PAG
User's Manual

Stephan Thesing, Florian Martin, Martin Alt and Oliver Lauer
{thesing|florian|alt|olauer}@cs.uni-sb.de

Version: 1.0
September 18, 1998

Contents

Chapter 1

Introduction

Historically a compiler is divided into a *frontend*, a *middleend* and a *backend*. The task of the frontend is to parse the input program, transform it into an *abstract syntax tree* and check the static constraints of the input language[1]. The middleend's task is to perform *optimizations* on the program[2], which are independent from the target language. The backend's task is to generate code for the output language. See figure 1.1.

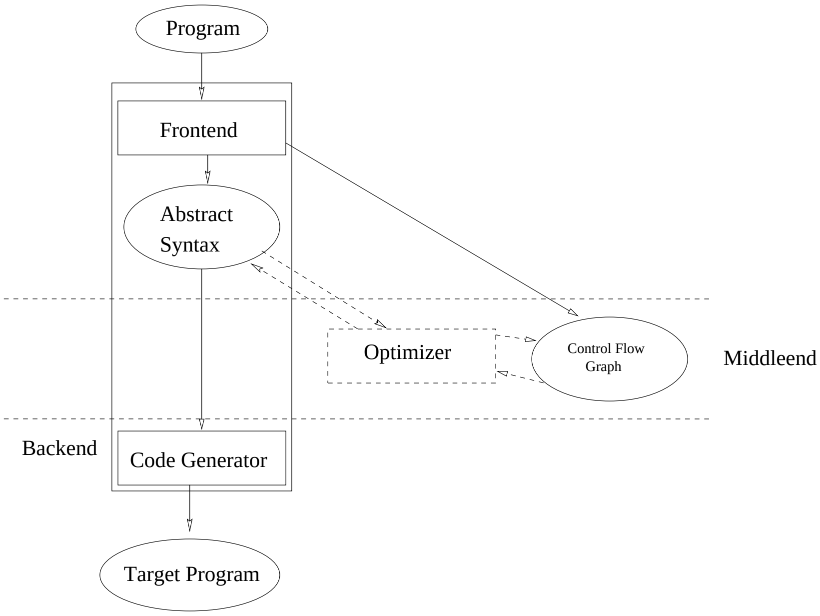


Figure 1.1: Compiler components

[1]E.g. scoping rules
[2]Resp. on its abstract syntax tree.

There are some widely used tools to generate code for the frontend's task of parsing[3], reducing the development costs of this phase. Also, there are some tools to generate code generators for backends. To support the generation of optimizers, PAG has been built: as a tool for the generation of *program analyzers*.
The task of an *optimizer* is to replace parts of the source program (i.e. its syntax tree) with a more efficient version, without changing the semantics of the program (e.g. if a variable always has a constant value at some point in the program, then the variable can be replaced by that constant value at compile time).
For this task the optimizer needs information about the program, e.g. whether a variable has a constant value and if, which. To obtain these information, one has to *analyze* the program, i.e. perform a *data flow analysis* (DFA) on it. This is done by simulating the execution of the program by computing a correct,i.e., valid for every input, approximation of the desired information. For this reason the program is transformed into a *control flow graph* (CFG), whose nodes are points[4] in the program and with an edge from a node to every possible next program point (node). See figure 1.2 for an example.

```
int fac(int n)
{
  int res;

  res=1;
    while (n>1) {
    res=res*n;
    n--;
  }
   return res;
}
```

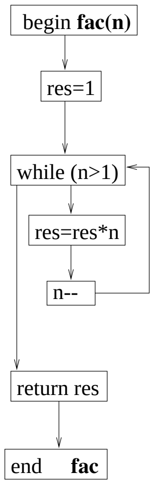


Figure 1.2: Program and it's CFG

Every node contains data, representing the desired information at this node. During the computation of the final information, interim data flows along the edges of the CFG to several nodes. Here the *incoming* information is transformed by a *transfer function*, according to the semantics of the program

[3] lex, yacc
[4] I.e. locations in the program.

fragment at this node and again propagated to all successor nodes. For nodes with multiple incoming edges, the data is mixed together by a special *combination operator*.
From this point of view, the CFG resembles a system of equations, with the information at each node as variables and the transfer functions at each node building the equations between the variables.
A solution of this system of equations gives the desired final information: every node contains the information for the corresponding program point.
To compute this solution, one may use an iteration algorithm. Every node is initialized to a start value. Then the data is propagated along the edges to other nodes, transformed and again propagated. This continues, until a stable state has been reached, i.e. the propagation of information leads to no changes at the nodes. Of course, the data and transform functions at the nodes must satisfy certain conditions to guarantee the existance and reachability of such a stable state (see section 2.1). In general, the information at each node has to form a complete lattice.
To generate an analyzer, one has to specify this lattice, the structure of the abstract syntax and the transfer functions for each node, along with some additional parameters of the analysis, as direction and initial values (see section 3.3).

1.1 PAG

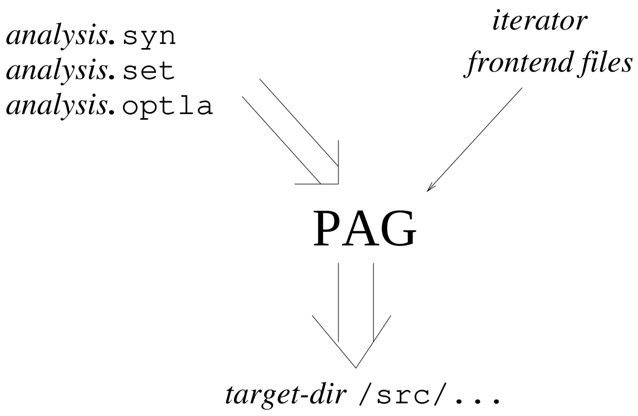
PAG takes the description of an analyzer and generates a set of **C** files in a directory.
From the compiler driver, the analyzer is called via a **C** function, which is passed in the actual CFG. The function returns a data structure, which associates each node of the CFG with the desired information for that node. The backend can then use this data in the optimization.
Naturally, the programmer must write (in **C**) an interface between the abstract syntax (AS) and the analyzer, e.g. functions to traverse the syntax tree. Anyhow, this interface must be written only once for every frontend and can be shared between analyzers in the same compiler.
Furthermore the frontend is responsible for the construction of the CFG itself and must offer functions to traverse the CFG.
PAG features a functional language to annotate the transfer (and supporting) functions, the definition of data types, specification of the analysis parameters and options to influence the generated analyzer at run time of the optimization. Naturally, PAG offers several iteration algorithms to select from.

1.1.1 Inputs

To generate an analyzer, PAG needs:

- The frontend used.
- The abstract syntax definition.
- The definition of the complete lattice of data values.
- The analysis definition:
  - The parameters (e.g. direction of analysis, carrier, initial values, see section 2.1).
  - The transfer functions for each node.
  - Support functions (optionally).

PAG reads the information from several files, which reside in a common directory and which are named *analysis*`.syn`, *analysis*`.set` and *analysis*`.optla`, where *analysis* stands for the name of the analysis.
The frontend used and the desired iteration algorithm are specified as options to the `pag` program, see figure 1.3 for the needed files and the command to produce the analyzer. Here *target-dir* stands for the target directory, where the generated `C` sources should be put (in the subdirectory `src`).



`pag` *frontend* *iterator*`.t spec/`*analysis* *target-dir*

Figure 1.3: Files needed and produced.

1.1.2 Interface to the Frontend

The interface provides types and functions:

- Analyzer ↔ Syntax Tree
  - **C** types for every syntactical type in the `.syn` file (see section 3.2.1).
  - Functions to access the components of a syntax tree.
  - Predicates to test the type of a syntax tree.
- Analyzer ↔ CFG
  - **C** types of nodes, edges in the CFG.
  - Functions to
    * init the CFG.
    * access successors, predecessors of a node.
    * access global data and node data in the CFG (see section 3.3.1).

1.1.3 Compiling the analyzer

PAG puts the generated **C**-Files into a subdirectory `src`. In addition, it generates a Bourne-shell (`sh`) script `compile` in that directory, which can be run to compile the source files. Furthermore, a makefile for the GNU make program is generated and put into the `src` subdirectory. It is called `GNUmakefile` and provides an alternative to running the `compile` script, but only if GNU make is used.

1.1.4 How to use the analyzer

Using the analyzer is simple: for every analysis, a function

`DFI_STORE` *analysis*`_doit(KFG` *k*`)`

is generated, which performs the actual analysis, taking a CFG k as input and returning the stored information, which has the type `DFI_STORE`. Here *analysis* stands for the name of the analysis.
In addition, functions

NodeType `dfi_get_post_info_all(DFI_STORE` *store*`, DFI_ID` *id*`)`

NodeType `dfi_get_pre_info(DFI_STORE` *store*`, DFI_ID` *id*`)`

are defined, which return the information, stored before/after the node *id* in the CFG. The type *NodeType* is the type of the information at each node of the CFG.

This information can then be used in further computations. Naturally, each data type offers **C** functions to manipulate the data obtained this way. See section 8.4 for details.

As another feature, FULA support functions can also be called from **C**, see section 10.8 for details.

Chapter 2

Theoretical Background

2.1 Data Flow Analysis

The task of data flow analysis is to compute static information about dynamic properties of a program, i.e. to compute at compile time information that is valid for all invocations of a program. So DFA can be seen as a special case of an *abstract interpretation*, see [CC77]. It was introduced in [Kil73] and [KU77].
DFA works by replacing the semantics of a program by an abstract one and 'tracing' the execution of the program. To solve such a *data flow problem* (DFP), the CFG of the program is built, which contains edges for every possible control transition and whose nodes represent parts of the program (see figure 1.2). Additionally, a unique start and end node is added to the graph.
Every node in the CFG is assigned a data value, representing the desired information (e.g. a mapping of all variables with constant values to those values). These values must form a *complete lattice*. The semantics of every program part is formulated for each node by a *transfer function*, operating on the data values.

2.1.1 Basic Definitions

A *control flow graph* (CFG) is a graph $G = (N, E, s, e)$, with a set N of *nodes*, a relation $E \subseteq N \times N$ of *edges*, a *start node* s and an *end node* e, $s, e \in N$. Iff $(n, m) \in E$, we say n is predecessor of m (m is successor of n). s is required to have exactly one successor and no predecessor, e to have exactly one predecessor and no successor. Additionally, we require the CFG to be *connected*, i.e. there are no nodes that can not be reached from s.

A *partial order* $(A, \sqsubseteq)$ is a set A together with a relation $\sqsubseteq$, which is reflexive, antisymmetric and transitive. We write $a \sqsubset b$, iff $a \sqsubseteq b$ and $a \neq b$.
A *lower* (*upper*) *bound* of a set $X \subseteq A$ is an element $a \in A$, such that $\forall_{a' \in X} : a \sqsubseteq a' (a' \sqsubseteq a)$.
It is called *greatest lower bound* or *glb* (*least upper bound* or *lub*), iff $\forall_{a' \in A}$: a' is lower (upper) bound of $X : a' \sqsubseteq a \, (a \sqsubseteq a')$.
*glb*s and *lub*s are unique, if they exist, so we write $\bigsqcap X, \bigsqcup X$ in this case.
A partial order $(A, \sqsubseteq)$ is called *complete lattice*, iff $\forall_{X \subseteq A} : \bigsqcap X, \bigsqcup X$ exist.
Sometimes we write a complete lattice $D = (A, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ as a six tuple, where $\bot = \bigsqcap A$, $\top = \bigsqcup A$ are called *bottom* and *top* elements of D. We write $x \sqcup y$ ($x \sqcap y$) for $\bigsqcup\{x, y\}$ ($\bigsqcap\{x, y\}$).
A function $f : D \to D$ is called *monotone*, iff $\forall_{d,d' \in D} : d \sqsubseteq d' \Rightarrow f(d) \sqsubseteq f(d')$.
f is called *distributive*, iff $\forall_{X \subseteq D} : f(\bigsqcup\{x | x \in X\}) = \bigsqcup\{f(x) | x \in X\}$.
A *chain* x_i is a sequence $x_0, x_1, \ldots$, such that $\forall_j : x_j \sqsubseteq x_{j+1}$. A chain x_i is called *strong ascending*, iff $\forall_j : x_j \sqsubset x_{j+1}$.
A *widening* is an operator $\nabla : D \times D \to D$, such that: $\forall_{x,y \in D} : x \sqsubseteq x \nabla y$, $\forall_{x,y \in D} : y \sqsubseteq x \nabla y$ and for every chain x_i the chain y_i, defined by $y_0 = x_0$, $y_{j+1} = y_j \nabla x_j$, must not be strong ascending.
Widenings can be used to improve termination behaviour of an iteration algorithm at the cost of less precise information.
A *path* π is a path in the CFG, beginning at the start node s and ending at some node n ($\pi = s.n_1. \cdots .n$).
The *path semantics* $[\![\pi]\!]$ of some path π is the combination of the transfer functions $tf : N \to D \to D$ along the path:

$$\begin{aligned} [\![s.\epsilon]\!] &= tf(s) \\ [\![p.n]\!] &= tf(n) \circ [\![p]\!] \end{aligned}$$

The semantics of a path π represents the semantics of a program execution along that path. The solution of the DFP is the union[1] of all path semantics, applied to the bottom element (Meet Over all Paths solution):

$$MOP(n) = \bigsqcup\{[\![\pi]\!](\bot) | \pi \text{ is path from } s \text{ to } n\}$$

I.e. all possible executions of a program, reaching node n are mixed together with $\bigsqcup$. The mixture of information ensures that the information is valid for all possible program executions, hence static.
Unfortunately, the MOP solution is not computable in general, so one uses the computable 'Minimal FixPoint' solution MFP:

$$MFP(n) = \begin{cases} tf(s)(\bot) & \text{, if } n = s \\ tf(n)(\bigsqcup\{MFP(n') | n' \text{ predecessor of } n\}) & \text{, otherwise} \end{cases}$$

[1] w.r.t. the complete lattice D.

Here the solutions of all predecessors of a node are mixed together. To ensure the computability of the MFP solution, all transform functions $tf(n)$ must be monotone, if they are even distributive, then $MFP(n) = MOP(n)\forall_n$[2]. If the monotonicity condition is met, it holds that

$$\forall_n : MOP(n) \sqsubseteq MFP(n),$$

i.e. $MFP(n)$ is a correct approximation of the MOP solution.

2.1.2 Computing the MFP solution

To actually compute the MFP solution, one uses iteration algorithms (since the MFP solution is recursively defined):
Every node is assigned an initial value of $\perp$. The $tf(s)(\perp)$ is computed (s is the start node of the CFG) and stored at s. For every successor n of s, $tf(\bigsqcup\{v(n')|n'$ predecessor of $n\})$ is computed and stored, where $v(n')$ is the value stored at node n'.
This is continued for all successors of n, until a stable state is reached, i.e. $v(n)$ will not change through further iterations for all n.
In this stable state $MFP(n) = v(n)\forall_n$.
Figure 2.1 presents the *worklist algorithm* to compute the MFP solution. Nodes, which have to be inspected are contained in a worklist.
The algorithm works by selecting nodes from the worklist, computing the lub of all incoming edges and applying the transfer function to it. Then the algorithm checks, whether the value $v(n)$ changed for a node n. If not, the value is not again propagated to the successors of the node, i.e. they are not entered into the worklist.
The algorithm leaves open the selection of the actual node from the worklist. There are several methods to select nodes from this list. For this reason, the list is ordered in special ways and a node is always selected from the head of the list. Section 2.3.3 lists all orderings defined in PAG,
Another iteration algorithm, *effect calculation* is introduced in section 2.3.2. A *widening* operator can be used to improve the termination behaviour of the worklist iteration. Here the line $v(n) := X$ is replaced by $v(n) := v(n)\nabla X$ in the algorithm, where ∇ is the widening operator.

2.2 Intra- vs. Interprocedural Analysis

Until now, we have looked only at programs with one procedure (the main procedure). But a typical program consists of many subroutines, which are

[2] And the MOP solution is computable in this restrictive case.

Input: A CFG $G = (N, E, s, e)$, a complete lattice D, a semantic functional $[\![\ ]\!] : N \to (D \to D)$.
Output: The annotation $v : N \to D$.

```
(* Init *)
FORALL n ∈ N DO
    v(n) := ⊥;
OD;
workset := {s};

(* Iteration *)
WHILE workset ≠ ∅ DO
    LET n ∈ workset (* select node from workset *)
        BEGIN
         workset := workset \ {n};
         X := [[n]](⊔{v(m)|(m,n) ∈ E});
         IF X ≠ v(n) THEN
           v(n) := X;
           workset := workset ∪ {m|(n,m) ∈ E};
         FI
        END
OD.
```

Figure 2.1: Work List Iteration

grouped into several procedures. The usage of several procedures, which may be called from different contexts and with different arguments, increases the complexity of program analysis. There are several approaches to incorporate procedures into DFA:

Inlining: Every call to a procedure is replaced by the body of that procedure. This is only feasible for non recursive procedures and the CFG grows exponentially in terms of the nesting depth.

Invalidating: This approach invalidates *all* CF data at every call to a procedure. This strategy is easy to implement[3] but very inprecise. It can be improved by invalidating only that information, which may be changed by the procedure call. This requires an extra alias analysis prior to the DFA.

Effect Calculation: For every procedure a function (effect) is calcu-

[3] The GNU C Compiler uses this strategy.

lated, which maps input values of the procedure to output values. These functions are then used to perform the analysis. See section 2.3.2 for the definition of the algorithms used herein.

Call String: The 'history' of a procedure call is incorporated into the underlying lattice D, allowing to analyse different call sequences separately, see [SP81].

Static Call Graph: This strategy looks at some statically defined call sequences and performs data analysis for each sequence separately. Sequences exceeding a certain length are mixed together and therefore loose precision. See section 2.3.1 for further details.

PAG implements the effect calculation and static call graph approaches.

2.3 PAG's approaches

To formalise the interprocedural analysis, the notion of a super graph and mappings are introduced below. The strategies implemented in PAG are summarized in section 2.3.2.

2.3.1 Basic definitions

The CFG of a program is constructed from the CFG of the single procedures and results in a super graph: Let $P_0, \dots, P_n$ be the procedures of the program, with the main procedure P_0. The *super graph* of the program P, $G^* = (N^*, E^*, s^*, e^*)$ is defined as follows. $G_0, \dots, G_n$ are the CFG of the procedures $P_0, \dots P_n$, modified such that every procedure call is replaced by 3 nodes.

1. A *call* node `Call` p_j,
2. a *local* node `Local` and
3. a *return* node `Return`.

See figure 2.2 for an example.
The nodes and edges of G^* are the union of the nodes and edges of the G_i, in addition, for every procedure call to a procedure P_j from procedure P_i, E^* contains the following edges:

- A call edge from `Call` p_j to the start node of P_j

```
main()
{
  f(); }

f()
{ i=1;
}
```

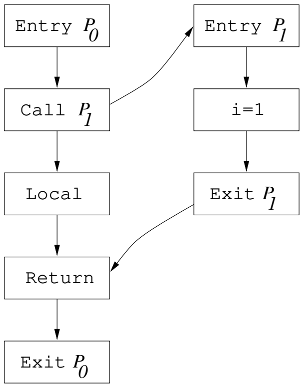


Figure 2.2: Super Graph

- an edge from `Call` p_j to `Local`
- a return edge from e_j to `Return` and
- an edge from `Local` to `Return`.

The super graph inherits the start and end edges from G_0; $s^* = s_0, e^* = e_0$. The local node and edge are introduced to allow for the propagation of information which can not be changed by the procedure call, e.g. values of local variables. See figure 2.3 for an example situation. Here, the variable x is local to P_1, whereas y is shared. The local node sets $y = \perp$, the `Exit` P_2 node sets $x = \perp$, so the value for x is preserved. This behaviour has to be encoded into the transfer functions at the local and exit nodes.
With the introduction of the super graph, the problem of interprocedural analysis can be reduced to an intraprocedural analysis problem on the super graph. The complexity of the analysis can further be reduced by observing, that not all paths in the super graph are legal call sequences, i.e. there are paths in the graph, that cannot refer to a valid program execution: consider figure 2.3 again. There are edges from the `Exit` P_2 node to the nodes 2 and 4. But since there is no call path 1.5.6.7.4 (the procedure call of P_2 has to return to node 2), we must not propagate information along the path 1.5.6.7.4 to node 4.
To keep different call paths separate, we annotate each node in the super graph with several data elements from D. For every procedure P_i, the number of data elements at nodes of this procedure is fixed and given by $arity(P_i)$, moreover $arity(P_0) = 1$, i.e. there is only one data element in the main procedure. We can extend $arity$ to a function $arity : N^* \rightarrow \mathbb{N}$ by $arity(n) := arity(P_i)$, iff n is a node in P_i.

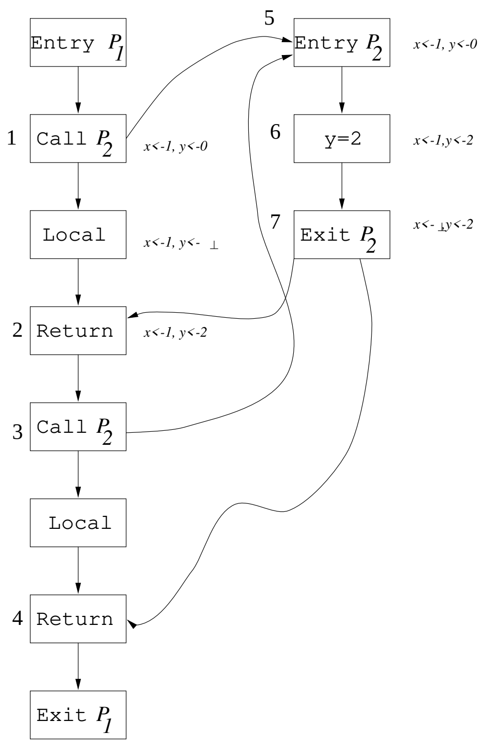


Figure 2.3: Local Node, valid paths

To define the static call graphs, a mapping $map_{\texttt{c}} : \{1, \ldots, arity(P_i)\} \rightarrow \{1, \ldots, arity(P_j)\}$ is required, which maps for every procedure call `c=Call` P_j in P_i the data elements of the call site to elements of the called procedure. Obviously, we can define another graph, the *extended super graph* with one data value per node, by building nodes (n, i), where $n \in N^*, i \in \{1, \ldots, arity(n)\}$:

$G_E^* = (N_E^*, E_E^*, s_E^*, e_E^*)$, with

- $N_E^* = \{(n, i) | n \in N^*, i \in \{1, \ldots, arity(n)\}\}$
- $s_E^* = (s^*, 1)$
- $e_E^* = (e^*, 1)$
- $((n_1, i_1), (n_2, i_2)) \in E_E^*$, iff $(n_1, n_2) \in E^*$ and one of the following conditions hold:
  - i.) $n_2 =$ `Entry` P_j and $i_2 = map_{\texttt{c}}(i_1)$
  - ii.) $n_1 =$ `Exit` P_j and $i_1 = map_{\texttt{c}}(i_2)$, where c=`Call` P_i is the call node corresponding to the return node n_2.

iii.) $i_1 = i_2$ and $n_1 \neq$ `Exit` p and $n_2 \neq$ `Entry` p

PAG uses this extended super graph to perform it's analysis. Still, the mappings map_{C} and arities $arity(P_j)$ can be adjusted to certain cases.

a.) $\forall_n : arity(n) = 1, \forall_{\mathrm{C}} : map_{\mathrm{C}} = id$. Only one data flow value is used per super graph node. No separation of different call paths is made: information is mixed together at call sides. This mapping is called **single**.

b.) Set $arity(P_i) = k_i$, where k_i is the number of incoming edges into `Entry` P_i, i.e. the number of static call sides. In addition set $arity(P_0) = 1$. for all c=`Call` P_i in P_j set $\forall_{i \in \{1,\ldots,arity(P_j)\}} : map_{\mathrm{C}}(i) = x$, where $x \in \{1, \ldots, arity(P_i)\}$ is fixed but unique for every call site of P_i.

   This mapping assigns a unique data element to every call of a procedure P. If there is a possible call path with two or more occurences of the same call site, information is mixed together for that procedure (e.g. at node **A** in figure 2.4). This mapping is called **static**.

c.) The mapping is choosen in the way that call paths up to a given length J are kept separate and is called **length**.

2.3.2 Strategies used in PAG

At the moment there are 3 different iteration algorithms implemented in PAG:

1. Effect calculation, using the algorithm in figure 2.5. The alogrithm works in two stages. First, the effect functions are calculated and stored in tables. Then the information is computed for all nodes by applying the function to all possible data values and building the lub of them. However, this strategy can only be used, if the underlying lattice is finite.
2. Worklist iteration. This is the algorithm, described in figure 2.1, with some optimizations.
3. Worklist iteration with basic blocks. This algorithm groups connected nodes into basic blocks and propagates information in these blocks sequentially. A basic block is defined as a maximum sequence of connected nodes, which does not contain nodes with multiple outgoing

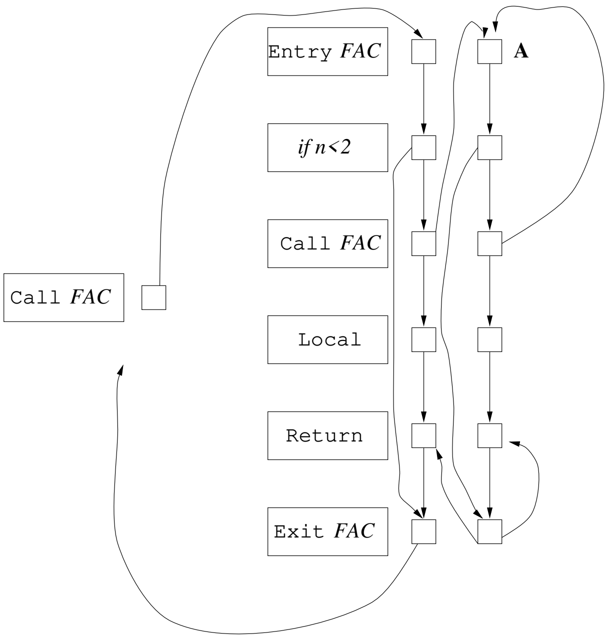


Figure 2.4: Mapping example

edges, except for the last node in the sequence. It may, however, contain nodes with multiple incoming edges, thus differing from the normal notion of a basic block.

All three algorithms can be combined with a widening operator.

2.3.3 Orderings supported by PAG

As already mentioned, the selction of a node from the worklist can be performed, according to special orderings of this worklist. PAG supports the following orderings:

- **dfs**: depth first order. Nodes are ordered in a *depth first* visiting strategy of the CFG.
- **bfs**: breadth first order. Nodes are ordered in a *breadth first* visting strategy of the CFG.
- **scc**$_d$: strongly connected components. This ordering first computes the scc (see e.g. [Meh84]) of the super graph[4]. They are ordered by a

[4]The *super graph* is defined in section 2.3.1.

topological ordering (see [Tod90]). The nodes of the scc are ordered depth first.

- $\mathbf{scc}_b$: strongly connected components. This ordering is similar to $\mathbf{scc}_d$, but orders the nodes of the scc breadth first.

- $\mathbf{hyp}_d$: This ordering computes the scc of the super graph, but orders them by the ATS ordering (defined in figure 2.7), related to the week topological orderings of [Bou93]. The nodes of the scc are ordered depth first.

- $\mathbf{hyp}_b$: This ordering differs from $\mathbf{hyp}_d$ only in sorting the nodes of the scc's breadth first.

Input: A collection of CFG $G^* = \bigcup_{P_i} G_{P_i}$ for all procedures P_i of a program. A complete lattice D, a semantic functional $[\![\ ]\!] : N^* \to D \to D$.

Output: The function $MFP : N \to D$.

Note: The effect functions Φ are represented by a two dimensional, partialy defined array $PHI : N^* \times D \to D$, such that $\forall_{n \in N^*, x \in D} : PHI(n, x) = \Phi_{\texttt{Entry}\ P}(x)$, where P is the procedure, containing n. The variable $workset$ contains pairs (n, x) from $N^* \times D$, for which $PHI(n, x)$ has changed and whose new values have not been propagated to the successors of n.

```
PROCEDURE init
  FORALL n ∈ N*, x ∈ D DO
   PHI(n, x) := ⊥;
  OD;
  workset := {(Entry main, ⊥)};
END

PROCEDURE propagate(x ∈ D, y ∈ D, n ∈ N*)
 z := PHI(n, x) ⊔ y;
 IF z ≠ PHI(n, x) THEN
   PHI(n, x) := z;
   workset := workset ∪ {(n, x)};
 FI
END

PROCEDURE post
 FORALL n ∈ N* DO
   MFP(n) := [[n]](⨆{PHI(n, x)|x ∈ D});
 OD
END
```

Figure 2.5: Effect calculation algorithm

```
PROCEDURE effects
 WHILE workset ≠ ∅ DO
  LET
   (n,x) ∈ workset;
   y = PHI(n,x);
   IN
    workset := workset \ {(n,x)};
    IF n ≡ CALL P in q THEN
     z := PHI(EXIT P,y);
     IF z ≠ ⊥ THEN (* Pair (n,y) not processed *)
      m := Return node belonging to n;
      propagate(x,z,m);
     ELSE
      propagate(y,y,ENTRY P);
     FI
    ELIF n ≡ Exit P THEN
     FORALL c = Call P DO
      m :=Return node belonging to c;
      FORALL u ∈ D WHERE PHI(c,u) = x DO
       propagate(u,y,m);
      OD
     OD
    ELSE
     FORALL m successor of n DO
      propagate(x,⟦m⟧(y),m);
     OD
    FI
   END
 OD
END

(* main *)
 init();
 effects();
 post();
END.
```

Figure 2.6: Effect calculation algorithm, cntd.

Order the sccs of a CFG.
The root R_e of an scc e is the node which is reached first (using an ordering on edges) by a depth first strategy.
Input: sccs of a flow graph $\{s_1, \ldots, s_m\}$
Output: total ordering ord of sccs.

```
scc := {s_1, ..., s_m};  count := 1;
∀_{i∈{1,...,m}} : ord(s_i) := 0;
WHILE scc ≠ ∅ DO
   IF ∃s ∈ scc with no predecessor  THEN
     sel := s; (* select minimal element *)
   ELSE
     select {y_1, ..., y_n} ⊆ scc with minimal
           po(R_{y_i}) and on_cycle(R_{y_i}) = true
     IF ∃i with y_i is procedure end node THEN
       sel := y_i;
     ELSE
       sel := y_1;
     FI
     ord(sel) := count++;
   FI
   scc := scc \ {sel}
OD
```

po (predecessors ordered) is defined as

$$po(x) = \|\{z | ord(R_z) \neq 0 \wedge \exists \text{edge } z \rightarrow x', R_{x'} = R_x\}\|$$

And $on\_cycle(x) = true$, iff x belongs to a cycle in the CFG.
Sccs are computed *intraprocedurally.*

Figure 2.7: ATS Algorithm

Chapter 3

Specifying the Inputs

To generate an analyzer, PAG needs the following inputs:

- A `.set` file, containing definitions of sets and lattices.
- A `.syn` file, containing the definition of the abstract syntax.
- A `.optla` file, containing the problem specification, transfer functions and support routines.

For the compilation of the analyzer, additional include and source files are needed, which are explained in chapter 4.
The iteration algorithm, which is used in the analyzer, is selected by a command line argument to the generator `pag`:

`pag` *front-end iterator* `.t` *specification target-directory*

Where *iterator* is one of

iterate2 This selects worklist iteration.

iterate3 This selects worklist iteration with basic blocks.

iternew This selects effect calculation.

The generated files are placed into a directory named `src` which is created in the directory given as *target-directory*.
front-end selects the front-end to use. The location of several include and source files is determined by the name given here, see chapter 4 for details.

The ordering, the iteration algorithm uses, is dynamically selected at run time by the contents of the global variable `cfg_ordering`.

`cfg_ordering`=1: **dfs**
2: **bfs**
3: $\mathbf{scc_d}$
4: $\mathbf{scc_b}$
5: $\mathbf{hyp_d}$
6: $\mathbf{hyp_b}$

The mapping of the work-list algorithms is also dynamically selected by the variable `sel_mapping`=1: This is the mapping **single** on page 14
2: Corresponds to mapping **static** on page 14.
3: This selects mapping **length** on page 14. The variable `J` holds the maximum length of call paths to be kept separate.

3.1 `.set` File

The `.set` file defines sets and lattices with the help of the DATLA language. Several primitive sets are predefined, like:

`unum` : The set of unsigned integers.

`snum` : The set of signed integers.

`real` : The set of real numbers[1].

`bool` : The set `{true, false}`.

`chr` : The set of characters.

`str` : The set of strings.

DATLA provides several operators to form new sets/lattices. See figure 3.1 for an example of a `.set` file.

3.1.1 DATLA: Sets, Lattices and Globals

Globals are declared in the optional `GLOBAL` section. In the `GLOBAL` section, declarations have the form *Id : expr*, where *Id* is an identifier and *epxr* can be

[1]As approximated by floating point numbers.

```
SET
  nums=list(unum)
DOMAIN
  varset=set(unum)
  vars=lift(varset)
```

Figure 3.1: An example `.set` file

- one of the given primitive types (e.g. `snum`). Then *Id* is declared as a global identifier, which can be used in following sections. The value of a global identifier is evaluated at runtime.

- `optlafunc`. Now *Id* is declared as a FULA function, which can be used in reduced product construction.

Sets are defined in the optional `SET` section, which is followed by lattice definitions in the `DOMAIN` section. See chapter 8 for a detailed definition of the grammar of `.set` files.
In the `SET` and `DOMAIN` sections, definitions have the form *Id*=*expr*, where *Id* is an identifier. The `SET` section provides the following expressions:

- Id_A`->`Id_B: Set of functions from A to B.

- *Id*`++`*Id*$\cdots$: Disjoint set union of two or more sets.

- *Id*$*$*Id*$\cdots$: Cross product.

- `list(`*Id*`)`: Set of lists over set *Id*.

- `set(`*Id*`)`: Power set of set *Id*.

- `[`*exp*`..`*exp*`]`: Section, where *exp* is either a number, a global *Id* or an expression with at most one global *Id*. An expression is constructed with the natural numeric operators '+', '*', '-', '/' and '%'.

- Enumerations. See below for an explanation.

The `DOMAIN` section also provides the `->`, $*$ and `[`$\cdots$`]` operators, but takes lattices instead of sets as parameters. In addition, the following operators are provided:

- `set(`*Id*`)`: Power set of set *Id*, ordered by $\subseteq$, with bottom element $\emptyset$ and top element *Id*.

- `flat(`*Id*`)`: Constructs a lattice from set *Id* by adding a top and a bottom element. See chapter 8 for details.
- `lift(`*Id*`)`: Lifting of lattice *Id*.
- `revord(`*definition*`)`: Reverse order construction of lattice *definition*, where *definition* is the right hand side of a domain definition. In other terms, the order of lattice *definition* is reversed.
- `redprod(`*definition*`,`*Id*`)`: Construction of the reduced product of lattice *definition*, where *Id* is a global FULA function.

  Conceptually *Id* is applied to all elements of lattice *definition*, so that elements of *definition*, which are in the argument range of *Id*, never occur.

Enumerations may occur in both sections. They take the form

`(`*Id*`,`...`,`*Id* | *relation* | *top* | *bottom*`)`

Here the elements of the set (lattice) are enumerated, followed by the definition of a relation and the top and bottom element of that relation. The relation definition is not used for sets, although it may be given. Top and bottom elements must always be specified, though they are again not used for sets. An *edges* definition takes the form *Id*<*Id*,...,*Id*<*Id*.
Please note that lattices may always be used as sets, e.g. with `flat` operators. Their ordering is simply discarded, then.
With these definitions, figure 3.1 becomes clear: First a set `nums` is defined as the set of lists over unsigned integers. Then a lattice `varset` is defined as the power set of unsigned integers, ordered by set inclusion. `vars` is then defined as the lifted lattice to `varset`.
Finally, `varset`$= (A, \sqsubseteq)$, where A contains the elements

$$A = \{\bot, \top, \{\}, \{1\}, \ldots\}$$

and $\sqsubseteq$ is indicated in figure 3.2.
Note, that the definitions in the `.set` file can be recursive, i.e. set/lattice names can be used in definitions, before they are themself defined.

3.2 .syn File

The `.syn` file defines the abstract syntax with the help of an extended *tree grammar*.

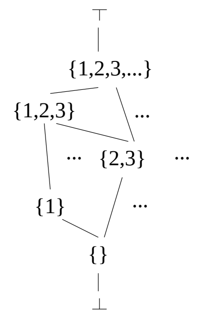


Figure 3.2: Ordering of figure 3.1

3.2.1 Syntaxtree Definition

Figure 3.3 shows an excerpt from a `.syn` file.

```
SYNTAX
START: mirStmt

mirStmt:  CFGCall(exp:  mirExpr)
       | CFGEndCall(exp:  mirExpr, sym:  mirSymbol*)
        ⋮
       ;

mirExpr:  ⋮
       | mirChar(str:  CHAR, type:  mirType)
        ⋮
       ;

CHAR  ==  chr;
INT   ==  snum;
   ⋮
```

Figure 3.3: An example `.syn` file

`START` gives the start symbol (or type) of the tree grammar. Every node in the CFG must be associated with an abstract syntax tree or term) of this type.
The productions of the grammar are grouped by type. E.g. `mirStmt` consists of an constructor `CFGCall`, who takes one argument, `exp` of type `mirExpr`.

`CFGEndCall` is another constructor of type `mirStmt`, with two arguments. The `*` at the second argument indicates that `sym` should be a *list* of terms from `mirSymbol`. PAG provides this shortcut, since lists are often needed.
`mirExpr` is another type in the grammar. It contains a constructor `mirChar`, who has an argument `str` of type `CHAR`. The type `CHAR` is special, since it does not correspond to a type in the grammar with constructors, but rather corresponds to the build in primitive set `chr` (see section 3.1).
All types that correspond to such primitive types (sets) must be declared after the constructors of the whole grammar. This is done by lines of the form *Name*`==`*primitive set*, as in figure 3.3. To be able to convert from the internal representation of these types to PAG's representation of it's primitive sets, **C** functions of the form

```
char *CHAR_get_value(CHAR i)
```

have to be specified as part of the interface. See chapters 4 and 11 for details.

3.3 `.optla` File

The `.optla` file specifies the problem, the transfer functions and support functions. The transfer and support functions are defined in a first order functional language: FULA.
Figure 3.4 shows a sample `.optla` file.

3.3.1 Problem Specification

The problem specification consists of:

- Declaration of global data
- Declaration of local data
- Declaration of problem parameters

Only the problem parameters must actually be present. The global data declares global parameters[2], their names and types. Local data is associated with every node of the CFG and declared in the same way as global data.
Global data may be referenced in transfer and support functions, like `result` in figure 3.4. Local data may only be referenced in transfer functions, like `locals` in figure 3.4.
The problem parameters are made up of (cf. figure 3.4):

- The name of the problem, given by the `PROBLEM` *name* command.

[2]Which map to global variables in the generated analyzer.

```
GLOBAL
result:  unum

NODE
locals:  *mirSymbol
defines:  *mirSymbol

PROBLEM live_var
 direction:  backward
 carrier:    vars
 init:       bot
 init_start: lift({})
 combine:    comb
 widening:   www
 equal:      eq

TRANSFER
 CFGEnter(_,_,_),_:  bot;
 CFGExit(_,_,_,_),_:  lift({result});
 CFGEndCall(_,_),N: lift(bot);
 _,_:  let loc= sym2set(locals) lub sym2set(params); in
        if lift(loc) <= @ then
          lift(loc) glb add(@,uses)
        else @
        endif;

SUPPORT
eq(a,b) = a=b;
www(a,b) = b;
comb(a,b)= a lub b;

sym2set([!])  ={};
sym2set(Obj(_,id,_,_)::ar) = sym2set(ar) ~ unum(val-INT(id));
    ⋮
```

Figure 3.4: Example `.optla` file

- The `direction` of the problem, this may be either `forward` or `backward`. Some analyzes require that the propagation of information is made *against* the direction of the control flow in the program[3]. This is simply achieved by inverting every edge in the CFG. But since there may be several analyzes with differing directions, this can not be handled by the front-end alone. PAG allows the definition of direction and does propagation of information properly according to the direction.

- The `carrier` command defines the underlying lattice of the analysis. This must be defined in the `.set` file.

- The initial value at each node of the CFG is given with the `init` command. It is followed by a FULA expression, denoting the initial value.

- The initial value of all nodes that have no incoming edges[4] is given by the `init_start` command.

- The `combine` command gives the name of a FULA function, which is used to combine information at nodes with multiple incoming edges. Rather than always using $\bigsqcup$ for the mixing of information, PAG allows a combination functions that mixes two values together and thus has type $D \times D \to D$. Mixing of three or more incoming values is done iteratively by folding the combination operator into the values: $comb(\cdots comb(comb(d_1, d_2), d_3) \cdots)$.

  So the function should be commutative and associative. Moreover, the init value given with the `init` command should be the neutral element of the combination function.

- A widening may be given with the `widening` command, which takes the name of a function of type $D \times D \to D$, where the first argument denotes the value of the last iteration at that node and the second value is the new value for that node. If no widening should be used, a default function of the form `www(a,b)=b` has to be used as argument to the `widening` command.

- The iteration algorithms stops propagation of values, as soon as the value of a node has not changed since the last visit to it (see line $X \neq v(n)$ in figure 2.1 and line $z \neq PHI(n, x)$ in figure 2.6). PAG allows to use another equivalence relation apart from the predefined syntactical equivalence. This is done by giving the name of a function

[3]E.g. liveness analysis.

[4]Especially the start node

with type $D \times D \to$ `bool` as argument to the `equal` command. Again, the standard equality has to be given to the `equal` command as a function of the form `eq(a,b)= a=b`.

3.3.2 FULA: Transfer- and Supportfunctions

In section 2.3.1 a `Local` node was introduced into the CFG to allow for the modelling of local behaviour across procedure calls. PAG uses an convenient abbreviation, by not introducing an extra (`Local`) node, but rather *labeling* the edges and having different transfer functions for different edges.
The transfer functions are then defined by pattern matching on node labels[5] and edge labels, e.g. `CFGEndCall(_,_),N: lift(bot)` in figure 3.4. Here, at every node that is labeled with a syntax term of the form `CFGEndCall(`x`,`y`)` and every outgoing edge, labeled with `N`, the transfer function computes `lift(bot)` as value.
Edges may only be labeled with names, which are defined in a file called `edges` in the frontend directory. See chapter 4 for details of the format of this file.
Note that the labeling of edges is not restricted to procedure call `Local` nodes. Every edge in the CFG must be labeled and thus makes possible, e.g. the destinction of the *true* and *false* branches of an `if` statement.
Naturally, the same behaviour can be reached by introducing new nodes into the CFG and using appropriate transfer functions on nodes alone, as we did with the `Local` node. So the edge labeling is just another convenient abbreviation, offered by PAG.
As already mentioned, transfer functions are annotated in a functional style by using pattern matching on node and edge labels. These patterns are matches from left to right and top to down.
The annotation language FULA is an eager evaluated, first order functional language with a monomorphic strong type system. All sets (and thus lattices), defined in the `.set` file of an analysis, are known in the `.optla` file as types to FULA, too. Naturally, apart from the transfer functions, support functions can be defined in the `SUPPORT` section.
FULA offers most of the constructs, found in nowadays functional languages, together with some extensions for DFA.
FULA borrows very much from the functional language **ML**. Thus, a look at [Ull94] helps the understanding of FULA, too. See [MTH90, MT90] for a real deep look at the semantics of **ML**.

[5] I.e. abstract syntax terms, as defined in the `.syn` file.

3.3.2.1 Data types in FULA

FULA supports the following data types:

- primitive: `unum`, `snum`, `real`, `bool`, `chr`, `str` (see section 3.1). Characters are annotated as `'a'`, strings with double quotes: `"A string"`. `true` and `false` denote the boolean values. Numbers are entered in the usual notation.

- sets/lattices: All sets/lattices defined in the `.set` file are also known in FULA. Sets can be enumerated: `{`e_1`,`$\ldots$`,`e_n`}`,$n \geq 0$. The top and bottom elements of a lattice can be written as `top` and `bot`, resp.

- syntactical terms: The syntax, defined in the `.syn` file is known as syntax terms in FULA patterns. As a special case, syntactical lists[6] are denoted by `[!]` for the empty list or h`::`tl for a list with head h and tail tl. There is no notation (and no need) for syntactical lists outside of patterns.

- tupels are written `(`e_1`,`$\ldots$`,`e_n`)`,$n \geq 2$.

- lists over any type. Lists can be enumerated: `[`e_1`,`$\ldots$`,`e_n`]`,$n \geq 0$ or constructed with a cons operator: e_1`:`e_2`:`$\cdots$`:[]`.

- functions. FULA knows two kinds of functions: on the one hand, the functions that are elements of function sets, defined in the `.set` file[7]. And on the other hand, functions defined in the `TRANSFER` and `SUPPORT` sections of the `.optla` file. The first kind of functions, *dynamic functions*, can be defined and used in `FULA` expressions. The dynamic function that maps all its arguments to a default value, denoted by the expression v, is annotated in FULA by `[->` v`]`. Updating of a dynamic function f is written f`\[`n_1`->`v_1`,`$\ldots$`,`n_k`->`v_k`]`,$k \geq 1$, where (n_i, v_i) are argument-value pairs.

  Dynamic functions and *static functions*, i.e. functions defined by function definitions in the `TRANSFER` and `SUPPORT` sections of the `.optla` file are not equivalent. Dynamic functions, e.g. can be used in ZF expressions as generators of new functions, whereas static functions can only be called: `sym2set(locals)`[8]. Application of dynamic functions to values is annotated differently: fe`{!`exp`!}`, where fe is an FULA

[6] Defined e.g. as `mirSymbol*` in figure 3.3.

[7] E.g. `funcs = unum -> snum`

[8] The only exception is the `crunch` predefined function, which takes the name of a static function as an argument.

expression, evaluating to a dynamic function and *exp* is the argument expression to the function.

FULA allows the type specification of (static) function definitions and expressions. To specify the type t of an expression e, we write t `$` e. The type of a function can be given with a *function declaration* of that function, e.g.

```
eq::  vars,vars -> bool
```

FULA types are constructed of type names for primitive types and types declared in the `.set` file and the syntactic list type, annotated `*`*styp*, e.g. `*mirSymbol`.

3.3.2.2 Pattern

FULA defines functions by pattern matching. Patterns may also be used in case expressions and local definitions within ZF expressions.
A *pattern* consists of either a simple pattern, a constructor pattern, a list pattern, a tupel pattern, a type specification or an alias pattern:

- A *simple pattern* is either a *variable*, with is bound to the corresponding expression at that place during matching, an universal match (_), matching every expression or a constant, that is: a number, a boolean value, a char, a string, the top/bottom element of a lattice (`top`, `bot`), the empty list `[]` or the empty set `{}`.

- A *constructor pattern* is made up either by the name of the syntactic constructor from the `.syn` file and it's argument patterns, e.g. `Obj(_,id,_,_)` in figure 3.4, or by the name of an element of an enumerated set from the `.set` file.

- A *list pattern* is either a cons list p_1`:`p_2, matching lists with heads that match p_1 and tails that match p_2, or a *syntactical list pattern.* These can be either `[!]` for the empty syntactial list or p_1`::`p_2 for patterns, matching lists with a head and tail component.

- *Tupel pattern* are annotated `(`p_1`,`$\ldots$`,`p_n`)`, where the elements e_i of a tupel are matched against the pattern p_i.

- *Type specifications* are not used to match expressions, but rather to specify the type of a pattern, e.g. `snum $ 1`. Type specification is only needed, if the type inference algorithm of FULA can not determine the correct type of the pattern itself, or where the type should be restricted.

- *Alias pattern* give names to patterns, so that an expression matching that pattern can be bound to a variable and later be referenced as a whole, e.g. `(a,b) as t`. This pattern matches pairs with elements `a,b`. If the whole tupel is needed for later calculations, it can simply be referenced by `t`, instead of constructing a new tupel `(a,b)`.

3.3.2.3 Operators

FULA provides, along with a set of predefined functions, which are listed in section 10, the following operations on data elements:

- arithmetic: Addition, subtraction, multiplication, division, modulo, potentiation on numbers are annotated by the binary infix operators `+,-,*,/,%,^`, resp. Negation of numbers is annotated with the prefixed unary `-`. Bit shifts of a number n, b positions to the left or right are annotated by v`<<`b, v`>>`b. Bit wise AND and OR operations on integers are annotated by the `&` and `|` binary infix operators.

- logic: Comparison of data elemnts can be done by the infix relations `<=, >=, <, >, =, !=`, where `!=` stands for 'not equal'. The other symbols retain their usual meaning. Note that these operations are supported for every data type, except that `chr` and `str` elements may only be tested for equality (or non equality). The same restriction applies to elements of non-lattices, defined in the `.set` file, since no `<,<=,>,>=` relations are defined for them.
  Boolean values b can be negated by `!`b, values a,b can be combined by the binary infix operators `&&` and `||`. Here `&&` denotes logical AND and `||` denotes logical OR. The operators are lazy, i.e. arguments are not evaluated, if they can't change the value of the whole expression, as in `true ||` e, `false &&` e.

- Concatenation of strings s_1, s_2 is written s_1`.`s_2.

- sets: The elements of an enumerated set in the `.set` file are known as constants to FULA. Addition/subtraction of an element e from a set s is annotated by s`~`e, s`#`e, resp. To test for the presence of an element in a set, we write e`?`s.
  For sets that are declared as the disjoint union of sets, e.g. `foo = snum++real++chr`, FULA offers constructs to extract and incorporate elements from/to these sets. The operator `up-`*id*`-`*i*`(`*exp*`)` incorporates the value of *exp* into the i-th element of the set *id*, e.g. `up-foo-2(1.0)` converts the real number into an element of the set `foo`. Note that

i must be an integer, not an expression. Note that the components of a disjoint set are numbered from 1 on from left to right (so `real=2` in the above example). To convert an element of a disjoint set to an element of a member set, the `down-`id`-`i`(`exp`)` construct exists, e.g. `down-foo-2(up-foo-2(1.0))=1.0`. To test, if an element of a disjoint set belongs to a special member set, the `is-`id`-`i`(`exp`)` construct exists. So, e.g. `is-foo-2(up-foo-2(1.0))=true`, but `is-foo-2(up-foo-3('a'))=false`.

- In addition to these operations, lattices support the building of least upper and greatest lower bounds ($\sqcup, \sqcap$) by providing the binary infix operators `lub` and `glb`. The incoming data flow value of a node in the CFG can further be annotated by `@` in transfer functions.

- List concatenation of (non syntactical) lists l_1, l_2 is written l_1`++`l_2.

- Elements of tupels can be selected with the tupel select construct. It is written t`!`i`!`n, where t is a tupel, i and n are numbers[9]. This selects the i-th element of the n-tupel t, i.e. `(`e_1`,...,`e_n`)!`i`!`$n \equiv e_i$.

- function application. The application of a static function named `f` to arguments a_i is written `f(`a_1`,...,`a_n`)`. Application of a dynamic function, denoted by expression fe to *an* argument a is written fe`{!`a`!}`. Note that dynamic functions always have axactly one argument, although this argument may be a tupel, naturally.

- ZF expressions[10]. FULA knows three kinds of ZF expressions for lists, sets and functions.

  - Lists: `[`e`!!`zf_1`,...,`zf_n`]`, $n \geq 1$
  - Sets: `{`e`!!`zf_1`,...,`zf_n`}`, $n \geq 1$
  - Functions: `[`e_1`->`e_2`!!`zf_1`,...,`zf_n`]`, $n \geq 1$

  These ZF expressions enumerate list, sets and functions by iterating *generators* in zf_i and thus binding variables in the expression e. For example, `[i+1 !! i<-[1,2,3,4], if i<3]` gives the list `[2,3]`. `i<-[1,2,3,4]` is a generator. It succesively binds `i` to the values `1,2,3,4`, the expression `i+1` is evaluated for every binding. Besides generators, *filter* and *local definitions* can be present in the zf_is. An

[9] This must not be arbitrary expressions, but rather plain integer numbers, e.g. `('a','b')!1+1!0+2` is incorrect.

[10] Sometimes called *list comprehensions*

example of a filter is `if i<3` above. A local definition *pat*=*exp* introduces new names, which are known in zf_js further to the right, as are the names introduced by generators. The left hand sides of generators and local definitions may be patterns, so that e is only evaluated for generated values, that match the pattern at the left hand side of the generator.

ZFs for sets and functions follow the same rules. There are three kinds of generators. Beside the list generator, that takes a list as right hand argument and enumerates the elements of that list, there is also a set generator. It takes a set as right hand argument and enumerates the elements of that set. It takes the form *pat* `<--` *exp*. A function generator *pat* `<-` *exp**exp* takes as first right hand argument an expression that evaluates to a dynamic function. The second argument, after the \ is an exclusion argument. This generator enumerates all argument, value pairs of the dynamic function, excluding those that have a value equal to the second argument.

E.g.
`{i+1!!i<--{1,2,3,4,4}}` = `{2,3,4,5}`
`{i+j!!i<-[5,6],j<--{1,2}}` = `{6,7,8}`
`[(x,y)->6!!x<-[2,3], y=x-1]`$\approx \{(2,1),(3,2),(4,6),(5,6),\ldots\}$

3.3.2.4 Control constructs

FULA offers two control constructs. The conditional evaluation of the `if` ... `then` ... `else` ... `endif` construct and the multi case pattern matching alternative of the `case` construct.
An expression of the form `if` e_1 `then` e_2 `else` e_3 `endif` evaluates to e_2, if e_1 evaluates to `true`, to e_3, otherwise. So the type of the `if` can be stated by `if::bool,`α`,`$\alpha \to \alpha$, where α stands for any type, but the three ocurences of it must denote the same type.
The `case` construct allows a multi pattern choice. An expression tupel is in turn matched against a sequence of pattern tupel. As soon as the first match succeeds for all expressions, the expression associated with that branch is evaluated and returned. If no alternative matches, a run time error occurs.
E.g. given

```
case 1,[4,5] of x,[] => x;
                x,y:ys => x+y;
endcase
```

this expression evaluates to `5`.
Naturally, the types of all expression at right hand sides of alternatives (e.g. `x+y`) must be the same. The same goes for the types of all pattern sequences

at the left hand sides. These, in addition, must be the same as the types of the expression sequence after the `case`.

3.3.2.5 Local definitions

Local definitions are introduced by the `let` construct:

$$\texttt{let } id_1\texttt{=}exp_1\texttt{;}\ldots\texttt{;}id_n\texttt{=}exp_n \texttt{ in } exp$$

This introduces new names $id_1, \ldots, id_n$, which are bound to the values of the expressions exp_i. Then the expression exp is evaluated and its value is the value of the whole construct. Note that the expressions exp_i are evaluated *before* exp is evaluated, corresponding to the eager evaluation strategy of FULA.
As a special case, FULA offers the *strict binding* of local names, by using id`<=`exp' instead of id`=`exp'. This is possible, if exp' is of a lattice type. If it evaluates to the bottom (top) element of the lattice, *the whole let construct* evaluates to the bottom (top) element of the lattice. This implies, that the exp in the body has a lattice type, too. Order of bindings is significant, so, e.g. `let x<=`$\top$`; y<=`$\bot$` in` ... evaluates to $\top$, since the binding for `x` is evaluated first. If the bindings of `<=` do not evaluate to the bottom or top element, then the identifier is bound to the value of `drop(`exp'`)`[11], i.e. the lattice must be of the form `lift(...)`.

3.3.2.6 Other constructs

The `print(`e_1`)`e_2 construct prints the value of expression e_1 and then evaluates and returns the value of expression e_2.
As already mentioned, the type of an expression can be specified by the *type* `$` *exp* construct.
FULA can perform type casts to/from the primitve types with expressions like *btype*`(`*exp*`)`, e.g. `unum('A')` = `65`. There are some restrictions on the combination of *btype* and the type of *exp*, which are sampled in the following table:

[11] `drop` converts a value from a lifted lattice to one of the underlying lattice, see chapter 10.

btype	*exp* type
unum	snum
	unum
	chr
	intervall
snum	snum
	unum
	chr
	intervall
real	snum
	unum
str	α
intervall	snum

Here, 'intervall' stands for an intervall type, e.g. if the `.set` file contains a line `foo = [1..10]`, then `foo(2)` performs a type cast to the intervall type (from `snum`). In the line for `str`, α stands for an arbitrary type, which is not a syntactic type from the `.syn` file.
Function declarations of static functions can be given by, e.g.

```
f::snum,snum->snum
```

In the `.syn` file, some types were declared equivalent to primitve FULA types, e.g. `CHAR==chr` in figure 3.3. To access parts of syntax term, that have this type, an explicit cast has to be performed. So one has to write `val-CHAR(`e`)`, if e is a syntax term of type `CHAR` to convert it to the primitive type `chr`.

Chapter 4

Building the Interface

The interface between the abstract syntax and the analyzer has to be provided by the compiler builder. Anyhow, this interface has to be specified and programmed only once for every compiler.
This chapter gives an overview of the components that have to be provided for the interface. Chapter 11 contains a detailed collection of the needed objects.
The following section summarizes the components and function of the interface. The next three sections explain, what types, functions and files have to be provided.
At the moment, PAG requires some low-level programming from the compiler builder in order to establish an interface. To change this situation, research for a *frontend generator* is under way to simplify this process.

4.1 What is needed for the interface?

The interface provides access to the CFG and AS from the analyzer. It has to provide **C** types for the components of the CFG and abstract syntax tree, i.e. types for CFG nodes, edges, syntax types, etc.
Functions to obtain and access parts of the syntax tree, traversing the CFG, accessing local/global variables and casting alias types have to be provided, too.
Aside from this, **C** header files `interface.h` and `syntax_tree.h` must be presented, along with a file `edges` that defines the edge types for the CFG.
Information on these types, functions and files is given in the following three sections.
The abstract syntax tree types and functions implement the description of the syntax tree in the `.syn` file. The analyzer accesses the syntax tree by calling

C functions, whose names are derived from the names of the constructors in the `.syn` file.
The CFG, too, is manipulated by the analyzer by calling specially named **C** functions, which have to be provided by the interface. Also, the analyzer depends on certain **C** types, whose definitions have to be given by including one of the header files `interface.h` and `syntax_tree.h`. E.g. the analyzer assumes that a node in the CFG is of type `KFG_NODE`.
Note that the CFG must be build by the frontend, according to the rules stated in chapter 2, i.e. there must be an `Entry` and `Exit` node for every procedure and every call of a procedure must be modeled by a `Call` node with an edge to the `Entry` node of the called procedure and a `Return` node, which has incoming edges from the `Exit` node of the called procedure and the `Call` node.

4.2 C types to provide

4.2.1 CFG

Naturally, a type `KFG` for the CFG itself has to be provided. The CFG consists of two components: nodes and edges between nodes. Every node must have an *identifier*, which represents the node, but is (conceptually) different from the node itself. Since there can be nodes of different types, e.g. `Return` nodes, `Call` nodes, etc., a node *type* has to be defined (which must be of integer '`int`' or `enum` type). Since PAG manipulates lists of CFG nodes[1], a type for lists of nodes has to be given, too.
Since there are also different types of edges in a CFG, a type for edge types has to be defined as an integer or `enum` type, e.g. by `typedef enum _edget {normal_edge, local_edge} EDGE_TYPE;`.
The names of these types to define are:

Name	Description
`KFG`	The CFG
`KFG_NODE_ID`	Node identifier
`KFG_NODE_TYPE`	Type of node
`KFG_NODE`	The node itself
`KFG_NODE_LIST`	List of nodes.
`KFG_EDGE_TYPE`	Type of edge

[1]E.g. the worklist.

4.2.2 Abstract syntax tree

For every type that is defined in the `.syn` file, a corresponding **C** type with the same name has to be defined. This includes also the alias types, defined via `==`. If syntactical lists are used, a **C**, named `LIST_`T, has to be defined for every list over type T. See chapter 11 for the detailed reference.

4.3 C functions to provide

4.3.1 CFG

An detailed listing of the various functions to provide is given in chapter 11. Here we will only concentrate on some special issues.
The two functions `kfg_get` and `kfg_create` are used to obtain the CFG and to initialize it. There must be only one CFG during the analysis and at any time, `kfg_get` must return it. The initialization function is called once by PAG before the analysis is started.
As stated in chapter 2, especially section 2.3.1, there are different node types in the CFG. There are `Start` and `End` nodes for every procedure in the program, as well as `Call` and `Return` nodes[2].
PAG frequently needs to obtain a `Start` node of a procedure or a `Return` node that corresponds to a `Call` node. The interface has to provide functions to obtain corresponding nodes. These are

```
kfg_entry
kfg_exit      , kfg_numproc
kfg_entrys    , kfg_calls
kfg_returns   , kfg_exits
kfg_get_call  , kfg_get_return
kfg_get_start , kfg_get_end
```

The procedures of a program have to be numbered (from 0 on, where 0 is for the main procedure). To obtain the `Start` node of procedure i, `kfg_numproc(`k`,`i`)` has to be called.
For further deatils on these and the remaining functions, refer to chapter 11.

4.3.2 Abstract syntax tree

To access parts of the syntax tree, for every constructor from the `.syn` file, **C** functions have to be written, that extract an argument of the constructor.

[2]In PAGs implementation, the `Local` node is replaced by a `Local` *edge*.

In addition, for every constructor a function has to be provided that tests, if a syntax tree is rooted by that constructor. To handle syntactical lists, functions to test for emptiness, extract the head and tail of the list must be defined. Again, see chapter 11 for a detailed specification.
To be able to convert from an alias type to the referenced DATLA type, for every alias type a conversion function has to be provided that returns the ASCII representation of the type. See section 11.2.2.

4.4 Files to provide

The frontend has to provide four files. Two **C** header files, a **C** code file and one file that defines the edge types of the CFG. These files must reside in the directory *prefix*`/include/frontend/`*frontend*, where *prefix* is the directory prefix where PAG was installed (see chapter 13), usually `/usr/local`. *frontend* is the frontend that is specified as first argument when PAG is invoked, see sections 1.1.1 and 14.1.
The header files are named `interface.h` and `syntax_tree.h`. The first must, when included, define all types that have to be provided by the interface. The later file must provide all types for the abstract syntax tree. Note that you need not write real functions for all the required functions in the interface, but can, where appropriate, use the `#define` facility of the **C** preprocessor instead. If you do this, place the definitions into the two header files. The source file is named `syntax_tree.c` and should contain the code for access functions for , e.g., alias types.
The last file that defines the edge types of the CFG is called `edges`. The edge types are defined as a DATLA enumeration data type and source files `o_edges.`[`ch`] are generated, see section 8.4 for details. This enumeration is translated into a **C** `enum` type called `o_edges`. The names of the `enum` constants is given in the `edges` file: the format of the file is a number of lines, seperated by newlines and one name on each line. This defines the constants starting from `0`, i.e. if the file has lines with names n_1,...,n_k in it, then the `enum` is defined to be `enum {o_`n_1`=0, ..., o_`n_k`=`k`-1} o_edges;`
Note that you must also provide a **C** data type `KFG_EDGE_TYPE`, as explained above. You should use the same enum construct, as is generated in the `o_edges.`[`ch`] files for the definition of that type.

Chapter 5

A very simple Example Generation

In this section a very simple example of an analyzer generation is given. The analysis is called *reaching definition* analysis.

5.1 The analysis

The reaching definition analysis determines for each program point, which assignments may have been made and not overwritten, when program execution reaches this point. The results of this analysis can be used for simple constant folding. The reaching definition analysis is more properly called reaching assignments but reaching definition is the traditional name.
An assignment reaches a program point, if there exists an execution path to a point at which the assignment is made and no furthur assignment to the same variable is made later on that path to the program point.
Formally we can describe this by the following equation system:

$$RD_{entry}, RD_{exit} : \text{Nodes} \rightarrow \mathcal{P}(V \times \text{Nodes})$$

$$\begin{aligned} RD_{entry}(n) &= \begin{cases} \bot & \text{if } n \text{ is the startnode} \\ \bigcup\limits_{e:n' \rightarrow n} RD_{exit}(n') & \text{otherwise} \end{cases} \\ RD_{exit}(n) &= (RD_{entry}(n) \setminus kill(n)) \cup gen(n) \end{aligned}$$

Here V is the set of variables and $e : n' \rightarrow n$ means that there is an edge from node n' to node n in the CFG. $RD_{entry,exit}$ return pairs consisting of a variable and the node where an assignment was made.

The functions *kill* and *gen* are defined on CFG-nodes. *kill* produces the set of variables, node pairs that are overriden by the actual node. *gen* generates the set of assignments that are merged with the already computed assignments. Only assignments kill or generate definitions.

$$kill : \text{Nodes} \to \mathcal{P}(V \times \text{Nodes})$$

$$kill(n) = \begin{cases} \{(\mathtt{x}, n') | n' \in \text{Nodes}\} & \text{, if } Attr(n) = \mathtt{x\!:=}expr \\ \emptyset & \text{, otherwise} \end{cases}$$

Here $Attr(n)$ is the syntax tree at node n.

$$gen : \text{Nodes} \to \mathcal{P}(V \times \text{Nodes})$$

$$gen(n) = \begin{cases} \{(\mathtt{x}, n)\} & \text{, if } Attr(n) = \mathtt{x\!:=}expr) \\ \emptyset & \text{, otherwise} \end{cases}$$

5.2 The implementation

The analysis is implemented for the `CLAX` frontend. Figure 5.1 shows the parts of the `.syn` file for `CLAX` that are relevant for our analysis.
Each node in the CFG is assigned a unique number and so we use numbers instead of nodes. We use a pair of variable number and node to record an assignment. The variable number gives the variable which is assigned to and the node component gives the CFG-node. As carrier for the analysis we choose a lifted set of tuples. The `.set` file is given in figure 5.2.
We initialize every node with the element `bot`, the bottom element of `dataflow`. The combination function at each node of the CFG is the least upper bound operator, as can be seen from the definition, and it corresponds to set union. We use no widening, so the widening operator is the default one (`www(a,b)=b`). Similar, the equality test is the syntactic equality: `eq(a,b)= a=b`.
Since this is just an intraprocedural analysis, no care has been taken for `Call, Entry, Exit` and `Return` nodes. Reading a variable from the terminal (`M_Read(v)`) is the same as an assignment of that variable to a constant. The definition of the transferfunctions is straightforward according to the equations in the equation system.
The complete `.optla` file for the reaching definition analysis is given in section 5.3. The `PROBLEM` section and transfer functions are obvious and should be clear after the previous explanation. The meaning of the support functions should be clear from the comments in the code.

```
SYNTAX
START     : Unlabstat
Block     : M_Block(stats:StatSeq)
            | M_Decl_Block(decl:Decllist,stats:StatSeq);
Decllist  : M_Decls(list:Decllist,elem:Decl)
            | M_NoDecls();
Decl      : M_VarDecl(id:Identifier,typ:Type)
            | M_LabDecl(id:Identifier)
            | M_ProcDecl(head:ProcHead,code:Block);
ProcHead  : M_ParamHdr(id:Identifier,forms:Formals)
            | M_Hdr(id:Identifier);
Formals   : M_Formals(tail:Formals,head:Formal)
            | M_NoFormals();
Formal    : M_Ref(id:Identifier,type:Type)
            | M_Val(id:Identifier,type:Type);
Exp       : M_Binop(op:Op,e1:Exp,e2:Exp)
            | M_Unop(op:Unop,e:Exp)
            | M_Var_exp(var:Var)
            | M_Int_exp(int:Integer)
            | M_Real_exp(real:Real)
            | M_True_exp()
            | M_False_exp();
Var       : M_simpl_var(id:Identifier)
            | M_select_var(var:Var,exp:Exp);
Unlabstat : M_Assign(var:Var,exp:Exp)
            | M_If1(exp:Exp,then:StatSeq,else:StatSeq)
            | M_If2(exp:Exp,then:StatSeq)
            | M_Goto(id:Identifier)
            | M_While(exp:Exp,stats:StatSeq)
            | M_Call(id:Identifier,params:Params,called:Decl)
            | M_Read(var:Var)
            | M_Write(exp:Exp)
            | M_WriteStr(str:String)
            | M_Nop()
            | M_Entry(decl:Decl)
            | M_Exit(decl:Decl)
            | M_Return(id:Identifier,params:Params,called:Decl);
Params    : M_NoExps()
            | M_Exps(tail:Params,head:Exp);

Identifier == snum;
```

Figure 5.1: CLAX frontend `.syn` file.

```
SET
  tup = snum * unum
DOMAIN
labelset = set(tup)
dataflow = lift(labelset)
```

Figure 5.2: Reaching definition `.set` file.

5.3 The `.optla` file

```
NODE
number : unum

PROBLEM reaching_definition
direction: forward
carrier: dataflow
init: bot
init_start: lift({})
combine: comb
widening: www
equal: eq

TRANSFER

// Only assignments and reads change the dataflow-value
M_Assign(var,exp),_ : let flow <= @; in
                       lift((setsubtr(flow , kill(flow,var))) ~
                        (getvar(var),number));

M_Read(var),_ : left flow <=@; in
                 lift((setsubtr(flow , kill(flow,var))) ~
                  (getvar(var),number));


_,_: @ ;



SUPPORT

// Get the identifier of a variable
// val-Identifier(id) transforms cfg-node id
```

```
// into an Indentifier's snum-value
getvar:: Var -> snum;
getvar(M_simpl_var(id))= val-Identifier(id);
getvar(M_select_var(v,e))= getvar(v);

// Kill like in formal definition
// returns set of tuples whose variable component is not Var
kill:: labelset,Var -> labelset;
kill(ls,v) = {lab !! lab<--ls , if lab!1!2 = getvar(v)};


// Setminus : Return set of tuples in s1 but not in s2
setsubtr:: labelset, labelset -> labelset;
setsubtr(s1,s2) = { el !! el<--s1, if !(el?s2)};


eq(a,b) = a=b;
www(a,b) = b;
comb(a,b) = a lub b;
```

Chapter 6

Faint variables: An Example Generation

As an example for an analyzer generation we present the implementation of the *faint variables* analysis. This analysis is a generalization of the *dead variables* analysis.

6.1 The analysis

A variable is said to be dead at a point in the program, if its value is not needed at program points that can follow this point.
Assignments to a dead variable can be dropped from the program (if there are no side effects in the right-hand side of the assignment). This transformation does not change the semantics of the program. Anyhow, by removing an assignment, variables can become dead that were alive before, i.e. variables in the right-hand side of the dropped assignment can become dead variables at program points prior to the assignment.
The dead variables analysis has to be repeated after every transformation step, in order to detect all dead variables. Naturally, this is too expensive. For this reason the faint variables analysis has been invented: Its results remain valid, if assigments to dead variables are eleminated.
The analysis differs from the dead variables analysis only in one point: If, at an assigment `v:=`e, the variable `v` is already known to be faint (dead), then the variables in the expression e are *not* made alive. In the dead variables analysis the variables in e are always made alive, whether or not `v` is already dead or not. This was the reason that the results of that analysis are not invariant under elmination of dead assignments.
More formally, the dead variable analysis searches for a solution of the fol-

lowing system of equations:

$$DEAD(m) = \bigcap_{e:m\rightarrow n} KILL(PROG(n)) \setminus USED(PROG(n))$$

Here $DEAD(m)$ is a set of variables that are dead at node m in the CFG, $PROG$ returns the abstract tree fragment associated with node n and $e : m \rightarrow n$ is an edge in the CFG. Furthermore, $USED(p)$ returns the set of variables used in p, whereas $KILL(p)$ returns the set of variables that are changed in p (assigned to in our case).
The faint variables analysis looks similar:

$$FAINT(m) = \bigcap_{e:m\rightarrow n} \begin{array}{ll} \text{let} & p = PROG(n) \text{ in} \\ & \text{if } \; KILL(p) \subseteq FAINT(n) \text{ then} \\ & \quad FAINT(n) \text{ else} \\ & \quad (FAINT(n) \cup KILL(p)) \setminus USED(p) \\ & \quad \text{fi} \end{array}$$

Here, the variables in $USED(p)$ are not removed from the set of faint variables (which is what $FAINT(m)$ returns for node m), if the variables that are changed in p are already faint.
Obviously, both problems call for a backward analysis: The iteration starts with all variables faint at the end of the program and removes variables that are not faint.
In our implementation, instead of computing faint variables, we will compute *strong* variables, where strong variables are defined by

$$STRONG(m) := ALLVARS \setminus FAINT(m),$$

i.e. $STRONG(m)$ returns the variables that are not faint (=strong) at node m (in the same way, we can compute *live* variables instead of dead variables).
The equation system then becomes:

$$STRONG(m) = \bigcup_{e:m\rightarrow n} \begin{array}{ll} \text{let} & p = PROG(n) \text{ in} \\ & \text{if } \; KILL(p) \cap STRONG(n) = \emptyset \text{ then} \\ & \quad STRONG(n) \text{ else} \\ & \quad (STRONG(n) \setminus KILL(p)) \cup USED(p) \\ & \quad \text{fi} \end{array} \tag{6.1}$$

6.2 The Implementation

We will implement the strong variables analysis for the CLAX frontend. Figure 6.1 shows the parts of the `.syn` file for CLAX that are relevant for our analysis. The complete file is distributed with PAG and (as the complete faintvars analysis) can be found in the `PAG/EXAMPLES/CLAX/faintvars` directory.
Every node in the CFG is labeled with a syntax tree of type `Unlabstat`.
Figure 6.2 shows the `.set` file for the analysis. As carrier for the analysis we choose a lifted set of `snums`, since variables are represented in the CLAX frontend as `Identifier==snum`.
The types `varlist` is a list of variable identifiers, `varset` is a lattice: a set of identifiers and `vars` is the lifted lattice to `varset`. For the analysis, we initialize every node with the value `bot`, the bottom element of `vars`, except the start node[1], which is initialized with `lift({})`, i.e. the value that states that no variable is strong at the end of the program (since there will be no more usages of any variable).
The combination function at each node of the CFG is the least upper bound operator ($\bigsqcup$), as can be seen from the definition of the analysis in equation 6.1. We use no widening, so the widening operator is the default one (`www(a,b)=b`). Similar, the equality test is the syntactic equality: `eq(a,b)= a=b`.
Since we want to build an interprocedural analyzer, special care has to be taken in defining the transfer functions for the `Call`, `Entry`, `Exit` and `Return` nodes of the CFG. These nodes are represented by the CLAX frontend as syntactic constructors `M_Call`, `M_Entry`, `M_Exit` and `M_Return`, cf. figure 6.1.
In CLAX, a `M_Call` node takes the identifier of the procedure to call, the actual parameters of the call and the declarartion of the called procedure as arguments of types `Identifier`, `Params` and `Decl`. The same arguments are again present at the corresponding `M_Return` labeled node. The `M_Entry` and `M_Exit` nodes take the declaration of the procedure as argument.
The other node types of `Unlabstat` are simple and straightforward to implement, by applying equation 6.1. It only has to be determined, which variables are used and which are changed at the node to determine the sets $USED(p)$ and $KILL(p)$.
CLAX contains arrays of values and a left-hand side of an assignment may contain an *array select* expression, such as in `a[i+1]:=1;`. So variables in

[1] Which is the `Exit` node of the main procedure in this case, because we perform a backward analysis.

```
SYNTAX
START : Unlabstat
Block     : M_Block(stats:StatSeq)
         | M_Decl_Block(decl:Decllist,stats:StatSeq);
Decllist  : M_Decls(list:Decllist,elem:Decl)
         | M_NoDecls();
Decl      : M_VarDecl(id:Identifier,typ:Type)
         | M_LabDecl(id:Identifier)
         | M_ProcDecl(head:ProcHead,code:Block);
ProcHead  : M_ParamHdr(id:Identifier,forms:Formals)
         | M_Hdr(id:Identifier);
Formals   : M_Formals(tail:Formals,head:Formal)
         | M_NoFormals();
Formal    : M_Ref(id:Identifier,type:Type)
         | M_Val(id:Identifier,type:Type);
Exp       : M_Binop(op:Op,e1:Exp,e2:Exp)
         | M_Unop(op:Unop,e:Exp)
         | M_Var_exp(var:Var)
         | M_Int_exp(int:Integer)
         | M_Real_exp(real:Real)
         | M_True_exp()
         | M_False_exp();
Var       : M_simpl_var(id:Identifier)
         | M_select_var(var:Var,exp:Exp);
Unlabstat : M_Assign(var:Var,exp:Exp)
         | M_If1(exp:Exp,then:StatSeq,else:StatSeq)
         | M_If2(exp:Exp,then:StatSeq)
         | M_Goto(id:Identifier)
         | M_While(exp:Exp,stats:StatSeq)
         | M_Call(id:Identifier,params:Params,called:Decl)
         | M_Read(var:Var)
         | M_Write(exp:Exp)
         | M_WriteStr(str:String)
         | M_Nop()
         | M_Entry(decl:Decl)
         | M_Exit(decl:Decl)
         | M_Return(id:Identifier,params:Params,called:Decl);
Params    : M_NoExps()
         | M_Exps(tail:Params,head:Exp);
Identifier == snum;
```

Figure 6.1: CLAX frontend `.syn` file.

this select expression (`i` in our example) have to be included in the $USED(p)$ set. Also, writing a variable to the screen (`M_Write(`e`)` node type) uses

```
SET
   varlist = list(snum)
DOMAIN
   varset = set(snum)
   vars = lift(varset)
```

Figure 6.2: CLAX `.set` file.

that variable[2]. Reading a variable from the terminal (`M_Read`(v)) is the same as an assignment of that variable to a constant (anyhow v may again be a select expression). The conditional expressions (`M_If1()`, `M_If2` and `M_While` nodes) use the variables in the expression[3].
The complete `.optla` file for the analysis is presented in section 6.3.
The `PROBLEM` section and most of the transfer function cases should be clear after the previous explanations.
The meaning of the support functions should be clear from the comments in the code.

6.3 The `.optla` file

```
PROBLEM Faintvars
direction : backward
carrier   : vars
init      : bot
combine   : comb
init_start: lift({})
widening : www
equal    : eq

TRANSFER
// A variable is faint, if it is not in drop(@)

// if v is faint then don't add variables in e as strong else
//    variables in e and the variables in a select var expression
//     in v are strong
M_Assign(v,e),_: let thevars<=@; thevar=getvar(v); in
                  if  !(thevar?thevars) then @ else
                   lift((thevars#thevar) lub used(e) lub usedv(v))
```

[2] $KILL$(`M_Write`(e)) $= \emptyset$ in this (and other) case.

[3] This and the write statement are a priori the only possibilities to make a variable strong.

```
                  endif;

// Conditionals: Variables in e are strong
M_If1(e,t,el),_: let thevars <=@; in lift(thevars lub used(e));
M_If2(e,r),_: let thevars<=@; in lift(thevars lub used(e));
M_While(e,s),_: let thevars<=@; in lift(thevars lub used(e));

// Writing a variable makes it strong
M_Write(e),_: let thevars <=@; in lift(thevars lub used(e));

// Reading a variable is like an assignment of the variable
//  to a constant value
M_Read(v),_: let thevars<=@; thevar=getvar(v); in
              if !(thevar?thevars) then @ else
               lift((thevars#thevar) lub usedv(v))
              endif;

//
M_Call(id,params,called),_: let thevars<=@;
                                parvars=getparvars(params); in
                                 lift(thevars lub parvars);
M_Return(_,_,_),local: bot;

// No change
_,_: @ ;

// support functions
SUPPORT
// Get variables in actual parameters
getparvars:: Params -> varset;
getparvars(p)=getparvars2(p,{});

// Help function: iterate parameter list
getparvars2:: Params, varset -> varset;
getparvars2(M_NoExps(),s) = s;
getparvars2(M_Exps(tl,e),s)= getparvars2(tl,s lub used(e));

// Setminus: Return set of variables in s1 but not in s2
subtract:: varset, varset -> varset;
subtract(s1,s2) = {el!! el<--s1, if !(el?s2)};

// Get the identifier of a variable
getvar:: Var -> snum;
```

```
getvar(M_simpl_var(id))=val-Identifier(id);
getvar(M_select_var(v,e))=getvar(v);

// Get the variables that are used in an Expr e
used:: Exp -> varset;
used(M_Binop(op,e1,e2))=used(e1) lub used(e2);
used(M_Unop(op,e))=used(e);
used(M_Var_exp(v))={getvar(v)};
used(_)={};

// Get the variables that are used in a Var
//  i.e. the variables in a select expression
usedv:: Var -> varset;
usedv(M_simpl_var(id))={};
usedv(M_select_var(v,e))= usedv(v) lub used(e);

// The combination function is lub
comb:: vars, vars -> vars;
comb(x,y) = x lub y;

// We use no widening
www:: vars, vars -> vars;
www(x,y) = y;

// Equality is syntactic equality
eq:: vars, vars -> bool;
eq(a,b) = a=b;
```

Chapter 7

Tuning up the analyzer

This chapter is describes how you can speed up the analyzer.

7.1 The datatype generator

The datatype generator is used to generate C-code from a DATLA-input. The generated C-code represents the functionality of the underlying lattice of the analyzer. Since it is an aim of PAG to generate efficient analyzers, the code for the dataflow values should also be efficient. To obtain such code you need information about the datatypes, e.g. the power of a type or the existence of an order.
Therefore the generator tries to calculate these information at first, so that it can be used at generation time. However there is information that can not be calculated statically. So the user has the possibility to annotate the DATLA-source to provide additional information.
The datatype generator constructs efficient C-code with the help of all statically calculated information.

7.2 Primitive datatypes

A datatype is *primitive* if it can be coded as a `long` and thus is no heaptype. Composed types can also be primitive, if their subtypes are primitive and do not exceed certain limits. The sets `bool`, `snum`, `unum` and `chr` are primitive. Additionally, you can implement tuples, sums, flated and lifted domains, sections and sets in bitvector representation as primitive types.
The datatype generator tries to calculate statically, whether a type can be implemented as primitive. Therefore, it also has to calculate statically the

power of a type. If the generator can implement a type as primitive, then it will do so.
The advantage of primitive types is that they use less heap memory. So the memory usage and the runtime of the garbage collection is decreased. Furthermore, operations on primitive objects are faster than operations on non-primitive ones.

7.3 Representation of datatypes

Datatypes are realized straightforward, e.g. tuples are implemented as `struct`'s. In this section only sets and functions will be described, because they are the only constructs, whose implementation can influence the efficiency of the analysis.

7.3.1 Sets

There are two possibilities to represent a set S with elements of type T. The first one is an additive representation in which a set is represented through the elements belonging to it. In the case that T is an unlimited type this representation does not give you the possibility to begin with the set of all elements from T in a dataflow analysis. The second one is subtractive representation of the set. Here exactly those elements are managed, which do not belong to the set.
In the concrete implementation a combined representation was choosen. There is a flag, that indicates, whether a set is represented additive or subtractive, and a container, in which elements from T are managed.
If the flag indicates, for example, a subtractive representation and the container contains just the element x, then this represents the set of all elements from T without x. Therefore, you can represent the set of all elements from T, even in some cases when T has unlimited power.

7.3.2 Functions

The representation of functions from a type T_1 to type T_2 consists of a default element from T_2 and a container of value pairs (x, y), with $x \in T_1$ and $y \in T_2$. By default the container is realised as a list of arrays of pairs. The datastructure is illustrated in figure 7.1.
To ensure consistency the following invariants are maintained:

1. No element occurs twice in the list.

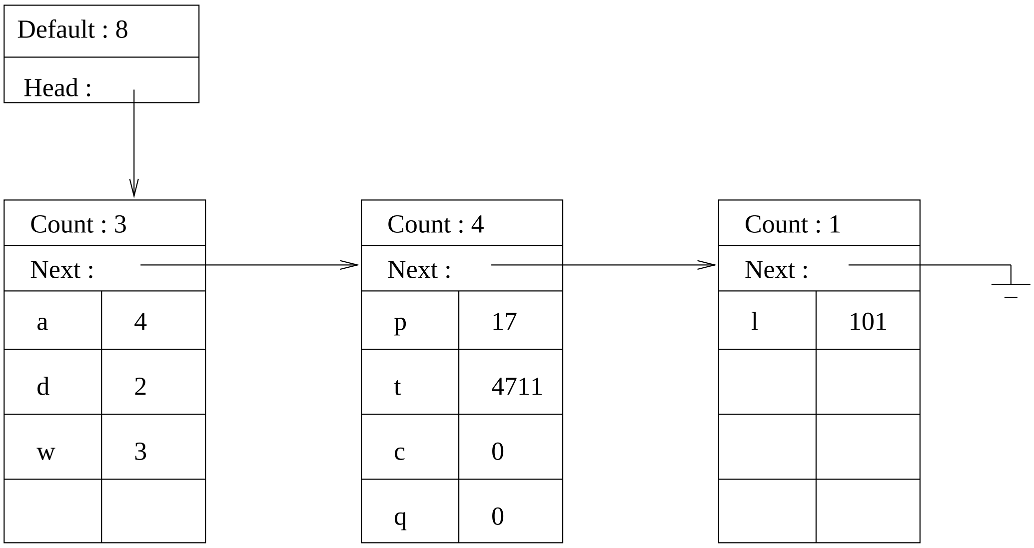


Figure 7.1: Datastructure for functions

2. There are no empty arrays.
3. No element in the list has the default element as value.

7.4 Available Implementations

An important property of these implementations is that an update operation of a container does not change it's content, but constructs a new one with the corresponding content. Therefore, they are called *persistent*.
The operations that are required on containers are searching, insertion, deletion, union, intersection, efficient iteration and rest-construction. Rest-construction is a special operation which is needed for sets and constructs a container in which all elements of a container c_1 are inserted that are not in a container c_2.
There are several possibilities to implement sets and functions, depending on the underlying types. These are described in the next subsections.

7.4.1 Search Trees

One possibility to realize a container is by using search trees. For this the elements, which are managed in the container, must be equipped with a total order ($\leq$).

The keys in a binary search tree follow the binary tree property :

Binary tree property If x is a node in a binary tree and y is a node in the left subtree of x, then $key(y) < key(x)$. If y is a node in the right subtree, then $key(x) \leq key(y)$.

The heigth of a binary tree can vary from 1 to n, when there are n nodes. Therefore, it is better to guarantee a logarithmic heigth. This can be done with several methods. The generator implements AVL-trees and red-black trees, where AVL-trees are selected as default.
The execution time for the required operations are listed below, where n is the number of nodes in the tree.

Operation	Time
Insert	$O(\log n)$
Delete	$O(\log n)$
Searching	$O(\log n)$
Union	$O(n_1 \log n_2)$
Intersection	$O(n_1 \log n_2)$
Rest-construction	$O(n_1 \log n_2)$
Iteration over all	$O(n \log n)$

7.4.2 Hashing

Hashing requires no ordering on the elements of a container. Here a hashvalue $H(x)$ of an element x is calculated and the element is stored in an array T, the hashtable, at position $H(x)$. H is called the hashfunction.
Since different elements may pass the same hash-value, lists of such elements are stored in the hashtable. An example of such a hashtable is given in figure 7.2.
The execution time of operations on hashtables depends on the loadfactor $\alpha = \frac{n}{s}$, where n is the number of entries in the hashtable and s is the size of the hashtable. The goal is that $\alpha \approx 1$.
By using dynamic hashing the size of the hashtable is changed at runtime, so that $\alpha = O(1)$ is guaranteed. So the execution time of the operations (see the table below) is achieved by an amortization analysis.

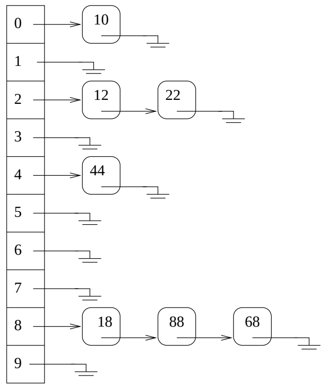


Figure 7.2: Collision resolution with linked lists . Hashfunction $h : U = \{0, \ldots, 99\} \rightarrow \{0, \ldots, 9\}$ with $h(S) = S\%10$, for all $S \in U$.

Operation	Time
Insert	$O(1)$
Delete	$O(1)$
Insert persistent	$O(n)$
Delete persistent	$O(n)$
Searching	$O(1)$
Union	$O(n^2)$
Intersection	$O(n_1^2)$
Rest-construction	$O(n_1^2)$
Iteration over all	$O(n)$

It is possible to produce non unique hashvalues for two equal objects in DATLA. This can be done by using hashing for a set-type in which hashing is also used for a subtype, which is finite. This is because of the subtractive and additive representation of a set. Thus, you can only use hashing when the subtype is infinite or the subtype is represented uniquely. The same can be said for the argument type of functions. In such a case the generator provides you with a warning. For example:

```
SET
inter = [0..9]
pset = set( inter )
ppset = set( pset )
```

If the element $\{0, 1, 2, 3, 4\}$ from `pmenge` is given, it is possible to represent it, as mentioned, in two ways. Additive it is $\{0, 1, 2, 3, 4\}$ and subtractive it is

$\{5, 6, 7, 8, 9\}$. However, the computation of a hashvalue leads to two different values for each representation of the element and so $\{0, 1, 2, 3, 4\}$ could be inserted twice into an element of `ppset`. Therefore you can not use hashing to implement `ppset`.

7.4.3 Bitvectors

Sets can be implemented by using a bitvector. The i-th bit of the vector indicates whether the i-th element is in the set or not. To use this representation of sets the power of the element type must be limited, e.g. a section.
An example is given in figure 7.3

0	1	0	1	1	0	0	0	0	1	0	1	0	1	1	1	0	0	0	1
19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 7.3: Representation of set $S = \{0, 4, 5, 6, 8, 10, 15, 16, 18\}$ with elements from $[0..19]$ as bitvector

Set operations are performed through boolean operations over bitvectors.
A bitvector of lenght n can be implemented on a w-bit machine through $\lceil n/w \rceil$ machine words.
If there are just few elements in the set you also have to use a whole bitvector. Thus, in general you can trade time against space.

Operation	Time
Insert	$O(1)$
Delete	$O(1)$
Insert persistent	$O(n/w)$
Delete persistent	$O(n/w)$
Searching	$O(1)$
Union	$O(n/w)$
Intersection	$O(n/w)$

7.4.4 Arrays

A function can be implemented by using an array of the size of the argument types power. Naturally the power of the argument type has to be limited, e.g. a section. The i-th element of the argument type corresponds to the i-th array position. An example is given in figure 7.4.

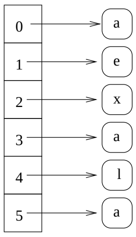


Figure 7.4: Representation of a function with default value a as array

Operations on arrays are very fast. The disadvantage is that you always need a whole array of size n, even if there are only a few elements which are different from the default value.

Operation	Time
Insert	$O(1)$
Delete	$O(1)$
Insert persistent	$O(n)$
Delete persistent	$O(n)$
Searching	$O(1)$

7.4.5 Binary Decision Diagrams

Binary decision diagrams (BDD's) are used to represent boolean functions $f : B^n \to B$. The function is represented through a rooted directed acyclic graph. By reducing the graph representation they get more compact and the computation of boolean manipulations gets simpler.
Each node is assigned a variable and has two edges. One edge corresponds to the case where 1 is assigned to the variable (high edge) and another edge corresponds to the case where 0 is assigned to the variable (low edge).
A boolean expression is constructed bottom-up. First the subexpressions are build and then they are composed to the whole expression.
BDD's can also be used to represent sets. Then the element type must be representable by a bitvector a. The set is represented by its characteristic function:

$$X_S = \sum_{a \in S} \prod_{1 \leq i \leq n} \rho_i(a)$$

$\rho_i(a)$ is the i-th component of the bitvector a.
Set operations can be realized with boolean operations over their characteristic functions.

$$\begin{aligned} X_{S\cup T} &= X_S + X_T \\ X_{S\cap T} &= X_S \cdot X_T \\ X_{S\setminus T} &= X_S \cdot \overline{X_T} \end{aligned}$$

The use of BDD's is only proposed if the functions or sets get big, otherwise their performance is poor.

7.5 Information calculation

Like the definition of sets and domains the calculation of information about them is done bottom-up. Information over primitive sets are known. If the generator gets to a non primitive type it calculates first all information about the subtypes and combines them to information about the whole type according to the construction operation.
However, there are global identifiers in DATLA, about which the generator only knows their primitive type. The user can now annotate them with information the generator can use later on, e.g. restrict the size of such an identifier.
In the following example the information calculation is illustrated.

```
GLOBAL
store_line_min:snum
store_line_max:snum
a_ways:snum < 10

SET
store_lin    = [store_line_min..store_line_max] small

DOMAIN
cache_line   = [0..a_ways]
ccache       = store_lin -> trip
trip         = cache_lines * cache_lines * cache_lines
cache_lines  = lift(cache_line)
```

Here there are three global identifiers of type `snum` among which `a_ways` has at most a value of 10. Therefore, the generator can calculate that section `cache_line` has at most power of 10 and so the lifted domain `cache_lines`

has power at most 12. Therefore `cache_lines` can be implemented as primitive and also the tuple `trip`.
For the function lattice `ccache` various implementations are possible. You can use AVL-trees or hashing. Because section `store_lin` is declared as `small` the generator knows that its size wouldn't get too big. Therefore, it can choose the array implementation of functions for `ccache`, too. Putting `small` after `store_line_min` and `store_line_max` would have the same effect as putting it after `store_lin`.
Specifying a type or identifier as `small` means that you don't exactly know its power, but it's not too big so that it can be implemented as primitive. You can restrict global identifiers more tightly by providing an value, the identifiers value will not exceed.
Refer to chapter 8 for further size restrictions which the user can make.
The user also has the possibility to choose a certain implementation for a type by putting `use(` *name* `)` after the type definition. Then the generator will use the given implementation for that type.
In the following table all names of type implementations are listed.

Name	Description
lift	Impl. of `lift`
plift	Impl. of primitive `lift`
flat	Impl. of `flat`
pflat	Impl. of primitive `flat`
inter	Impl. of section
list	Impl. of `list` as simple linked list
list2	Impl. of `list` as special linked list
slist	Impl. of `list` as sorted simple linked list
slist2	Impl. of `list` as sorted special linked list
power	Impl. of `set` with lists
avlpower	Impl. of `set` with AVL-trees
rbpower	Impl. of `set` with red-black trees
hashpower	Impl. of `set` with dynamic hashing
bitset	Impl. of `set` with bitvectors
pbitset	Impl. of `set` with primitive bitvectors
bddpower	Impl. of `set` with BDD's
func	Impl. of `->` with special lists
avlfunc	Impl. of `->` with AVL-trees
rbfunc	Impl. of `->` with red-black trees
hashfunc	Impl. of `->` with dynamic hashing
continued on next page	

continued from previous page	
Name	Description
arrayfunc	Impl. of `->` with arrays
bddfunc	Impl. of `->` with BDD's

Chapter 8

DATLA Reference

This chapter gives a more precise definition of the DATLA language and semantics. It starts with the grammar of the language. The next section gives the context dependent restrictions of DATLA. The chapter ends with a more precise definition of the meaning of DATLA constructs, than those given in section 3.1.
For a more exact description of terminals `use` and `small` see section 7

8.1 Grammar

The grammar uses the usual constructs of a context free grammar, plus some convenient abbreaviations:

- Non terminals are typeset in italics: *NonTerm*
- Terminal symbols are typeset in underlined typewriter: `SET`
- Optional constructs are enclosed in square brackets: [Optional]
- Alternatives are seperated by |: $a \mid b$
- Zero or more occurences are enclosed in curly braces with a star: { zero or more }*
- One or more occurences take a '+': { one or more }+
- The construct $a \;||\; b$ stands for $a \,|b\, |a\ b$
- Sequences of constructs are seperated by spaces.
- Ranges are annotated by a dash, e.g. `0`-`9`

- Grouping by brackets: { a | b } c
- Comments may be introduced at productions: %A comment

Primitive tokens are numbers *Number* ({0-9}$^+$) and identifiers *Id* ({A-Z|a-z}{_|A-Z|a-z|0-9}*).
Comments are introduced with a double slash and end at the end of the currents line: //{.}*
Whitespace may be inserted everywhere between tokens.

spec	::=	[`GLOBAL` { *globdef* }*] [`SET` { *setdef* }*] `DOMAIN` { *domaindef* }*
globdef	::=	*Id*`:`*primitive* [`small`] [{`<` | `>`} *Number*] | %Declaration of a global identifier
		Id`:`*primitve* [`small`] [`in` `[` *Number* `..` *Number*`]`] | %Declaration of a global identifier
		Id`:optlafunc` %Declaration of a FULA function
primitive	::=	`bool` |
		`snum` |
		`unum` |
		`real` |
		`chr` |
		`str`
type	::=	*primitive* |
		Id
setdef	::=	*Id*`=`*type* `->` *type* [`use(`*Id*`)`] | %Set of functions from a to b
		Id`=`*type* `++` *type* { `++` *type* }* [`use(`*Id*`)`] [`small`] | %Disjoint union of sets
		Id`=`*type* `*` *type* { `*` *type* }* [`use(`*Id*`)`] [`small`] | %Cross product
		Id`=list(` *type* `)` [`use(`*Id*`)`] | %Lists: Set->Set
		Id`=set(` *type* `)` [`use(`*Id*`)`] | %Power set
		Id`=` *enum* | %Enumeration of elements
		Id`=[` *exp* `..` *exp* `]` [`use(`*Id*`)`] [`small`] %Intervall
exp	::=	*Id* |
		Number |
		exp `+` *exp* |
		exp `-` *exp* |
		exp `*` *exp* |

exp / *exp* |
exp % *exp* |
(*exp*) |
- *exp*

enum ::= (*Id* { , *Id* }* | [*edge* { , *edge* }*] | *Id* | *Id*)
%Enumeration of elements — relation — top — bottom

edge ::= *Id*<*Id*

domaindef ::= *Id*= *type* -> *type* [`use(`*Id*`)`] | %Function domain over a b
Id= `flat(` *type*) [`use(`*Id*`)`] [`small`] | %Flat ordering
Id= *type* * *type* { * *type* }* [`use(`*Id*`)`] [`small`] |
%Cross product
Id= `set(` *type*) [`use(`*Id*`)`] | %Power set
Id= *enum* | %Enumeration
Id= `lift(` *type*) [`use(`*Id*`)`] [`small`] | %Lifting of domain
Id= `revord(` *def*) [`use(`*Id*`)`] | %Reverse order
Id= `redprod(` *def* , *Id*) [`use(`*Id*`)`] | %Reduced product
Id= [*exp* .. *exp*] [`use(`*Id*`)`] [`small`] %Intervall

def ::= *type* -> *type* [`use(`*Id*`)`] |
`flat(` *type*) [`use(`*Id*`)`] [`small`] |
type * *type* { * *type* }* [`use(`*Id*`)`] [`small`] |
`set(` *type*) [`use(`*Id*`)`] |
enum |
`lift(` *type*) [`use(`*Id*`)`] [`small`] |
[*exp* .. *exp*] [`use(`*Id*`)`] [`small`]

8.2 Additional Restrictions

Besides the context free restrictions of DATLA programs, given by the grammar, some more restrictions have to be obeyed:

- There must not be multiple definitions for the same identifier.
- All identifiers used as arguments in definitions must be defined somewhere in the `.set` file. The only exception are identifiers in section definitions (see below) and predefined sets.
- Identifiers must not interfere with reserved words, i.e. terminals of the language.

- The most severe restriction is that the *right hand sides* of definitions must be unique. There must not be two identifiers, defined to the same right hand side. E.g. '`foo = set(unum)   bar=set(unum)`' is illegal.

8.3 Semantics – sort of

The definitions in the SET and DOMAIN sections of a DATLA file introduce new names and bind definitions of sets or lattices to them.
Note that recursive definitions are allowed, i.e. a name may occur in a definition of the same name, even if this goes across some other definitions.
The recursivity of the definitions makes the semantics definitions hard. In fact, some possible definitions, like `A = set(A)`, do not possess a semantics at all[1]. So we always assume that the definitions are well defined, i.e. they possess a semantics, and restrict our efforts to giving this semantics.
Note in the following that a lattice can always occur, where a set can. The ordering of the lattice is discarded in this case.

8.3.1 SET definitions

A =

$\texttt{set(}B\texttt{)}$: $A \equiv \mathcal{P}(B)$

$\texttt{list(}B\texttt{)}$: $A \equiv \bigcup_i L_i(B)$, where $L_0(B) := \{\texttt{[]}\}$, $L_{i+1}(B) := \{a\texttt{:}as | a \in B, as \in L_i(B)\}$.

$A_1\texttt{*}\cdots\texttt{*}A_n$: $A \equiv \{(a_1, \ldots, a_n) | a_i \in A_i\}$

$A_1\texttt{++}\cdots\texttt{++}A_n$: $A \equiv \{(i, a_i) | 1 \leq i \leq n, a_i \in A_i\}$

$\texttt{\{}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{|}e\texttt{|}t\texttt{|}b\texttt{\}}$: $A \equiv \{a_1, \ldots, a_n\}$.

$\texttt{[}i\texttt{..}j\texttt{]}$: $A \equiv \{k \in \mathcal{Z} | i \leq k \leq j\}$

$A\texttt{->}B$: $A \equiv \{f : A \to B\}$

As mentioned in section 3.1, the ordering and top, bottom elements of an enumerated *set* are not used and have no impact on the semantics.
The section definition [*exp*..*exp*] allows an identifier as start and/or end point. This identifier is not defined in the `.set` file but rather is part of the

[1]At least they can not be represented as sets.

interface. It's value has to be provided by the return value of a **C** function `snum global_`*Id*`();`. This gives the opportunity to have *dynamic* start/end points with different calls to the analyzer. Anyhow, the values must remain static during the invocation of an analyzer, and thus can only be changed between runs of the same analyzer.

8.3.2 `DOMAIN` definitions

We give the semantics of these constructs in a similar way as we did for the set definitions. Lattices are always written as a (set, ordering) pair $(A, \sqsubseteq)$.
A `=`

> `set(`B`)` : $(A, \sqsubseteq) \equiv (\mathcal{P}(B), \subseteq)$
>
> $(A_1, \sqsubseteq_1)$`*`$\cdots$`*`$(A_n, \sqsubseteq_n)$: $(A, \sqsubseteq) = (A_1 \times \cdots \times A_n, \sqsubseteq_*)$, where we have $(a_1, \ldots, a_n) \sqsubseteq_* (b_1, \ldots, b_n) \iff \forall_i : a_i \sqsubseteq_i b_i$.
>
> `flat(`B`)` : $(A, \sqsubseteq) \equiv (B \cup \{\top, \bot\}, \sqsubseteq_f)$, where $\{\bot, \top\} \cap B = \emptyset$ and $\sqsubseteq_f = \{(\bot, \top)\} \cup \{(\bot, b), (b, \top) | b \in B\}$, by writing the relation as a subset of $A \times A$.
>
> `lift(`$(B, \sqsubseteq_B)$`)` : $(A, \sqsubseteq) \equiv (B \cup \{\top, \bot\}, \sqsubseteq_l)$, where $\{\top, \bot\} \cap B = \emptyset$ and $\sqsubseteq = (\sqsubseteq_B \cup \{\bot, \top\} \cup \{(\bot, b), (b, \top) | b \in B\})^*$, the transitive, reflexive closure of the given relation.
>
> `[`i`..`j`]` : $(A, \sqsubseteq) \equiv (\{k \in \mathcal{Z} | i \leq k \leq j\}, \leq)$
>
> `{`a_1`,`$\ldots$`,`a_n`|`a_{i_1}`<`a'_{i_1}`,`$\ldots$`,`a_{i_k}`<`a'_{i_k}`|`a_t`|`a_b`}` : $(A, \sqsubseteq) \equiv (\{a_1, \ldots, a_n\}, \sqsubseteq_e)$, iff $a_t = \bigsqcup A$ and $a_b \bigsqcap A$, where $\sqsubseteq_e = \{(a_{i_m}, a'_{i_m}) | 1 \leq m \leq k\}^*$, the transitive, reflexive closure of the given ordering.
>
> A_1`->`$(A_2, \sqsubseteq_2)$: $(A, \sqsubseteq) \equiv (\{f : A_1 \to A_2\}, \sqsubseteq_{1\to2})$, where $f \sqsubseteq_{1\to2} g \iff \forall_{a \in A_1} : f(a) \sqsubseteq_2 g(a)$.
>
> `revord(`$(B, \sqsubseteq_B)$`)` : $(A, \sqsubseteq) \equiv (B, \sqsupseteq_B)$
>
> `redprod(`$(B, \sqsubseteq_B)$`,` **Id** `)` : $(A, \sqsubseteq) \equiv (Id(B), \sqsubseteq_B)$, where $Id : B \to B$

The ordering for lattices, defined as power sets is set inclusion. The same remarks about sections for sets apply to lattices, too. Lattices defined by sections use the natural ordering $\leq$ on natural numbers. For lattices defined by lifting, flattening, new top/bottom elements $\top, \bot$ are added, that do not occur in the previous set/lattice.

It is important to note that lattices defined by enumerations are not checked for well-formdness, i.e. if they really are lattices. The user has to make sure that the given ordering fullfills the requirements. Anyhow, all other lattice definitions result in lattices, if the arguments were themselves lattices.

8.4 Accessing Data Types

All data types, primitives ones and those defined in the `.set` file, but not the syntactic types from the `.syn` file, PAG generates **C** functions to access the data types in various ways. All these accesses are restricted to read-only accesses, since updating of data type interferes with PAG's internal handling of data structures. In addition to the access functions, several **C** structures and typedefs are generated.
Some functions are defined for every data type, some are specific to a given type (a list, a set, ...).
For every data type that is defined by a line of the form `name = ...` in the `.set` file, a header file `o_name.h` is generated and therein a **C** type `o_name` is defined. In the following, for a data type *name*, we abbreviate `o_name` by T, if *name* is clear from the context (or irrelevant).
To be able to use the functions for a given type, the file `T.h` must be included (it is generated by PAG in *targetdir*`/src` directory.
In the following section, we first describe the functions that are generated for every data type, or every domain type. Thereafter we present an enumeration of all functions that are specific to a data type, grouped by data type.

8.4.1 Common functions

The type `FLO_BOOL` is defined in the header file `global.h` in PAG's include directory.

Prototype	Description
`char * T_print(T)`	Returns the ASCII representation of the element
`void T_print_fp(FILE *, T)`	Writes the ASCII representation of the element to the file
`FLO_BOOL T_eq(T,T)`	Returns `FLO_TRUE`, if the two elements are equal, `FLO_FALSE` otherwise.
`FLO_BOOL T_neq(T,T)`	Returns `FLO_TRUE`, if the two elements are not equal, `FLO_FALSE` otherwise.

The following functions are only defined for domain types, i.e. types defined in the `DOMAIN` section of the `.set` file.

Prototype	Description
`FLO_BOOL` *T*`_istop(`*T*`)`	Returns `FLO_TRUE`, if the element is the top element ($\top$) of the data type, `FLO_FALSE` otherwise.
`FLO_BOOL` *T*`_isbottom(`*T*`)`	Likewise for the bottom ($\perp$) element of the data type.
`FLO_BOOL` *T*`_le(`*T*`,`*T*`)`	Returns `FLO_TRUE`, if the first argument is smaller than the second argument, according to the ordering of the data type. It returns `FLO_FALSE` otherwise[2]
`FLO_BOOL` *T*`_leq(`*T*`,`*T*`)`	Returns `FLO_TRUE`, if the first argument is smaller than or equal to the second argument.
`FLO_BOOL` *T*`_ge(`*T*`,`*T*`)`	Returns `FLO_TRUE`, if the first argument is greater than the second.
`FLO_BOOL` *T*`_geq(`*T*`,`*T*`)`	Returns `FLO_TRUE`, if the first argument is greater than or equal to the second.
T *T*`_glb(`*T*`,`*T*`)`	Returns the glb ($\sqcap$) of the two arguments.
T *T*`_lub(`*T*`,`*T*`)`	Returns the lub ($\sqcup$) of the two arguments.

8.4.1.1 Cursor functions

Although there are plenty of access functions for data types, one often wants to iterate about all components of an element or even wants to enumerate all elements of a data type.
For this reason, *cursors* are provided. They allow to enumerate the components of functions, lists or sets, as well as to enumerate all elements of any data type.
There are two classes of cursor functions:

1. Functions to enumerate the components of a function, a list or a set. Since the remaining data types (enumerations, sections, products,...) do not have components but are distinct elements in their own right, there are no such functions for them.

[2]From now on, we will leave out the 'otherwise' case, whenether a function returns `FLO_BOOL`.

2. Functions to enumerate all elements of a given data type. These functions are defined for every countable data type[3].

We call the later *abstract cursors*.
PAG generates for every data type appropriate **C** types in the header file for that data type, called `o_N_cur` and `o_N_acur`[4] for cursors and abstract cursors, respectively (We will abbreviate them as `T_cur` and `T_acur` again). Here N is the name of the data type (like, e.g., it occurs in a line `N=...` in the `.set` file). These types are defined as pointers to **C** structures, called `_T_cur` and `_T_acur`, repectively.
For every cursor, four functions are defined:
`T_cur_reset()`, `T_cur_next()`, `T_cur_is_empty()` and `T_cur_get()`[5].
To use a (abstract) cursor, one first calls `T_cur_reset()` for a given type, this resets the cursor to the beginning of the enumeration. To test if there is nothing left to enumerate, one calls `T_cur_is_empty()`, which returns 1 in this case. To get the enumeration, a call to `T_cur_get()` has to be made. And to advance the enumeration to the next element, `T_cur_next()` has to be called.
Here is a small **C** code fragment, to demonstrate the procedure for a list data type over `snums` (i.e. there was a line `foo = list (snum)` in the `.set` file).

```
/* Declare a cursor structure and a cursor */
_o_foo_cur mycurstruct;
o_foo_cur mycur = &mycurstruct;
snum mysnum;

...
/* Init the cursor */ o_foo_cur_reset(mycur);
while (!o_foo_cur_is_empty(mycur)) {
    mysnum=o_foo_cur_get(mycur);
            ...
        /* Massage mysnum here */
            ...
    o_foo_cur_next(mycur);
}
```

Figure 8.1: Example of cursor usage

We will now first describe the (non abstract) cursor functions for functions,

[3]Functions from `snum` to `snum` are not countable, e.g.
[4]The first is only generated for lists, functions and sets.
[5]For abstract cursors, the functions are named `T_acur_reset()`, etc.

lists and sets. For all of these the following functions are defined (with the given prototypes):

Prototype	Description
`void` T`_cur_next(`T`_cur)`	Advances the cursor to the next element
`FLO_BOOL` T`_cur_is_empty(`T`_cur)`	Returns `FLO_TRUE`, if the enumeration is exhausted.

The reset functions for function cursors has the prototype

`void` T`_cur_reset(`T`_cur,` T`,` *tart*`)`

The third argument is an element of the target type of the function[6]. This tells the enumeration to drop all argument/value pairs, where the value of the function is equal to the third argument of the reset function. Normally, this argument will be the default value of the function. You cannot enumerate a function, if the source type (n_1) is infinite, or if the third argument is different from the default function result.
For lists and sets, the reset functions looks like

`void` T`_cur_reset(`T`_cur,`T`)`

The result of a `_get` cursor function call is

- For lists and sets an element of the underlying data type, e.g. for lists over `snum`, the result is of **C** type `snum`.
- For functions, an argument/value pair is returned. If the function type was declared by N `=` n_1 `->` n_2 an element of n_1`*`n_2 is returned. If there had been no definition with this right hand side in the `.set` file, a new data type is introduced by PAG, named `o_`N`_fupair`. Header and source files for this new data type are generated, as if it had been defined in the `.set` file.

For function types, the `_get` function has the prototype

rt T`_cur_get(`T`_cur)`

If there is a definition N' `=` n_1`*`n_2 in the `.set` file, then rt=`o_`N', otherwise rt=`o_`N`_fupair`, as explained above.

[6] I.e. if the function was declared by N `=` n_1 `->` n_2 in the `.set` file, then *tart* is `o_`n_2.

For lists and sets, the function looks like

`o_`N' `T_cur_get(T_cur)`

Here N' is the data type of the underlying element, i.e. the definition was N `= list` N' in the `.set` file (replace `list` with `set` for sets, obviously).
For abstract cursors, four functions are defined for every type:

Prototype	Description
`void T_acur_reset(T_acur)`	Resets abstract cursor.
`void T_acur_next(T_acur)`	Advances the cursor to the next element.
`T T_acur_get(T_acur)`	Get element.
`FLO_BOOL T_acur_is_empty(T_acur)`	Returns `FLO_TRUE`, if the enumeration is exhausted.

8.4.2 Specific functions

8.4.2.1 Primitive data types

For the primitive types `snum`, `unum`, `real`, `bool`, `chr` and `str`, several **C** functions are defined.
For the numeric data types (`snum`, `unum`, `real`) the following functions are provided by PAG:

Prototype	Description
`T T_add(T,T)`	Performs addition.
`T T_min(T,T)`	Performs subtraction of the second argument from the first.
`T T_mal(T,T)`	Performs multiplication.
`T T_geteilt(T,T)`	Performs division of the first argument by the second.
`T T_modulo(T,T)`	Performs the modulo operation.
`T T_hoch(T,T)`	Performs exponentiation of the first argument by the second.
`T T_abs(T)`	Returns the absolute value of the argument.
`T T_unmin(T)`	Performs negation of the argument.
`snum T_sgn(T)`	Returns the signum of the srgument (`-1, 0, +1`).

For `snum` and `unum`, the following functions are defined:

Prototype	Description
`T T_bor(T,T)`	Performs a bitwise OR of the arguments.
`T T_band(T,T)`	Performs a bitwise AND of the arguments.
`unsigned int T_to_unsigned(T)`	Performs cast to unsigned integer.
`T int_to_T(int)`	Performs cast from int.
`T T_not(T)`	Performs bitwise NOT on argument.

For `snum` additional functions are defined:

Prototype	Description
`unum snum_to_unum(snum)`	Cast to `unum`.
`int snum_to_int(snum)`	Cast to int.
`snum str_to_snum(char *)`	Read in `snum` from string.

For `real`, the following functions are provided:

Prototype	Description
`real snum_to_real(snum)`	Cast from `snum`.
`snum real_to_snum(real)`	Cast to `snum`.
`unum real_to_unum(real)`	Cast to `unum`.
`snum real_trunc(real)`	Truncate real number after decimal point.
`real int_to_real(int)`	Cast from integer.

The type `real` is defined to be `typedef double real;`.
For the type `bool`, which is defined by `typedef int bool;`, the following functions are defined:

Prototype	Description
`unum bool_to_unum(bool)`	Cast to `unum`.
`snum bool_to_snum(bool)`	Cast to `snum`.

For `chr`, which is defined by `typedef char chr;`, these functions are provided:

Prototype	Description
`chr snum_to_chr(snum)`	Cast from `snum`.
`unum char_to_unum(chr)`	Cast to `unum`.
`snum char_to_snum(chr)`	Cast to `snum`.

For `str` these functions are defined:

Prototype	Description
`real str_to_real(str)`	Scans a `real` from a string.
`void str_to_charp(str, char *)`	Copies the contents of the string to the location, pointed to by the second argument.
`str unum_to_str(unum)`	Returns ASCII representation of the `unum`.
`str snum_to_str(snum)`	Returns ASCII representation of the `snum`.
`str int_to_str(int)`	Returns ASCII representation of the integer.
`snum str_slen(str)`	Returns the length of the string as `snum`.
`str str_conc(str,str)`	Returns the concatenation of the second argument to the first.
`int str_length(str)`	Returns the length of the argument as integer.

8.4.2.2 Sets

For sets, the following functions are defined, where *ET* denotes the type of the elements of the set.

Prototype	Description
`FLO_BOOL` *T*`_member(`*ET*`,`*T*`)`	Returns `FLO_TRUE`, if the element is in the set.
`int` *T*`_size(`*T*`)`	Returns the cardinality of the set. Returned is either an integer $\geq$ 0 or the constant `UNENDLICH` for infinity. `UNENDLICH` is defined in the header file `analyzer/header.h`.
`FLO_BOOL` *T*`_subset(`*T*`,`*T*`)`	Returns `FLO_TRUE`, if the first argument is a subset of or equal to the second argument ($\subseteq$).
`FLO_BOOL` *T*`_supset(`*T*`,`*T*`)`	Returns `FLO_TRUE`, if the first argument is a superset of or equal to the second argument ($\supseteq$).
`FLO_BOOL` *T*`_subset_real(`*T*`,`*T*`)`	Returns `FLO_TRUE`, if the first argument is a real subset of the second ($\subset$).
`FLO_BOOL` *T*`_supset_real(`*T*`,`*T*`)`	Return `FLO_TRUE`, if the first argument is a real superset of the second ($\supset$).

8.4.2.3 Lists

For lists, the following functions are defined, where *ET* denotes the type of the elements of the list.

Prototype	Description
`FLO_BOOL` *T*`_is_empty(`*T*`)`	Returns `FLO_TRUE`, if the list passed as argument is empty.
ET *T*`_head(`*T*`)`	Returns the first element of the list[7].
T *T*`_tail(`*T*`)`	Returns the tail of the list, i.e. the list without the first element[7].
`int` *T*`_length(`*T*`)`	Returns the length of the list.
ET *T*`_get_elem(`*T*`,int)`	Returns the ith element, where i is given by the second argument and starts with 1.
`FLO_BOOL` *T*`_is_elem(`*T*`,`*ET*`)`	Returns `FLO_TRUE`, if the second argument is in the list.

8.4.2.4 Functions

For functions, the following functions are defined, where *tart* denotes the type of the codomain of the function and *argt* denotes the type of the domain of the function, i.e. the function is of type *argt* `->` *tart*.

Prototype	Description
tart *T*`_get(`*T*`,`*argt*`)`	Returns the value of the function for the second argument.

8.4.2.5 Sections

For sections (`[`i`..`j`]`) the following functions are defined. Note that a section is an integer with a restricted value.

[7]The behaviour of these functions is undefined, if the argument list is empty!

Prototype	Description
snum *T*_to_snum(*T*)	Converts a section value to a snum value
T *T*_add(*T*,*T*)	Returns the section that is the result of adding the two argument sections.
T *T*_abs(*T*)	Returns the absolute value of the section.
T *T*_modulo(*T*,*T*)	Returns the value of the first argument modulo (%) the second.
T *T*_mal(*T*,*T*)	Performs a multiplication on the sections.
T *T*_min(*T*,*T*)	Performs subtraction of the second section from the first.
T *T*_hoch(*T*,*T*)	Performs exponentiation, i.e. a^b, where a is the first argument.
T *T*_geteilt(*T*,*T*)	Performs division of the first argument by the second
T *T*_unmin(*T*)	Negates the section
T *T*_bor(*T*,*T*)	Returns the bitwise OR of the two sections
T *T*_band(*T*,*T*)	Returns the bitwise AND.
snum *T*_sgn(*T*)	Returns the snum 0, if the section is zero, returns +1, if the section is positive, -1 otherwise.
unsigned int *T*_to_unsigned(*T*)	Casts the section to an unsigned integer.
T int_to_*T*(int)	Casts the integer to a section.
T *T*_not(*T*)	Does a bitwise complement of the section.
T *T*_get(int)	Returns the i–th value in the section.
int *T*_pos(*T*)	Returns the index of the section (inverse to *T*_get()).
T *T*_from_snum(snum)	Casts a snum to a section.
str *T*_to_str(*T*)	Casts a section to a string.

8.4.2.6 Cross Products

For cross products (N_1*N_2*$\cdots$*N_n) the following function is defined, where *elt-k* denotes the type of the k-th element in the product (N_k).

Prototype	Description
`elt-k T_select_k(T)`	Selects the k-th component from the product.

8.4.2.7 Disjoint unions

For disjoint unions (N_1++$\cdots$++N_n) the following functions are defined, where *elt-k* denotes the type of the k-th component (N_k).

Prototype	Description
`int T_selector(T)`	Returns the index k, that corresponds to the type N_k of the argument.
`FLO_BOOL T_is_k(T)`	Returns `FLO_TRUE`, if the argument is of type N_k.
`elt-k T_down_to_k(T)`	Converts the argument to an element of N_k.

8.4.2.8 Flat Domains

There are two functions defined for flat domains, where *elt* denotes the type of the underlying set (i.e. N in `flat (N)`).

Prototype	Description
`T T_lift(elt)`	Casts an element to the flat domain.
`elt T_drop(T)`	Casts an domain element to the underlying type.

8.4.2.9 Lifted Domains

For lifted domains, similar functions as for flat domains are defined.

Prototype	Description
T `T_lift(`*elt*`)`	Casts an element to the domain.
elt `T_drop(`*T*`)`	Casts an domain element to the underlying type.

8.4.2.10 Reverse Order

In this case all functions for the underlying domain are also defined for this type, but with the appropriate prefix.

8.4.2.11 Reduced Product

Here again all functions for the underlying domain are defined.

8.4.3 Accessing Dataflowinformation

To access the information at the edges of the CFG, five functions are provided:

Prototype	Description
NodeType `dfi_get_post_info_all(` `DFI_STORE` *st*, `DFI_ID` *id*`)`	Returns the combined value over all values at outgoing edges from node *id*.
NodeType `dfi_get_post_info(` `DFI_STORE` *st*, `DFI_ID` *id*, `KFG_EDGE_TYPE` *edge*`)`	Returns the element at the outgoing edge *edge* of node *id*.
NodeType `dfi_get_post(` `DFI_STORE` *st*, `DFI_ID` *id*, `int` *ind*, `KFG_EDGE_TYPE` *edge*`)`	AS interprocedural analyses may have multiple data-flow elements at each edge, corresponding to different calling contexts. This function returns the *ind*-th element at edge *edge* of node *id*. *ind* must be in the range [0..`kfg_arity_id(`*id*`)` − 1].
NodeType `dfi_get_pre_info(` `DFI_STORE` *st*, `DFI_ID` *id*`)`	Returns the element at the incoming edge of node *id*.
continued on next page	

continued from previous page	
Prototype	Description
NodeType `dfi_get_pre(` `DFI_STORE` *st*, `DFI_ID` *id*, `int` *ind*`)`	This function returns the *ind*-th element at the incoming edge of node *id* (cf. `dfi_get_post`).

8.5 Error messages

In this section the error messages and warnings of the DATLA generator are explained.

8.5.1 Errors

- `'power to big'`

  This means that the power of a type doesn't fit in a variable of type `ull`. The generator can't calculate any furthur. This can happen if you use set construction with sections too often . In this case you can go to infinity by using `snum` or `unum` instead of a section.

- `'illegal template in use'`

  If you use, for example, a set template for a function definition, this error occurs.

- Syntax and semantic errors are not listed. They are self-explaining.

8.5.2 Warnings

- `'Declared as small, but subtype is not'`

  If you specify a type as `small`, you also have to do so with its subtypes. This message can occur if you want to specifiy a type as small so that it can be implemented as primitive, but the generator calculates that it can't be implemented as primitive.

- `'You should not use hashpower.t for that type'`

  As mentioned hash-values of elements may not be unique. This warning states that if you use hashing to implement that type it can lead to such a situation.

- `'When using arrayfunc.t, typepower must be limited'`

  When you use arrayfunc for a function definition the type isn't allowed to be unlimited. If the generator calculates that the type power is unlimited, this warning occurs.

- `'When using bitset.t, typepower must be limited'`

  When you use bitset for a set definition the type isn't allowed to be unlimited. If the generator calculates that the type power is unlimited, this warning occurs.

Chapter 9

Providing own Implementations

To use own implementations of datatypes you have to provide a so called template-file in which the realization is given. After you have written a template-file you can use it in the `.set`-file with the `use` mechanism.

9.1 Templates

To generate C-Code for a new datatype template-files are used. Those are files, that describe how a new type T is constructed from subtypes and how it is implemented in an abstract way. There are syntactic patterns in a template-file which are replaced by typenames. This is done by a tool `create` and the output conists of a C-header-file and a C-source-file, in which the concrete implementation of the new type is given.
By providing a template for a single linked list for example, the tool can create a concrete implementation of a list of `int`-values, but also a concrete implementation of a list of `str`-values.
A template is divided into two sections. One for the C-header-file (`.h`) and another one for the C-file (`.c`). The header-section begins with the macro `#H_FILE` and the C-file-section starts with the macro `#C_FILE`. In both sections you can use C language constructs.
The pattern for the output-type T is `$T` and there is the possibility to use up to four patterns `$T1` ... `$T4` for the input-types $T_1 \ldots T_4$. The pattern for the output-type is used in every template-file, because it specifies the new type, and `$U` is a pattern for an additional unsigned integer value.

9.2 The interface

By a typedef a type `T` is defined, which can be a `long` or a pointer. If it is a pointer the type of the datastructure it points to should be `_T`. Heapmemory is managed by a memory-manager that also has the opportunity to perform a garbage collection.
The function names that are defined for the type `T` all have the prefix `T_`.
Because the datatypes are used by the implementation of a functional programming language with sharing of heap-objects, you should have the following in mind: A heap-object can be pointed to from several other objects. Therefore you aren't allowed to change an object by a function. If you want to change such an object you have to copy it before. Exceptions of this are functions that explicitly work destructive (have postfix `destr`) or types, which are marked as *noshare* at their registration.
An object can be shared by the memory-manager automatically. That means you can't be sure that there is only one reference to the object, even though you have generated no furthur reference to it.

9.2.1 Memory-manager

The memory-manager provides the following functions:

```
int GC_registertype( int size, void (*mark)(void*),
                               void (*mcopy)(void*, void*),
                               int (*equal)(void*,void*),
                               unsigned int (*hash)(void*),
                               int noshare );
```

This function is used at initialization for registration of the datatypes at the memory-manager, when `T` is a pointer to a heap-object. The return-value is a typenumber which is used for memory-allocation. The following parameters are necessary:

- `size`: the number of bytes that `T` allocates in the heap (`sizeof(_T)`).
- The functions `mark`, `mcopy`, `equal` and `hash` which are described closer later on. Normally this are the functions `T_mark`, `T_mcopy`, `T_eq` and `T_hash`.
- the flag `noshare`, when true, means that no object of this type is allowed to be pointed to twice.

```
void * GC_alloc( int type);
```

This function returns a pointer to a new heap-object of that type. The parameter is the typenumber you got from `GC_registertype`. The returned pointer is never `NULL`. If there is no more memory the memory-manager aborts.

```
void GC_free( void *prt );
```

Frees an heap-object explicitly. Normally heap-objects are freed through a garbage collection.

```
void GC_mark( void **prt);
void * GC_share_copy( void *prt );
```

These functions are used in a garbage collection or sharing phase. Therefore you have to implement the following functions for a type `T`:

```
void T_mark_self( T ptr );
```

This is a macro that calls `GC_mark` with the adress of `ptr`.

```
#define T_mark_self(x) (GC_mark((void**) &x))
```

For non-heap-types this macro also has to be defined. In this case it is empty.

```
void T_mark( T ptr );
```

The memory-manager calls this function to mark all pointers in the specified object. All objects of type $\texttt{T}_\texttt{i}$ have to be marked through $\texttt{T}_\texttt{i}$`_mark_self`.

```
T T_mcopy_self( T ptr );
```

This function performs a sharing copy of an object. For heap-types:

```
#define T_mcopy_self(x) ((T)GC_share_copy((void*)x))
```

and for non-heap-types:

```
#define T_mcopy_self(x) (x)
```

```
void T_mcopy( void *src, void *target );
```

The content of `src` is copied to `target`. Objects of type $\mathtt{T_i}$ have to be copied by $\mathtt{T_i}$`_mcopy_self` and other components by assignment.

```
int T_eq( T ptr1, T prt2 );
```

This function checks the structural equality of two objects. The function given to `GC_registertype` is used to test, if two objects can be shared.

```
unsigned int T_hash( T ptr );
```

By that function a hash-value for the given object `ptr` is calculated. It should return the same value for semantically equal objects.

```
void T_find_obj( void );
void T_copy_obj( void );
```

These two functions are used for marking static objects of type `T` in a garbage collection or a sharing phase. You have to ensure that these functions are executed only once until the next call to `T_clear_flag`.
All static objects have to be marked through `T_mark_self` or `T_mcopy_self` respectively. In addition the corresponding functions have to be called for the appropriate subtypes, e.g. $\mathtt{T_1}$`_find_obj`().

```
void T_clear_flag( void );
```

See `T_find_obj`

9.2.2 Additional functions

The following functions have to be implemented:

- Those referenced in sections 8.4.1 and following.
- Functions and variables in the following table:

Prototype	Description
`void T_init(void)`	Initialization of the type `T` and its subtypes.
`T T_duplicate(T)`	Duplicates an object of type `T`. Only pointers were copied not the whole content.
`int T_is_power_unendl`	The variable must be set to `1` if the power of that type is infinite. You can obtain the calculated value from the generator by defining it as `int T_is_power_unendl=$U;`
`ull T_power`	Here the power of type `T` is stored. Should be `UNENDLICH` if the power is infinite.
`T T_bottom`	This variable contains the bottom element of the domain. It is only required for domain-types.
`T T_top`	This variable contains the top element of the domain. It is only required for domain-types.
`char * T_len(void)`	Return the length of a domain as a character string. It is only required for domain-types.

- Lists

  For lists, the following functions have to be implemented, where `ET` denotes the type of the elements of the list.

Prototype	Description
`T T_new(void)`	Returns an empty list.
`T T_extend(ET,T)`	Returns a list with argument one as head and argument two as tail.
`T T_conc(T,T)`	Returns a list which is the concatenation of the two given lists.
`T T_set_elem(T,int,ET)`	Sets the i-th element, which is given by the second argument, of a list to the element that is given by the third argument.
`T T_append(T,ET)`	Appends an element given by the second argument to a list and returns the list.
`T T_remove(T,ET)`	Removes an element from a list.

- Sets

  For sets, the following functions have to be implemented, where `ET` denotes the type of the elements of the set.

Prototype	Description
`T T_union(T,T)`	Returns the union of two sets.
`T T_disjoint(T,T)`	Returns the intersection of two sets.
`int T_isempty(T)`	Returns 1 if the given set is the empty set.
`FLO_BOOL T_isall(T)`	Returns `FLO_TRUE` if the given set is the set of all elements of type `ET`, `FLO_FALSE` otherwise.
`T T_create(void)`	Returns the empty set.
`T T_remove(T,ET)`	Removes an element, given by the second argument, from a set and returns the resulting set.
`T T_insert(T,ET)`	Inserts an element in a set.
`T T_not(T)`	Returns the complement of a given set.

- Functions

  For functions, the following functions have to be implemented, where `AT` denotes the type of the arguments of the function and `IT` denotes the type of the image of the function. `PT` is the type of the argument/value pair as explained in section 8.4.1.1 and it is automatically substituted by the generator for pattern `$T3`.

<table>
<tr><th>Prototype</th><th>Description</th></tr>
<tr><td><code>T T_create(IT)</code></td><td>Creates a new function with default value, given by the argument.</td></tr>
<tr><td><code>T T_change_default(T,IT)</code></td><td>This function changes the default value of the given function to the value given by the second argument. This function can operate destructively, because the semantics is not changed.</td></tr>
<tr><td><code>T T_update(T,ET,IT)</code></td><td>Returns a function in which the function value of the given argument is changed to the value given by the third argument.</td></tr>
<tr><td><code>T T_update_destr(T,ET,IT)</code></td><td>This is the destructive version of the above function.</td></tr>
<tr><td><code>T T_update_pair_destr(T,PT)</code></td><td>The function should be defined like this:
<pre>
$T $T_update_pair_destr($T r,
                        $T3 p)
{
  return($T_update_destr(r,
           $T3_select_1(p),
           $T3_select_2(p)));
}
</pre></td></tr>
<tr><td><code>ull T_size(T)</code></td><td>Returns for the given function the number of values that are different from the default value.</td></tr>
<tr><td><code>FLO_BOOL T_isbottom(T)</code></td><td>Returns <code>FLO_TRUE</code> if the given function is equal to the bottom element of a function lattice. Otherwise it returns <code>FLO_FALSE</code>.</td></tr>
<tr><td><code>FLO_BOOL T_istop(T)</code></td><td>Returns <code>FLO_TRUE</code> if the given function is equal to the top element of a function lattice. Otherwise it returns <code>FLO_FALSE</code>.</td></tr>
</table>

- Cross products

  For cross products $(N_1 * N_2 * \cdots * N_n)$ the following functions must be implemented, where *elt-k* denotes the type of the k-th element in the product (N_k).

Prototype	Description
`T T_create(`*elt-1*`,...,`*elt-n*`)`	Creates a new n-tuple of the given components.
`T T_update_`*k*`(T,`*elt-k*`)`	Returns a tuple where the k-th element is updated with the given element.

- Disjoint unions

  For disjoint unions (N_1++$\cdots$++N_n) the following functions have to be implemented, where *elt-k* denotes the type of the k-th component (N_k).

Prototype	Description
`T T_create_from_`*k*`(`*elt-k*`)`	Creates a new element of the disjoint union of the given argument.

Chapter 10

FULA/OPTLA Reference

This chapter explains the FULA and OPTLA languages in greater detail. In fact, these two are so closely related that we will refer to them as *the language* only.
The chapter starts with the presentation of the context free grammar of the language. The next section describes the context sensitive restrictions[1]. The third section goes on with a description of the type system of the language in greater detail, while the next section gives the semantics of the basic FULA constructs. The concluding section of this chapter introduces and desribes the predefined functions of the language.

10.1 Grammar

This grammar uses, beside the *Number* and *Id* tokens from the DATLA grammar, additional tokens for integer numbers *Numbers* ({0-9}$^+$[U]), real numbers *Real* ({0-9}*.{0-9}$^+$[{e|E}[+|-]{0-9}$^+$]), characters *Char* ('ASCII') and strings *String* ("{ASCII}*"). The *int* token is an integer token, greater or equal to 1.
Unsigned integers are annotated by postfixing the letter `U`, as in the definition of *Numbers*.
Comments and whitespace may be introduced, as in the DATLA grammar.

spec	::= *global-spec node-spec problem-spec transfer-spec support-spec*	
global-spec	::= [`GLOBAL` { *node-def* }*]	%Global variables
node-spec	::= [`NODE` { *node-def* }*]	%Variables at each CFG node
node-def	::= *Id* `:` *type*	%Name and type of variable

[1] Sometimes called static semantics, too.

problem-spec ::= `PROBLEM` *Id* { *problem* }*

problem ::= `direction:` { `forward` | `backward` } |
`carrier:` *type* | %Name of lattice
`init:` *optla-exp* | %Init values for each CFG node
`equal:` *Id* | %Equality used
`combine:` *Id* | %Combination of DF values
`widening:` *Id* | %Widening operator
`init_start` *optla-exp* %Init value for start node

hash ::= [`hash` [`[`*Number*`]`] |
`cache` [`[` *Number*`]`]]

transfer-spec ::= `TRANSFER` *hash* { *tran-fkt* }+ %Transfer functions

tran-fkt ::= *pattern*`,`*simple-pattern*`:`*optla-exp*`;`
%Argument, edge pattern: result

support-spec ::= [`SUPPORT` { *fkt-def* }*]

const ::= *Number* | %Unsigned integer
bool |
Real |
Char |
String |
`[`*all]* | %Full set
`top` | %Top element of lattice
`bot` | %Bottom element of lattice
`[]` | %Empty list
`{}` %Empty set

bool ::= `true` | `false`

fkt-def ::= *Id*`(` [*pattern* { `,` *pattern* }*] `)` `=` *optla-exp*`;` |
%Function definition. Argument pattern = result
Id `::` *hash* [*type* { `,` *type* }*] `->` *type*`;`
%Type declaration with hash specification

type ::= *Id* | %Simple type
`*`*Id* | %Syntactical list
bas-type %Basic type

bas-type ::= `unum` | %Unsigned integers
`snum` | %Signed integers
`real` | %Real numbers
`bool` | %Boolean

	`chr` |	%Characters
	`str`	%Strings
simple-pattern	::= *Id* |	%Variable or enum element
	`_` |	%Anonymous variable
	const	%Constant
pattern	::= *simple-pattern* |	
	Id`(`*pattern* { `,` *pattern* }* `)` |	%Syntax constructor
	pattern`:` *pattern* |	%Cons listpattern
	`[!]` |	%Empty list for syntactical lists
	pattern`::`*pattern* |	%Cons listpattern for syntactical lists
	`(`*pattern* { `,` *pattern* }* `)` |	%Tupel
	type `$` *pattern* |	%Type specification
	pattern `as` *Id*	%Aliasing of pattern
optla-exp	::= *Id* |	%Variable or primitve
	optla-exp `<<` *optla-exp* |	%Bitshift left
	optla-ex `>>` *optla-exp* |	%Bitshift right
	bas-type `(` *optla-exp* `)` |	%Typecast
	`print(` *optla-exp* `)`*optla-exp* |	%Print a, return b
	const |	%Constant
	`if` *optla-exp* `then` *optla-exp* `else` *optla-exp* `endif` |	%Conditional
	`case` *optla-exp* { `,`*optla-exp* }* `of` { *case-pattern* }+ `endcase` |	%Case match
	`let` { *let-def* }+ `in` *optla-exp* |	%Local definitions
	`{` *optla-exp* { `,` *optla-exp* }* `}` |	%Set enumeration
	`[` *optla-exp* { `,` *optla-exp* }* `]` |	%List
	optla-exp `{!` *optla-exp*`!}` |	%Application of syn. function
	Id`(` [*optla-exp* { `o`*ptla-exp* }*] `)` |	%Application of static function
	`(` *optla-exp* `)` |	%Bracketning
	`-` *optla-exp* |	%Negation
	`!` *optla-exp* |	%Logical NOT
	`+` *optla-exp* |	
	optla-exp `-` *optla-exp* |	%Subtraction
	optla-exp `+` *optla-exp* |	%Addittion
	optla-exp `*` *optla-exp* |	%Multiplication
	optla-exp `/` *optla-exp* |	%Integer division
	optla-exp `%` *optla-exp* |	%Remainder

```
            optla-exp ^ optla-exp |                              %Exponentation
            optla-exp . optla-exp |                              %String concatenantion
            optla-exp && optla-exp |                             %Logical lazy AND
            optla-exp || optla-exp |                             %Logical lazy OR
            optla-exp | optla-exp |                              %Bitwise OR
            optla-exp & optla-exp |                              %Bitwise AND
            optla-exp glb optla-exp |                            %a ⊓ b
            optla-exp lub optla-exp |                            %a ⊔ b
            optla-exp = optla-exp |                              %Equality
            optla-exp != optla-exp |                             %Inequality
            optla-exp  { < | > | >= | >= }  optla-exp |
                                                                 %Comparison operators
            optla-exp \ [ [ update { , update }*  ] ] |          %Function update
            [ -> optla-exp ] |                                   %Function with default value
            optla-exp : optla-exp |                              %List cons
            optla-exp ++ optl-exp |                              %List concatenation
            optla-exp ~ optla-exp |                              %Add element to set
            optla-exp # optla-exp |                              %Subtract element from set
            optla-exp ? optla-exp |                              %Element of set predicate
            optla-exp!int!int |                                  %Tupel select. Tupel!Element!Arity
            ( optla-exp { , optla-exp }+ ) |                     %Tupel
            up-Id-int(optla-exp) |                               %Element to disj. union
            down-Id-int(optla-exp) |                             %disjoint union to element
            is-Id-int(optla-exp) |                               %Is disj. union of element type
            val-Id(optla-exp) |                                  %Synatctical type access
            type $ optla-exp |                                   %Type specification
            @  |                                                 %DF value at node
            [ optla-exp !! zf { , zf }* ] |                      %List ZF
            { optla-exp !! zf { , zf }* } |                      %Set ZF
            [ optla-exp -> optla-exp !! zf { , zf }* ]           % Function ZF

zf        ::= gen |                                              %generator
            ldef |                                               %local definition
            if optla-exp                                         %Filter

ldef      ::= pattern = optla-exp

gen       ::= pattern <- optla-exp |                             %List generator
            pattern <-- optla-exp  |                             %Set generator
            pattern <- optla-exp \ optla-exp                     %Function generator

update    ::= optla-exp -> optla-exp                             %Update argument, value pair

let-def   ::= Id = optla-exp; |                                  %Local definition
```

Id <u>`<=`</u> *optla-exp*<u>;</u> %Strict local definition

case-pattern::= [*pattern* { <u>,</u> *pattern* }*] <u>`=>`</u> *optla-exp*<u>;</u>

10.2 Additional Restrictions

10.2.1 `PROBLEM` section

In the `PROBLEM` section of the program the `direction`, `carrier`, `init`, `init_start`, `equal`, `combine` and `widening` commands must all be present (each one exactly once), but the order of the commands is free.

10.2.2 Scoping

The scope of an identifier is the part of the program, where it is *visible*. An identifier may only be used (in an expression or type), if it is visible. New identifiers[2] are introduced by definitions of sets/lattices in the `.set` file, definitions of syntactic types/constructors in the `.syn` file, static function definitions, definitions in the `GLOBAL` and `NODE` sections, variables in patterns (i.e. in transfer functions, static function definition parameters, case expressions and ZF expressions) and by `let` constructs.
A visible name may be hidden by introducing the same name again in the scope of that name, so that any references to the name reference the second introduction.
Apart from this hiding, the scope of a name is defined by some rules:

- Type names (i.e. primitive types, sets/lattices defined in the `.set` file, type names from the `.syn` file) are only visible in type expressions: *type*, `carrier`, `up-`*id*`-`*int*`()`, `down-`*id*`-`*int*`()`, `is-`*id*`-`*int*`()`, `val-`*id*`()`.
- Non type names are not visible in type expressions.
- Names of syntactic constructors from the `.syn` file are only visible in *pattern*.
- Names of static functions (including names of predefined functions) are visible in *optla-exp*, `widening`, `combine` and `equal` commands.
- Names from `GLOBAL` definitions (e.g. `a` in `a: unum`) are visible in *optla-exp* expressions.

[2]or *names*

- Names from the `NODE` section are visible in *optla-exp* expressions in the `TRANSFER` section.
- Names from pattern in static function definitions (i.e. from arguments of the definition), are visible in the body of the definition.
- Names from `case` patterns are visible in the branch of the `case`, where the pattern occured, e.g. `x:xs => x;`.
- Names from `let` expressions are visible in the body of the let expression *only*. I.e. `let x=2; in let x=1; y=x+1; in y` evaluates to `3`, not `2`.
- In ZF expressions, names can be introduced by the pattern of generators or local definitions.
  The *head* of a ZF expression is defined to be:
  $$head(zf) = \begin{cases} h & \text{, if } zf = \texttt{[}h\texttt{!!}zfs\texttt{]} \\ h & \text{, if } zf = \texttt{\{}h\texttt{!!}zfs\texttt{\}} \\ a & \text{, if } zf = \texttt{[}a\texttt{->}\ b\texttt{!!}zfs\texttt{]} \end{cases}$$
  Names in patterns of generators or local definitions of a ZF expression zf are visible in $head(zf)$ and in all generators, local definitions and filters *to the right* of the generator/local definition.
- The number of patterns in each branch of the `case` expression must be the same and coincide with the number of expressions after the `case` keyword.

10.2.3 Scope intersection

There must not be two introductions of a name with the same scope. This means that overloading of names is not allowed. Here, multiple definitions for the same static function name are considered as multiple cases in the same definition of the function.

10.2.4 Typing

optla-exp must be typed correctly, according to the type system of section 10.3. Static functions must possess a monomorphic type, i.e. their type must not contain type variables.

10.2.5 Data flow value

The actual data flow value `@` must only appear in the `TRANSFER` section.

10.3 FULA's type system

The type system of FULA is a restricted version of the **ML** type system, as described in [Ull94, MTH90, Mil78].
In essence, it is a monomorphic type system, i.e. the types of static functions must not contain type variables. To assure an efficient implementation, FULA further demands that every expression has a type that is either a primitive type, a syntactical list or a type defined in the `.set` and `.syn` files. This allows to produce efficient code for FULA expressions.
As noted already, FULA knows about the primitive types `snum, unum, real, bool, chr, str`. In addition, the sets and lattices defined in the `.set` file and the syntactical types, defined in the `.syn` file are known as basic types.
FULA knows some type constructors, which are used in building types. The type of a program is inferred automatically by PAG, using the inference algorithm of [Mil78]. Anyhow, sometimes types have to be specified to ensure that the resulting type of a function is monomorphic (e.g. `silly(snum $x)=x;` has type `snum` $\rightarrow$ `snum`, whereas `silly(x)=x` possesses the polymorphic type $\alpha \rightarrow \alpha$). The type constructors are:

- Static function type. A function, taking arguments of types $\tau_1, \ldots, \tau_n$ as arguments and returning a value of type τ has type $\tau_1, \ldots, \tau_n \rightarrow \tau$.
- A dynamic function, mapping arguments of type τ_1 to values of type τ_2 has type $\tau_1 \Rightarrow \tau_2$.
- A tupel with elements of types $\tau_1, \ldots, \tau_n$, with $n \geq 2$ has the type $(\tau_1, \ldots, \tau_n)$, where the brackets are used as type constructors.
- A list with elements of type τ has type $[\tau]$.
- A syntactical list of elements of type τ has type $*\tau$.
- A set or lattice A, defined in the `.set` file has type $\{A\}$.
- A lattice A, which is a lifted lattice of lattice B, i.e. $A = lift(B)$ has type $lift(B)$.

For an expression or program to be typed correctly, it has to possess a type, constructed from the above constructors and the basic types. There are some rules, how the type of an expression is made up from the types of it components:

- If `f(`$e1$`,...,`e_n`)` is the application of a static function `f` to arguments e_i, then it has the type τ, denoted by `f(`$e1$`,...,`e_n`)`$::\tau$, iff $e_i :: \tau_i \forall_i$ and `f`$::\tau_1, \ldots, \tau_n \rightarrow \tau$.
- A similar rule is valid for dynamic functions.
- If `(`e_1`,...,`e_n`)` is a tupel and $e_i : \tau_i \forall_i$ then the tupel has the type $(\tau_1, \ldots, \tau_n)$.
- If l is a list of type $[\tau]$ and $e :: \tau$ then e`:`$l::[\tau]$.
- The same is true for syntactical lists.
- If `e` is an enumerated element from a set A in the `.set` file, then `e`$::\{A\}$.
- The constant `all` has the type $\{\alpha\}$, it denotes the full set as an element of the power set. I.e. if `foo = set(bar)` is in the `.set` file, then `all` can be instantiated to {`bar`}.
- `let x`$_1$`=`e_1`;...x`$_n$`=`e_n`; in` $e::\tau$, iff $\forall_i : e_i :: \tau_i$ and $e :: \tau$, where `x`$_i::\tau_i$ is assumed in e. If there is one or more definition of the form `x`$_i$`<=`e_i, then e must have the same lattice type $lift(A)$ as e_i and in e, `x`$_i::A$ is assumed.
- `case` e_1`,...,`e_n `of` $::\tau$, iff
  p_1^1`,...,`p_n^1 `=>` e'_1`;`
  $\vdots$
  p_1^k`,...,`p_n^k `=>` e'_k`;`
  `endcase`
  $e_i :: \tau_i$ and $\forall_i : e'_i :: \tau$ and $\forall_{i,j} : p_i^j :: \tau_i$.
- `if` e_1 `then` e_2 `else` $e_3::\tau$, iff e_1`::bool` and $e_2, e_3 :: \tau$.
- Unsigned numbers, e.g. `123U` have type `unum`.
- Signed number have type `snum`.
- Real numbers have type `real`.
- The boolean values `true` and `false` have type `bool`.
- Characters have type `chr`.
- Strings have type `str`.
- If `A =A`$_1$`++`$\cdots$`++A`$_n$ is a set definition in the `.set` file then:

- $\texttt{up-A-}i\texttt{(}e\texttt{)}{::}A$, iff $e{::}A_i$.
- $\texttt{down-A-}i\texttt{(}e\texttt{)}{:}A_i$, iff $e{::}A$.
- $\texttt{is-A-}i\texttt{(}e\texttt{)}{::}\texttt{bool}$, iff $e{::}A$.

- If `A` is a syntactical alias type from the `.syn` file, i.e. there is a line $\texttt{A==}A'$, where A' is a FULA primitive type, then $\texttt{val-A(}e\texttt{)}{::}A'$, iff $e :: A$.

- ZF expressions.
  - $\texttt{[}h\texttt{!!}\mathit{zfs}\texttt{]}{::}[\tau]$, iff the *zfs* are well typed and $h :: \tau$
  - $\texttt{\{}h\texttt{!!}\mathit{zfs}\texttt{\}}{::}\{\tau\}$, iff the *zfs* are well typed and $h :: \tau$
  - $\texttt{[}h\texttt{->}d\texttt{!!}\mathit{zfs}\texttt{]}{::}\tau_1 \Rightarrow \tau_2$, iff the *zfs* are well typed and $h :: (\tau_1, \tau_2)$ and $d :: \tau_2$.

  For *zfs* the following rules must be met:

  - $\texttt{if }e.\ e :: \texttt{bool}$
  - $p\texttt{=}e.\ p :: \tau, e :: \tau$
  - $p\texttt{<-}e.\ p :: \tau$ and $e :: [\tau]$
  - $p\texttt{<--}e.\ p :: \tau$ and $e :: \{\tau\}$
  - $p\texttt{<-}e_1\backslash e_2.\ p :: (\tau_1, \tau_2)$, $e_1 :: \tau_1 \Rightarrow \tau_2$ and $e_2 :: \tau_2$

As already stated, every expression in a FULA/OPTLA program must possess a type and futhermore, this type must be either primitive, a syntactical list or the type must have a definition in the `.set` file, where the right hand side of that definition is equal to the type.
As a special case, the arithmetic operators have to be considered. They all take numbers as arguments, anyhow as there are three number types (`snum`, `unum` and `real`) the arguments must all be of the same type, i.e. all `unum` or all `snum`, etc.

10.4 Semantics of FULA constructs

In this section we only give an informal semantics of the basic FULA constructs. The semantics of the predefined functions is given in section 10.6.

- e `<<` f, e `>>` f: The bitshift operators. The numeric value of e is shifted f positions to the left (`<<`) or right (`>>`). f must denote an unsigned integer value.

- `print(`e`)` f: This prints the value of e to the terminal and evaluates f and returns that value.

- `if` b `then` e `else` f `endif`: The conditional evaluates b. If it is equal to `true`, then the value of e is returned, otherwise the value of f is returned.

- `case` e_1`,...,`e_n `of`
  p_1^1`,...,`p_n^1 `=>` f_1`;`
  $\vdots \quad \vdots \quad \vdots$
  p_1^k`,...,`p_n^k `=>` f_k`;`
  `endcase`
  First, the values of the e_i are computed. Then these values are matched against p_i^1, i.e. the value of e_1 is matched against p_1^1, the value of e_2 is matched against p_2^1, etc. If all these matches succeed, then f_1 is evaluated, where the variables from the patterns p_i^1 are first bound to the values from the match. The value of f_1 is the result in this case. If any of the matches failed, then the next branch p_i^2 is tried in the same way. If no branch was successfull, then the result of the expression is undefined.

- `let` $\mathtt{x}_1 \oplus e_1$`;...;` $\mathtt{x}_n \oplus e_n$`;` `in` e, where $\oplus \in \{$`=`,`<=`$\}$. Here, the e_i are evaluated from left to right. If j is an definition $\mathtt{x}_j$`<=`e_j, then the value of e_j is checked against $\{\bot, \top\}$. If it is one of these values, then this value is returned as the value of the `let` expression. If it is not, then `drop(`e_j`)` is taken to be the value of the variable $\mathtt{x}_j$ in the expression e afterwards and evaluation of $e_i, i > j$ continues. After all the e_i have been evaluated, the expression e is evaluated, where the $\mathtt{x}_i$ are bound to the values of e_i (or `drop(`e_i`)`). This value is the value of the `let` expression in this case.

- The arithmetic operators `-, +, *, /, %, ^` have the usual meaning.

- The bitwise AND, OR operators `&, |` take the bitwise AND/OR of the unsigned integer arguments: `96U & 40U`$= 32$.

- The logical NOT `!` inverses it's argument, e.g. `!true`$= false$.

- The logical AND/OR operators work on boolean values. They are lazy and evaluate their arguments only as far as is needed to determine the result, so e.g. in `true ||` e, e is not evaluated.

- $e.f$: The string concatenation operator returns the concatenated strings, e.g. `"Hello"." World!"`="Hello World!".

- e `lub` f: This evaluates e and f to their values e', f' and builds $e' \sqcup f'$ as result. e `glb` f works analogous.

- e`=`f: Equality test. This compares the values of e and f for equality and returns `true` in the positive case, `false`, otherwise. `!=` tests in the same way for inequality.

- `<,>,<=,>=`: The comparison operators compare their arguments, returning `true` or `false`. If the arguments are lattices, the ordering of the lattice is used, otherwise the natural ordering (e.g. $\leq$ for numbers) is used.

- A function update e`\[`e_1`->`f_1`,`$\ldots$`,`e_n`->`f_n`]` evaluates the dynamic function e and the new argument/value pairs (e_i, f_i) and returns the dynamic function that is equal to e, but returns for arguments e_i the value f_i.

- The default dynamic function `[->` e`]` returns for all arguments the value of e.

- e`++`f: The list concatenation operator evaluates the lists e and f and returns the concatenated list: `[1,2,3]++[4,5,6]`$= [1, 2, 3, 4, 5, 6]$.

- The set addition e`~`f evaluates e and f and adds the element that is the value of f to the set e, if it is not already in the set.

- The set subtraction e`#`f works analogous by removing the element f from e.

- The set test operator f`?`e tests, if the element, denoted by the value of f is a member of the set e and returns `true` in this case, `false` otherwise.

- `(`e_1`,`$\ldots$`,`e_n`)!`i`!`n: The tupel select operator selects the ith element of the n-tupel: `(1,2,3,4)!2!4`$=2$.

- `up-`id`-`i`(`e`)`: Evaluates e and converts the result to a disjoint set. If we look at disjoints sets as sets with elements of (tag, value) pairs, then the semantics can be given by: `up-`id`-`i`(`e`)`$= (i, e')$, where e' is the value of e.

- `down-`id`-`i`(`e`)`: Dropping from a disjoint set can likewise be written `down-`id`-`i`(`e`)`$= e'$, iff the value of e is (i, e').

- is-id-i(e)= $\begin{cases} \texttt{true} & \text{, if } e' = (i, e'') \\ \texttt{false} & \text{, otherwise} \end{cases}$, where e' is the value of e.
- val-id(e): This is the cast value of e, where the casted value of e is defined as the return value of the interface **C** function id_get_value(e).
- @: this is the data flow value at the current CFG node.
- ZF expressions. The head of an ZF expression is evaluated for every element, generated by the generators in the body of the ZF expression. The result is the list (set, dynamic function), built from all these values. Values may be filtered out by filters (if e). A value is generated only, if all filters evaluate to true. Local definitions in the body of the ZF expression (p=e) introduce new name bindings, the expression e is reavaluated for every (possible) generation of a new value and matched against the pattern. If the match failed, no value will be generated. Generators (p<-e, p<--e, p<-e_1\e_2) generate new values. First the expressions are evaluated, then the pattern is matched against the value. If the match succeeds, the names in the pattern are bound. For function generators, the generated value must be a pair (a, b). If b is equal to the value of e_2, then no value is generated.

  Generators to the right vary faster, than those at the beginning of the body. The order of generation is only defined for list generators: values are taken from the list e, in the order of the list. So the expression [i!!i<--{1,2}] may evaluate to [1,2] or [2,1].

  In function ZFs, the default element d is evaluated only once, at the beginning of the ZF evaluation. It denotes the default element of the function, i.e. for every argument value, for which no argument, value pair is generated by the zfs, the function maps that argument to the value of d.

10.5 Priorities

Table 10.1 gives the priorities of the operators. The entries are ordered from lower to higher priorities.

10.6 Predefined functions

FULA provides the following functions (where 'num' stands for one of snum, unum, real).

<table>
<tr><th>Level</th><th>Associativity</th><th>Operator</th></tr>
<tr><td>1</td><td></td><td><-, <--</td></tr>
<tr><td>2</td><td></td><td>let, in</td></tr>
<tr><td>3</td><td>left</td><td>||</td></tr>
<tr><td>4</td><td>left</td><td>&&</td></tr>
<tr><td>5</td><td>left</td><td>|</td></tr>
<tr><td>6</td><td>left</td><td>&</td></tr>
<tr><td>7</td><td>left</td><td>glb, lub</td></tr>
<tr><td>8</td><td></td><td>=, !=</td></tr>
<tr><td>9</td><td></td><td><, >, <=, >=</td></tr>
<tr><td>10</td><td>left</td><td><<, >></td></tr>
<tr><td>11</td><td></td><td>@</td></tr>
<tr><td>12</td><td>left</td><td>++</td></tr>
<tr><td>13</td><td>left</td><td>+, -</td></tr>
<tr><td>14</td><td></td><td>?</td></tr>
<tr><td>15</td><td>left</td><td>#, ~</td></tr>
<tr><td>16</td><td>right</td><td>:, ::</td></tr>
<tr><td>17</td><td>left</td><td>*, /, %</td></tr>
<tr><td>18</td><td>left</td><td>^</td></tr>
<tr><td>19</td><td>left</td><td>unary -, unary +, !</td></tr>
<tr><td>20</td><td>left</td><td>.</td></tr>
<tr><td>21</td><td>left</td><td>\</td></tr>
<tr><td>22</td><td></td><td>as</td></tr>
<tr><td>23</td><td></td><td>typecast, [ ], {}, {! !}</td></tr>
</table>

Table 10.1: FULA operator priorities

- `sgn`::`snum`→`snum`. The signum function. It takes a number n as argument and returns $\begin{cases} -1 & , \text{if } n < 0 \\ 0 & , \text{if } n = 0 \\ +1 & , \text{if } n > 0 \end{cases}$
- `error`::`str`→ α. The error function evaluates and prints it's argument to the terminal and exits the analyzer with an error. This function can take any type and thus occur at any position within an expression.
- `abs`::`snum`→`snum`. This function returns the absolute value of it's numeric argument.

- `trunc`::`real` $\rightarrow$ `snum`. This function takes a real number as argument and returns the portion before the decimal point, e.g. `trunc(4.5)`=`4`.

- `crunch`::$(\alpha \Rightarrow \beta), (\alpha \Rightarrow \beta), (\beta, \beta \rightarrow \beta) \rightarrow (\alpha \Rightarrow \beta)$. The crunch function allows the application of a static function to the results of dynamic function application. E.g. in `crunch(f,g,h)`, `f,g` are dynamic functions and `h` is the name of a static function. The semantics of the crunch function can be defined by `crunch(`f`,`g`,h)`$= \lambda x.\mathtt{h}(f(x), g(x))$. The result of the crunch function is a dynamic function.

- `reverse`::$[\alpha] \rightarrow [\alpha]$. This function reverses it's argument list, e.g. `reverse([1,2,3,4])`=`[4,3,2,1]`.

- `tl`::$[\alpha] \rightarrow [\alpha]$. The tail function returns the tail of it's argument list, i.e. the list without the first element: `tl([1,2,3])`=`[2,3]`. Application of `tl` to the empty list generates a run time error.

- `hd`::$[\alpha] \rightarrow \alpha$. The head function returns the first element of it's argument list:`hd([1,2,3])`=`1`. Application of `hd` to the empty list generates a run time error.

- `drop`::$lift(\alpha) \rightarrow \alpha$. This drops one element from a lifted lattice to the base of the lattice. E.g., if `A = lift(B)` and `a`::`A` then `drop(a)`=`a`, if `a`$\neq \top, \bot$. If `drop` is applied to the bottom or top element of `A`, then a run time error results.

- `lift`::$\alpha \rightarrow lift(\alpha)$. This is the inverse operation to `drop`. The function lifts an element of `B` to an element of `A`.

- `rnd`::`real`$\rightarrow$`snum`. This function rounds the real number argument to the nearest signed integer: `rnd(1.2)`=`1`, `rnd(1.7)`=`2`.

- `strlen`::`str`$\rightarrow$`snum`. This function returns the number of characters in it's string argument, e.g. `strlen("Hello World!!")`=`13`

- `union`:: $set(\alpha), set(\alpha) \rightarrow set(\alpha)$. This is union on sets.

- `intersect`:: $set(\alpha), set(\alpha) \rightarrow set(\alpha)$. This is set intersection.

10.7 Advanced features

For the sake of efficiency, PAG allows to store the result of function evaluations (both transfer functions and static functions) in a hash table (or, alternatively, in a cache). Further evaluations of the function will first look in the hash table (cache), if the function has already been called with the present arguments. If this is the case, the stored result will be reused, instead of recomputed. This can cause a vast speed improvement, at the expense of a higher memory usage.

In the grammar, the transfer functions can be given a hash specification (in *transfer-spec*): `TRANSFER` *hash* .... A hash specification may either be a hash table definition `hash [`*number*`]` or a cache definition `cache [`*number*`]`. The *number* gives the size of the hash table or cache in numbers of entries. If it is omitted, a default value is used.

Each static functions can be given a separate hash specification (see *fkt-def* in the grammar).

If a hash table is to be used, a hash value is calculated from the arguments of the function call. This hash value is used as an index into the hash table. Each hash table entry consists of a linked list of *(argument,value)* pairs. This list is searched for the actual argument values. If they are found, the corresponding value is retrieved and returned. If not, the function is evaluated and the result stored in the list.

A cache works differently, as it only stores the most recent argument,value pairs. Again, a hash function is applied to the actual arguments of a to be evaluated function call. The value is used as an index into the cache. Each entry contains at most *one* argument,value pair. The actual arguments are compared against the arguments in the entry. If they are the same, then the value from the entry is returned. If not, then the function is evaluated and the argument,value pair *overwrites* the entry in the cache and is then returned.

Note, however that through the presence of the `print` primitive, FULA is not totally referntial-transparent and though the use of the hash mechanism may cause `print` commands not to be executed.

10.8 Calling **FULA** functions from C

As already pointed out in section 8.4, PAG generates for every DATLA data type N a **C** typedef `o_`N.

In addition, PAG generates for every FULA function FN of type

$$FN\texttt{::}AT_1, \ldots, AT_n\texttt{->}RT$$

a **C** function with prototype

$$\texttt{o\_}RT\ \ \texttt{o\_}FN\texttt{(o\_}AT_1\texttt{,...,o\_}AT_n\texttt{)}$$

This function may be called from any **C** fragment, yielding the same result as an evaluation of the FULA function in the `.optla` file.

Chapter 11

Interface Reference

11.1 Types

11.1.1 CFG

The following types for the CFG have to be defined as **C** types.

Name	Description
KFG	The CFG itself.
KFG_NODE_TYPE	A type, describing the type of a node, must be an enumeration type.
KFG_NODE_ID	A type, describing the identifier of a node. In most cases this will be `int`.
KFG_NODE	A node in the CFG.
KFG_NODE_LIST	A type, representing a list of nodes.
KFG_EDGE_TYPE	A type, describing the type of edges between nodes. Probably an enumeration type.

Please note that the `KFG_NODE_TYPE` enumeration type *must* at least contain the enumeration constants `RETURN`, `CALL`, `ENTRY` and `EXIT`. And furthermore, every `Call`, `Return`, `Entry` and `Exit` node in the CFG must be assigned the appropriate enumeration constant by the frontend.

11.1.2 Syntax tree

For every type t from the `.syn` file, including the alias types, a **C** type with the same name has to be provided. If syntactical lists are used, e.g. `mirSymbol*`, a **C** type for the list has to be given. The name of that type must be `LIST_`T, where T is the name of the type to build the list on (e.g.

`mirSymbol` in the prior example). As an example, consider the program fragment in figure 3.3: here at least **C** types `mirStmt`, `mirExpr`, `LIST_mirSymbol`, `CHAR` and `INT` have to be provided. The definition of these types must be made available by including the file `syntax_tree.h`.

11.2 Functions

In the following, we present the functions, every frontend has to provide, grouped by functional category.

11.2.1 CFG manipulation

To access the CFG, the frontend has to provide the following functions.

Prototype	Description
`KFG kfg_get(void)`	Returns the actual CFG.
`KFG kfg_create(KFG)`	Initialize the CFG.
`int kfg_anz_node(KFG)`	Returns the number of nodes in the CFG.
`int kfg_edge_type_max(void)`	Returns the number of different edge types.
`int kfg_proc_anz(KFG)`	Returns the number of procedures in the CFG.
`KFG_NODE kfg_entry(KFG)`	Returns the entry node of the main procedure.
`KFG_NODE kfg_exit(KFG)`	Returns the exit node of the main procedure.
`char *kfg_proc_name(int` *i*`)`	Returns static pointer to the name of procedure *i*.
`KFG_NODE_LIST kfg_all_nodes(KFG)`	Returns list of all nodes.
`KFG_NODE kfg_numproc(KFG, int` *i*`)`	Returns entry node of procedure *i*.
`KFG_NODE_LIST kfg_entrys( KFG)`	Returns list of all entry nodes.
`KFG_NODE_LIST kfg_calls( KFG)`	Returns list of all call nodes.
`KFG_NODE_LIST`	Returns a list of all return nodes.
continued on next page	

continued from previous page

Prototype	Description
kfg_returns (KFG)	
KFG_NODE_LIST kfg_exits(KFG)	Returns list of all exit nodes.
KFG_NODE node_list_head(KFG_NODE_LIST)	Returns head of list.
KFG_NODE_LIST node_list_tail(KFG_NODE_LIST)	Returns list without the first element.
int node_list_is_empty(KFG_NODE_LIST)	Returns 1, if the list is empty, 0 otherwise.
int node_list_length(KFG_NODE_LIST)	Returns length of node list.
int kfg_procnumnode(KFG, KFG_NODE)	Returns the number of the procedure, the node belongs to.
KFG_NODE_TYPE kfg_node_type(KFG, KFG_NODE)	Returns the type of the node.
KFG_NODE_ID kfg_get_id(KFG, KFG_NODE)	Returns the identifier of the node.
KFG_NODE kfg_get_node(KFG, KFG_NODE_ID)	Returns the node with the given identifier.
int kfg_procnum(KFG, KFG_NODE_ID)	Returns the number of the procedure, the node with the given identifier belongs to.
char * kfg_node_infolabel_print(KFG_NODE)	Returns a static string, holding the description of the given node.
KFG_NODE_LIST kfg_predecessors(KFG, KFG_NODE)	Returns the list of predecessors of the node.
KFG_NODE_LIST	

continued on next page

continued from previous page	
Prototype	Description
kfg_successors(KFG, KFG_NODE)	Returns the list of successors of the node.
KFG_EDGE_TYPE kfg_edge_type(KFG_NODE s, KFG_NODE t)	Returns the type of the edge from node s to node t.
int kfg_which_in_edges(KFG_NODE)	Returns a bitmask, where for every edge type, that labels an incoming edge, the corresponding bit is set. See below.
int kfg_which_out_edges(KFG_NODE)	As kfg_which_in_edges, but for outgoing edges.
KFG_NODE kfg_get_call(KFG, KFG_NODE r)	Returns the call node, belonging to the return node r.
KFG_NODE kfg_get_return(KFG, KFG_NODE c)	Returns the return node, belonging to the call node c
KFG_NODE kfg_get_start(KFG, KFG_NODE e)	Returns the start node, belonging to the end node e.
KFG_NODE kfg_get_end(KFG, KFG_NODE s)	Returns the end node, belonging to the start node s.

The functions kfg_which_in_edges and kfg_which_out_edges return a bit mask, where a bit is set for every edge type, that labels an incoming (outgoing) edge.
A sample implementation of kfg_which_in_edges is shown in figure 11.1. Likewise, the function kfg_which_out_edges can be obtained, by replacing the function kfg_predecessors by kfg_successors and replacing kfg_edge_type(tar,node) by kfg_edge_type(node,tar).

11.2.2 Synatxtree manipulation

For every constructor $c(n_1 : t_1, \ldots, n_k : t_k)$ of type t in the .syn file, functions

$$t_i \;\; t\texttt{\_}c\texttt{\_get\_}n_i(t)$$

and

$$\texttt{int is\_op\_}t\texttt{\_}c(t)$$

```
int kfg_which_in_edges(KFG_NODE node)
{
  KFG_NODE tar;
  KFG_NODE_LIST preds = kfg_predecessors(kfg_get(), node);
  int erg=0;

  while (!node_list_is_empty(preds)) {
    tar = node_list_head(preds);
    preds=node_list_tail(preds);
    erg |=(1<< kfg_edge_type(tar,node));
  }
  return erg;
}
```

Figure 11.1: Sample implementation of `kfg_which_in_edges`

have to be provided. The first family of functions accesses elements of a syntax tree, the second tests, if a syntax tree is labeled with the constructor *c*. The functions return `1` in this case, `0` otherwise.
For every alias type *t*==*p* form the `syn` file, a function

`char *` *t*`_get_value(`*t*`)`

has to be provided, which returns the ASCII representation of the alias type, which can then be converted into the primitive FULA type *p* by the system. E.g. for the program fragment in figure 3.3 functions

```
mirExpr mirStmt_CFG_Call_get_exp(mirStmt)
mirType mirExpr_CFGEndCall_get_type(mirExpr)
LIST_mirSymbol CFGEndCall_get_sym(mirExpr)
char * CHAR_get_value(CHAR)
```

have to be provided (among others).
For every syntactcal list, e.g. `mirSymbol *` in figure 3.3, a function to test for emptiness of the list, functions to access the head and the tail of the list have to be provided. If *T* is the type the list is based on (e.g. `mirSymbol` in the above example), the following functions must be defined:

Prototype	Description
`int LIST_`*T*`_empty(LIST_`*T*`)`	Returns `1` if the list is empty.
T `LIST_`*T*`_hd(LIST_`*T*`)`	Returns the head of the list.
`LIST_`*T* `LIST_`*T*`_tl(LIST_`*T*`)`	Returns the tail of the list.

Last but not least, a function has to be provided that allows to access the syntax tree, associated with each CFG node. It is called `NO`:

$$t\ \texttt{NO(KFG\_NODE)}$$

Here t is the syntax tree type, given with the `START` statement in the `.syn` file (`mirStmt` in figure 3.3).

11.2.3 Global/local variables

For all variables, declared in the `GLOBAL` or `NODE` section of the `.optla` file, functions to access them have to be provided.
For all global variables declared by $v\!:\!t$, a **C** function

$$t'\ \texttt{global\_}v\texttt{(void)}$$

has to be provided. Here t' is the **C** type t, if t is a syntactical type from the `.syn` file. If it is a type from the `.set` file, then $t' = \texttt{o\_}t$.
For all local variables declared by $v\!:\!t$, a function

$$t'\ \texttt{kfg\_node\_get\_}v\texttt{(KFG\_NODE)}$$

has to be provided, where t' is build from the same rules as above.

Chapter 12

Examples

Chapter 13

Installation

The installation of PAG uses the GNU `autoconf` system ([Mac95]) and is thus very simple. Below, we descibe the necessary steps to get, unpack, configure and build the PAG system. By now, PAG has been successfully tested on the following architectures:

- SPARC stations under SunOS 4.1.3
- SPARC stations under Solaris 2.6
- PCs under Linux
- SGIs under IRIX 6.2
- Amiga under NetBSD 1.2 (should run on any NetBSD system!)

However, PAG will probably run under most UNIX versions and machines, but may require adjustment for system dependent details.

13.1 Getting PAG

PAG is available by anonymous ftp from .... To obtain it, do something like

```
ftp ......
[ftp messages]
login:  anonymous
Password:  your email address
[more messages]
ftp> bin
200 Type set to I.
ftp> cd .....
250 CWD command successful.
ftp> get PAG.tgz
[even more messages]
ftp> bye
```

13.2 Unpacking PAG

`cd` into a appropriate directory, move the `PAG.tgz` file there and type:

```
gzip -d PAG.tgz
tar xvf PAG.tar
```

This generates a directory `PAG` and several subdirectories and files.

13.3 Configuring PAG

Just `cd PAG` and there type `./configure`, this will automatically configure PAG for your system. Try `./configure --help` for an explanation of the various options of `configure`. Just typing `./configure` configure PAG for installation under `/usr/local`. If you want to install it with another prefix, just do `./configure --prefix=`*`prefix`* instead. In the following, we will refer to the prefix, under which PAG is to be installed as *prefix* (which defaults to `/usr/local`, as already mentioned).
`configure` will print various messages while it scans your system to determine various machine and OS dependent parameters.
Should `configure` abort with an error message, then PAG cannot be build on your machine. Contact one of the authors in this case.

13.4 Building PAG

After successfuly configuration, just type `make`. This will build and install the various parts of the system, including the `clax`[1] example frontend. If

[1]CLAX is a PASCAL like language.

you also want to build the `C` and `executable` frontend, type `make c` and/or `make exec`.
The installation process creates various directories:

- *prefix*`/share/pag`. This directory contains various PAG data files.
- *prefix*`/lib/pag`. This directory contains link libraries used in linking the generated analyzers.
- *prefix*`/libexec/pag`. This directory contains some programs, executed by `pag`
- *prefix*`/include/pag`. This directory contains various include files, needed to compile an analyzer.

In addition, the generator itself, `pag`, is installed in *prefix*`/bin`

13.5 Examples

The original directory, where you did the `configure`, contains a subdirectory `EXAMPLES`, which contains various example analyzer definitions.

13.6 Environment Variables

It is usefull to set the environment variable `PAGDIR` to the value of *prefix*. If you write your own Makefiles for analyzers later, the locations of the various include files, link libraries, etc. can then be specified relative to `${PAGDIR}`.

Chapter 14

Options Summary

This chapter lists and explains all options of PAG. This includes both the command line options to the `pag` program, as well as the compile options and the run-time options, which can be used to influence an analyzer at run-time.

14.1 Command line options

As already explained in section 1.1.1, `pag` takes exactly 4 arguments

`pag` *frontend iterator analysis target-dir*

frontend stands for the frontend to use. It determines the place of several files, which PAG includes and references. The place of the files is *prefix*`/share/pag/frontend/`*frontend*, where *prefix* is either `/usr/local` or the custom prefix, choosen while configuring PAG, see chapter 13. The files that are needed for each frontend are listed in chapter 11.
iterator is the iteration algorithm to use in the analyzer, see section 3 for a list of possible choices.
analysis is the prefix to a place, where the `.set`, `.syn` and `.optla` files can be found. These files are then located as *analysis*`.set`, *analysis*`.syn` and *analysis*`.optla`.
The last argument, *target-dir* is the path to the directory, where PAG creates the `src` directory, in which all generated files will be placed.

14.2 Compile-time options

When compiling the generated files, some options can be given by defining or undefining **C** preprocessor symbols via the `-D`*symbol*`=`*value* **C** compiler

flag.
PAG uses the following symbols:

Symbol	Description
`GC_MEMSIZE`	The value of this symbol, which is 8MB by default, determines the size of one memory bank for the FULA heap (see below).
`ANIM`	If defined, an animation of the analyzer (see section 16) may be created and is controlled by the run-time option `animation`, see below.
`DFI_WRITE`	If defined, a **C** function `dfi_write_()` has to be provided, which emits `vcg` node descriptions for debugging output, see chapter 16 for details.
`NDEBUG`	If defined, no assertion checking code is generated. Use this for the final version of the analyzer (see man page for `assert(3)` on your system.).
`NO_RCHECK`	If defined, no error checking code for access to lattice elements via `drop` is generated. Use in the final version of your code.

In addition, for every datatype T, one can define the symbols *T*`_WRITE` and *T*`_VCGWRITE`. These control, how elements of the given type T are printed.
If *T*`_WRITE` is defined, the user must provide his own implementation of the function `char *`*T*`_print(`*T el*`)`, which returns the string representation of *el*. If the symbol is not defined, a default implementation is generated by PAG.
If *T*`_VCGWRITE` is defined, the user must provide his own implementation of the function `char *`*T*`_vcgprint(`*T el*`)`, which returns a `vcg` description of element *el* (see chapter 16.3). If the symbol is not defined, a default implementation is generated by PAG.

14.3 Run-time options

Run-time options are realized as **C** variables in the generated analyzer. Their effects is defined by the values of these variables. In this section, we will

give the declarations of these variables, together with a description of their meaning.
There are three classes of options:

1. Analysis options. These influence the behaviour of the analysis itself.
2. Garbage collection options. These influence the behaviour of the garbage collector, which is part of the FULA language.
3. Debugging options. These control the existence and form of debugging.

14.3.1 Analysis options

These options were already discussed in chapter 3:

Declarartion	Description
`int cfg_ordering`	Selects the ordering to use on the work list.
`int J`	Selects the length of the longest call path to keep seperate, when `sel_mapping`=3.
`int sel_mapping`	Selects the mapping of call paths to the array of data values at each node.

For a more verbose description of the effects of these options, refer to chapter 3.

14.3.2 Garbage collection options

FULA, as most functional languages, allocates new data structures on a heap. There is no explicit disposal of data structures that are no longer needed. Instead, a garbage collector is invoked, whenever the heap fills up above a given level, who searches the active data and frees any unreferenced cells in the heap. This searching and freeing is called *garbage collection*. By 'GC' we refer in the following both to the garbage collection as well as the garbage collector.
Another feature of FULA is that common data structures in the heap can be *shared*. This means that at a GC, the system searches the whole heap for data structures that are identical and let all references to such a structure point to one instance of that structure, freeing the other instances. Since FULA is a referential transparent language, this is possible without changing the semantics of the language.
The behaviour of the GC in FULA can be influenced in several ways. To understand the effects of the options, one has to know that FULA allocates

place for the heap in terms of several *memory banks* of fixed size (8MB). All banks together form the heap.
The run-time options for the GC are listed in the following table.

Prototype	Description
`int startbanks`	This gives the number of banks to allocate for the heap at the initialisation of the analysis.
`int low_perc`	The value ($0 \leq, \leq 100$) gives the percentage of free heap, at which a GC is started. I.e. if the amount of free space in the heap is less than `low_perc`% of the total heap size, then a GC is initiated.
`int high_perc`	If, after a GC, less than `high_perc`% of space in the heap is free, a new bank is allocated and added to the heap.
`int share_min`	This holds the number of banks, which must be filled, before sharing in the heap is activated.
`int share_anz`	This variable holds the number of GCs, after which sharing is again performed (i.e. the distance between sharings).

14.3.3 Debugging options

As explained in chapter 16, there are five options to control the amount of debugging.

Declaration	Description
`int verbose`	If the value of this variable is set to 1, the analyzer prints out dots and some letters to the terminal, while it is running. Useful to validate that the analyzer is still alive and doing something.
`int debug_stat`	If the value of this variable is 0, no statistics is printed. If it's 1, some statistics is printed. If the value is > 1, in addition statistics about the usage of hash tables and/or caches in FULA is printed.
continued on next page	

continued from previous page

Prototype	Description
`char *animation`	This variable is only defined, if the analyzer was compiled with the `-DANIM` flag. If this is `NULL`, no animation of the changing data flow values at the nodes of the CFG are output. If it's not `NULL`, then it must hold the path of to a directory, in which the `vcg` files, describing the animation are stored.
`int global_print_post_info`	If 1, information after the nodes of the CFG are output.
`int global_print_pre_info`	If 1, information before the nodes of the CFG are output.
`int global_mapping_edges`	If 1, edges according to the mapping between data-flow values are output.
`int global_proc_as_graphs`	If 0, no supgraphs for procedures are output in the `vcg` files. If > 0, procedures are output as subgraphs in the `vcg` files. The latter requires `vcg` version ≥ 1.4.

Chapter 15

Error Messages

There are two kinds of messages that can occur:

- Warnings. They state that something is probably not right, but that PAG can continue. Anyhow the result may be quite different from what was intended.
- Errors. They state that something is definitely wrong and that PAG can not continue. Actually two kinds of errors exist:
  - Compile-time errors. They are discovered during the generation of an analyzer.
  - Run-time errors. They occur during the excution of the analyzer, e.g. by applying `drop` to `bot`.

In this chapter we will first present the overall structure of messages and then introduce the warnings and errors in the following sections. We will restrict ourselves to presenting the messages that are not self-explanatory.

15.1 Messages

Generally messages have the form shown in figure 15.1.

filename: *text* `in line` *line* `pos` *pos*
explanation

Figure 15.1: Format of messages.

Here, *filename* is the name of the source file, where the message occured. *text* is the text of the message, *line* and *pos* give the line number and position

in that line, related to the message. *explanation*, if present, gives further information.
Note however, that the *line* and *pos* parameters can only be considered as rough approximations to the real line number and position, where the message occured. That is, they should only be treated as hints for the region in the file to look at.

15.2 Warnings

The following warnings deserve explanation:

- `No unique mapping K=`K `proc_anz=`p: PAG cannot construct a mapping for the iteration algorithm, with the parameter K, which is $N^{\mathtt{J}}$, where N is the node list length and `J` is described in chapter 3. Solution: Increase the parameter `J`. This warning can only occur at the run-time of the analyzer.

- `Patterns never match (`t`)`: The patterns, given in the term t can never match and are thus useless. This is discovered at compile-time. Anyhow, note that this warning often occurs if previous patterns have been specified wrong.

15.3 Errors

15.3.1 Compile-time errors

Aside from obvious error messages the following exist:

- `function` f `has too many arguments`: The function named f is called with too many arguments (there can be at most 16 arguments).

- `sum type` t `has too many elements`: There can be at most 16 components of a sum type in a `.set` file.

- `Tuple too large`: A tuple with more than 16 elements was specified.

- `can't unify types`: There is a type error in the program. The message gives the types it has derived for some terms. These types should be unifyable[1] but are not. Type variables are annotated with capital letters. The location of the error is given as a range, e.g. `(3,1-3,16)` means line 3, positions 1 upto line 3, position 16 in the file.

[1] I.e. there should be a common instance of them.

- `underspecified type`: All types must be instantiated to monomorphic types in the end. This was not the case here.
- `unknown syntactic constructor '`c`'`: The syntax constructor c was used in a pattern but not defined in the `.syn` file.

15.3.2 Run-time errors

Interesting errors at run-time are:

- `arity too large (`a`) try shorter callstring length`: The specified callstring length (parameter `J`, see chapter 3) is too large.
- `parameter sel_mapping invalid`: An invalid value of `sel_mapping` was given, see chapter 3 for valid values.
- `no pattern matched in case expression`: No pattern of a `case` expression matched the argument.
- `no pattern matched in function` f: No pattern in the definition of function f matched the argument.
- l `is not a well formed lattice`: Somehow it was discovered that l, although defined as lattice, does not form a lattice. This can, e.g., happen if the lattice was defined by an enumeration.

Chapter 16

Debugging Analyzers

16.1 Overview

Debugging of analyzers is a hard task, due to the complex nature of program analysis. Old fashioned debugging methods, like `printf` statements, inserted at some places in the code or even more modern methods, as the use of graphical source level debuggers, like `ddd` e.g., are too low-level to give great help in the debugging process. Since PAG aims at generating analyzers from higher-level descriptions, a better suited debugging method has to be provided by the system.
PAG uses the approach to *visualize* the changing data flow values at the nodes of the CFG. This is done by emiting for each step of the iteration algorithm of the analyzer a graph description, representing the CFG and the actual data flow values. The graph description is annotated using the `vcg` language. `vcg` ([San95]) is a tool for the description and layout of graphs. See the `vcg` Manual for a detailed description of the language and program. The `vcg` files, output during iteration, can then be viewed in sequence using the `anim` tool, which is distributed with PAG. This tool takes as an argument the path to the directory, where the `vcg` files have been output. It then opens a window and displays the files in sequence, building an animation. So the changes of data flow values at the nodes of the CFG can be viewed and errors can be localized.
Beside this, PAG allows the printing of some statistics about the analyzer, which is output after the analyzer is finished. This statistics includes the time needed by the analysis, some information about the iteration and the garbage collection and memory usage, as well as information about the hash and/or cache efficiency for FULA functions.
In the next section we will describe, how this statistics can be output and

what they mean. The next section describes the printing of `vcg` graphs and the usage of the `anim` tool and how the graphs can be influenced by user defined functions.

16.2 Statistic output

For statistics to be enabled, the analyzer has to be compiled with the `-DANIM` **C** compiler flag. Then, before the analyzer is started, the global integer variable `debug_stat` has to be set to `1` or `2`, as shown in figure 16.1.

```
extern int debug_stat;

main()
{
KFG mykfg;
DFI_STORE dfi;
   ...
  debug_stat=1; /* Enable statistics */
  dfi=domin_doit(mykfg); /* Do the analysis*/
   ...
}
```

Figure 16.1: Enabling statistics

Setting `debug_stat` to `1` prints information about the analysis, as shown in figure 16.2.
The first line gives the times spent in inializing the analysis, doing the iterations and sorting the worklist. The number of iteration steps and nodes in the CFG is printed in the next line.

```
domin analyzer done in 3.13 sec
0.01sec initalizing 3.12sec iterating 0 sorting
48 steps, 45 nodes
0.437500 avg worklist tests, 1.458333 average worklistlength
=> 0.065 sec per step
0 sec for garbage collection
7284k Memory used total
Longest Chain of o_Nodes:  2^unum
```

Figure 16.2: Analysis statistics

Statistics about the worklist is printed in the following line. During iteration, it has often to be tested, if a node n is already in the worklist W: $n \in W$. The total number of comparisons $n = w$ for nodes in W divided by the number of tests $n \in W$ is the `avg worklist tests` number. The average length of the worklist is shown as the seond number. The remaining statistics is self explanatory.
By setting `debug_stat` to 2, additional statistics about the use of hash tables and chaches for FULA functions (see section 10.7) is printed. FULA can keep *(argument,value)* pairs for functions in hash tables or a cache. The usage of these for each function is printed by setting `debug_stat=2;`. See figure 16.3 for an example output for one function.

```
o_eq:
size=10
used=5
max=1
average=1.000000
cache hit=6
empty=5
cache quality=1.200000
```

Figure 16.3: Cache usage statistics

The figure shows the usage statistics for a cached function. It begins with the name of the function (with a prefixed `_o`). The the number of entries in the cache is shown. `used` gives the number of used entries in the cache. The values of `max` and `average` are always one[1]. The total number of cache hits is given by `hit`, the number of cache entries, which have never been used is given through `empty`. The quality of the cache is given as the ratio `hit/used`. For functions, which are hashed statistics as in figure 16.4 is printed.
Here `size` gives the number of slots in the hash table (each slot holds a list of elements, called *bucket*). `used` gives the number of entries used in the hash table. This is the sum of the length of all buckets. The length of the longest bucket is given by `max`, the average length by `average`. The number of hits in the hash table, i.e. the number of elements found in the table during the lookup is given by `hit`. The quality of the hash table is given by `quality` as the ratio `hit/used`. Sometimes it is important to see, if the replacement of the memory intensive hash table by a (fixed sized) cache is desirable. For this reason, PAG displays the number of cache hits, that would have occured,

[1]They represent the number of used entries per cache slot, which is always one in the current implementation.

```
o_eq:
size=10
used=47
max=12
average=9.400000
cache hit=6
empty=5
hit=6
quality=0.127660
cache quality=0.127660
```

Figure 16.4: Hash usage statistics

if a cache had been used, and the quality of that cache by `cache hit` and `cache quality`.
Another useful run-time variable is `verbose`. If set to 1, the analyzer prints dots and some letters, while it is running. This can be used to verify that the analyzer is still alive and running.

16.3 Output of CFGs

To enable the use of `vcg` output, the analyzer has to be compiled with the `-DANIM` **C** compiler flag.
The global char pointer `animation` holds the path of a directory, to which the `vcg` files are written during iteration. If `animation` is `NULL`, no `vcg` files will be written.
Beside from printing the graphs during iteration, the actual CFG and it's data flow values can be output by calling the **C** function `vcg_create()`[2].
Setting the integer variable `global_mapping_edges` to 1 includes edges between the data flow elements in the `vcg` graphs to be shown, according to the mapping in effect (see chapter 2.3).
The appearance of the `vcg` graph can be influenced by providing a function `dfi_write_()`: If the **C** preprocessor symbol `DFI_WRITE` is defined, the function

```
void dfi_write_( FILE *fp, char *name, char *attr,
                 dfitype dfivalue, int nodeid)
```

has to be provided, which emits the description for a node in the CFG. The arguments to the function are

[2]The printing of graphs during iteration also uses this function.

- `fp` A file pointer, referencing the file, to which the node description has to be printed.
- `name` A string, describing the name of the node.
- `attr` A string, describing the attributes of the `vcg` node.
- `dfivalue` This is of the type of the domain, underlying the analysis, e.g. `o_vars dfivalue`, if the line `carrier:  vars` was in the `.optla` file.
- `nodeid` This is the node id, as returned from the frontend interface for the node of the CFG.

The function has to output a `vcg` node description to the file, referenced by `fp`. The default output, that is used by PAG if `DFI_WRITE` was not defined, is shown in figure 16.5.

```
node:  {
  title:  "name"
  attr
  label:  "dfirep"
}
```

Figure 16.5: Default `vcg` node output

There, *name* and *attr* are the strings, pointed to by the `name` and `attr` arguments of `dfi_write_()`, respectively. *dfirep* is an ASCII string, representing the `dfivalue` argument[3].
To print one graph into a file, the function `vcg_create()` can be called. The prototype of this function is

```
void vcg_create(char *filename, int nodeid, int pos, int typ)
```

Here,

- `filename` is the name of the file, to which the graph is to be printed.
- `nodeid` is the number of the node in the CFG, which shall be highlighted in the `vcg` graph. If this is `-1`, no node is highlighted.

[3]Which is printed by calling `T_print_fp(dfivalue)` in the systems default output routine, see section 8.4.

- `pos` is the index into the data flow value array at the node, which should be highlighted.
- `typ` is the type of the edge, which shall be highlighted.

An example of the usage of both the iteration animation and the use of `vcg_create()` is shown in figure 16.6.

```
extern char *animation;
main()
{
KFG mykfg;
DFI_STORE dfi;
    ...
  animation="anim/"; /* enable animation */
  dfi=domin_doit(mykfg); /* do the analysis */
   /* print the result graph */
  vcg_create("result.vcg",-1,0,0);
    ...
}
```

Figure 16.6: Animation example

16.3.1 Using `anim` to view the animation

To view the results of the animation output, invoke

`anim` *dir*

Where *dir* is the path to the directory, as used in `animation="`*dir*`"`.
`anim` displays the steps of the iteration in a `vcg` window. After each display, it emits a prompt and the user can enter simple commands. The commands are:

Command	Description
`+`	Go to the next step.
`-`	Go to the previous step.
`q`	Quit `anim`.
`*`	Display the remaining steps without further command prompting.
`&`	Display the steps backward until the first, without command prompting.
`g`	The user is prompted for the number of the step to go to.
`f`	The user is prompted for a number n. The next step is n steps away.
`b`	Like `f`, but backwards.

Appendix A

Reference Count Garbage Collector

This chapter describes the interface of an additional garbage collector, which manages memory by reference counting. That means it counts the number of references to an object. If the counter becomes 0, the object is inserted into a free-list, in which objects are stored that can be overwritten. The user is responsible for maintaining the correct reference count.
The following functions provide the interface to the memory manager:

```
unsigned long rgc_stat(void);
```

This function returns the amount of memory the memory manager has actually allocated. The value is returned in KB.

```
void rgc_init(void);
```

You have to call this function first, before you use the memory manager. The best place for initializing it is in your `T_init`-function.

```
int rgc_registertype(unsigned int size, int freeondec);
```

The call of this function returns a typenumber, by which you can allocate new memory. The parameters are:

size The number of bytes that an object of this type needs in the heap, usually `sizeof(_T)`.

freeondec If 1, means that the object is inserted in the free-list, when its reference count becomes 0. Otherwise it is not inserted into the free-list and the user has to free the object manually by a call to `rgc_free`.

```
void rgc_free(void *ptr);
```

This function inserts an object `ptr` into the free-list, so that its memory can be reused. The reference count of the object is not tested for 0.

```
void *rgc_incref(void *ptr);
```

Increments the reference count of object `ptr` and returns the object. If the reference count of the object is already 0xffff then it is not furthur increased.

```
void *rgc_decref(void *ptr);
```

Decrements the reference count of object `ptr` and returns the object. If the reference count becomes 0 the object is inserted into the free-list, except of you registered the type not to do so.

```
int rgc_getref(void *ptr);
```

Returns the actual reference count of object `ptr`.

```
void *rgc_alloc(int type);
```

This function allocates memory for an object of type `type`. `type` is the typenumber you got at the registration.
First the garbage collector looks in the free-list if there is an object of that type that can be reused. If so, the memory is initialized by zeros an the object is returned.
Otherwise, new memory is allocated and returned. If there is no more memory to allocate the memory manager aborts with an error-message.

Appendix B

Example of a template-file

Here is an example of a template-file given, which realizes a function definition (`->`) in a `.set`-file. The implemention is that which was described in chapter 7 for function representations.

```
#H_FILE

#ifndef __$T_h_
#define __$T_h_

#ifndef $T_FUNCSIZE
#define $T_FUNCSIZE 16
#endif

#include "header.h"  /* global header-file */

/* header for subtype one, the argumenttype */
#include "$T1.h"

/* header for subtype two, the imagetype */
#include "$T2.h"

/* header for value-pair tuple in .set-file */
#include "$T3.h"

#ifdef GC
#include "gc_mem.h"    /* gargage collector interface */
#endif
```

```
typedef struct
{
  $T1 s;
  $T2 t;
} $T_mpair;              /* value pair */

typedef struct $T_list_base_
{
  int howmany;
  $T_mpair arr[$T_FUNCSIZE];
  struct $T_list_base_ *next;
} *$T_li,_$T_li;


typedef struct
{
  $T2 def;
  $T_li head;
} *$T,_$T;             /* the implemented functiontype */


typedef struct
{
  int dir;
  union
  {
    struct
    {
      $T_li block;
      int idx;
    } c2;
    struct
    {
      _$T1_acur c;
      $T2 wo;
      $T s;      /* NULL if reached the end */
    } c1;
  } inh;
} _$T_cur,*$T_cur;
```

```
typedef struct
{
  _$T2_acur *werte;          /* array from 0 to $T1_power */
} _$T_acur,*$T_acur;



#define $T_neq(x,y)   (!$T_eq(x,y))
#define $T_ge(x,y)    $T_le(y,x)
#define $T_geq(x,y)   $T_leq(y,x)
#define $T_leq(x,y)   ($T_le(x,y) || $T_eq(x,y))



/* for Gargage Collector */
#define $T_mark_self(x) (GC_mark((void**)&x))
#define $T_li_mark_self(x) {if((x)!=NULL)\
                             GC_mark((void**)&x);}
#define $T_mcopy_self(x) (($T)GC_share_copy((void*)x))
#define $T_li_mcopy_self(x) ((x)==NULL?(x):\
                            ($T_li)GC_share_copy((void*)x))




extern int $T_mytype,$T_litype;

extern $T $T_top;
extern $T $T_bottom;


extern ull $T_power;
extern int $T_is_power_unendl;


extern FLO_BOOL $T_flag;
```

```
/* Now function prototypes */

#if __STDC__
$T $T_crunch($T f1, $T f2,$T2 f($T2,$T2));
INLINE $T $T_create ( $T2 );
void $T_init ( void );
char *$T_len ( void );
INLINE void $T_remove_from_list ( $T, $T_li, $T_li );
INLINE unsigned int $T_li_hash ( $T_li );
INLINE unsigned int $T_hash ( $T );
INLINE void $T_remove_from_table ( $T_li, int );
char * $T_print ( $T );
char * $T_vcgprint ( $T );
void $T_print_fp ( FILE *, $T );
INLINE $T $T_update_destr ( $T, $T1, $T2 );
INLINE $T $T_update_pair_destr ( $T, $T3 );
INLINE $T $T_dublicate ( $T );
INLINE $T $T_update ( $T, $T1, $T2 );
INLINE ull $T_size ( $T );
INLINE $T $T_change_default ( $T, $T2 );
INLINE FLO_BOOL $T_get_diffs ( $T, $T1, $T2 * );
INLINE FLO_BOOL $T_eq ( $T, $T );
INLINE FLO_BOOL $T_li_eq ( $T_li, $T_li );
INLINE $T2 $T_get ( $T, $T1 );
INLINE FLO_BOOL $T_isbottom ( $T );
INLINE FLO_BOOL $T_istop ( $T );
$T $T_lub ( $T, $T );
$T $T_glb ( $T, $T );
INLINE FLO_BOOL $T_le ( $T, $T );
INLINE void $T_cur_correct ( $T_cur );
INLINE void $T_cur_reset ( $T_cur, $T, $T2 );
INLINE FLO_BOOL $T_cur_is_empty ( $T_cur );
INLINE void $T_cur_next ( $T_cur );
INLINE $T3 $T_cur_get ( $T_cur );
$T $T_acur_get ( $T_acur );
void $T_acur_next ( $T_acur );
FLO_BOOL $T_acur_is_empty ( $T_acur );
void $T_acur_reset ( $T_acur );
#else
```

```
$T $T_crunch();
INLINE $T $T_create ();
void $T_init ();
char *$T_len ();
INLINE void $T_remove_from_list ();
INLINE unsigned int $T_li_hash ();
INLINE unsigned int $T_hash ();
INLINE void $T_remove_from_table ();
char * $T_print ();
char * $T_vcgprint ();
void $T_print_fp ();
INLINE $T $T_update_destr ();
INLINE $T $T_update_pair_destr ();
INLINE $T $T_dublicate ();
INLINE $T $T_update ();
INLINE ull $T_size ();
INLINE $T $T_change_default ();
INLINE FLO_BOOL $T_get_diffs ();
INLINE FLO_BOOL $T_eq ();
INLINE FLO_BOOL $T_li_eq ();
INLINE $T2 $T_get ();
INLINE FLO_BOOL $T_isbottom ();
INLINE FLO_BOOL $T_istop ();
$T $T_lub ();
$T $T_glb ();
INLINE FLO_BOOL $T_le ();
INLINE void $T_cur_correct ();
INLINE void $T_cur_reset ();
INLINE FLO_BOOL $T_cur_is_empty ();
INLINE void $T_cur_next ();
INLINE $T3 $T_cur_get ();
$T $T_acur_get ();
void $T_acur_next ();
FLO_BOOL $T_acur_is_empty ();
void $T_acur_reset ();
#endif /* __STDC__ */


#include "$T.inline_h" /* for inlining purpose */
```

```
#C_FILE

#include <string.h>
#include "$T.h"


#if __STDC__
#ifdef GC
static void $T_mcopy(void *,void * );
static void $T_list_mcopy(void *,void * );
static void $T_mark(void * );
static void $T_list_mark(void * );
#endif
#else
#ifdef GC
static void $T_mcopy();
static void $T_list_mcopy( );
static void $T_mark( );
static void $T_list_mark();
#endif
#endif /* __STDC__ */


$T $T_top;
$T $T_bottom;

ull $T_power;
int $T_is_power_unendl;


#ifdef GC
int $T_mytype =-1,$T_litype =-1;
#endif

#ifdef SYNTAX_GC
extern int syntaxtype;
```

```
extern int e_syntaxtype;
#endif



INLINE $T $T_create($T2 def)
{
  $T help;

#ifdef GC
  help = ($T) GC_alloc($T_mytype);

  if (help==NULL)
    fatal_error("memory exhausted");
#else
  MALLOC(help,$T);
#endif
  help->def = def;
  help->head=NULL;
  return(help);
}



/* function to calculate power */
INLINE long $T_fupower(long p1, long p2)
{
  double d;

  assert((p1!=UNENDLICH) && (p2 != UNENDLICH));

  d = pow((double)p2,(double)p1);
  if( d  > (double) LLUINT_MAX)
    fatal_error("Functionpower to big for long!");
  return ((long)d);
}



void $T_init(void)
{
```

```
  static FLO_BOOL flag = FLO_FALSE;

  if( !flag )
  {
    flag = FLO_TRUE;
    $T1_init();
    $T2_init();
#ifdef GC
    $T_mytype = GC_registertype(sizeof(_$T),$T_mark,$T_mcopy,
                                $T_eq,$T_hash,FLO_FALSE);
    $T_litype = GC_registertype(sizeof(_$T_li),$T_list_mark,
                                $T_list_mcopy,
                                $T_li_eq,$T_li_hash,FLO_FALSE);
#endif
    $T_top = $T_create($T2_top);
    $T_bottom = $T_create($T2_bottom);
    $T_is_power_unendl = $U;
    $T_power = $T1_power==0 || $T2_power==0?0
                :$T2_power==1 ?1
                :$T1_power==UNENDLICH||$T2_power==UNENDLICH ?
                   UNENDLICH
                :$T_fupower($T1_power , $T2_power);
  }
}
```

```
#ifdef GC
static void $T_mcopy(void * x,void *y )
{
  $T p1 = ($T) x;
  $T p2 = ($T) y;
  $T_li help;

  p2->def = $T2_mcopy_self(p1->def);
  p2->head = $T_li_mcopy_self(p1->head);
}
```

```
static void $T_list_mcopy(void * x,void *y)
{
  $T_li p1 = ($T_li) x;
  $T_li p2 = ($T_li) y;
  int i;
  $T_mpair *h1,*h2;

  p2->howmany = p1->howmany;
  for(i=0,h1=p1->arr,h2=p2->arr;i<p1->howmany;i++,h1++,h2++)
  {
    h2->s = $T1_mcopy_self(h1->s);
    h2->t = $T2_mcopy_self(h1->t);
  }
  p2->next = $T_li_mcopy_self(p1->next);
}


static void $T_mark(void * x )
{
  $T p = ($T) x;

  $T2_mark_self(p->def);
  $T_li_mark_self(p->head);
}


static void $T_list_mark(void * x)
{
  $T_li p = ($T_li) x;
  int i;
  $T_mpair *h;

  for(i=0,h=p->arr;i<p->howmany;i++,h++)
  {
    $T1_mark_self(h->s);
    $T2_mark_self(h->t);
  }
  $T_li_mark_self(p->next);
}
```

```
FLO_BOOL $T_flag = FLO_FALSE;

INLINE  void $T_copy_obj(void)
{
  if (!$T_flag)
  {
    $T_flag = FLO_TRUE;
    $T_top = $T_mcopy_self($T_top);
    $T_bottom = $T_mcopy_self($T_bottom);
    $T1_copy_obj();
    $T2_copy_obj();
  }
}


INLINE  void $T_find_obj(void)
{
  if (!$T_flag)
  {
    $T_flag = FLO_TRUE;
    $T_mark_self($T_top);
    $T_mark_self($T_bottom);
    $T1_find_obj();
    $T2_find_obj();
  }
}


INLINE void $T_clear_flag(void)
{
  if ($T_flag)
  {
    $T_flag = FLO_FALSE;
    $T1_clear_flag();
    $T2_clear_flag();
  }
}
```

```
#endif    /* ifdef GC */
```

```
char *$T_len(void)
{
  char *s1;
  char *s;
  int len;

  s1 = $T2_len();
  len = strlen(s1);
  len += 200;
#ifdef SYNTAX_GC
  s = (char *) gc_tmp(len);
#else
  s = (char *) gc_malloc(len);
#endif
  if (s==NULL)
    fatal_error("memory exhausted");
  if (!$T1_is_power_unendl /*power != UNENDLICH*/)
    sprintf(s,"(%s)*%ld",s1,$T1_power);
  else
    sprintf(s,"(%s)*$T1",s1);
#ifndef SYNTAX_GC
  gc_free(s1);
#endif
  return(s);
}
```

```
INLINE void $T_remove_from_list($T func, $T_li elem,$T_li old)
{
  assert(elem!=NULL);
  assert(func!=NULL);
  assert(func->head!=NULL);
  assert(elem==func->head ||
```

```
         (func->head->next != NULL && old != NULL));

  if (elem==func->head)
  {
    func->head = elem->next;
  }
  else
  {
    assert(old == NULL || old->next==elem);
    old->next = elem->next;
  }
#ifdef GC
  GC_free((void *)elem);
#else
  gc_free((void*)elem);
#endif
}




INLINE unsigned int $T_li_hash($T_li help)
{
  unsigned int h=0;
  int i,*pr;
  $T_mpair *h1;

  for(i=0,h1=help->arr,pr=prime;
      i<help->howmany&&pr<prime+anz_prime;
      i++,h1++)
  {
    h += *pr++ *$T1_hash(h1->s) + *pr++ * $T2_hash(h1->t);
  }
  return(h);
}



INLINE unsigned int $T_hash($T e)
{
  $T_li help = e->head;
```

```
  int i=1,*pr=prime;
  unsigned int h;

  h=$T2_hash(e->def);
  while(help!=NULL)
  {
    h+=*pr++ +$T_li_hash(help);
    if (pr>=prime+anz_prime)
      break;
    help = help->next;
  }
  return(h);
}



INLINE void $T_remove_from_table($T_li help,int n)
{
  int anz;

  assert(n>=0);
  assert(help!=NULL);
  assert(n<help->howmany);

  help->howmany-=1;
  anz = help->howmany-n;
  assert(anz>=0);
  if (anz==0)
    return;
  memmove((void*)((help->arr)+n),(void*)((help->arr)+n+1)
          ,sizeof($T_mpair)*anz);
}



char * $T_print($T func)
{
  $T_li help ;
  int i;
  char *s,*q,*w,*s1,*zz;
  int a,len;
```

```
  $T_mpair *h;

  len = 1024;
  a = 0;
#ifdef SYNTAX_GC
  s = (char *) GC_alloc(e_syntaxtype);
#else
  s = (char*) gc_malloc(sizeof(char)*len);
#endif
  if (s == NULL)
    fatal_error("memory exhausted");
  sprintf(s, "[%s:",zz=$T2_print(func->def));
#ifndef SYNTAX_GC
  gc_free(zz);
#endif
  a = strlen(s);
  help = func->head;
  while (help != NULL)
  {
    for (i = 0,h=help->arr; i < help->howmany; i++,h++)
    {
      q = $T1_print(h->s);
      w = $T2_print(h->t);
      a += strlen(q)+ strlen(w)+3;
      if (a>len-10)
      {
        len *= 2;
#ifdef SYNTAX_GC
        s1 = (char*)gc_tmp(sizeof(char)*len);
#else
        s1 = (char*)gc_malloc(sizeof(char)*len);
#endif
        if (s1 == NULL)
          fatal_error("memory exhausted");
        strcpy(s1,s);
#ifndef SYNTAX_GC
        gc_free(s);
#endif
        s = s1;
      }
      strcat(s,q);
```

```
      strcat(s, "->");
      strcat(s,w);
      strcat(s,",");
#ifndef SYNTAX_GC
      gc_free(q);
      gc_free(w);
#endif
    }
    help = help->next;
  }
  strcat(s, "]");
  return(s);
}




void $T_print_fp(FILE *f ,$T func)
{
  char *s;

  s = $T_print(func);
  fprintf(f,"%s",s);
#ifndef SYNTAX_GC
  gc_free(s);
#endif
}



INLINE $T $T_update_destr($T root, $T1 s, $T2 t)
{
  int i;
  $T_li free,help,help_old;
  $T_mpair *h;

  assert(root!=NULL);
  free = NULL;
  help_old=NULL;
  help = root->head;
```

```
  while(help!=NULL)
  {
    if (help->howmany!=$T_FUNCSIZE)
      free = help;
    for (i=0,h=help->arr; i<help->howmany;i++,h++)
    {
      if ($T1_eq(h->s,s))
      {
        if ($T2_eq(root->def,t))
        {
           $T_remove_from_table(help,i);
           if (help->howmany==0)
             $T_remove_from_list(root,help,help_old);
           return(root);
        }
        h->t = t;
        return(root);
      }
    }
    help_old = help;
    help = help->next;
  }
  if ($T2_eq(t,root->def))
    return(root);
  if (free == NULL)
  {
#ifdef GC
    free = ($T_li) GC_alloc($T_litype);
      if (free==NULL)
        fatal_error("memory exhausted");
#else
    MALLOC(free,$T_li);
#endif
    free->next = root->head;
    free->howmany = 0;
    root->head = free;
  }
  free->arr[free->howmany].s = s;
  free->arr[free->howmany].t = t;
  free->howmany += 1;
  return(root);
```

```
}



INLINE $T $T_update_pair_destr($T root, $T3 pair)
{
  return($T_update_destr(root,
                        $T3_select_1(pair),
                        $T3_select_2(pair)));
}



INLINE $T $T_dublicate($T root)
{
  $T help;
  $T_li s,l=root->head;
  $T_li *s_old;

#ifdef GC
  help = ($T) GC_alloc($T_mytype);
#else
  MALLOC(help,$T);
#endif
  if (help==NULL)
    fatal_error("memory exhausted");
  help ->def = root->def;
  s_old= &(help->head);
  while(l!=NULL)
  {
#ifdef GC
    s = ($T_li) GC_alloc($T_litype);
#else
    MALLOC(s,$T_li);
#endif
    if (s==NULL)
      fatal_error("memory exhausted");
    *s_old = s;
    s_old = &(s->next);
    memmove((void*)s->arr,
            (void*)l->arr,
```

```
            sizeof($T_mpair)*l->howmany);
    s->howmany = l->howmany;
    l = l->next;
  }
  *s_old = NULL;
  return(help);
}
```

```
INLINE $T $T_update($T root, $T1 s, $T2 t)
{
  return($T_update_destr($T_dublicate(root),s,t));
}
```

```
INLINE ull $T_size($T f)
{
  ull anz=0;
  $T_li help;

  help = f->head;
  while(help!=NULL)
  {
    anz+=help->howmany;
    help=help->next;
  }
  return(anz);
}
```

```
INLINE $T $T_change_default($T f,$T2 def)
{
  int i;
  $T_li help,p,ho;
  $T_mpair *h;
```

```
  f->def = def;
  p=NULL;
  help = f->head;
  while(help!=NULL)
  {
    for(i=0,h=help->arr;i<help->howmany;i++,h++)
    {
      if($T2_eq(def,h->t))
      {
        $T_remove_from_table(help,i);
        h -= 1;
        i -= 1;
      }
    }
    ho = p;
    p = help;
    help = help->next;
    if (p->howmany==0)
      $T_remove_from_list(f,p,ho);
  }
  return(f);
}



INLINE FLO_BOOL $T_get_diffs($T f,$T1 elem,$T2 *p)
{
  int i;
  $T_li help;
  $T_mpair *h;

  help = f->head;
  while(help!=NULL)
  {
    for(i=0,h=help->arr;i<help->howmany;i++,h++)
    {
      if ($T1_eq(elem,h->s))
      {
        *p = h->t;
        return(FLO_TRUE);
```

```
      }
    }
    help = help->next;
  }
  return(FLO_FALSE);
}




INLINE FLO_BOOL $T_eq($T s, $T t)
{
  $T_li help;
  int num,i;
  ull sm,tm;
  $T2 tar;
  $T_mpair *h;

  if ($T1_is_power_unendl /*power == UNENDLICH*/)
  {
    if (!$T2_eq(s->def,t->def))
      return(FLO_FALSE);
    sm = $T_size(s);
    tm = $T_size(t);
  }
  else
  {
    sm = $T_size(s);
    tm = $T_size(t);
    assert($T1_power>=sm);
    assert($T1_power>=tm);
    if ($T1_power==sm && $T1_power !=tm)
    {
      if ($T2_eq(s->def,t->def))
        goto doit;
      return($T_eq(t,$T_change_default(s,t->def)));
    }
    else
      if ($T1_power==tm && $T1_power !=sm)
      {
        if ($T2_eq(s->def,t->def))
```

```
          goto doit;
        return($T_eq(s,$T_change_default(t,s->def)));
      }
    if ($T1_power>sm)
      if (!$T2_eq(s->def,t->def))
        return(FLO_FALSE);
  }
doit:
  if (sm!=tm)
    return(FLO_FALSE);
  help = s->head;
  num=0;
  while (help != NULL)
  {
    for (i = 0,h=help->arr; i < help->howmany; i++,h++)
    {
      if (!$T_get_diffs(t,h->s,&tar))
        return(FLO_FALSE);
      if (!$T2_eq(tar, h->t))
        return(FLO_FALSE);
    }
    help = help->next;
    num+=1;
  }
  return(FLO_TRUE);
}


INLINE FLO_BOOL $T_li_eq($T_li s, $T_li t)
{
  int i;
  $T_li h1=s,h2=t;
  $T_mpair *s1,*s2;

  while(h1!=NULL && h2!=NULL)
  {
    if (h1->howmany != h2->howmany)
      return(FLO_FALSE);
    for(i=0,s1=h1->arr,s2=h2->arr;i<h1->howmany;i++,s1++,s2++)
    {
```

```
      if (!$T1_eq(s1->s,s2->s))
        return(FLO_FALSE);
      if (!$T2_eq(s1->t,s2->t))
        return(FLO_FALSE);
    }
    h1 = h1->next;
    h2 = h2->next;
  }
  return(FLO_FALSE);
}
```

```
INLINE $T2 $T_get($T f, $T1 s)
{
  $T2 help;

  if ($T_get_diffs(f,s,&help))
    return(help);
  return(f->def);
}
```

```
INLINE FLO_BOOL $T_isbottom($T f)
{
  ull i;

  i = $T_size(f);
  if ($T2_isbottom(f->def))
    return(i==0);
  if (!( !$T1_is_power_unendl /*power!=UNENDLICH*/
          && i==$T1_power))
    return(FLO_FALSE);
  return($T_size($T_change_default(f,$T2_bottom))==0);
}
```

```
INLINE FLO_BOOL $T_istop($T f)
{
  ull i;

  i = $T_size(f);
  if ($T2_istop(f->def))
    return(i==0);
  if (!( !$T1_is_power_unendl /*power!=UNENDLICH*/
         && i==$T1_power))
    return(FLO_FALSE);
  return($T_size($T_change_default(f,$T2_top))==0);
}
```

```
$T $T_lub($T f1, $T f2)
{
  return($T_crunch(f1,f2,$T2_lub));
}
```

```
$T $T_glb($T f1, $T f2)
{
  return($T_crunch(f1,f2,$T2_glb));
}
```

```
$T $T_crunch($T f1, $T f2,$T2 f($T2,$T2))
{
  $T help;
  $T_li lauf;
  $T2 ll;
  int i;
  $T_mpair *h;
```

```
  ll = f(f1->def, f2->def);
  help = $T_create(ll);
  lauf = f1->head;
  while (lauf != NULL)
  {
    for (i = 0,h=lauf->arr; i < lauf->howmany; i++,h++)
    {
      ll = $T_get(f2, h->s);
      $T_update_destr(help, h->s, f(h->t, ll));
    }
    lauf = lauf->next;
  }
  lauf = f2->head;
  while (lauf != NULL)
  {
    for (i = 0,h=lauf->arr; i < lauf->howmany; i++,h++)
    {
      if (!$T_get_diffs(f1, h->s,&ll))
        $T_update_destr(help, h->s, f(f1->def,h->t));
    }
    lauf = lauf->next;
   }
  return(help);
}
```

```
INLINE FLO_BOOL $T_le($T f1, $T f2)
{
  $T_li lauf;
  $T2 ll;
  int i;
  FLO_BOOL erg=FLO_FALSE;
  $T_mpair *h;

  if ($T1_is_power_unendl /*power == UNENDLICH*/
      || $T1_power == $T_size(f1)
      || $T1_power == $T_size(f2))
```

```
  {
    if (!$T2_leq(f1->def, f2->def))
      return(FLO_FALSE);
    if ($T2_le(f1->def, f2->def))
      erg = FLO_TRUE;
  }
  lauf = f1->head;
  while (lauf != NULL)
  {
    for (i = 0,h=lauf->arr; i < lauf->howmany; i++,h++)
    {
      ll = $T_get(f2, h->s);
      if (!$T2_leq(h->t, ll))
        return(FLO_FALSE);
      if (!erg)
        if ($T2_le(h->t, ll))
          erg = FLO_TRUE;
    }
    lauf = lauf->next;
  }
  lauf = f2->head;
  while (lauf != NULL)
  {
    for (i = 0,h=lauf->arr; i < lauf->howmany; i++,h++)
    {
      if (!$T_get_diffs(f2, h->s,&ll))
      {
        if (!$T2_leq(h->t, f1->def))
          return(FLO_FALSE);
        if (!erg)
          if ($T2_le(h->t, f1->def))
            erg = FLO_TRUE;
      }
    }
    lauf = lauf->next;
  }
  return(erg);
}
```

```
INLINE void $T_cur_correct($T_cur co)
{
  $T1_acur c = &(co->inh.c1.c);
  $T *s = &(co->inh.c1.s);

  assert(co->dir==1);
  if ($T1_acur_is_empty(c))
  {
    *s=NULL;
    return;
  }
  if (*s==NULL)
    return;
  while($T2_eq($T_get(*s,$T1_acur_get(c)),co->inh.c1.wo))
  {
    $T1_acur_next(c);
    if ($T1_acur_is_empty(c))
    {
      *s=NULL;
      return;
    }
  }
}
```

```
INLINE void $T_cur_reset($T_cur c,$T f,$T2 wo)
{
  if ($T2_eq(f->def,wo))
  {
    c->inh.c2.block=f->head;
    c->inh.c2.idx=0;
    c->dir=2;
    assert(c->inh.c2.block==NULL||
           c->inh.c2.block->howmany!=0);
  }
  else
  {
```

```
    if ($T1_is_power_unendl /*power==UNENDLICH*/)
      fatal_error("Generating infinitely");
    $T1_acur_reset(&(c->inh.c1.c));
    c->inh.c1.s=f;
    c->inh.c1.wo=wo;
    c->dir=1;
    $T_cur_correct(c);
  }
}



INLINE FLO_BOOL $T_cur_is_empty($T_cur c)
{
  if (c->dir==1)
    return(c->inh.c1.s==NULL);
  assert (c->dir==2);
  return(c->inh.c2.block==NULL);
}



INLINE void $T_cur_next($T_cur co)
{
  if (co->dir==1)
  {
    $T1_acur_next(&(co->inh.c1.c));
    $T_cur_correct(co);
    return;
  }
  assert(co->dir==2);
  if(++(co->inh.c2.idx)>=co->inh.c2.block->howmany)
  {
    co->inh.c2.block=co->inh.c2.block->next;
    assert(co->inh.c2.block==NULL||
           co->inh.c2.block->howmany!=0);
    co->inh.c2.idx=0;
  }
}
```

```
INLINE $T3 $T_cur_get($T_cur co)
{
  if (co->dir==1)
  {
    $T1 t1 = $T1_acur_get(&(co->inh.c1.c));
    return($T3_create(t1,$T_get(co->inh.c1.s,t1)));
  }
  assert(co->dir==2);
  {
    $T_mpair *p=((co->inh.c2.block)->arr)+(co->inh.c2.idx);
      return($T3_create(p->s,p->t));
  }
}
```

```
INLINE $T $T_acur_get($T_acur c)
{
  long i;
  $T ret;
  _$T1_acur t1ac;

  ret = $T_create( $T2_acur_get( & c->werte[0]));

  $T1_acur_reset( &t1ac);

  for(i = 1,$T1_acur_next(&t1ac);
      i < $T1_power;
      i++,$T1_acur_next(&t1ac))
  {
    assert( !$T1_acur_is_empty(&t1ac));
    ret=$T_update_destr(ret,
                        $T1_acur_get(&t1ac),
                        $T2_acur_get(&c->werte[i]));
  }
  return ret;
}
```

```
INLINE void $T_acur_next($T_acur c)
{
  long i;

  i = 0;

  while( !$T2_acur_is_empty( &c->werte[i]) && (i<$T1_power))
  {
    $T2_acur_next(&c->werte[i]);
    if( $T2_acur_is_empty( &c->werte[i]))
    {
      $T2_acur_reset( &c->werte[i]);
      i++;
    }
    else
      break;
  }

  if( i == $T1_power )
    c->werte = NULL;
}


INLINE FLO_BOOL $T_acur_is_empty($T_acur c)
{
  return (c->werte == NULL );
}


INLINE void $T_acur_reset($T_acur c)
{
  long i;

  if( $T1_is_power_unendl /*power == UNENDLICH */)
    fatal_error("Generating infinitely !\n");

  c->werte = ($T2_acur) gc_tmp($T1_power * sizeof(_$T2_acur));
  for( i = 0; i<$T1_power;i++)
    $T2_acur_reset(& c->werte[i]);
```

```
}
```

Index

Bibliography

[AFMW96] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache Behavior Prediction by Abstract Interpretation. To appear in: *SAS'96 Static Analysis Symposium*, 1996.

[AM95] Martin Alt and Florian Martin. Generation of efficient interprocedural analyzers with PAG. In *SAS'95, Static Analysis, LNCS 983*, pages 33–50. Springer, 1995.

[AMW95] Martin Alt, Florian Martin, and Reinhard Wilhelm. Generating Analyzers with PAG. Technical Report A10/95, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, 1995.

[Bou93] Francois Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Conference on Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, pages 128–141. Springer, 1993.

[CC77] Patrik Cousot and Radhia Cousot. A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on principles of Programming Languages*, pages 194–206, Boston, Massachussetts, October 1973.

[KU77] J.B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[Mac95] David MacKenzie. *Autoconf. Creating Automatic Configuration Scripts.* FSF- Free Software Foundation, Inc., 2.7 edition, November 1995.

[Mar95] Florian Martin. Entwurf und Implementierung eines Generators für Datenflussanalysatoren. Master's thesis, Fachbereich 14, Universität des Saarlandes, Saarbrücken, 1995.

[Meh84] Kurt Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness.* Springer Verlag, 1984.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, **17**:348–375, 1978.

[MT90] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1990.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[San95] Georg Sander. *VCG – Visualization of Compiler Graphs.* Universität des Saarlandes, 66041 Saarbrücken, GERMANY, v1.30 edition, 1995.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Application*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[Tod90] Toda. On the complexity of topological sorting. *Information Processing Letters*, 35, 1990.

[Ull94] Jeffrey D. Ullman. *Elements of ML Programming.* Prentice-Hall, 1994.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design.* Addison-Wesley, 1995.