A Canonical Form of Arithmetic and Conditional Expressions

Torsten Görg, Mandy Northover
University of Stuttgart
Universitaetsstr. 38, 70569 Stuttgart, Germany
{torsten.goerg, mandy.northover}@informatik.uni-stuttgart.de

Abstract: *This paper contributes to code clone detection by providing an algorithm that calculates canonical forms of arithmetic and conditional expressions. An experimental evaluation shows the relevance of such expressions in real code. The proposed normalization can be used in addition to dataflow normalizations.*

1 Introduction

Clone detection techniques [1] try to find program code fragments that are semantically equivalent or similar. It is not possible to solve this problem completely because semantic equivalence is not decidable for arbitrary code fragments. A common approach to compute equivalence is to use a normal form. Because of the undecidability of semantic equivalence, a unique normal form is also not achievable for usual source code. Nevertheless, it is useful to establish canonical code representations as partial normalizations to support clone detection. E.g., program dependence graphs (PDG) are used in several clone detection tools to normalize data flows [2]. Roy and Cordy [1] also mention a transformation step as part of a general clone detection process.

We introduce a canonical form of arithmetic and conditional expressions. Through mathematical term transformations, many code variations are possible on expressions. Most of these variations are not handled by PDG. Our normalization is based on heuristics, so that most expressions occuring in real code are mapped to a unique canonical form.

2 Relevance Evaluation

As a first step, we evaluated the relevance of a canonical form of simple expressions like operators on basic type values, literals, variable access, and reads on components of arrays and records. Assignments, function calls, and control constructs like loops and gotos were excluded. The expressions obeying these constraints were located as subtrees at the bottom of the AST. We measured the amount of such expressions in relation to the total code size for several open source systems. Table 1 shows the results for *make*, *bison*, *bash*, *gnuplot*, and *unzip*. Our measurements were based on the program analysis framework Bauhaus and its intermediate representation (IML) [3]. The total code size is measured in SLOC and the number of all nodes in the IML graph, including the declarative parts of the code. The following columns show the number and percentage of nodes in simple expressions. The

Name	SLOC	#all	#exp	%exp	Avg
make	*17427*	*82521*	*32643*	*39.6%*	*2.85*
bison	*20395*	*197226*	*74844*	*37.9%*	*2.55*
bash	*88401*	*514265*	*200790*	*39.0%*	*2.97*
gnuplot	*61494*	*549746*	*247497*	*45,0%*	*2.91*
unzip	*49127*	*82480*	*35535*	*43.1%*	*3.33*

Table 1. Measured relevance of simple expressions

average number of nodes in these subtrees are given in the last column to indicate the size of the expressions. An average of 41% in the second last column indicates a high relevance of simple expressions.

3 Normalization Process

To normalize expressions, we define a set of rewrite rules. For gaining unique normal forms, termination and confluence have to be guaranteed, i.e., equivalent expressions have to be mapped to the same form [4]. All our transformations terminate. Confluence is heuristically approximated.

As described by Metzger and Wen [5], many variations result from permutations on the operands of commutative operators like *add* or *multiply*. To eliminate these variations we define a partitial order on expressions and sort the operands of commutative operators based on this order. The transformation is done in the following steps:

1. As is usual in intermediate representations, Bauhaus IML constructs expressions from binary and unary operators. To handle arbitrary sums, cohesive binary *add* nodes are contracted to a single *sum* node, based on these rules:

$$\texttt{add}(o_1,o_2) \rightarrow \texttt{sum}(o_1,o_2)$$

$$\texttt{sum}(\ldots,\texttt{add}(o_3,o_4),\ldots) \rightarrow \texttt{sum}(\ldots,o_3,o_4,\ldots)$$

Multiply nodes are contracted to products in the same way, as are logical and bitwise disjunctions

and conjunctions. Although logical operations are usually evaluated lazily, this is no problem here because the constraints specified in the previous section exclude side effects and guarantee referential transparency.
Inverse operations are also included in the contracted representation, e.g., subtracts in sums. To express this, each operand has a sign:

$$sub(o_1, o_2) \rightarrow sum(o_1, -o_2)$$

Divisions are handled similarly.
2. Unary plus operators are simply eliminated because they have no semantic effect in arithmetic expressions.
3. A unary minus operator toggles the signs of the operands that it dominates. Thus it is also integrated in the contracted representation.
4. The operands of each contracted sum or product are reordered based on several sorting criteria (beginning with the highest priority):

- The type of the root node of the subtree representing the operand.
- The number of operands.
- Successive comparison of operands.
- The value of a literal.
- The IML node ID of the declaration for a variable access.

After the contractions and operand reorderings, further transformations are processed to improve the confluence:
1. Constant folding reduces the number of literal nodes. The reordering in the previous step has already grouped literal operands together. Because calculations on literal values may introduce rounding imprecisions, the comparison of canonical forms allows some tolerance.
2. The contractions may result in sums and products in multiple layers:

$$sum(\ldots, product(\ldots, sum(\ldots), \ldots)$$

Applying the distributive law eliminates this.
3. Additional mathematical laws are applied, e.g., absorption, idempotence, and complement.

Another problem is the unification of corresponding variable accesses. Semantically equivalent expressions usually access different free variables. To get a more unique form, variable accesses are replaced by numbered surrogate nodes, e.g.:

$$a + b + 2*b^2 \rightarrow s_2 + s_1 + 2*s_1^2$$

But in some cases this is still not unique. To cope with this problem, a heuristic approach simliar to the technique described by Metzger and Wen [5] is used. It identifies the unique variable accesses in a term to order them uniquely. Numbering a variable may unify a variable access that was ambiguous previously. In the example above, b is uniquely represented by s_1 because of the unique subterm $2*b^2$. Subsequently a is uniquely numbered as s_2. Variables that are not orderd uniquely have to be renumbered during the comparison of normalized terms.

4 Application

The suggested normalization can easily be combined with PDG techniques. After normalizing and comparing the expressions, they are contracted to surrogate nodes. All variable accesses are incoming data flows. The result of an expression is its only outgoing data flow. In contrast to the fine grained PDG technique of Krinke [2], this handles more variations and reduces the number of PDG nodes. An evaluation of our approach is future work.

5 Related Work

Metzger and Wen [5] use canonical forms to handle variations in code that hamper the recognition of known algorithms taken from a knowledge base. They focus on reordering the operands of commutative operations and do not process any further transformations.
Zhou and Burleson [6] apply canonical arithmetic expressions to identify datapaths with equivalent path predicates in designs of digital signal processing systems.

References

[1] Chanchal Kumar Roy and James R. Cordy, "A survey on software clone detection research," technical report, Queen's University, Canada, 2007.

[2] Jens Krinke, "Identifying Similar Code with Program Dependence Graphs," in Proc. Eight Working Conference on Reverse Engineering (WCRE 2001), Stuttgart, Germany, pp. 301-309, October 2001

[3] Aoun Raza, Gunther Vogel, and Erhard Plödereder, "Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering," in Proceedings of Ada Europe 2006, LNCS 4006, pp. 71-82.

[4] N. Dershowitz, "Rewrite systems," in "Handbook of Theoretical Computer Science," pp. 243-320, Elsevier Science Publishers B.V., 1990.

[5] Robert Metzger and Zhaofang Wen, "Automatic Algorithm Recognition and Replacement – A New Approach to Program Optimization," MIT, 2000

[6] Zheng Zhou and Wayne Burleson, "Equivalence Checking of Datapaths Based on Arithmetic Expressions," in Proceedings of 32nd ACM/IEEE Design Automation Conference, ACM, 1995.