated analyses is a key issue in language implementation technology.

Many papers investigating data flow analysis (in particular of imperative programming languages) use a fixed, often rather simple flow graph model as basis for the program semantics. Data flow analyses are specified by *local relations* between the information that corresponds to adjacent nodes in the graph. This approach was very successful to set up the fundamental theory and to develop basic algorithms. However, this approach has the following drawbacks when it comes to the development of powerful data flow analyses for realistic programming languages:

1. Realistic programming languages cannot be modeled with simple flow graphs.
2. Specifications of data flow analyses using local relations between graph nodes often do not capture the entire original problem; i.e. they are too restrictive.

Examples to illustrate the first drawback are e.g. recursive procedures, procedure parameters or pointers, method selection techniques in object oriented languages, parallelism. Whereas there is already some work done to find more expressive settings, the second drawback is almost not touched (cf. related work). To illustrate the second drawback, let us consider deadness analysis of variables: A variable V is called dead at a program point P, if it is not used until it is defined in computations following P.

```
     proc P(): if y=0 then z:=x else x:=y-1 fi
(1)     ... y:=3 ; P() ;
```

[1] Supported by DFG grant Po 432/2-1.
[2] currently moving to Munich

According to this informal definition, the variable `x` is dead before the assignment to `y` in line (1), because whenever `P` is called in line (1) the value of `y` is 3 so that the else–branch in `P` is executed where `x` is defined. But to give a formal account of this definition using local relations between graph nodes leads to very complex specifications. The main reasons for that are: Parts of the language semantics has to be encoded in the analysis; and the analysis has to use information from backward and forward analyses.

Approach We show how to specify data flow problems with temporal logic formulas. The interpretation of such formulas is defined in terms of the programming language semantics, which is given by evolving algebras, a very expressive semantics framework (cf. section 2). The high–level temporal specifications directly reflect the informal definitions of data flow problems. We use the specifications to prove fixpoint implementations of DFAs correct. As the specifications are based on the set of computations of a program, the fixpoint implementations are usually suboptimal (in general, semantic–based specifications are undecidable; see [RL77] for an example). And this is an advantage of our method, because it enables refinements and allows to stepwise approximate the specification by more and more sophisticated analyses. As an example, we develop a powerful interprocedural deadness analysis that uses constant information depending on the context where the active procedure was called. We show how to prove such a combination of backward and forward analyses correct, thereby illustrating the use of specifications in correctness proofs.

Related Work The correctness of fixpoint analyses was classically proved correct w.r.t. corresponding merge over all path analyses (MOP). In the abstract interpretation framework, the corresponding correctness and optimality results are expressed by the coincidence theorems[3] (see [KU77] and [KS92] for an interprocedural setting). In these works, the correctness of the MOP analyses is simply taken for granted. In [CC79], the correctness of abstract interpretations w.r.t. a strongest post–/weakest precondition semantics over simple flow graphs is investigated. F. Nielson developed a denotational framework for data flow analysis (see e.g. [Nie87]). Both frameworks directly relate the analyses to the language semantics like we do here. But in contrast to our work, history sensitive analyses (like deadness or available expression) are difficult to specify in those frameworks, in particular when it comes to realistic programming languages.

In [Ste91], B. Steffen used modal logic to specify data flow analysis and to use model checkers to perform the analysis. His work is similar to ours in that he looks for a higher specification level. But in his framework, the formulas are interpreted w.r.t. programs and not w.r.t. runs of programs as we do; i.e. the models do not capture the language semantics sufficiently detailled to express history sensitive analyses in full generality.

The interprocedural analysis developed in this paper profited from [KS92] and from the idea of semantical refinement as it is e.g. used in [WZ85]. Many aspects of our practical goals are similar to [Ven89] (cf. conclusions).

Overview Section 2 gives a short introduction into evolving algebras, our semantical framework. It defines temporal logic formulas as specification language and introduces the basic proof rule for computational induction. Section 3 explains the specification of different kinds of data flow analyses and relates them to the specification of an example language with recursive procedures. In section 4, we describe a fixpoint implementation of a deadness analysis and prove it correct w.r.t. its specification. Section 5 refines that analysis. This refinement provides a very powerful interprocedural deadness analysis. In addition, it illustrates that sophisticated analyses may have very complex fixpoint definitions whereas their specifications can be very

[3] Notice that the induction proofs used there only apply to backward analysis of terminating computations, whereas our approach applies as well to non–terminating computations.

simple. Section 6 contains the conclusions and some remarks on implementation issues further motivating the use of evolving algebras.

2 Specification Framework

This section describes the specification framework for programming language semantics and data flow problems. After a short introduction to evolving algebras, we define temporal formulas and their validity w.r.t. computations of an evolving algebra. Then, we define a weakest precondition transformer on state formulas that relates the step semantics given by the evolving algebra to the temporal nexttime operator.

2.1 Evolving Algebras as Semantics Framework

Evolving algebras are a powerful framework for specifying and reasoning about programming languages (for an introduction see [Gur91]). Evolving algebras have been used to specify the semantics of many different programming languages including C ([GH92]), PROLOG ([BR92]), and Occam ([GM89]). In contrast to e.g. denotational, action, or structural operational semantics, they directly provide a trace semantics which can be used as the semantics for the linear time temporal logic. Beside this, they force the language specifier to make explicite the data structures controling execution, whereas in other frameworks these data structures are only implicite in the richer formalism (e.g. in evolving algebras expression evaluation is done by a store semantics (see [MS76]), the control stack for recursive procedures is visible). This is an advantage for our purposes, because it allows to refer to these data structures from the logic. Last not least, they provide a flexible basis for implementations (cf. section 5). On the other hand, other semantics could be used as well, if they have a trace semantics (like most transition systems) or are refined to a trace semantics (which can at least be done for operational semantics). In the following, we give the basic definitions of evolving algebras; an example is presented in section 3.2. Informally, evolving algebras can be considered as a set of guarded multiple assignments that allow pointwise updates of functions.

A *signature* is a family of finite sets $(FUNC_s)_{s \in \mathbb{N}}$ of function symbols. A *first-order structure* S with signature Σ is given by a set U called the universe of S and an interpretation φ of function symbols, $\varphi = (\varphi^s)_{s \in \mathbb{N}}$, $\varphi^s : FUNC_s \rightarrow \mathcal{F}(U^s, U)$, where $\mathcal{F}(U^s, U)$ denotes the functions from U^s to U. A *computation* with signature Σ is given by a universe and a sequence $(\varphi_i)_{i \in \mathbb{N}}$ of interpretations; φ_i is called the i-th *state* of the computation.

Let VAR be a set of variables. Σ*–terms*, Σ*–ground–terms*, and *quantifier–free* Σ*–formulas* over $=$, $\vee$, $\wedge$, $\neg$ are defined as usual. We will use an interpretation φ as well to denote its canonical extension to ground terms. Let ν be a valuation, i.e. $\nu : VAR \rightarrow U$. By $\varphi[\nu]$ we denote the interpretation of terms under valuation ν. Furthermore, we assume the classical notions of validity in first–order logic with equality.

Definition 2.1 Let Σ be a signature.

- An *update* (over Σ) is an expression of the form $f(t_1, \ldots, t_n) := t$ where $f(t_1, \ldots, t_n)$ and t are Σ–ground–terms.

- A *transition rule* is an expression R of the form **if** *guard(R)* **then** *updates(R)* where *guard(R)* is a quantifier–free closed Σ–formula and *updates(R)* is a finite set of updates. A transition rule R is called *consistent*, if its guard implies that for each pair of updates $\langle f(s_1, \ldots, s_n) := s\, ,\, f(t_1, \ldots, t_n) := t \rangle$ the disjunction $s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n$ holds.

- An *evolving algebra* (over Σ) is given by a finite set of transition rules.

- An evolving algebra is called *consistent* if all its rules are consistent; it is called *complete*, if the disjunction of the guards is a tautology.

- A *computation* of a consistent and complete evolving algebra EA is a computation such that for each $i \in \mathbb{N}$ there is a rule R in EA transforming φ_i into φ_{i+1}, i.e.:

  - *guard(R)* is valid in the structure (U, φ_i).
  - Denoting the updates of R by $f_1(\vec{s}_1) := t_1, \ldots, f_r(\vec{s}_r) := t_r$, we have

  $$\varphi_{i+1}(f)(\vec{u}) =_{def} \begin{cases} \varphi_i(t_j) & \text{if } f \equiv f_j \text{ and } \vec{u} = \varphi_i(\vec{s}_j) \\ \varphi_i(f)(\vec{u}) & \text{otherwise} \end{cases}$$

  where j ranges over $1, \ldots, r$ and $\vec{u} \in U^s$.

□

Definition 2.1 follows the introduction of evolving algebras in [GR92]. This setting is a bit different from the one given in [Gur91] where all rules with true guards are executed simultaneously. Whereas the latter setting provides more flexibility in writing down specifications, the focus here is to keep formal definitions simple. Each evolving algebra EA can be made consistent by adding the corresponding restrictions to the guards and complete by adding a rule that has the conjunction of the negated guards of EA as guard and an empty set of updates.

A *programming language* (Π, EA) is given by a set Π of Σ–structures and by a consistent and complete evolving algebra EA over Σ. Each element of Π models one *program* together with its initial state and the basic data types. The *computations of a program* $\text{P} \in \Pi$ are those computations of EA that start with P. The function symbols occurring in outermost position on the left hand side of updates in an EA are called *dynamic*. Zero–ary dynamic function symbols are often called as well *dynamic* or *flexible*[4] *variables.*

2.2 The Specification Language: Temporal Logic Formulas

This section introduces temporal formulas and their semantics. We use a subset of the operators defined in [MP92]. Our formulas (and semantics) are slightly simpler in that we do not allow quantification over dynamic variables. On the other hand, the semantics here is a bit more general[5] in that computations can change the interpretation of n-ary function symbols for $n > 0$, and not only that of dynamic variables.

Definition 2.2 Let Σ be a signature and *VAR* be a set of variables. The set of *(temporal) formulas* over Σ is inductively defined as follows:

- if t_1, t_2 are Σ terms, then $t_1 = t_2$ is a formula;

- if F, G are formulas and X is a variable, then $\neg F$, $F \rightarrow G$, $\circ F$, $F\,\mathbf{until}\,G$, $F\,\mathbf{since}\,G$, $\forall X.F$ are formulas.

□

The operators $\circ$, **until**, **since** express temporal properties: $\circ F$ is valid in the current state if F is valid in the next state; $F\,\mathbf{until}\,G$ is valid in the current state if F is valid until G becomes valid; $F\,\mathbf{since}\,G$ is valid in the current state if F is valid since G was valid the last time (the formal definitions are given below). A formula not containing a temporal operators is called a *state formula.* In the following, we assume that the signatures contain the 0-ary function symbol *true* (to denote the boolean value "true") and that the universes contain two distinct

[4]The term "flexible variable" is often used in the temporal logic community.

[5]If necessary, e.g. to use existing temporal calculi, this can be eliminated.

elements to represent the boolean values. To improve readability, we denote $true = true$ by TT, write $f[\ldots]$ for f(...)=true, use $\leftrightarrow$, $\vee$, $\wedge$, $\exists$ as abbreviations with their usual meaning, and call boolean valued functions *predicates*; furthermore, we use the following temporal abbreviations:

- $\Diamond F \;\equiv\; \mathrm{TT}\,\mathbf{until}\,F$, called "eventually",
- $\Box F \;\equiv\; \neg\Diamond\neg F$, called "always",
- $F\,\mathbf{unless}\,G \;\equiv\; \Box F \vee (F\,\mathbf{until}\,G)$.

Definition 2.3 Let $\mathcal{C} = (\,U, (\varphi_i)_{i\in\mathrm{N}}\,)$ be a computation, ν be a valuation, and F and G be temporal formulas.

- The validity of formulas F w.r.t. ν in state φ_i (denoted by $(\varphi_i,\nu) \models F$) is inductively defined:

$(\varphi_i,\nu) \models F$	iff F is a state formula and is valid in φ_i w.r.t. ν in the usual sense of first-order logic with equality;
$(\varphi_i,\nu) \models \neg F$	iff $(\varphi_i,\nu) \models F$ does not hold;
$(\varphi_i,\nu) \models F \rightarrow G$	iff $(\varphi_i,\nu) \models F$ implies $(\varphi_i,\nu) \models G$;
$(\varphi_i,\nu) \models \circ F$	iff $(\varphi_{i+1},\nu) \models F$ holds;
$(\varphi_i,\nu) \models F\,\mathbf{until}\,G$	iff there is a $k \geq i$ such that $(\varphi_k,\nu) \models G$ and for all j, $i \leq j < k$: $(\varphi_j,\nu) \models F$;
$(\varphi_i,\nu) \models F\,\mathbf{since}\,G$	iff there is a $k \leq i$ such that $(\varphi_k,\nu) \models G$ and for all j, $i \geq j > k$: $(\varphi_j,\nu) \models F$;
$(\varphi_i,\nu) \models \forall X.F$	iff $(\varphi_i,\nu') \models F$ holds for all valuations ν' with $\nu(Y) = \nu'(Y)$ for $Y \neq X$.

- F is said to be valid in a computation $\mathcal{C}$ w.r.t. ν (denoted by $(\mathcal{C},\nu) \models F$) if $(\varphi_i,\nu) \models F$ holds for all states φ_i of $\mathcal{C}$. F is said to be valid in $\mathcal{C}$ (denoted by $\mathcal{C} \models F$), if $(\mathcal{C},\nu) \models F$ holds for all valuations. F is said to be valid in a program P (denoted by P $\models F$) if it is valid in all computations of P. F is said to be valid in a programming language (Π,EA) (denoted by $(\Pi,\mathrm{EA}) \models F$) if it is valid in all its programs. F is said to be valid if it is valid in all computations. □

It is easy to see that for a state formula F not containing dynamic function symbols P $\models F$ holds iff F is valid in structure P in the sense of classical first-order logic with equality.

2.3 Relating Evolving Algebras to the Logic

The proofs in the following sections are carried out by computational induction, either by showing that the proposition holds in the first state and is invariant during the computation, or by showing that the proposition holds always sometimes in the future and remains invariant going back in the computation history. In order to proof the induction step, we have to refer to the evolving algebra. Therefore, we define the weakest precondition transformer **wp** associated with an evolving algebra. **wp** transforms a state formula F into the weakest state formula such that $\mathbf{wp}[F] \rightarrow \circ F$ is valid in the programming language.

As it is sufficient for the purposes of this paper and technically simpler, we define **wp** here only for state formulas that contain no function symbol with arity $n > 0$. (The general case can be handled by a technique described in [GR92], p. 189). Under these restrictions, we can use the predicate transformer technique associated with Hoare logic, in particular the transformers for multiple assignments and alternative commands (see e.g. [Gri81], p. 121f and 131f). But it should be clear that even though the technique is the same, the usage is quite different: There the transformers apply to program parts; here the transformer applies to the whole program. There the pre-/postconditions cannot refer to program counter, valuation, and control stacks; here we refer to them in the formulas. There the so-called Hoare triples relate information before a program part to information that holds after it; here we relate information of two successive states in a computation.

Definition 2.4 Let EA $= \{ R_1, \ldots, R_n \}$ be an evolving algebra and let us denote the vector of dynamic variables occurring on the left hand side in the updates of R_i by $\vec{x}_i$ and the corresponding vector of right hand sides by $\vec{u}_i$. Furthermore, let F be a state formula not containing a function symbol with arity $n > 0$. Denoting the simultaneous substitution of a vector of 0-ary function symbol with $\vec{x}$ by a vector of terms $\vec{u}$ in a formula F by $F[\vec{x}/\vec{u}]$, the weakest precondition transformer **wp** is defined as follows:

$$\mathbf{wp}[F] \equiv_{def} \quad \bigwedge_{i=1}^{n} (\ guard(R_i) \ \rightarrow \ F[\vec{x}_i/\vec{u}_i]\ ) \ .$$

□

The characteristic property of **wp**, namely $\mathbf{wp}[F] \rightarrow \circ F$, follows from defintion 2.1 and the definition of the nexttime operator.

3 Specifying Data Flow Problems

The first part of this section explains how data flow problems can be specified by temporal formulas. In the second part, we give the evolving algebra of a procedural example language and show how specifications of data flow problems are made complete w.r.t. programming language specifications.

3.1 Specifying Data Flow Problems by Temporal Formulas

Data flow analyses usually provide information that some proposition is true at a point of a program whenever the program reaches this point during execution. E.g. constant propagation determines that an expression E has always the same value C at program point P. Assuming that the programming language specification (Π, EA) provides a dynamic variable PC for the program counter and the dynamic function VAL for the valuation of program variables and expressions in a state, we can specify constant propagation as follows: Find a predicate

$$const : (\text{Variables} \cup \text{Expressions}) \times \text{Points} \times \text{Values} \ \rightarrow \ \text{Bool}$$

that satisfies the specification:

$$(\Pi, \mathrm{EA}) \models \ \ const[E, P, C] \ \rightarrow \ (\ \mathrm{PC} = P \ \rightarrow \ \mathrm{VAL}(E) = C\ )\ )$$

Assuming that *const* captures the information determined by a constant propagation implementation, the formula expresses the correctness of that implementation: It requires that $const[E, P, C]$ may only hold if $\mathrm{VAL}(E) = C$ holds in all states where the program counter is at P (in all computations of all programs). Implementations are partially ordered by the implication, i.e. we say an implementation *const1* is better/more precise than *const2* if both satisfy the specification and

$$(\Pi, \mathrm{EA}) \models \ \ const2[E, P, C] \ \rightarrow \ \ const1[E, P, C]$$

It is a well-known fact that in general optimal solution according to our definition cannot be computed ([RL77]).

Many data flow analysis determine whether something has happened before a point is reached or will happen in the future after a point is reached. Typical examples for the first kind — called *forward analyses*, as the analyses follow the normal flow of program execution — are available expression analysis (*avexp*), and initialization analysis (*init*). Typical examples for the second kind -- so called *backward* analyses — are very busy expression analysis (*vbe*), and liveness analysis. With the liveness specification we will as well show how to handle so called any-path problems. When looking at the data flow specifications, one should notice that the form of the conclusions provide a fine classification for DFAs: A DFA is history insensitive if the conclusion is a state formula (e.g. *const*); whether it is forward or backward, depends on the used temporal operators.

An expression E is called available at a point P, if in every computation that reaches P, E was not killed since it was defined:

$$(\Pi, \mathrm{EA}) \models \quad avexp[E, P] \quad \rightarrow \quad (\, \mathrm{PC} = P \rightarrow (\, \neg \mathrm{KILL}(E) \, \mathbf{since} \, \mathrm{DEF}(E)\,)\,)$$

where KILL(E), DEF(E) abbreviate formulas expressing that E is killed/defined (see below). Initialization analysis determines that whenever a computation reaches a point P, a variable V was defined sometimes in the past:

$$(\Pi, \mathrm{EA}) \models \quad init[V, P] \quad \rightarrow \quad (\, \mathrm{PC} = P \rightarrow (\, \mathrm{TT} \, \mathbf{since} \, \mathrm{DEF}(E)\,)\,)$$

An expression E is called very busy at a point P, if it is not killed until it is used:

$$(\Pi, \mathrm{EA}) \models \quad vbe[E, P] \quad \rightarrow \quad (\, \mathrm{PC} = P \rightarrow (\, \neg \mathrm{KILL}(E) \, \mathbf{until} \, \mathrm{USE}(E)\,)\,)$$

A variable V is called live at a program point P if there is a computation in which V is not defined until it is used after the computation reached P. As linear temporal logic can only express properties that hold in all computations, we specify the dual problem: A variable V is called dead at a program point P, if it is not used unless[6] it is defined in computations following P:

$$(\Pi, \mathrm{EA}) \models \quad dead[V, P] \quad \rightarrow \quad (\, \mathrm{PC} = P \rightarrow (\, \neg \mathrm{USE}(V) \, \mathbf{unless} \, \mathrm{DEF}(V)\,)\,)$$

In the following, we use the deadness problem as example, because it allows us to show the correctness of a backward analysis even for non-terminating computations and because it is more familiar than *vbe*.

3.2 Data Flow Problems and Language Specification

The specifications in the last section contain "wholes": The formulas KILL(E), USE(V), DEF(V) are not specified. Whereas these properties are often trivial for toy languages, they may be a rather complex in realistic programming languages: E.g. what does it mean precisely that an expression containing a procedure call is not killed[7]; what does is mean in the presence of aliasing that a variable is used. (And in both cases data flow analysis will be necessary to determine approximate solutions.) Of course, these properties are usual language depend. In this section we give examples how the specification of such properties can be done based on our language specification framework. In order to do so, we first introduce an imperative example programming language, called SPL (for simple procedural language) in the rest of this paper.

[6] To be precise: $\neg(\neg \mathrm{DEF}(E) \, \mathbf{until} \, \mathrm{USE}(E)\,)$ is equivalent to $\neg \mathrm{USE}(E) \, \mathbf{unless} \, (\mathrm{DEF}(E) \wedge \neg \mathrm{USE}(E)\,)$; but we assume here $\mathrm{DEF}(E) \rightarrow \neg \mathrm{USE}(E)$ which gives us the above specification (cf. 3.2).

[7] This information is e.g. very helpful to optimize expressions computing the bound in a C for–statement.

Language Specification This paragraph focuses on the relevant parts of the SPL specification. Appendix A contains the abstract syntax and the function *succ* on points that defines the relation between the syntax tree and the flow graph (we use the notation of the MAX system (cf. [PH93b]). SPL has recursive procedures and global variables; to keep it in manageable size, SPL does not have parameters and local variables, and the language only deals with values of type integer. The universes of the structures modeling programs (cf. section 2.1) contain the nodes of the syntax tree and two program points for each expression, statement, and procedure (declaration) node, representing the points before and after that construct. Boolean functions are used to describe a sorting of the universe; in particular there are the sorting functions: *Var* for the variable (declaration) nodes, *Procedure* for the procedure (declaration) nodes, *IntConst* for the nodes that are constants, *Exp* for the expression nodes, *CallStm* for the call statements, and *Point* for the program points.

The elements of the universe carrying values during runtime are called objects; in our notation, we write

```
Object  =  Var  |  IntConst  |  Exp
```

meaning that an object is a variable, constant[8], or expression node. The control flow of a program is modeled by a graph the nodes of which are either program points or so called tasks (cf. appendix A). Here are the sort defintions for the tasks:

```
Task     =  Branch | Move  | Succ | Call | Return | End
Succ     =  Point  | BinOp | Start
Branch   (  Exp.cond   Task.ttsucc   Task.ffsucc  )
Move     (  Object.src    Object.dst   Task.succ  )
BinOp    (  Operator.op   Exp.left  Exp.right  Exp.dst  Task.succ  )
Call     (  Point.contin  Task.succ  )
Start    (  Task.succ  )
Return   (  Procedure.proc  )
End      ()
```

The subsort *Succ* is introduced for later use. A branch task has three components: The expression node representing the condition in a loop or if–statement (selected by *cond*), and the successor task for the true and false case. A move task has as well three components: the source and destination object and the successor task. A binary operation task has the operator, the two argument and the result expression and its successor as components. A call task has the point where execution continues after the call and the entry in the called procedure (succ) as component. The return task has the procedure where it returns from as component.

The SPL specification contains the dynamic function VAL and two dynamic variables: PC, the program counter; and CTR_S, the control stack.

```
VAL      :  Object  ->  Int
PC       :  ->  Task
CTR_S    :  ->  Stack(Task)
```

The evolving algebra of SPL defines for each task sort how to change the dynamic functions/variables. The function *eval* evaluates a binary operator on the given arguments.

```
if  Branch[PC] ∧ VAL(cond(PC)) = 1   then   PC := ttsucc(PC)
if  Branch[PC] ∧ VAL(cond(PC)) = 0   then   PC := ffsucc(PC)

if  Point[PC]   then  PC := succ(PC)
if  Move[PC]    then  PC := succ(PC),   VAL(dst(PC)) :=  VAL(src(PC))
if  BinOp[PC]   then  PC := succ(PC),   VAL(dst(PC)) :=  eval( op(PC),
                                                 VAL(left(PC)), VAL(right(PC)) )
if  Call[PC]    then  PC := succ(PC),   CTR_S := push(contin(PC),CTR_S)
```

[8] To treat constants as (read only) objects makes the semantics a bit simples

```
if  Start[PC]  then  PC := succ(PC),    CTR_S := push( End(), EmptyStack() )
if  Return[PC] then  PC := top(CTR_S),  CTR_S := pop( CTR_S )
if  End[PC]    then  skip
```

Filling the Wholes The notable aspect of formal language specifications as the one sketched in the previous subsection is that they provide a powerful basis for specifying a wide variety of static and dynamic properties. E.g. they provide the sets of program variables, program points, and expressions for a given program (compare this to frameworks where the program is modeled by an element of a term algebra). These sets are very important for the specification of properties like KILL(E), USE(V), DEF(V). In order to complete the deadness specification given in 3.1, we have to specify what USE(V) and DEF(V) mean in SPL:

$$\begin{array}{lll} \mathrm{USE}(V) & \equiv_{def} & \mathit{Var}[V] \wedge \mathit{Move}[\mathrm{PC}] \wedge \mathit{src}(\mathrm{PC}) = V \\ \mathrm{DEF}(V) & \equiv_{def} & \mathit{Var}[V] \wedge \mathit{Move}[\mathrm{PC}] \wedge \mathit{dst}(\mathrm{PC}) = V \end{array}$$

i.e. a variable V is used/defined in a state if the program counter is at a move task and the source/destination of that task equals V. In SPL, there is never a read and write to a variable in one step, i.e. $\neg(\mathrm{DEF}(V) \wedge \mathrm{USE}(V))$ is valid in SPL. This is due to the fact that we use a store semantics where the value of a variable on the right hand side of an assignment is first moved to the corresponding expression node (see appendix A).

4 Correctness of Fixpoint Implementations

As shown in the last section, a temporal specification of a data flow problem is given by a temporal logic formula with say free variables $V_1, \ldots, V_n$. A corresponding implementation is considered to be a n-ary predicate *dfa*; we say *dfa* captures the implementation. To be more precise, each program structure in Π is extended by the *dfa* predicate and possibly some auxiliary predicates; the extension of Π is denoted by $\Pi_{\{dfa,\ldots\}}$. The properties of an implementation are expressed by a finite number of state formulas $F_1, \ldots, F_m$ over $\Sigma' \cup \{dfa_{(n)}, \ldots\}$ where Σ' is the signature of the programming language without the dynamic symbols; i.e.:

$$(\Pi_{\{dfa,\ldots\}}, \mathrm{EA}) \models \;\; F_1 \wedge \ldots \wedge F_n$$

Usually, the axioms F_i express the well-known fixpoint properties of data flow analyses (see below) and can often be directly used as an input to a fixpoint finding evaluator (see the remarks in the conclusions). Proving correctness of an implementation then means to prove that

$$(\Pi_{\{dfa,\ldots\}}, \mathrm{EA}) \models \;\; dfa[V_1, \ldots, V_n] \rightarrow \mathrm{SPEC}(V_1, \ldots, V_n)$$

holds where $\mathrm{SPEC}(V_1, \ldots, V_n)$ denotes the temporal specification of the problem. Usually the axioms F_i do not uniquely characterize the implementation, but define necessary conditions for the correctness of an implementation. E.g. the implementor may choose the fixpoint that provide the "best" information.

This section gives a fixpoint characterization of a simple deadness analysis for SPL, called *sdead*, and proves it correct w.r.t. the specification of the previous section. The following section investigates more complex implementations providing better information. The analysis *sdead* satisfies the following axiom:

$$\begin{array}{lll} sdead[V,T] & \leftrightarrow & \mathit{Var}[V] \wedge \mathit{Task}[T] \\ \quad \wedge \; (\mathit{Move}[T] & \rightarrow & V = dst(T) \vee (V \neq src(T) \wedge sdead[V, succ(T)])\,) \\ \quad \wedge \; (\mathit{Branch}[T] & \rightarrow & sdead[V, ttsucc(T)] \wedge sdead[V, ffsucc(T)]\,) \\ \quad \wedge \; (\mathit{Succ}[T] & \rightarrow & sdead[V, succ(T)]\,) \\ \quad \wedge \; (\mathit{Call}[T] & \rightarrow & sdead[V, succ(T)]\,) \\ \quad \wedge \; (\mathit{Return}[T] & \rightarrow & \forall P : \mathrm{CONTINP}(P, proc(T)) \rightarrow sdead[V, P]\,) \end{array}$$

where CONTINP($P, PROC$) holds if P is the point after a call site of procedure $PROC$ (the formal definition of CONTINP is given in appendix A). Recalling that *proc* yields —given a return task—the returning procedure, the last line says that a variable is dead at a return task if it is dead after all call sites of the returning procedure in the program. Let us denote this axiom by FIXPDEAD(V, T); it essentially spells out a simple graph based fixpoint specification of deadness using our logical framework. Obviously, FIXPDEAD(V, T) is a state formula not containing dynamic symbols. And, there are predicates satisfying FIXPDEAD(V, T), because *sdead* occurs only positive on the right hand side of the equivalence (cf. [GS86]). The correctness of *sdead* is expressed by the following theorem where $\mathrm{SPL}_{\{sdead\}}$ denotes the extension of SPL.

Theorem 4.1 Each implementation *sdead* satisfying FIXPDEAD(V, T) is correct w.r.t. to the temporal specification of the section 3; i.e. if

$$\mathrm{SPL}_{\{sdead\}} \models \quad \mathrm{FIXPDEAD}(V, T)$$

then

$$\mathrm{SPL}_{\{sdead\}} \models \quad sdead[V, T] \rightarrow (\mathrm{PC} = T \rightarrow (\neg\mathrm{USE}(V)\ \mathbf{unless}\ \mathrm{DEF}(V)))$$

$\Box$

The proof of theorem 4.1 is given in some detail to demonstrates the basic proof principles needed for showing data flow analyses correct. In particular, the proof demonstrates how correctness can be established even for non-terminating computations.

Proof Assuming that FIXPDEAD(V, T) is valid in $\mathrm{SPL}_{\{sdead\}}$, we have to show

$$\mathrm{SPL}_{\{sdead\}} \models \quad sdead[V, \mathrm{PC}] \rightarrow \neg\mathrm{USE}(V)\ \mathbf{unless}\ \mathrm{DEF}(V)$$

Let us abbreviate this formula by CORR(V). The proof runs by backward computational induction, i.e we show

1. $\mathrm{SPL}_{\{sdead\}} \models \ \Diamond\, \mathrm{CORR}(V)$
2. $\mathrm{SPL}_{\{sdead\}} \models \ \circ\, \mathrm{CORR}(V) \rightarrow \mathrm{CORR}(V)$

Ad 1: In order to show $\Diamond\, \mathrm{CORR}(V)$, we consider the cases $\Box\neg\mathrm{USE}(V)$ and $\Diamond\mathrm{USE}(V)$.

<u>Case 1:</u> The definition of **unless** gives $\Box\neg\, \mathrm{USE}(V) \rightarrow (\neg\mathrm{USE}(V)\ \mathbf{unless}\ \mathrm{DEF}(V))$ which implies $\Box\neg\, \mathrm{USE}(V) \rightarrow \mathrm{CORR}(V)$; by the law $F \rightarrow \Diamond F$ and the transitivity of the implication we get $\Box\neg\mathrm{USE}(V) \rightarrow \Diamond\, \mathrm{CORR}(V)$.

<u>Case 2:</u> Using the fact that variables are not simultaneously read and written, i.e. $\mathrm{USE}(V) \rightarrow \neg\mathrm{DEF}(V)$ and expanding the definition of USE we get

$$\mathrm{USE}(V) \rightarrow \neg\mathrm{DEF}(V) \wedge \mathrm{Var}[V] \wedge \mathrm{Move}[\mathrm{PC}] \wedge src(\mathrm{PC}) = V$$

By expanding DEF and propositional simplification, we get

$$\mathrm{USE}(V) \rightarrow \mathrm{Var}[V] \wedge \mathrm{Move}[\mathrm{PC}] \wedge V \neq dst(\mathrm{PC}) \wedge V = src(\mathrm{PC})$$

As the conclusion implies the negated right hand side of FIXPDEAD(V, PC), we conclude $\mathrm{USE}(V) \rightarrow \neg\, sdead[V, \mathrm{PC}]$; and because of $F \rightarrow G \vdash \Diamond F \rightarrow \Diamond G$ we have

$$\Diamond\mathrm{USE}(V) \rightarrow \Diamond\mathrm{CORR}(V)$$

As $\Box\neg\mathrm{USE}(V) \vee \Diamond\mathrm{USE}(V)$ is valid, proposition 1 is proved.

Ad 2: The prove of the backward induction step consists of two parts: First we show that

$$\text{INV} \equiv_{def} \neg\text{DEF}(V) \wedge sdead[V, \text{PC}] \rightarrow \mathbf{wp}[sdead[V, \text{PC}]]$$

is valid in $\text{SPL}_{\{sdead\}}$. Then, it is shown that INV implies $\circ\,\text{CORR}(V) \rightarrow \text{CORR}(V)$.

Step 1: Using FIXPDEAD and performing some simple propositional simplifications, it is easy to see that the premise of INV implies

$$\begin{array}{ll} & (\ Branch[\text{PC}] \rightarrow sdead[V, ttsucc(\text{PC})] \wedge sdead[V, ffsucc(\text{PC})]\ ) \\ \wedge & (\ Move[\text{PC}] \vee Succ[\text{PC}] \vee Call[\text{PC}] \rightarrow sdead[V, succ(\text{PC})]\ ) \\ \wedge & (\ Return[\text{PC}] \rightarrow \forall P : \text{CONTINP}(P, proc(\text{PC})) \rightarrow sdead[V, P]\ ) \\ \wedge & \ sdead[V, \text{PC}] \end{array}$$

The weakest precondition of $sdead[V, \text{PC}]$ is

$$\begin{array}{lll} & (\ Branch[\text{PC}] \wedge \text{VAL}(cond(\text{PC})) = 1 & \rightarrow sdead[V, ttsucc(\text{PC})]\ ) \\ \wedge & (\ Branch[\text{PC}] \wedge \text{VAL}(cond(\text{PC})) = 0 & \rightarrow sdead[V, ffsucc(\text{PC})]\ ) \\ \wedge & (\ Point[\text{PC}] & \rightarrow sdead[V, succ(\text{PC})]\ ) \\ \wedge & (\ Move[\text{PC}] & \rightarrow sdead[V, succ(\text{PC})]\ ) \\ \wedge & (\ BinOp[\text{PC}] & \rightarrow sdead[V, succ(\text{PC})]\ ) \\ \wedge & (\ Call[\text{PC}] & \rightarrow sdead[V, succ(\text{PC})]\ ) \\ \wedge & (\ Start[\text{PC}] & \rightarrow sdead[V, succ(\text{PC})]\ ) \\ \wedge & (\ Return[\text{PC}] & \rightarrow sdead[V, top(\text{CTR\_S})]\ ) \\ \wedge & (\ End[\text{PC}] & \rightarrow sdead[V, \text{PC}] \end{array}$$

For all cases except the return case propositional reasoning is sufficient to show the implication. The return case is shown using the lemma

$$Return[\text{PC}] \rightarrow \text{CONTINP}(\ top(\text{CTR\_S}), proc(\text{PC})\ )$$

which simply says that on return of a procedure P there is a call site of P on top of the control stack. The prove of this lemma is sketched in appendix B.

Step 2: The prove that INV implies $\circ\,\text{CORR}(V) \rightarrow \text{CORR}(V)$ uses the basic property of **wp**, i.e. $\mathbf{wp}[sdead[V, \text{PC}]] \rightarrow \circ sdead[V, \text{PC}]$; uses $\text{USE}(V) \rightarrow \neg\, sdead[V, \text{PC}]$ (which was proved in **Ad 1**); and uses the inductive characterization of **unless**. Details are given in appendix B.

q.e.d.

5 Refining Fixpoint Implementations

Of course, the simple fixpoint solution presented in the previous section provides a rather rough approximation to the specification. E.g. in the program

```
    proc  Q() :  if  y=0  then  z:=y  else  y:=y-1 ; Q()  fi

(1)     ...  Q() ; x:=0 ;
(2)     ...  Q() ; z:=x ;
```

the variable `x` is dead before the call to `Q` in line (1), but this cannot be detected by *sdead*, because *sdead* "identifies" all continuations of `Q` and `x` is obviously live after the call in line (2).

This section presents two refinements of the simple deadness analysis. The first one makes use of an idea underlying most interprocedural data flow analyses; the second uses information from constant propagation to improve the result. For both refinements, we will sketch how to prove them correct w.r.t. the temporal specification.

Interprocedural Analysis The main idea of interprocedural data flow analysis is to first compute the mapping procedures perform on data flow information and then to use these mappings to compute the data flow information itself (cf. [KS92]). For the refined deadness analysis *ipdead* (for interprocedural deadness), we introduce an auxiliary predicate *pdead* that tells us that a variable V is dead at a task in a procedure P assuming that it is dead at the end of P. The characterization for *ipdead* is identical to that of *sdead* except for the call case:

$$
\begin{array}{lll}
ipdead[V,T] \leftrightarrow & & \ldots \\
\quad \wedge \; (\; Call[T] & \rightarrow & ipdead[V, succ(T)] \\
 & & \vee \; (\; pdead[V, succ(T)] \wedge ipdead[V, contin(T)] \;)\;) \\
\quad \wedge & & \ldots
\end{array}
$$

$$
\begin{array}{lll}
pdead[V,T] \leftrightarrow & & ipdead[V,T] \vee (\; Var[V] \wedge Task[T] \\
\quad \wedge \; (\; Move[T] & \rightarrow & V = dst(T) \vee (V \neq src(T) \wedge pdead[V, succ(T)]) \;) \\
\quad \wedge \; (\; Branch[T] & \rightarrow & pdead[V, ttsucc(T)] \wedge pdead[V, ffsucc(T)] \;) \\
\quad \wedge \; (\; Succ[T] & \rightarrow & pdead[V, succ(T)] \;) \\
\quad \wedge \; (\; Call[T] & \rightarrow & pdead[V, succ(T)] \wedge pdead[V, contin(T)] \;) \\
)
\end{array}
$$

An analysis that computes the greatest fixpoint of the above axioms determines that variable `x` is dead before the call to `Q` in line (1), as $pdead[x, Q_0]$ holds for Q_0 being the first point in procedure `Q`. The correctness proof for *ipdead* has the same structure as the correctness proof of *sdead*. The difference is that instead of $\mathrm{CORR}(V)$ we have to use

$$ipdead[V, \mathrm{PC}] \vee (\; pdead[V, \mathrm{PC}] \wedge ipdead[V, top(\mathrm{CTR\_S})] \;) \rightarrow \neg\mathrm{USE}(V) \;\textbf{unless}\; \mathrm{DEF}(V)$$

in the induction step.

Two Refinements Using Constant Infomation Even though the above analysis gives us optimal results (in the sense of [KS92]) for fixpoint specifications that take only uses and definitions of variables into account, it is obviously not the best solution we can get. Here, we show how to use information obtained by constant propagation to produce better approximations to the temporal deadness specification. Of course, other analyses could be used as well (or in addition) like e.g. interval analysis. The main point here is that the simple temporal specification remains the valid reference to prove correctness whatsoever sophisticated the analysis is. In addition to this, the refinement exemplifies how to combine data flow analyses and how to use their specifications in proofs.

The two refinements discussed in this section will solve the problem illustrated by the program fragment in the introduction. To focus on the main issues, we refine here the simple deadness analysis of section 4. The corresponding refinements for *ipdead* are spelled out in appendix C. The first refinement uses the constant information captured by *const* introduced in section 3. The refined version of *sdead* is obtained by replacing the branch case in its fixpoint characterization by

$$
\begin{array}{lll}
\ldots & & \\
\wedge \; (\; Branch[T] \wedge const[cond(T), T, 1] & \rightarrow & sdead[V, ttsucc(T)] \;) \\
\wedge \; (\; Branch[T] \wedge const[cond(T), T, 0] & \rightarrow & sdead[V, ffsucc(T)] \;) \\
\wedge \; (\; Branch[T] \wedge \neg const[cond(T), T, 1] \wedge \neg const[cond(T), T, 0] & & \\
\qquad\qquad \rightarrow \; sdead[V, ttsucc(T)] \wedge sdead[V, ffsucc(T)] \;) & &
\end{array}
$$

i.e. whenever we know the outcome of the test, we claim the deadness only for the taken branch. The correctness proof of implementations characterized by this axiom differs from that of theorem 4.1 only in that we have to use lemmas of the form $sconst[cond(\mathrm{PC}), \mathrm{PC}, 1] \rightarrow$

$\mathrm{VAL}(cond(\mathrm{PC})) \neq 0$ to prove $\neg\mathrm{DEF}(V) \wedge sdead[V,\mathrm{PC}] \rightarrow \mathbf{wp}[sdead[V,\mathrm{PC}]]$. And they are almost direct consequences from the *const* specification.

Assuming a reasonably powerful implementation for *const*, our refined deadness analysis detects the deadness of x in the program fragment of the introduction — at least if the call to P is the only call of P in the program. If there is a second call to P with a different context for y, e.g.

```
(2)     ...  y:=0 ; P() ;
```

sdead again fails to detect the deadness of x at the beginning of line (1), because the constant propagation *const* can not distinguish different calling contexts of P. But in applications, it is by far more interesting to analyse a procedure in its different calling contexts (in particular to take actual parameter values into account) than to profit only from dead code elemination as the above analysis does. The following refinement provides such an analysis by making the constant and deadness information at a task depend on the call site from which the procedure enclosing the task was called. By giving this refinement, we not only present a very powerful deadness analysis (in particular in the interprocedural version), but provide as well a good example to show the power of our framework. It goes without saying that the used technique applies as well to refine other data flow analyses.
For the refinement, we need a constant propagation satisfying the following specification:

$$cconst[E,T,SITE,C] \;\rightarrow\; (\ \mathrm{PC}=T \wedge top(\mathrm{CTR\_S}) = SITE \;\rightarrow\; \mathrm{VAL}(E)=C\ )$$

i.e. if $cconst[E,T,SITE,C]$ holds, then expression E has the value C whenever control reaches task T and the active procedure was called from site $SITE$ (*cconst* stands for **c**ontext sensitive constant propagation). To reuse above definitions, we represent call sites by the corresponding continuation point, i.e. the point after the call statement; accordingly, the call site of the main procedure is represented by the end task (this was til now hidden in the definition of CONTINP). The refined deadness analysis is characterized by two axioms: The first axiom is the fixpoint equivalence for the deadness analysis *csdead* where calling contexts are taken into account. The second axiom expresses that a variable is dead if it is dead according to all call sites:

$$\begin{array}{lll}
csdead[V,T,S] \leftrightarrow & Var[V] \wedge Task[T] \wedge \mathrm{CONTINP}[S, encl_proc(T)] & \\
\quad \wedge\ (\ Move[T] & \rightarrow\ \ V = dst(T) \vee (V \neq src(T) \wedge csdead[V, succ(T), S])\ ) & \\
\quad \wedge\ (\ Branch[T] \wedge & cconst[cond(T),T,S,1] \ \rightarrow\ csdead[V, ttsucc(T), S]\ ) & \\
\quad \wedge\ (\ Branch[T] \wedge & cconst[cond(T),T,S,0] \ \rightarrow\ csdead[V, ffsucc(T), S]\ ) & \\
\quad \wedge\ (\ Branch[T] \wedge & \neg cconst[cond(T),T,S,1] \wedge \neg cconst[cond(T),T,S,0] & \\
 & \rightarrow\ \ csdead[V, ttsucc(T), S]\ ) \wedge csdead[V, ffsucc(T), S]\ ) & \\
\quad \wedge\ (\ Succ[T] & \rightarrow\ \ csdead[V, succ(T), S]\ ) & \\
\quad \wedge\ (\ Call[T] & \rightarrow\ \ csdead[V, succ(T), contin(T)]\ ) & \\
\quad \wedge\ (\ Return[T] & \rightarrow\ \ \forall S' : \mathrm{CONTINP}(S', encl_proc(S)) \rightarrow csdead[V,S,S']\ ) &
\end{array}$$

$$\begin{array}{ll}
dead[V,T] \leftrightarrow & Var[V] \wedge Task[T] \\
\quad \wedge\ (\ \neg\, Start[T] & \rightarrow\ \ \forall S : \mathrm{CONTINP}(S, encl_proc(T)) \rightarrow csdead[V,T,S]\ ) \\
\quad \wedge\ (\ Start[T] & \rightarrow\ \ csdead[V, succ(T), End()]\ )
\end{array}$$

The interesting cases of the axiom for *csdead* are the call and return task. For the call case it is required that the variables have to be dead at the beginning of the called procedure w.r.t. the call site (represented by the continuation point). For the return case, variables are required to be dead after the current call site in all possible calling contexts S' of the procedure enclosing S. The function *encl_proc* yields for each task the enclosing procedure, and the dummy element *nil* for the start and end task; i.e. for the return task of the main procedure we get that CONTINP(S', $encl\_proc(End())$) is not valid for any S'. This means that every variable is dead at the return task of the main procedure and is the desired behaviour. The special treatment of the start task in the axiom for *dead* is necessary, because there is no calling context for the start task.

Again the correctness prove for *csdead* follows the lines of 4.1; but instead of CORR(V) we have to show $csdead[V, \mathrm{PC}, top(\mathrm{CTR\_S})] \rightarrow \neg\mathrm{USE}(V)$ **unless** DEF(V) using the specification of *cconst*.

6 Conclusions

The development of data flow analyses can benefit from high–level semantics–based specifications. Such specifications allow to formally define the general goal of a data flow problem in an intuitive way based on the programming language semantics whereas graph–based specifications often exclude interesting solutions from the very beginning. Complex and powerful analyses can then be proved correct w.r.t. the high–level specification. We chose evolving algebras as semantics framework, because they were successfully used for the specification of many different, realistic programming languages (cf. section 2.1) and provide a trace semantics which is helpful if not necessary to handle history sensitive analyses. As specification language for data flow problems we proposed formulas of linear time temporal logic and defined their validity w.r.t. evolving algebras.

After showing how to use temporal formulas to specify some classical data flow analyses, we stepwise refined a deadness analysis. The development of this example analysis had four goals: It should

1. illustrate correctness proofs in our framework;
2. show how different analyses (here a forward and backward analysis) can be combined;
3. demonstrate that sophisticated fixpoint–based data flow analyses can become very complex even for simple data flow problems which is here understood as a motivation for using higher–level specifications;
4. give an idea how refinements can be constructed by approximating state information; first, we approximated the function VAL, then the top of the control stack; therefore it is necessary to have all state information explicite in the language specification as it is the case for evolving algebras.

The presented work should be understood as a further step towards semantic–based development of efficient language implementation. As [Ven89], we are interested in constructing tools to support such developments. Currently, we are implementing a system that generates interpreters from a combination of an attribute grammar like formalism and evolving algebras (cf. [PH93a] and the specification in appendix A). This system provides all the notions we have used throughout this paper. In a next step, we want to integrate a fixpoint evaluator to the system that allows prototyping data flow analysis based on special forms of axioms. Concerning the correctness proofs, our goal is to apply proof development systems; the use of a formal logic framework is a first step in this direction.

References

[AU77] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison–Wesley, 1977.

[BR92] E. Börger and D. Rosenzweig. A simple mathematical model for full Prolog. Technical Report TR–33/92, Dipartimento di Informatica, Universita di Pisa, 1992.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. *Conference Record of the 6th ACM Symposium on Principles of Programming Languages*, 1979.

[GH92] Y. Gurevich and J. Huggins. The semantics of the C programming language. In E. Börger et al., editor, *Computer Science Logic*, pages 274–308, 1992. LNCS 702.

[GM89] Y. Gurevich and L. Moss. Algebraic operational semantics and Occam. In E. Börger et al., editor, *Computer Science Logic*, pages 176–192, 1989. LNCS 440.

[GR92] P. Glavan and D. Rosenzweig. Communication evolving algebras. In E. Börger et al., editor, *Computer Science Logic*, pages 182–215, 1992. LNCS 702.

[Gri81] D. Gries. *The Science of Programming.* Springer, 1981.

[GS86] Y. Gurevich and S. Shelah. Fixed-point extensions of first-order logic. *Annals of pure and applied logic*, 32, 1986.

[Gur91] Y. Gurevich. *Evolving Algebras*, volume 43, pages 264–284. EATCS Bulletin, 1991.

[KS92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In P. Pfahler U. Kastens, editor, *Compiler Construction*, 1992. LNCS 641.

[KU77] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica 7*, pages 309–317, 1977.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer–Verlag, 1992.

[MS76] R. Milner and C. Stratchey. *A Theory of Programming Language Semantics.* Chapman and Hall, 1976.

[Nie87] F. Nielson. Denotational theory of abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages.* Ellis Horwood Limited, 1987.

[PH93a] A. Poetzsch-Heffter. Developing efficient interpreters based on formal language specifications. To appear, 1993.

[PH93b] A. Poetzsch-Heffter. Programming language specification and prototyping using the MAX system. In J. Penjam M. Bruynooghe, editor, *Programming Language Implementation and Logic Programming*, 1993. LNCS 714.

[RL77] J. H. Reif and R. Lewis. Symbolic evaluation and the global value graph. *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[Ste91] B. Steffen. Data flow analysis as model checking. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, 1991. LNCS 526.

[Ven89] G. A. Venkatesh. A framework for construction and evaluation of high-level specification of program analysis techniques. *ACM Conference on Progamming Language Design and Implementation*, 1989.

[WZ85] M. N. Wegman and F. K. Zadek. Constant propagation with conditional branches. *Conference Record of the 12th ACM Symposium on Principles of Programming Languages*, 1985.

A Specification of SPL

This part of the appendix contains the abstract syntax of SPL and the function *succ* on points that defines the data flow graph. We use the notation of the MAX system (cf. [PH93a] for details). Besides this, we define the initial state of the dynamic valuation function VAL and CONTINP($P, PROC$).

```
Program        ( DeclList )
DeclList       * Decl
Decl           = Var  | Procedure
Var            ( Ident  )
Procedure      ( Ident StatList  )

StatList       * Stat
Stat           = While | IfStat | Assign | CallStat
While          ( Exp  StatList  )
IfStat         ( Exp  StatList  StatList  )
Assign         ( Ident  Exp  )
CallStat       ( Ident  )

Exp            = IntConst | VarId | Call | BinExp
IntConst       ( Int  )
VarId          ( Ident  )
BinExp         ( Operator  Exp  Exp  )
Operator       = Add  |  Sub  |  Mul  |  Div  |  Eq  |  Lt
```

The following function *succ* defines the successors to program points and thereby the data flow graph. It uses the functions *before* and *after* yielding the point before and after a node; the function *node* yielding the node corresponding to a point; the function *next* yielding the next point according to a left to right tree traversal; and the function *decl* yielding for each identifier the corresponding declaration node. Let us explain the used notation for the case

```
| Assign<ID,N> A        :  Move( N, decl(ID), after(A) )
```

where the parameter `P` is a point after its corresponding node `N` and `N` represents the right hand side of an assignment statement `A`. If this is the context of `P` than its successor is the move task that has `N` as source, the variable `decl(ID)` as destination, and the point after the assignment as successor.

```
FCT  succ( Point P ) Task:
  LET  N == node(P) :
  IF  NOT ( Procedure[N]  OR  Exp[N]  OR  Stat[N]  OR  StatList[N] ) :  nil()
   |  P = before(N) :
      IF  Procedure<_,_,_,SL> N :  before(SL)
       |  Assign<ID,E> N        :  before(E)
       |  VarId<ID> N           :  Move( decl(ID), N, after(N) )
       |  CallStat<ID> N        :  Call( after(N), before(decl(ID)) )
       |  BinExp@<_,LO,_> N     :  before(LO)
      ELSE  next(P)
   |  P = after(N) :
      IF  Procedure N           :  Return( N )
       |  While<N,BD>  W        :  Branch( N, before(BD), after(W) )
       |  While<E,N>            :  before(E)
       |  IfStat<N,TB,EB>       :  Branch( N, before(TB), before(EB) )
       |  IfStat<_,N,_> IS      :  after(IS)
       |  Assign<ID,N> A        :  Move( N, decl(ID), after(A) )
       |  BinExp<O,L,N> E       :  BinOp( term(O), L, N, E, after(E))
      ELSE  next(P)
  ELSE  nil()
```

The initial state of the dynamic valuation function is as follows:

```
FCT  VAL( Object OBJ ) Int:
   IF  Var OBJ          :   0
    |  IntConst<I> OBJ :   I
   ELSE  nil()
```

Finally, we give the definition of CONTINP($P, PROC$):

$$\text{CONTINP}(P, PROC) \equiv_{def}$$
$$Procedure[PROC] \wedge ((Point[P] \wedge PROC = decl(firstson(node(P)))) \vee End[P])$$

B Proof Refinements

This part of the appendix provides the steps left out in the proof of theorem 4.1.

The Lemma: First, we sketch the proof of the lemma

$$Return[\text{PC}] \rightarrow \text{CONTINP}(top(\text{CTR\_S}), proc(\text{PC}))$$

which says that on return of a procedure $P = proc(\text{PC})$ there is a call site of P on top of the control stack. The idea is to prove

$$\begin{aligned} \text{SPL} \models \quad & i > 0 \wedge S = push(\text{PC}, \text{CTR\_S}) \rightarrow \\ & \quad \text{CONTINP}(top(i+1, S), encl\_proc(top(i, S))) \\ & \vee\ top(i+1, S) = nil \vee Start[\text{PC}] \end{aligned}$$

by computational induction where $top(i, S)$ stands for applying top i-times to S ($i > 0$). Let us denote the above formula by INV2. For $i = 1$, INV2 simplifies to

$$\text{CONTINP}(top(\text{CTR\_S})), encl\_proc(\text{PC})))$$

and because of $Return[\text{PC}] \rightarrow proc(\text{PC}) = encl\_proc(\text{PC})$ this is sufficient to proof the lemma.

The base case of the computational induction is trivial as $Start[\text{PC}]$ holds in the initial state. Because of $\text{wp}[\text{INV2}] \rightarrow \circ\, \text{INV2}$, it suffices to show for the induction step that $\text{INV2} \rightarrow \text{wp}[\text{INV2}]$ is valid in SPL. The proof of that is straightforward, but rather technical and therefore not carried out here.

Second Step of Ad 2: Because of $\mathbf{wp}[sdead[V, \text{PC}]] \rightarrow \circ sdead[V, \text{PC}]$ the formula INV implies

$$\neg\text{DEF}(V) \wedge sdead[V, \text{PC}] \rightarrow \circ sdead[V, \text{PC}]$$

implying

$$\circ sdead[V, \text{PC}] \vee \neg sdead[V, \text{PC}] \vee \text{DEF}(V) \vee \circ(\text{USE}(V) \,\mathbf{unless}\, DEF(V))$$

which is equivalent to

$$(\circ sdead[V, \text{PC}] \wedge \neg \circ (\text{USE}(V) \,\mathbf{unless}\, DEF(V))) \vee \text{CONCL}$$

where $\text{CONCL} \equiv_{def} \neg sdead[V, \text{PC}] \vee \text{DEF}(V) \vee \circ(\text{USE}(V) \,\mathbf{unless}\, DEF(V))$. As $\neg sdead[V, \text{PC}] \vee \neg\text{USE}(V)$ is valid in $\mathcal{C}$ (s. above), CONCL is equivalent to

$$(\neg sdead[V, \text{PC}] \vee \neg\text{USE}(V)) \wedge \text{CONCL}$$

Applying distributivity twice, we get

$$\neg sdead[V, \mathrm{PC}] \ \vee \ (\neg \mathrm{USE}(V)) \wedge \mathrm{DEF}(V)) \ \vee \ (\neg \mathrm{USE}(V) \ \wedge \circ(\ \mathrm{USE}(V)\mathbf{unless}\,\mathrm{DEF}(V)\,)\,)$$

Because of $(DEF(V) \ \rightarrow \ \neg\mathrm{USE}(V)$ this implies

$$\neg sdead[V, \mathrm{PC}] \ \vee \ \mathrm{DEF}(V) \ \vee \ (\neg \mathrm{USE}(V) \ \wedge \circ(\ \mathrm{USE}(V)\mathbf{unless}\,DEF(V)\,)\,)$$

Using the inductive formulation of **unless**, we get

$$\neg sdead[V, \mathrm{PC}] \ \vee \ \mathrm{USE}(V)\mathbf{unless}\,DEF(V)$$

And it is easy to sea that

$$(\neg sdead[V, \mathrm{PC}] \ \vee \ \neg\mathrm{USE}(V)) \ \wedge \ \neg sdead[V, \mathrm{PC}] \ \vee \ \mathrm{USE}(V)\mathbf{unless}\,DEF(V)$$

is equivalent to CONCL.

C Interprocedural Deadness Analysis Using Context Dependent Constant Propagation

This part of the appendix provides the fixpoint axiomatization for the refinement of *ipdead/pdead* by context dependent constant information as illustrated for *sdead* in section 5. The resulting predicates are called *cipdead/cpdead*:

$$\begin{array}{lll}
cipdead[V,T,S] \leftrightarrow & Var[V] \ \wedge \ Task[T] \ \wedge \ \mathrm{CONTINP}[S, encl_proc(T)] & \\
\ \wedge \ (\ Move[T] & \rightarrow \quad V = dst(T) \ \vee \ (V \neq src(T) \wedge cipdead[V, succ(T), S])\ ) & \\
\ \wedge \ (\ Branch[T] \ \wedge & cconst[cond(T), T, S, 1] \ \rightarrow \ \ cipdead[V, ttsucc(T), S]\ ) & \\
\ \wedge \ (\ Branch[T] \ \wedge & cconst[cond(T), T, S, 0] \ \rightarrow \ \ cipdead[V, ffsucc(T), S]\ ) & \\
\ \wedge \ (\ Branch[T] \ \wedge & \neg cconst[cond(T), T, S, 1] \wedge \neg cconst[cond(T), T, S, 0] & \\
 & \rightarrow \quad cipdead[V, ttsucc(T), S]\ ) \wedge cipdead[V, ffsucc(T), S]\ ) & \\
\ \wedge \ (\ Succ[T] & \rightarrow \quad cipdead[V, succ(T), S]\ ) & \\
\ \wedge \ (\ Call[T] & \rightarrow \quad cipdead[V, succ(T), contin(T)] & \\
 & \quad \vee\ (\ cpdead[V, succ(T), contin(T)] \ \wedge \ cipdead[V, contin(T), S]\ )\ ) & \\
\ \wedge \ (\ Return[T] & \rightarrow \quad \forall S' : \mathrm{CONTINP}(S', encl_proc(S)) \ \rightarrow \ cipdead[V, S, S']\ ) &
\end{array}$$

$$\begin{array}{lll}
cpdead[V,T,S] \ \leftrightarrow & (\ Var[V] \ \wedge \ Task[T] \ \wedge \ \mathrm{CONTINP}[S, encl_proc(T)] & \\
\ \wedge \ (\ Move[T] & \rightarrow \quad V = dst(T) \ \vee \ (V \neq src(T) \wedge cpdead[V, succ(T), S])\ ) & \\
\ \wedge \ (\ Branch[T] \ \wedge & cconst[cond(T), T, S, 1] \ \rightarrow \ \ cpdead[V, ttsucc(T), S]\ ) & \\
\ \wedge \ (\ Branch[T] \ \wedge & cconst[cond(T), T, S, 0] \ \rightarrow \ \ cpdead[V, ffsucc(T), S]\ ) & \\
\ \wedge \ (\ Branch[T] \ \wedge & \neg cconst[cond(T), T, S, 1] \wedge \neg cconst[cond(T), T, S, 0] & \\
 & \rightarrow \quad cpdead[V, ttsucc(T), S]\ ) \wedge cpdead[V, ffsucc(T), S]\ ) & \\
\ \wedge \ (\ Succ[T] & \rightarrow \quad cpdead[V, succ(T), S]\ ) & \\
\ \wedge \ (\ Call[T] & \rightarrow \quad cpdead[V, succ(T), contin(T)] \ \wedge \ cpdead[V, contin(T), S]\ ) & \\
\ ) \ \vee & cipdead[V, T, S] &
\end{array}$$

$$\begin{array}{lll}
dead[V,T] \leftrightarrow & Var[V] \ \wedge \ Task[T] & \\
\ \wedge\ (\neg\, Start[T] & \rightarrow \ \ \forall S : \mathrm{CONTINP}(S, encl_proc(T)) \ \rightarrow \ cipdead[V, T, S]\ ) & \\
\ \wedge\ (\ Start[T] & \rightarrow \ \ cipdead[V, succ(T), End()]\ ) &
\end{array}$$