Semantic Analysis of Straight Line C Code with Pointers

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

29 February 1992

Résumé

Automatic parallelization of C language programs is a subject rarely addressed: this can be explained by the difficulties inherent to the language, such as side effects within expressions or access to memory via pointers. The method we describe here is based on a semantic analysis of programs. The goal is to detect parallelism in complex programs using pointer arithmetic, thanks to the type system of the language. In order to perform this analysis, we define semantic domains suited to C. Then, we use them to synthesize effect of the program through functional objects called transformations.

Keywords: automatic parallelization, compilation, pointers, side effects, C language, types.

Résumé

La parallélisation automatique des programmes en langage C est un sujet rarement traité, ce qui peut être expliqué par les difficultés inhérentes au langage telles que les effets de bord dans les expressions ou les accès à la mémoire via des pointeurs. La méthode que nous décrivons ci-après est basée sur une analyse sémantique des programmes. Le but est de détecter le parallélisme dans les programmes complexes utilisant l'arithmétique sur les pointeurs, grâce au système de types du langage. Pour faire cette analyse, nous avons défini des domaines sémantiques adaptés au C. Nous les utilisons avec des objets fonctionnels nommés "transformations" pour synthétiser l'effet du programme.

Mots-clefs : parallélisation automatique, compilation, pointeurs, effets de bord, langage C, types.

1 Introduction

In the past few years, a considerable amount of work has been done in the field of automatic parallelization [2, 12, 15, 16, 17, 18, 21]. 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 [19]. 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 make some parallelized versions of C ([14] for example). In this paper, however, we focus on the difficult problems of automatic detection of parallelism. The most important one are:

- side effects in expressions (including assignment) hinder further analysis,
- some operators (`&&` for example) are sequential ones: they create a control dependence in expressions,
- the `for` loop, on which all parallelizers rely, is not reliable: they are disguised `while` loops and therefore cannot be used as the basis for parallelization,
- using small functions is considered good programming style: this makes parallelization of loops containing function calls difficult,
- 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 ([7, 10] among others), but rare are those directly applied to C [5]. 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. The intermediate form used is higher level than in traditional compilers, `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.

   Parafrase-2 parallelizer [15] includes a pre-processor and a post-processor for the C language. However, they are not sufficient to make the parallelizer fluent with C. Authors admit that dependence analysis with 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 in this paper. 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).

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. It is implemented as a pattern matching in a syntax tree and a set of transformation rules.

In the next section, we build the framework for semantic analysis. We have carefully designed this framework to be valid for the C complex syntax. Given this basis, we can use it to synthesize effect of instructions.

In the last section, we give some informations about the implementation we have made.

2 Input Language

Parallelization of C language programs leads to a lot of new and interesting research problems. Among them are the numerous data types, complex expressions with side effects and the very difficult problem of memory access via pointers.

The base language of our parallelizer is ANSI C [4, 11, 13]. 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 (especially parameter passing, where types are not verified when there is no function prototype) have been removed of our input language. Since there is no function call and cast, there is no dynamic memory allocation.

2. Problems not specific to C, such as interprocedural analysis [20] or program restructuring [3] are not dealt with.

   Function calls have already been removed. In addition, all non structured constructs are removed: `goto`, `continue`, `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 simple generalization of `if` statement. Note however that `switch` can also be a totally unstructured statement in some circumstances.

   `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 but cast and function call,

- has sequence of instructions, `if` statement, and `while` and `for` loops even if they are not yet presented.

Note that all expressions, even cluttered ones, are accepted. For example:

```
*(a++) = i%(n=3) && ((j||!k) ?
      *(b=tab)[c+=5] : 5[tab],*y),z ;
```

The input language is a non-trivial subset including a large part of difficulties mentioned earlier. Most removed features have been already addressed in previous works and we can concentrate on new and interesting problems.

3 Program Normalization

We impose few restrictions on our input language so it seems difficult to parallelize an expression like the one in the example above. Side effects are really frequent in C programs and we must deal with them.

However, since semantic analysis with side effects is not a simple task, we have chosen to remove them. The next sections explain how to do that.

The first one expose the language our semantic analyzer expects, the second one explains how to translate our input language into this new language.

3.1 Goal

The 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 more an operator, but an instruction. This forbids multiple assignments such as `a=b=0;`.
- Side-effect operators (`++`, `+=`, etc.). They are short-hand notations and they can easily be replaced by an equivalent sequence.
- Sequential operators (`&&`, `||` and `?:`). Here, the side effect is on program counter. For example, in the expression `x||y`, `y` is evaluated if and only if `x` evaluation returns a non zero value.
- 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 from expressions.

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

- assignment is now an instruction. Unlike true C where an expression is an instruction, expressions are only part of assignments and `while` or `if` tests.
- `for` loops are replaced by equivalent `while` based code. As there is a well-known equivalence between C `for` and `while` loops, no information is lost in the translation. We think that it is as complex to check that a `for` loop is a *true* `for` (i.e. semantics equivalent to Pascal `for` or Fortran `DO`) as to convert `while` loop to *true* `for`.

When a C language program is converted to this new language, semantic analysis can be done.

3.2 Normalization

Normalization is the translation of a program from the input language to the language we defined above. This normalization must obviously respect the semantics of the program.

We have defined a set of step-by-step transformations in a syntax tree. Normalization is just a pattern matching in this tree. Let us illustrate the idea with an example: assignment operator in the input program can be located anywhere in the syntax tree. We must make it move up, until it become the direct child of an instruction sequence node (every instruction is a child of an instruction sequence node in our implementation).

Here is an example of such syntactic transformations:

```
                          tmp = *r++,
*p++ = *q++ = *r++  <=>   *q++ = tmp,
                          *p++ = tmp
```

where `tmp` is a temporary variable created for this transformation. This variable is needed to guarantee only one evaluation of modified addresses. Comma is used because the new expression can be a part of a more complex expression.

Sequential operators must be converted to language instructions. For example:

```
          a = (b && c) ;
               ⇕
    a = (b ? (c ? 1 : 0) : 0) ;
               ⇕
b ? (c ? (a=1) : (a=0)) : (a=0) ;
               ⇕
if (b)
{
     if (c)
          a = 1 ;
     else a = 0 ;
}
else a = 0 ;
```

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 part introduces notations we use to build this model. The second part defines objects we manipulate. The third part uses these objects to define expressions evaluation. The remaining parts are devoted to each instruction type.

4.1 Notations

Set names are typeset with calligraphic letters (like $\mathcal{A}$). Cartesian product is noted $\times$ and $\langle e_1, e_2, \ldots e_n\rangle$ is a member of $\mathcal{E}_1 \times \mathcal{E}_2 \times \ldots \mathcal{E}_n$. We use the $+$ operator to note the disjoint sum of two sets, i.e. $\mathcal{E}_1 + \mathcal{E}_2 = \mathcal{E}_1 \cup \mathcal{E}_2$ if $\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset$. We note $P(\mathcal{E})$ the set of all subsets of $\mathcal{E}$.

We define the type of a function with $f : \mathcal{I} \rightarrow \mathcal{O}$, which means that function f has a domain $\mathcal{I}$ and a codomain $\mathcal{O}$. Function definitions are written using the lambda-notation, i.e. $\lambda p.e$ where p is the parameter and e is the expression. Application of a to the f function is noted fa. In order to avoid writing partially defined functions, we use the $\top$ (with a unique member $\top$) as a part of the output set. We note $f = \cup f_i$ the function built from functions f_i with disjoint input sets.

Lastly, we use the logical operators $\wedge$ (*and*), $\vee$ (*or*) and $\neg$ (*not*).

4.2 Semantic Domains

In order to study semantic analysis of C language programs, we must carefully define semantic domains. Since C is not a simple language, semantic domains are necessarily complex.

4.2.1 Base Semantic Domains

First, we define the base 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 object identifiers

There is no provision for floating point values. Addition of floating point support does not change our method and for brevity, we did not include it.

The domain $\mathcal{I}$ contains all identifiers that reference an object in data memory, either in stack or in static data storage. Procedure names, `struct` and `union` names, field names and `typedef` names are not elements of $\mathcal{I}$.

4.2.2 Address Domain

With these domains, it is possible to construct the *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. The address domain is particular to one program: intuitively, if two different programs use the same symbol i, there is no reason to bind the same address to this identifier. Following the traditional convention, we will prepend an underscore to each C language symbol.

Each memory object has a size, which limits its offsets within a memory object. In other words, an address $\langle i, k \rangle$ is valid if and only if $0 \leq k < d_m(i)$, where $d_m(i)$ is the size (maximum displacement) of the object referenced by i.

We can now define some operations on addresses:

- addition and subtraction of an integer:

$$\begin{aligned} + :\ & \mathcal{A} \rightarrow \mathcal{N} \rightarrow \mathcal{A} \\ & \lambda\langle x, e_1\rangle.\lambda e_2.\langle x, e_1 + e_2\rangle \\ - :\ & \mathcal{A} \rightarrow \mathcal{N} \rightarrow \mathcal{A} \\ & \lambda\langle x, e_1\rangle.\lambda e_2.\langle x, e_1 - e_2\rangle \end{aligned}$$

- address difference, if addresses are references inside the same object:

$$\begin{aligned} - :\ & \mathcal{A} \rightarrow \mathcal{A} \rightarrow \mathcal{N} \\ & \lambda\langle x, e_1\rangle.\lambda\langle x, e_2\rangle.e_2 - e_1 \end{aligned}$$

- inequality operators ($<$, $\leq$, $>$ and $\geq$), if addresses both reference the same object:

$$\begin{aligned} \text{op} :\ & \mathcal{A} \rightarrow \mathcal{A} \rightarrow \mathcal{B} \\ & \lambda\langle x, e_1\rangle.\lambda\langle x, e_2\rangle.e_1 \text{ op } e_2 \end{aligned}$$

- equality operators:

$$
\begin{aligned}
&=: \mathcal{A} \rightarrow \mathcal{A} \rightarrow \mathcal{B}\\
&\quad \lambda\langle x_1, e_1\rangle.\lambda\langle x_2, e_2\rangle.(x_1 = x_2 \wedge e_1 = e_2)\\
&\neq: \mathcal{A} \rightarrow \mathcal{A} \rightarrow \mathcal{B}\\
&\quad \lambda\langle x_1, e_1\rangle.\lambda\langle x_2, e_2\rangle.(x_1 \neq x_2 \vee e_1 \neq e_2)
\end{aligned}
$$

The semantic domain $\mathcal{A}$ is too large for our purpose. In particular, there is no type notion. We introduce it by splitting $\mathcal{A}$: we state that it 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. Informally, n in $\mathcal{A}_n$ is the number of stars in the declaration of objects included in $\mathcal{A}_n$. For example :

`int i ;`	$\langle \_\mathtt{i}, 0\rangle \in \mathcal{A}_0$
`int *pi ;`	$\langle \_\mathtt{pi}, 0\rangle \in \mathcal{A}_1$
`int **ppi ;`	$\langle \_\mathtt{ppi}, 0\rangle \in \mathcal{A}_2$

This partition is designed to deal with pointers. We could extend it to handle all types and re-partition each $\mathcal{A}_n$ in subsets according to the basic type ($\mathcal{A}_n^{\mathtt{char}}$, $\mathcal{A}_n^{\mathtt{int}}$, etc.). The implementation uses this double partitioning scheme, but it is omitted in this article for the sake of clarity.

Partitioning in sets $\mathcal{A}_n$ is guaranteed by the semantic rules of C and our restrictions: absence of casts, function calls and `union` types make type violation impossible, and access to variables out of bound is forbidden by the norm. Hence, a variable is used as it is declared. We can formalize this rule by:

Property 1 $\forall\, i \in \mathcal{I}, \langle i, 0\rangle \in \mathcal{A}_n \Longrightarrow$
$\forall\, k \in \mathcal{N}\ (0 \leq k < d_m(i)), \langle i, k\rangle \in \mathcal{A}_n$

What about arrays? C language sometimes mixes arrays and pointers. For example, `t1[i]` is equivalent to `*(t1+i)` to access the $(i+1)$-th array element. This rule is valid only for the last dimension of an array. For example, `t3[i][j][k]` is not equivalent to `*(*(*(t3+i)+j)+k)`. Only one memory reference is needed, hence `t3[i][j][k]` is equivalent to `*(t3[i][j]+k)` and `t3[i][j]` is an address computed with `i` and `j` without any memory reference. The rule is: *whatever the number of dimensions of an array, access to one element is made with one and only one memory reference.* This allow us to think of an array as an association of simple variables, hence the definition:

Definition 1 *A variable (addresses $\langle i, k\rangle$) is in $\mathcal{A}_n$ iff the declaration of i contains n stars.*

In other words, there is $n+1$ memory references (implicit or explicit with the `*` operator) to get the pointed value. Let us give some examples:

`int ti[10] ;`	$\langle \_\mathtt{ti}, 0\rangle \in \mathcal{A}_0$
`int *tpi[10] ;`	$\langle \_\mathtt{tpi}, 0\rangle \in \mathcal{A}_1$
`int (*pti)[10] ;`	$\langle \_\mathtt{pti}, 0\rangle \in \mathcal{A}_1$
`int *(*ptpi)[10] ;`	$\langle \_\mathtt{ptpi}, 0\rangle \in \mathcal{A}_2$
`int (*tpti[10])[20] ;`	$\langle \_\mathtt{tpti}, 0\rangle \in \mathcal{A}_1$

To be exhaustive, we must speak of *virtual* arrays: function parameters declared as n-dimensions arrays are not true arrays. They are pointers to $(n-1)$-dimensions arrays. If we have the parameter `int t[10][20]`, we must rewrite it to the equivalent form `int (*t)[20]`. This last form implies $\langle \_\mathtt{t}, 0\rangle \in \mathcal{A}_1$, which is compatible with parameter passing rules.

4.2.3 Partitioning the $\mathcal{I}$ Domain

As we partitioned the $\mathcal{A}$ domain, we must partition the $\mathcal{I}$ domain with the relation:

$$\begin{array}{c}(i,j) \in \mathcal{I}, i \text{ is related with } j \Longleftrightarrow \\ \exists\, n/\langle i,0\rangle \in \mathcal{A}_n \text{ and } \langle j,0\rangle \in \mathcal{A}_n\end{array}$$

As this relation is an equivalence relation, it defines a partition.

In set $\mathcal{I}_n$, 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.

4.2.4 Memory Domain

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:

$$\begin{array}{rrcl} & M_0 & : & \mathcal{A}_0 \rightarrow \mathcal{V} + \top \\ \forall n > 0, & M_n & : & \mathcal{A}_n \rightarrow \mathcal{A}_{n-1} + \top \end{array}$$

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.

Definition 2 *A memory is the function* $M = \bigcup_{n \geq 0} M_n$

Definition 3 *The set of all memories is called* $\mathcal{M}$.

For example, after execution of the following program segment:

```
x = 1 ;
*(t + 0) = 2 ;
*(t + 1) = 3 ;
p = t ;
```

the memory looks like:

$$M = \lambda a. \begin{cases} 1 & \text{if } a = \langle \_\mathtt{x}, 0\rangle \\ 2 & \text{if } a = \langle \_\mathtt{t}, 0\rangle \\ 3 & \text{if } a = \langle \_\mathtt{t}, 1\rangle \\ \langle \_\mathtt{t}, 0\rangle & \text{if } a = \langle \_\mathtt{p}, 0\rangle \\ \top & \text{else} \end{cases}$$

4.3 Expression Interpretation

Let us define exactly the syntax of expressions:

$$\begin{array}{lcl} E & ::= & R|L \\ R & ::= & R_0|R_1|\ldots|R_n|\ldots \\ R_0 & ::= & \text{const}|\text{ident}_0|(R_0)|R_0 \text{ op}_2\, R_0 \\ & & |\text{op}_1\, R_0|R_i - R_i|{*}R_1 \\ R_n & ::= & \text{const}_{n-1}|\text{ident}_n|(R_n)|R_n\mathtt{+}R_0 \\ & & |R_n - R_0|{*}R_{n+1}|\&L_{n-1} \\ L & ::= & L_0|L_1|\ldots|L_n|\ldots \\ L_n & ::= & \text{ident}_n|(L_n)|{*}L_{n+1} \end{array}$$

Where op_1 and op_2 are unary and binary operators respectively. 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 :

$$E : \mathcal{X} \to \mathcal{M} \to \mathcal{A} + \mathcal{V}$$

$$\begin{array}{rl}
R_0[\![\text{const}]\!] & \equiv \lambda\mu.\text{const} \\
R_0[\![\text{ident}_0]\!] & \equiv \lambda\mu.\mu(\langle\text{ident}_0, 0\rangle) \\
R_0[\![(R'_0)]\!] & \equiv \lambda\mu.R_0[\![R'_0]\!]\mu \\
R_0[\![R'_0 \text{ op}_2\ R''_0]\!] & \equiv \lambda\mu.R_0[\![R'_0]\!]\mu \text{ op}_2\ R_0[\![R''_0]\!]\mu \\
R_0[\![\text{op}_1\ R'_0]\!] & \equiv \lambda\mu.\text{op}_1\ R_0[\![R'_0]\!]\mu \\
R_0[\![R'_i - R''_i]\!] & \equiv \lambda\mu.R_i[\![R'_i]\!]\mu - R_i[\![R''_i]\!]\mu \\
R_0[\![*R'_1]\!] & \equiv \lambda\mu.\mu(R_1[\![R'_1]\!]\mu) \\
 & \\
R_n[\![\text{const}_{n-1}]\!] & \equiv \lambda\mu.\langle\text{const}_{n-1}, 0\rangle \\
R_n[\![\text{ident}_n]\!] & \equiv \lambda\mu.\mu(\langle\text{ident}_n, 0\rangle) \\
R_n[\![(R'_n)]\!] & \equiv \lambda\mu.R_n[\![R'_n]\!]\mu \\
R_n[\![R'_n + R'_0]\!] & \equiv \lambda\mu.R_n[\![R'_n]\!]\mu + R_0[\![R'_0]\!]\mu \\
R_n[\![R'_n - R'_0]\!] & \equiv \lambda\mu.R_n[\![R'_n]\!]\mu - R_0[\![R'_0]\!]\mu \\
R_n[\![*R'_{n+1}]\!] & \equiv \lambda\mu.\mu(R_{n+1}[\![R'_{n+1}]\!]\mu) \\
R_n[\![\&L'_{n-1}]\!] & \equiv \lambda\mu.L_{n-1}[\![L'_{n-1}]\!]\mu \\
 & \\
L_n[\![\text{ident}_n]\!] & \equiv \lambda\mu.\langle\text{ident}_n, 0\rangle \\
L_n[\![(L'_n)]\!] & \equiv \lambda\mu.L_n[\![L'_n]\!]\mu \\
L_n[\![*R_{n+1}]\!] & \equiv \lambda\mu.R_{n+1}[\![R_{n+1}]\!]\mu
\end{array}$$

We can prove that:

$$\begin{array}{rl}
R_0 : & \mathcal{X}_r \to \mathcal{M} \to \mathcal{V} \\
\forall n \geq 1, R_n : & \mathcal{X}_r \to \mathcal{M} \to \mathcal{A}_{n-1} \\
\forall n \geq 0, L_n : & \mathcal{X}_l \to \mathcal{M} \to \mathcal{A}_n
\end{array}$$

We note $L = \bigcup_{n\geq 0} L_n$ and $R = \bigcup_{n\geq 0} R_n$.

4.4 Assignment

With the expression syntax above, assignment is simply:

$$\text{assignment} ::= \text{L}_n\texttt{=}\text{R}_n\,;$$

The effect of this assignment is to modify the memory. If the memory before assignment is called M, the new memory become:

$$M' = \lambda a. \begin{cases} \text{if } a = L_n[\![\text{L}_n]\!]M & \text{then } R_n[\![\text{R}_n]\!]M \\ & \text{else } Ma \end{cases}$$

The simple form of this new memory is due to the absence of side effects in expressions. Thus, the same memory is used to evaluate both the left hand side and the right hand side.

An assignment can be viewed as a function which modifies a memory. We call this a *transformation*, and we write:

$$T : \mathcal{M} \to \mathcal{M}$$

How can we deal efficiently with these transformations? Our implementation needs an easily manipulated representation of objects. One possible representation is based on the fact that elements used to define the M' function are the two expressions lhs_n and rhs_n, the two interpretation functions L_n and R_n, and the memory M. Among these elements, definition of M is not used, functions L_n and R_n were defined in the previous subsection and will not change. Hence, the only variable elements are expressions L_n and R_n. The transformation depends on these two expressions, elements of $\mathcal{X}$. This leads us to the representation we are looking for:

Definition 4 *We represent the transformation T (for the assignment $l = r$) by a substitution, i.e. the value $\langle l, r\rangle \in \mathcal{X}_l \times \mathcal{X}_r$, 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. For example, if we need to represent the transformation for `i=5`, we can write $\sigma = [\mathtt{i} \leftarrow 5]$.

4.5 Sequence

In this subsection, we define the transformation for a sequence of assignments as the composition of individual assignments. But ambiguity appears in the representation, which leads us to a new representation of transformations.

4.5.1 Composition of transformations

A sequence of instructions is a list of assignments. Transformation for the sequence is the *composition* of individual transformations. For example, given the program segment (`int` variables assumed):

```
i = 1 ; j = 2 ;
```

The composition of the transformations T_1 (for `i=1`) and T_2 (for `j=2`) gives:

$$\begin{aligned} T_3 M &= (T_1 \circ T_2) M \\ &= \lambda a. \begin{cases} \text{if } a = \langle \_\mathtt{i}, 0 \rangle & \text{then } 1 \\ \text{elsif } a = \langle \_\mathtt{j}, 0 \rangle & \text{then } 2 \\ & \text{else } Ma \end{cases} \end{aligned}$$

And we can write the representation:

$$\sigma = [\mathtt{i} \leftarrow 1, \mathtt{j} \leftarrow 2]$$

4.5.2 New representation

However, an important problem arises with this representation: pointers lead to ambiguities. If we are given the following program segment:

```
i = 1 ; *p = 2 ;
```

Composition gives the transformation (the order of conditions is important):

$$TM = \lambda a. \begin{cases} \text{if } a = M(\langle \_\mathtt{p}, 0 \rangle) & \text{then } 2 \\ \text{elsif } a = \langle \_\mathtt{i}, 0 \rangle & \text{then } 1 \\ & \text{else } Ma \end{cases}$$

Which is unambiguous: if we are interested in the value of variable `i`, if `p` contains the address of variable `i`, then the value is `2` else `1`. However, the representation of this transformation is: $[\mathtt{*p} \leftarrow 2, \mathtt{i} \leftarrow 1]$. The value assigned to variable `i` depends on the address pointed to by `p`.

As soon as an indirection via a pointer appears in a transformation, representations become ambiguous. We formalize this idea by the notion of *consistency*:

Definition 5 *The representation*

$$[l_1 \leftarrow r_1, \ldots l_n \leftarrow r_n]$$

of a transformation is consistent iff:

$$\forall M \in \mathcal{M}, L[\![l_i]\!]M = L[\![l_j]\!]M \Longrightarrow i = j$$

In other words, a representation is consistent if all left hand side expressions address different memory cells, whatever the memory configuration is.

How to make a representation consistent? We adopt a pragmatic approach: if representations are ambiguous, we must extend these representations. The idea of the solution we use is to define functions which map a memory to a boolean value ($G : \mathcal{M} \rightarrow \mathcal{B}$). The role of these functions is to select memory configurations when there is an ambiguity. To ease manipulations, we need to deal with a limited form of these functions, which we call *guards*.

Definition 6 *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 is the function:*

$$\begin{array}{ll} B: & (\mathcal{X}_r + \mathcal{B}) \rightarrow \mathcal{M} \rightarrow \mathcal{B} \\ & \lambda x.\lambda\mu. \begin{cases} R_0 x\mu \neq 0 & \textit{if } x \in \mathcal{X}_r \\ x & \textit{if } x \in \mathcal{B} \end{cases} \end{array}$$

A guard function is defined by its terms, each one being defined by the e_i expression. Let us take again our previous example:

```
i = 1 ; *p = 2 ;
```

The ambiguity is due to the fact that the `p` pointer could address the `i` variable. Hence, we must distinguish two cases ({`p == &i`} and {`p != &i`}) in order to write a consistent transformation representation:

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

We thus extend the transformation representation for a single assignment (a couple of expressions) into a new representation for a sequence of instructions.

When there is no ambiguity, there is no need to use guards. But, in order to be homogeneous, we use the *true guard* (noted {true}). Now we can define the general representation of transformations:

Definition 7 *We represent the transformation T (for an instruction or a sequence of instructions) by a set τ of couples $\langle\gamma_i, \sigma_i\rangle$ where $\gamma_i \in P(\mathcal{X}_r) + \mathcal{B}$ and $\sigma_i \in P(\mathcal{X}_l \times \mathcal{X}_l)$*

4.5.3 Abstraction and Concretisation

So far, we have seen transformations on one side, and representations on the other one. We must be sure that representations *represent* effectively transformations, because implementation knows only representations. In other words, we must be able to compute the representation from the transformation (we call this operation a *concretisation*) and vice-versa (we call this operation an *abstraction*). Since these operations and their proofs are long and tedious, we just give the method outlines. All details are in [9].

First, we give a general form, more precise, of transformations. This is:

$$TM = \lambda a. \begin{cases} \text{if } a = f_1 M \wedge g_1 M & \text{then } h_1 M \\ \quad \vdots & \\ \text{elsif } a = f_n M \wedge g_n M & \text{then } h_n M \\ & \text{else } Ma \end{cases}$$

where, with $1 \leq i \leq n$, f_i is a function $\mathcal{M} \rightarrow \mathcal{A}$, g_i is a function $\mathcal{M} \rightarrow \mathcal{B}$ and h_i is a function $\mathcal{M} \rightarrow \mathcal{A} + \mathcal{V}$. Assignment trivially gives transformation in the above form. Next, we prove that this form is stable by composition. This is verified by exhaustive examination of cases of expression syntax for f_i, g_i and h_i functions.

Abstraction is only a matter of rewriting with L and R functions.

Concretisation, however, is more complex. We have seen that the order of conditions is important. The first thing is to eliminate this constraint. We proceed by adding terms $\neg f_i$ to all lines j ($j > i$). This introduces *or* operators which we suppress by lines duplication. Next, we introduce L^{-1} and R^{-1} functions which translate a formula into an expression of the C language (member of $\mathcal{X}$). At last, we can rewrite our system to a representation.

4.6 Conditional

The syntax of conditional statement is:

$$\begin{array}{rl} \text{cond} ::= & \texttt{if (}\mathrm{R}_n\texttt{)}\ \text{inst} \\ | & \texttt{if (}\mathrm{R}_n\texttt{)}\ \text{inst}\ \texttt{else}\ \text{inst} \end{array}$$

Let us look at the general (if-else) case. We note T_t and T_f transformations for the *true* part and for *false* part respectively. If there is no *false* part, T_f is the identity function ($\mathrm{id}_\mathcal{M} : \mathcal{M} \rightarrow \mathcal{M}$). The final transformation is thus:

$$TM = \begin{cases} \text{if } R_n[\![\mathrm{R}_n]\!]M \neq 0 \text{ then} & T_v M \\ \text{else} & T_f M \end{cases}$$

As T_v and T_f are two conformant transformation, the development of T yields to a conformant transformation. Hence, abstraction and concretisation operations are still valid.

Note that integration of conditional statements is very easy, thanks to the definition of a robust framework and to the introduction of *guards*.

5 Implementation

This section describes the implementation we realized. First, we present the syntactic analyzer and intermediate representation. Next, we present the program normalizer and structure of the rules we used. The last part describe semantic analysis.

5.1 Syntactic analyzer

The input to the syntactic analyzer is a program in ANSI C. It is translated into a syntax tree written as Lisp expressions. This is our intermediate representation, which is described in detail in [8]. One important point of this representation is the precision of memory access operators, without bothering with data types peculiarities. At this stage, the syntax tree still contains some side-effects operators.

This translator itself is written in C language using `lex` and `yacc` tools. It totals 15 000 lines, heavily commented. Its internal structure (data and code) looks like a Lisp interpreter [6], with a reader and a printer and all classical operations on conses.

Time used by the translator is negligible when compared with other phases. For example, a program containing 10 functions, each with 100 assignments `*p1=x1+x2+*p1+*p2;`, is translated in 6.6 seconds on a 68040 based workstation (including pre-processing time). Note that the native compiler (without assembly phase) took 10.7 seconds on the same example.

5.2 Normalizer

The program normalizer accepts a syntax tree in intermediate form as input. Its engine is a pattern matching filter. It uses rules which are made of:

1. pattern to look for,
2. pattern to replace with,
3. temporary variables to create if any.

For example, a typical rule is:

```
((unary-op (if e0 e1 e2))
 (if e0 (unary-op e1) (unary-op e2))
 nil
 )
```

meaning that if the expression `-(a?b:c)` is encountered, it is replaced with `a?-b:-c` without any temporary variable creation.

The normalizer totals 800 lines in Common Lisp, of which more than one half is devoted to the rules. Analysis of a typical function (10 lines) is done in less than 10 seconds.

5.3 Semantic Analyzer

The 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. This module compute successive 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.

6 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 to 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 article.

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] Allen J., Kennedy K., *Automatic Translation of Fortran Programs to Vector Form.* ACM Trans. on Programming Languages and Systems, Vol. 9, No 4, October 1987.

[3] Ammarguellat Z., *Restructuration des programmes Fortran en vue de leur parallélisation.* Thèse de l'Université Paris 6, Rapport MASI 90.3, 1988.

[4] American National Standard Institute, *Draft Proposed American National Standard for Information Systems – Programming Langage C.* Draft X3J11/88-002., January 1988.

[5] Bodin F., *Data Structure Analysis in C Programs.* Int'l Workshop on Compilers for Parallel Computers, Paris, December 1990.

[6] Chailloux J., Devin M., Hulot J.M., *Le_Lisp, A Portable and Efficient Lisp System.* INRIA Res. Rep. 319, July 1984.

[7] Chase D., Wegman M., Zadeck K., *Analysis of Pointers and Structures.* ACM-PLDI90, June 1990.

[8] David P., *Une représentation intermédiaire pour la compilation du langage C.* Rapport MASI 300, Octobre 1989.

[9] 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).

[10] Horwitz S., Pfeiffer P., Reps T., *Dependence Analysis for Pointer Variables.* ACM-PLDI89, 1989

[11] Harbison S., Steele G. Jr, *C, A Reference Manual, second edition.* Prentice Hall, 1987.

[12] Kuck D., Kuhn R., Leasure B., Wolfe M., *The Structure of an Advanced Vectorizer for pipelined Processors.* Fourth Int'l Computer Software and Applications Conference IEEE, October 1980.

[13] Kernighan B., Ritchie D., *The C Programming Langage, second edition.* Prentice Hall, 1988.

[14] Li K., *A Note on the Vector C Language.* ACM Symp. on Compiler Construction, SIGPLAN Notices Vol. 21, No 1, January 1986.

[15] 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.* Proc. of the 1989 Int'l Conference on Parallel Processing, 1989.

[16] Padua D., Kuck D., Lawrie D., *High Speed Multiprocessors and Compilation Techniques.* IEEE Trans. on Computers, C29, September 1980.

[17] Padua D., Wolfe M., *Advanced Compiler Optimizations for Supercomputers.* CACM, Vol. 29, No 12, December 1986.

[18] Tawbi N., Dumay A., Feautrier P., *PAF : un paralléliseur pour Fortran.* Rapport MASI 185, Mai 1987.

[19] Thornburg J., *Number Crunching in C.* SIGPLAN Notices, Vol. 21, No 6, June 1986.

[20] Triolet R., Irigoin F., Feautrier P., *Direct Parallelization of Call Statements.* ACM Symp. on Compiler Construction, SIGPLAN Notices Vol. 21, No 7, July 1986.

[21] Zima H., Chapman B., *Supercompilers for Parallel and Vector Computers.* ACM Press, Addison Wesley, 1991.