B L A C K W O O D   D E S I G N S

Principles of Program Analysis

OVERVIEW

This is a review of the contents of *Principles of Program Analysis* (2005) 2ed Flemming Nielson, Hanne Riss Nielson, Chris Hankin. It covers techniques for better understanding the behaviour of a program (esp in the edge case)

BENEFITS

Smaller memory footprint, faster execution, often easier to understand than other methods

USES

Compilers
Finding calculations that may have flaws
Tracing erroneous outcomes to fault points
Find code that is unbounded
Testing software
Analyzing software for incomplete implementation
Optimizing program elements

STRUCTURES

Lattice
Expression tree
Graph
Control flow graph
Data flow graph
Annotations
Lists
Sparse matrix

SURGEON GENERALS WARNING – Prolonged butt-scratching may result in Repetitive Stress Injury.

FILE: J:\My Documents\Experimental Languages and VMs\Book Program Analysis;2.doc

BOOK REVIEW

Principles of Program Analysis

RANDALL MAAS

This note is a review of the book “Principles of Program Analysis,“ to help understand the narrative. The book uses a too-complex method of description.

1. Statements into a graph, expressions & sub-expressions into nodes
2. Form base set of attributes for nodes
3. Form complete attributes of each node
4. Answer questions about procedure, etc.

“Principles of Program Analysis” 2ed
Flemming Nielson
Hanne Riss Nielson
Chris Hankin
2005, Springer

Supplements at: http://www2.imm.dtu.dk/~riis/PPA/ppasup2004.html

Compilers use these techniques (or similar) to:

1. Remove superfluous computations (dead code propagation, constant propagation)
2. Merge redundant computations
3. To schedule computation and other operations

Analysis tools may also suggest that these exhibit possible implementation mistakes.

1 Notation

Syntax:

> The language is broken down along syntax into nodes. Implicitly there is only one operation per node. Expressions are decomposed into separate sub-expression for each action

Semantics:

semantics – p211

1. Set of values, state, variables and their type, sets of variables (closures)
2. Specifies how a program transforms one value into another

Program analysis:

1. Set of properties
2. Specifies how a program transforms one property into another

Nodes are assigned a unique *numerical identifier*.

labels – p6

> A node could be identified by an internal pointer. Using a file-line-column-span (e.g. mapping to the source file) is not recommended. Constant folding and merging

duplicate code operations, make it possible for several different source-file locations to map to the same node.

The text prefers to use a small number of base abstractions.

- "Lattices" are used for structures
- The process of applying rules, broadly, uses the concept of fixed point
- Working thru constraints is handled by work-lists

To look at an analysis, the book often defines a small grammar; some massaging is often needed to make it doable.

1.1 Fixed point

Fixed point is used, idiomatically, to mean repeatedly resolving references – e.g. values expressions – until no more can be resolved. Specific examples include:

fixed point – p8

- Producing a trace
- Constant folding
- Dead Code elimination
- Abstract interpretation
- Approximating fixed point

approximating fixed point – p221-22

The technical meaning of fixed point is a value that a function (when given it as an argument) returns. In this case, the "value" is the set of variables and their values (or unresolved expression, as the case may be). The function is the process of resolving expressions into values. This is repeated until nothing more can be resolved this way.

1.2 Lattice

The text prefers to make structures into *complete lattice* for its analysis. Lattices are essentially tree structures: the set of child nodes (of two nodes) don't partially overlap – they are either a subset, the same, or share no common elements. In, complete lattices all children (subsets) have a greatest lower bound, a least upper bound, a least and a greatest element. The right most child node is often the left most child of a sibling.

complete lattice– p393

Treating lattices with bit vectors, although not clearly defined.

1.3 Work-lists

Work-list builds a set of items that satisfies constraints. These constraints are in a graph structure, and numbered. These algorithms relate to repeatedly applying the rules until solved (see *fixed point)*

work list – p368

2 Analysis

The techniques should be sound and complete; discussion on how to tell. Start with a restricted class analysis. Define correctness relations for each type of analysis. Starts with simple and expands to more intermediary steps in the analysis. This leads to what the elements analysis are:

- Values
- Expression
- Type
- Heap
- Pointer
- State
- Property
- Selector
- Location

- Variable
- Label
- Constraint

Types of analysis by pairs of these elements

State x State : Constant propagation analysis
Env x Env : Control flow analysis
Var x Label : Data flow
Values x Properties : Abstract interpretation

2.1 Value and Data flow

The data flow for each node and the variables it affects are tracked – linking each variable to nodes that may have assigned it. *Reaching definitions – p3*

Traditionally, this is a transposed dataflow matrix. This maps a node to the symbols it changes. Each row corresponds to a symbol (variable), each column corresponds to a node that may have assigned it a value; this matrix is often quite sparse.

Other similar attributes include mapping to possible value sets, and variable aliasing.

Nodes have an entry and exit (transposed) dataflow matrices, defining its action. The book needs two matrices are needed since the matrices are not placed on the arcs of the graph. A node may have many inputs (e.g. branch targets). Its entry matrix is the union of the exit matrices of the nodes that link to it. The exit matrices may be used to identify *what is affected when a node is modified*.

1. A node that does not modify a variable will duplicate the variable's row in the entry matrix to the exit matrix.
2. A variable assignment to a value based on a previous value will duplicate the variable's row in the entry matrix to the exit matrix and add this node
3. A variable assignment to an expression's value that does not depend on its previous value (e.g. a constant) will have a single entry for this node. It will not duplicate the variable's row in the entry matrix to the exit matrix.

The EQUATIONAL APPROACH and the CONSTRAINT BASED APPROACH. Each nodes exit is defined in terms of its entry matrix, and optional replacement row for a variable. The entry matrices are the union of all the exit matrices that feed into the node.

These dataflow matrices are referred to as Var x Label *Var x Label – p7*

Most of these matrices are sparse; the text introduces a notation of tracking an optional reference (e.g. a previous node) and the changes from that matrix.

Useful for determining which formulae are not used (e.g. Dead Code elimination), or the limits of their inputs. *least solution – p7*

Constant folding (an example of a code transform), aka constant value propagation. *constant folding – p27*

1. In all expressions, replace variables that have only a constant value, with that value
   a. Analysis may find that a variable has a constant value whenever a given node is entered; The variable may be eliminated for that node
2. Evaluate sub-expressions that involving only constants
3. Repeat for any variables that have been found to have a constant value

Values – simple values, states, closures, etc. *values – p211*

2.2 Types

Underlying types are the types the language nominally specifies (also called the *ordinary types*) *underlying type – p295*

Free variables and their type; *type environment –*

Annotated types, with the annotations including: *annotated type – p287*

- Rules to the effect: When statement S is executed: if properties ABC are true, then it specifies the resulting properties XYZ after execution. *annotated type system– p17*
- A set of functions that return an item of a given type
- Program points (a node). Regions of points don't partially overlap – they are either a subset, the same, or share no common elements. *program point – p284*

2.3 Variables

Variables hold values; most of the tracking is best done with variables.

These techniques are combined in abstract interpretation to find: *abstract interpretation – p13*

- All possible values a variable (or expression) may (or may not) have at each point.
- A *trace* – the "record [of] where the variables have obtained their values in the course of the computation." This is found by rippling thru all the places where a variable can be set and used. *trace – p13*

Not all variables are named – such as those implicit as the output step of an expression.

Different incarnation of variable (relative to procedure). Author uses a location. Single assignment variables

Definition point – points where function abstractions are created; variables assigned value. *definition point – p145*

Use point – points where functions are applied; variable value is accessed. *use point – p145*

Ranking includes:

- Dead – the value is not used at all
- Faint variable – dead or is used only to calculate new values for faint variables *faint variable – p136*
- Live variable – if any successor uses the variable before it is redefined *live variable – p49*
- Strongly live – live but not faint *strongly live variable – p136*

2.4 Expressions

Ranking of expressions based on whether the result is used on all paths, some paths or none.

- Killed expression
- Generated expression – evaluated, none of its inputs are modified
- Very busy expression – result is always used before any inputs are redefined *very busy expression– p46*
- Available expressions – expressions that have been computed and not modified later. *available expressions – p39*

Various analyses to tell, for a given expression, its:

- The type of its result, given its inputs. *type judgment – p286*
- Which storage locations have been created, accessed and assigned *side effect analysis – p320*
- Which exceptions may result *exception analysis – p325*
- The regions of program points involved

A notation that has each statement specify

> When statement S is executed: if properties ABC are true, then it has the following the effect it. *effect system– p17*

Communication analysis to determine the communication behaviour of each expression: *communication – p339*

- Allocating channels
- Entities sent over channels
- Entities received
- Behaviour of the process being generated
- Establishing temporal order and causality

2.5 Reference and Shape Analysis

1st order analysis – p212

The analysis techniques proceed from the simple to more complex:

- Variables are allocated statically
- Variables can be allocated statically or on the stack
- Variables can be allocated statically, on the stack, or dynamically elsewhere

Location – where a variable can be stored. Dynamic allocation allows an unbounded number of variables can be allocated. In this case representing the location in a more abstract manner (a bit more symbolically) and compactly to be tractable. *location – p105* *abstract location – p110*

Pointers are a class of variables that can refer to locations. The analysis looks only at pointers, operations, and expressions of. *pointer expression – p107*

Translate into single assignment form, where each variable is assigned only once:

1. Identify points where flow of control may join and special assignments are to be inserted
2. Rename variables

Regions – inference, region names, variables, static regions.

Shape analysis: the finite characterization of the shape of data structures – which could conceivably be unbounded. Predict null dereference. Then apply heap analysis. A heap is a set of links between locations: *shape analysis – p104* *heap – p104* *abstract heap – p111*

A heap is not the kind referred to in C memory allocation

- Table of edges between two locations and selector name on edge.
- Selector names are essentially field names *selector names – p104*
- Pointers have edges with empty selector names.

One approach is to, apply a Knuthian transform to the data to make it a binary tree. Selector names include next, prev, address of variable.

Equivalent strings of selectors.

Other operations translated to three address codes *transfer function – p110*

Shape graph *shape graph – p109*

With dynamic allocation, the analysis proceeds: *abstract heap – p111*

1. Split heap into separate heaps, and see how the heaps refer to each other (otherwise ignoring the internal structure) *sharing information – p112*
2. Shape of heap's internal structure; approximate the access paths

2.6 Control Flow analysis

2nd order analysis – p212

Developed for functional languages

Dead code. To be useful this process must repeatedly strip off those that aren't live to find cases where variable is *dead* but increments itself (appearing live, at least locally). Remove expression with effect or use.

Performing loop unrolling. Divide and conquer by splitting graph (by duplicating nodes) into portions that can be partially evaluated and those that cannot. Then analyze.

A basic block is a sequence of statements (executed in order). The first statement is the entry point, and the only exit point is the last statement (which may be a branch). *basic block – p136*

Path: the list of blocks traversed to reach the current one (see also trace).

Valid paths: paths with proper nesting of calls. This helps a lot of the analysis of calls and returns. *valid paths– p89*

Call Strings with unbounded length (e.g. recursive) p95

Call string with bounded length p97

Dynamic dispatch is hard. OOP is hard too, depending on the style of call. The text doesn't make a distinction between the dynamic dispatch of Objective-C (etc) and the complexities of C++, Java, C#.

Cartesian product algorithm – developed for object-oriented languages. *Cartesian Product Algorithm – p145*

Adding in data flow analysis helps.

2.7 Constraint Based Analysis

– p141

Constraints on types, variables, operations. Constraints include:

- Information true (as labels) on entry to a block
- Information true on exit from a block

Correctness relations

3 Data Analysis

data analysis – p35

Intraprocedural analysis limits itself to operations with in procedure; calls are treated as simple operation with large number of side-effects.

Interprocedural analysis – use of call strings

Structural Operational Semantics. Each node is called a *configuration,* and is a state (variable binding) and an optional statement. Transitions are statement and state mapping to a configuration.

State (aka *model)* variable's and their values (including linkages) at a given point in time

Monotone framework *monotone framework – p68*

- Complete lattice, which satisfies the ascending chain condition
- A set of space functions
- A finite flow
- A finite set of labels
- Extremal value
- A mapping from labels to transfer function

4 Abstract Interpretation *– p212*

Correctness relation: R : Values x Properties

Design of Property Spaces. Their functions and computations, relationships between them.

Galois Connections and Galois Insertions. Means of making property space less costly can generate further analysis. Extraction functions.

Design of Galois Connection:

1. Sequential Composition
2. Catalogue of combination techniques
   a) Independent attribute method
   b) Relational method
   c) Total function space
   d) Monotone function space
   e) Direct product
   f) Direct tensor product
3. Induced operations

Sets of state analysis p265

5 Type and Effect System *– p282*

Safety properties: if point X is reached, properties XYZ will hold

6 Algorithms *– p222*

Flow variable *flow variable – p366*

Strong Components: "maximally strongly connected subgraphs" *strong components – p381*

Induction – mathematical induction; structural induction