Semantic Analysis for Parallelizing C

Pierre David

Laboratoire MASI - Institut Blaise Pascal
Laboratoire LPTL
Université Pierre et Marie Curie
75252 Paris Cedex 05 - France
email : pda@masi.ibp.fr

Abstract. Automatic parallelization of C language programs is a difficult task due, for example, to side effects within expressions and access to memory via pointers. We describe a method based on semantic analysis of the language, in order to detect parallelism in complex programs using pointer arithmetic. We define semantic domains which respect the type system of the language, and use them to synthesize effect of the program through functional objects called transformations.
Keywords: automatic parallelization, compilation, pointers, side effects, C language, types.

1 Introduction

In the past few years, a considerable amount of work has been done in the field of automatic parallelization [13]. However, most of these efforts have been applied to the Fortran language, which is undoubtedly the most widely used language in science and engineering. In comparison, very little time has been devoted to more recent imperative languages such as Pascal, C or Ada.

C offers some interesting facilities, even in the area of intensive computing [11]. Among its strong features, C is widely used and taught, it is stable (there is a norm), it simplifies portability problems, it is efficient, it is adapted to many problems from numeric simulations to graphic manipulations (domains where a large demand exists for parallel computing).

Some efforts have been made to specify some parallelized versions of C ([9] for example). In this paper, however, we focus on the problems one must face in order to automatically detect parallelism in C programs. The most important ones are:

- side effects in expressions (including assignment) hinder further analysis,
- some operators (`&&` for example) are sequential,
- numerous data types and sizes of manipulated objects,
- the `for` loop, on which all parallelizers rely, does not have the same semantics as the Fortran `DO` loop,
- using small functions is considered good programming style,
- operator `&` allows variable modification in a subtle way, making obsolete all methods to build the dependence graph used for Fortran,
- pointer arithmetics makes determination of modified variables much more complex, and the presence of cast operator, `union` types and parameter passing allow type system violation.

Some attempts have been made to solve some of these problems. These works fall into two categories:

1. works done on pointer and dynamically allocated data structures. Many of them are not applied to a particular language ([5, 8] among others), but rare are those directly applied to C [4]. In these works, major problems we cited, such as pointer arithmetic or side-effects in expressions, are not addressed.
2. works dedicated to the C language and some of the problems we mentioned.
   A compiler for a multi-processor and vector machine is described in [1]. The pragmatic approach, in a industry context, is to use the AT&T compiler as the front-end of a parallelizer. `For` loops are converted to `while` loops, side effects are removed from expressions and procedure inlining is described. However, authors do not detail their work.
   The Parafrase-2 parallelizer [10] includes a pre-processor and a post-processor for the C language. However, they are not sufficient to do a good job on C. Authors admit that dependence analysis in the presence of pointers is not implemented.

As we can see, very few satisfying solutions have been given to our problem. We cannot afford to address all problems at the same time. So, our goal is to be able to parallelize significant programs (using arrays, pointers and pointer arithmetic, complex expressions). This paper describes the first step in this direction: semantic analysis of assignments, conditionals and sequences (no loops). Due to space requirements, this is just an outline. More detailed discussion is in [6] or [7].

In the second section, we define the input language our parallelizer will admit. In particular, pointers as well as all expressions, even the most complicated ones, are allowed. In the third section, we describe how we suppress all side effects from expressions. This phase is called the *normalization* of programs. In the last section, we build the framework for semantic analysis. We have carefully designed this framework to be valid for the C complete syntax. Given this basis, we can use it to synthesize effect of instructions.

2 Input Language

The base language of our parallelizer is ANSI C [3]. However, we cannot plan to solve all problems in this study. Hence, we must apply some restrictions on the input language of our parallelizer. There are three kinds of restrictions:

1. Difficult problems. In fact, there is only one major problem: transgression of the C language type system. Hence, the *cast* operator, `union` types and function call have been removed from our input language.
2. Problems not specific to C, such as interprocedural analysis [12] or program restructuring [2] are not dealt with. Function calls have already been removed. In addition, all non structured constructs are removed: `goto`, `break`, `return`, etc.
3. Tedious problems. They are direct extensions of constructs we treat in this paper or we plan to in the future. For example, `switch` is a generalization of `if` statement. `Do-while` loop and `switch` choice are removed. We plan to treat `while`

and `for` loops in a near future. Another simple generalization of mechanisms presented in this paper is `struct` data type: it is an aggregate of multiple distinct objects with a common name. Hence, we remove the `struct` data type, and bit-fields.

To summarize, our input language is based on the syntax of C, and:

- has all data types (including arrays and pointers) but `struct`, `union` and the exotic bit-field,
- use all expressions operators (even cluttered ones) but cast and function call,
- has sequence of instructions, `if` statement, and `while` and `for` loops even if they are not yet presented.

which is a non-trivial subset including a large part of difficulties mentioned earlier.

We have written a syntactic analyzer for this language in C with `lex` and `yacc` tools. Programs are translated into a syntax tree written as Lisp expressions, which is our intermediate representation [6]. Time used by the translator is negligible when compared with other phases.

3 Program Normalization

We impose few restrictions on our input language so it seems difficult to parallelize when expressions contain many side effects, but they are really frequent in C programs and we must deal with them. Since semantic analysis with side effects is not a simple task, we have chosen to remove them. This section exposes the output of the program normalizer, i.e. the language our semantic analyzer expects.
This new language is basically the same as the input one. All types allowed are unchanged. No new type is introduced. The largest area of change is the expressions syntax. We must define a set of operators such as there is no side effect. Hence, we must remove some operators:

- Assignment. This is no longer an operator, but an instruction.
- Side-effect operators (`++`, `+=`, etc.). They are only short-hand notations.
- Sequential operators (`&&`, `||` and `?:`). Here, the side effect is on program counter.
- Array access. Access to array elements is replaced by an explicit memory access via a pointer since both notations are equivalent.
- Comma operator. This operator is obsolete when all side effects are removed.

There are few instructions in our input language, and they differ from C:

- assignment is now an instruction. Expressions are now only part of assignments and `while` or `if` tests.
- `for` loops are replaced by equivalent `while` based code.

The program normalizer is implemented as a pattern matching filter which accepts a syntax tree in intermediate form as input and outputs a modified syntax tree. A typical function (10 lines) is processed in less than 10 seconds. When a C language program is converted to this new language, semantic analysis can be done.

4 Semantic Analysis

In this section, we present a method to synthesize the effect of a program. The analysis can be used as the dependence analysis phase of an automatic parallelizer, or as the framework of a loop analysis for normalization/parallelization. The hard work, when faced with such a complex language as C, is to build a model for analysis.

The first step is to define the classic semantic domains:

$\mathcal{N}$	$\ldots -1, 0, 1 \ldots$	Natural values	$\mathcal{B}$	{true, false}	Boolean values
$\mathcal{V}$	$\mathcal{N} + \mathcal{B}$	All possible values	$\mathcal{I}$	`a`, `b` ...	All data object identifiers

This allows us to build address domain $\mathcal{A}$ which is included in the cartesian product $\mathcal{I} \times \mathcal{N}$. In other words, an address $a = \langle i, k \rangle$ is built from the symbol i, which identifies the start of an object in memory, and a integer offset k inside the object referenced by i. We define operations on addresses: difference, equality and inequality, addition and subtraction with integers. However, this semantic domain is too large for our purpose: there is no type notion. We introduce it by splitting $\mathcal{A}$: we state that there exists a partition of $\mathcal{A}$ in subsets $\mathcal{A}_n$ (with $n \geq 0$). What does n mean in $\mathcal{A}_n$? The idea is to include types, pointers in particular, in our formalism. A variable (addresses $\langle i, k \rangle$) is in $\mathcal{A}_n$ iff the declaration of i contains n stars. In other words, there are $n+1$ references to get the pointed value. For example, if types of `i` and `ppi` are `int` and `int **` respectively, then $\langle \texttt{i}, 0 \rangle \in \mathcal{A}_0$ and $\langle \texttt{ppi}, 0 \rangle \in \mathcal{A}_2$. Disjonctions of the sets $\mathcal{A}_n$ are guaranteed by the semantic rules of C and our restrictions: type violation is impossible. Hence, a variable is used as it is declared. Arrays are special pointers: they are constant pointers, and access to each cell can be done in only one memory reference. Then, a variable of type `int[][]` belongs to $\mathcal{A}_0$. We could extend this partition to handle all types and re-partition each $\mathcal{A}_n$ in subsets according to the basic type ($\mathcal{A}_n^{\texttt{char}}$, $\mathcal{A}_n^{\texttt{int}}$, etc.). The implementation uses this double partitioning scheme, but it is omitted in this article for the sake of clarity.

We partition the $\mathcal{I}$ domain in the same way that we partitioned the $\mathcal{A}$ domain, and we note $ident_n$ the set of all identifiers matching simple variables (i.e. not arrays), and $const_n$ (with $n \geq 1$) the set of all array identifiers.

The $\mathcal{A}_n$ sets allow us to define the memory. Each cell contains a scalar variable, a pointer, or an array of scalars or pointers. Consider the function family $M_n : \mathcal{A}_n \rightarrow \mathcal{A}_{n-1} + \top$, where $n \geq 0$, $\mathcal{A}_{-1} = \mathcal{V}$ and $\top$ is the set containing the only element $\top$. This family associates the content of a memory cell, either a value (M_0) or an address (M_n, with $n > 0$), with an address. If the cell is neither defined nor assigned, we write $\top$ (*top*) its content. A memory is the function $M = \bigcup_{n \geq 0} M_n$. The set of all memories is called $\mathcal{M}$.

We are now able to define the syntax of expressions:

$$
\begin{array}{llll}
E & ::= R|L & R & ::= R_0|R_1|\ldots|R_n|\ldots \\
R_0 & \multicolumn{3}{l}{::= \mathrm{const}|\mathrm{ident}_0|(R_0)|R_0 \text{ binary-op } R_0|\text{unary-op } R_0|R_i - R_i|{*}R_1} \\
R_n & \multicolumn{3}{l}{::= \mathrm{const}_{n-1}|\mathrm{ident}_n|(R_n)|R_n{+}R_0|R_n - R_0|{*}R_{n+1}|\&L_{n-1}} \\
L & ::= L_0|L_1|\ldots|L_n|\ldots & L_n & ::= \mathrm{ident}_n|(L_n)|{*}L_{n+1}
\end{array}
$$

Note that expressions have been split into two families: R (for *right hand side*) and L (for *left hand side*). We will use the set $\mathcal{X}$ of all expressions, and the two subsets $\mathcal{X}_l$ and $\mathcal{X}_r$ of L-expressions and R-expressions respectively. Finally, the subscript n

in R_n or L_n is the number of stars in the type of the result. All expressions evaluate in the context of a memory. Hence, the type of interpretation equations is: $E : \mathcal{X} \rightarrow \mathcal{M} \rightarrow \mathcal{A} + \mathcal{V}$. For example, the L_n function is defined by: $L_n[\![\mathrm{ident}_n]\!] \equiv \lambda\mu.\langle \mathrm{ident}_n, 0\rangle$, $L_n[\![(L'_n)]\!] \equiv \lambda\mu.L_n[\![L'_n]\!]\mu$, and $L_n[\![*R_{n+1}]\!] \equiv \lambda\mu.R_{n+1}[\![R_{n+1}]\!]\mu$.

With the expression syntax above, assignment is simply: L_n = R_n. The effect of this assignment is to modify the memory. If the memory before assignment is called M, the new memory become: $M' = \lambda a.(\text{if } a = L_n[\![\mathrm{L}_n]\!]M \text{then } R_n[\![\mathrm{R}_n]\!]M \text{else } Ma)$. An assignment can be viewed as a function which modifies a memory. We call this a *transformation*, and we write: $T : \mathcal{M} \rightarrow \mathcal{M}$

Our implementation needs an easily manipulated representation of objects. We represent the transformation T (for the assignment $l = r$) by a substitution, i.e. the expressions $\langle l, r\rangle \in \mathcal{X}_l \times \mathcal{X}_r$, which is written $\sigma = [l \leftarrow r]$. In other words, the representation of a transformation is the mechanical writing of the two expressions involved in the assignment. When given a sequence of instructions, we must compose associated transformations. However, the representation we have chosen become ambiguous as soon as an indirection via a pointer appears. What is the value of `i` after execution of code synthesized by $[\mathtt{i} \leftarrow 1, *\mathtt{p} \leftarrow 2]$? To solve this problem, we introduce functions (*guards*) to select memory configurations. A guard is a function $G : \mathcal{M} \rightarrow \mathcal{B}$ with the form: $G = \lambda\mu. \bigwedge_{1\leq i\leq n} Be_i\mu$ where $\bigwedge$ is the conjonction operator (*and*) and $B((\mathcal{X}_r + \mathcal{B}) \rightarrow \mathcal{M} \rightarrow \mathcal{B})$ is the function $B \equiv \lambda x.\lambda\mu.(R_0 x\mu \neq 0 \text{ if } x \in \mathcal{X}_r, x \text{ if } x \in \mathcal{B})$. It is now possible to write a *consistent* representation as:

$$\tau = \left| \begin{array}{l} \{\mathtt{p == \&i}\}/[\mathtt{i} \leftarrow 2] \\ \{\mathtt{p\ != \&i}\}/[\mathtt{i} \leftarrow 1, *\mathtt{p} \leftarrow 2] \end{array} \right.$$

How can we be sure that our representations *represent* effectively transformations? This is accomplished by defining explicitly two operations: the *concretisation* to convert a transformation into its representation, and *abstraction* which is the reverse.

Conditionals are a simple extension of previous mechanisms. If we note T_t and T_f transformations for the *true* part and for *false* part respectively, the final transformation is thus: $TM = (\text{if } R_n[\![\mathrm{R}_n]\!]M \neq 0 \text{ then } T_v M \text{ else } T_f M)$. Note that integration of conditional statements is very easy, thanks to the definition of a robust framework and to the introduction of *guards*.

The implementation of our semantic analyzer accepts a normalized syntax tree in intermediate form as input. It computes the corresponding transformation. There are four main steps:

1. Instruction analysis, to compute compositions of transformation associated with each instruction.
2. Guard simplification: brute composition of transformations generates a lot of impossible guards. This module removes some of these guards using the memory partitions based on the complete type system.
3. Expression simplification.
4. Expression substitution. Given a transformation and a substitution, this module compute a new transformation where all possible expressions have been substituted.

5 Conclusion

We have defined a significant subset of the C language including pointers and pointer arithmetic, as well as side effects in expressions. Next, we have translated our program to remove side effects. Then, we elaborate a framework for semantic analysis. This framework benefits from the memory we partitioned following the type system. This partition scheme allowed us a fine grain simplification of expressions and guards. All these points were implemented in order to experiment with our approach.

The techniques exposed here can be applied to other domains: program verification against semantic rules of the C norm, interprocedural parallelization, or dependence graph computation.

However, our goal is now to perform the second step toward program parallelization: loop analysis. Our approach will be, given the transformation for the loop body, to compute approximations of the iterates of this transformation until it is stable. In performing these approximations, we will recognize *true* `for` loops, suitable for parallelization. But this will be the subject of another paper.

Acknowledgments: I would like to thank J. Taillandier for his assistance with the preparation of this article, and Pr. P. Feautrier for his helpful comments.

References

1. Allen R., Johnson S., *Compiling C for Vectorization, Parallelization, and Inline Expansion.* ACM PLDI88, June 1988
2. Ammarguellat Z., *Restructuration des programmes Fortran en vue de leur parallélisation.* Thèse de l'Université Paris 6, Rapport MASI 90.3, 1988.
3. American National Standard Institute, *Draft Proposed American National Standard for Information Systems – Programming Langage C.* Draft X3J11/88-002., January 1988.
4. Bodin F., *Data Structure Analysis in C Programs.* Int'l Workshop on Compilers for Parallel Computers, Paris, December 1990.
5. Chase D., Wegman M., Zadeck K., *Analysis of Pointers and Structures.* ACM-PLDI90, June 1990.
6. David P., *Analyse sémantique des programmes en langage C en vue de leur parallélisation.* Thèse de l'Université Paris 6, 1991 (Rapport MASI 91.60).
7. David P., *Semantic Analysis of Straight Line C Code with Pointers.* Rapport MASI 92.39 June 1992
8. Horwitz S., Pfeiffer P., Reps T., *Dependence Analysis for Pointer Variables.* ACM-PLDI89, 1989
9. Li K., *A Note on the Vector C Language.* ACM Symp. on Compiler Construction, SIGPLAN Notices Vol. 21, No 1, January 1986.
10. Polychronopoulos C., Girkar M., Haghighat M., Ling Lee C., Leung B., Schouten D., *Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors.* Int'l Conf. on Parallel Processing, 1989.
11. Thornburg J., *Number Crunching in C.* SIGPLAN Notices, Vol. 21, No 6, June 1986.
12. Triolet R., Irigoin F., Feautrier P., *Direct Parallelization of Call Statements.* ACM Symp. on Compiler Construction, SIGPLAN Notices Vol. 21, No 7, July 1986.
13. Zima H., Chapman B., *Supercompilers for Parallel and Vector Computers.* ACM Press, Addison Wesley, 1991.