The Program Compaction Revisited: the Functional Framework

Marc Pouzet

VERIMAG, Miniparc-ZIRST, Rue Lavoisier, 38330 Montbonnot St-Martin, France
e-mail Marc.Pouzet@imag.fr.

Abstract. This paper presents a general method to compact the first-order part of functional languages with call-by-value semantics for fine-grain parallel machines like VLIW or super-scalars. This work extends previous works on compaction in two ways. First, it defines a new formal system for the compaction problem usable to design a meta-compiler for these machines. Second, the compaction is directly applied to functional expressions instead of graph based representations (control flow or dependence flow based representations) leading to a very uniform and simple presentation.

1 Introduction

VLIW (Very Long Instruction Word) [14] and super-scalar architectures [11, 5] are fine grain parallel and compiled architectures in the sense that they can execute many instructions per cycle, gathered together by a compiler. VLIW are controlled by a single instruction stream (one program counter) where each processor executes a dedicated field of a long instruction. In a super-scalar machine, the processor executes successive RISC-like instructions belonging to a small window by analyzing at runtime their dependencies.

Fine grain parallelization for these architectures, or *compaction*, statically recognizes and schedules groups of elementary operations that can be executed in parallel. Compaction can be *local* — it is then limited to expressions without branches, or *global* — and it treats conditional expressions. Loops are compiled by *Software Pipelining.*

We propose in this paper a new and formal presentation of compaction, directly applied to functional expressions (terms). It can be used as a first step in the problem of designing meta-compacting compilers but also as a first step toward fast and efficient implementations of functional languages on fine-grain parallel machines.

This paper is organized as follows. The first part motivates the point of view adopted in this paper. We then give some examples. The next part is the formalization: we first define a functional language and its operational semantics. The classical notion of dependence is defined by an equivalence relation named *structural equivalence*, between programs having the *same dependences.* It serves as a guide for the formalization: every program transformations will have to preserve this equivalence. Then we build a transformational system on terms that improves programs for a given machine.

2 Motivations

Compacting compilers and, more generally, parallelizing compilers, can be decomposed into two classes. Compilers in the first class are applied to control flow based representations (basic bloc graphs) where elementary instructions "percolate" in the graph [8, 7, 1, 15, 6]. In this framework, software pipelining is done by iterative methods with a controlled inlining of loops, looking for a repeated behavior [1, 6]. For compilers in the second class, compaction is applied to dependence graph representations of programs, extending scheduling techniques to programs containing control structures [18, 13]. Software pipelining is then applied to graphs with cycles. The tradeoff is between *generality* and *efficiency*. Control based representation is the most general framework since it is the low level representation of every program. Nonetheless, it leads to very poor implementations — operations moves are done in linear time. Dependence graph based representation leads to efficient implementations — operation moves are done in constant time — but only a subset (without tests) of a real language is well treated in this framework. In particular, software pipelining is applied to elementary **do** loops with no tests or nested loops. When control is added (e.g, Lam's *hierarchical reduction* system [13]), the control is boxed, which forbids the best possible compaction.

This paper investigates a formalization of compaction, thus we have to define an intermediate language and an abstract notion of fine-grain parallel machines. We have to balance this tradeoff: the intermediate language has to be general enough to represent a real subset of classical languages, but it also has to be practical (the representation must lead to efficient implementations). Moreover, this formalization has to be simple enough to allow reasoning about compaction. Then, some program transformations (renaming, duplication, unrolling, etc.) should be possible. Lastly, to be general enough, it has to compact recursive and functional programs: to compact such programs, it makes no sense to translate them into some low level control graph representation and then retrieve their dependences, since in a functional language, dependences are simply read off the syntax. Moreover, in a functional language, sections of programs can be guaranteed to be free of side-effects by the type system [1] and have the Static Single Assignment (SSA) property. As it fails to represent the control of all kind of programs, the dependence graph representation alone is not suited either as an intermediate representation of programs. The solution proposed here is between control flow based representations and dependence flow based representations.

Compaction will be defined by a set of transformations directly applied to functional *terms*, each of them *improving* the term to some extent. The language is an extension of dependence graph representation — a dependence graph is a particular case of a functional expression [2] — but is able to represent the control of program. With the functional representation, the compiler manage at the same

[1] Though they manage data-structures (lists,...) represented with pointers.

[2] The let definition in a functional expression defines the sharing, i.e, Directed Acyclic Graphs.

time the dependence and the control information on the program and program transformations are based on the well known substitution principle. In order to define a formal system for the compaction problem, we propose an operational semantics to give a measure to expressions — every compacting transformations will have to improve programs — and a notion of ***dependences.*** The operational semantics defines the way a fine-grain parallel machine executes a program. The notion of dependences is also important. Indeed, as compaction usually respects this notion in the classical framework, how this notion applies in the functional case, in order to have the same expressiveness? In other words, what are the program transformations a reasonable compiler can do? Here, the notion will be defined by an equivalence relation, named ***structural*** that will permit classical transformations. Finally, because functional terms can be represented with sharing, it allows better implementations than with the classical control-flow based representation.

3 Examples

Let us try first to compact the simple program P_1 given on the left of the following figure. It executes an addition ($+_1$) between a variable and a constant. It binds the result to the variable x and then executes the operation $+_2$ and then $-_3$, the test and one of the two branches. Finally, it computes $*_5$ between the value of $(x +_2 4) -_3 x$ and the value of the conditional expression.

$$\begin{array}{rl} P_1 = & \mathsf{let}\, x = \overline{y +_1 2} \\ & \mathsf{in}\ ((x +_2 4) -_3 x) *_5\ \mathsf{if}\ \overline{z =_2 0} \\ & \qquad\qquad\qquad\quad \mathsf{then\ let}\, m = \overline{4 +_3 y} \\ & \qquad\qquad\qquad\qquad\quad \mathsf{in}\ m *_4 n \\ & \qquad\qquad\qquad\quad \mathsf{else}\, x *_3 2 \end{array}$$

$$\begin{array}{rl} P_2 = & \mathsf{let}\, x = y +_1 2 \\ & \mathsf{in\ if}\, z =_2 0 \\ & \quad \mathsf{then}\ ((x +_2 4) -_3 x) *_5 (\,\mathsf{let}\, m = \overline{4 +_3 y}) \\ & \qquad\qquad\qquad\qquad\qquad\ \ \mathsf{in}\ m *_4 n \\ & \quad \mathsf{else}\ ((x +_2 4) -_3 x) *_5 (x *_3 2) \end{array}$$

$$\begin{array}{rl} P_3 = & \mathsf{let}\, x = y +_1 2 \\ & \qquad m = 4 +_3 y \\ & \mathsf{in\ if}\, z =_2 0 \\ & \quad \mathsf{then\ let}\, x_1 = x +_2 4 \\ & \qquad\qquad\quad x2 = m *_4 n \\ & \qquad\quad \mathsf{in\ let}\, x_1 = x_1 -_3 x \\ & \qquad\qquad \mathsf{in}\ x_1 *_5 x_2 \\ & \quad \mathsf{else\ let}\, x_1 = x +_2 4 \\ & \qquad\qquad\quad x_2 = x *_3 2 \\ & \qquad\quad \mathsf{in\ let}\, x_1 = x_1 -_3 x \\ & \qquad\qquad \mathsf{in}\ x_1 *_5 x_2 \end{array}$$

Consider now a machine $\mathcal{M}$ able to execute this program in a left-to-right evaluation order and where three successive and independent instructions can be executed in parallel. The execution time of P_1 on $\mathcal{M}$ is 5. Let us show how a simple transformation *improves* this execution time. Consider the elementary computations that can be executed immediately in parallel. We term these computations *ready sub-terms.* They are overlined here. $+_3$ may be executed speculatively before the test because it is a *safe* computation [3]. We apply a move-up rule to the test, getting P_2. Now, $+_3$ can be moved-up. We then continue the

[3] A computation is *safe* when it can be executed speculatively without modifying the semantics.

compaction with the two branches, getting at the end P_3. This program can not be compacted further and its execution time on $\mathcal{M}$ is 4. It is decomposed into groups of independent instructions executed in parallel.

Consider now the case of a simple recursive function F written in a PCF style [10]. It is an iteration on a list where [] stands for the empty list, *hd* and *tl* for the head and the tail of the list (x and a are the arguments).

$$F = \mathsf{Fix}_f \left(\begin{array}{l} \lambda\,[x;a].\,\mathsf{if}\, x = [\,] \\ \qquad\quad \mathsf{then}\, a \\ \qquad\quad \mathsf{else}\, f[\overline{tl(x)}; -(\overline{hd(x)}) + a] \end{array} \right)$$

The global compaction on this program would compact the body of the function, ignoring the possible overlapping between iterations, thus the parallelism. For this reason, we apply an inlining rule: $\mathsf{Fix}_f(a) \to a[f \backslash \mathsf{Fix}_f(a)]$ meaning that all occurrences of the free variable f are replaced by its definition. We get the following term given on the left. Now the only ready sub-term is the test. We then move the overlined ready sub-terms in the false branch, getting the program on the right.

$$\begin{array}{l} \lambda\,[x;a].\,\mathsf{if}\, x = [\,] \\ \qquad\quad \mathsf{then}\, a \\ \qquad\quad \mathsf{else}\, F[\overline{tl(x)}; -(\overline{hd(x)}) + a] \end{array} \qquad \begin{array}{l} \lambda\,[x;a].\,\mathsf{if}\, x = [\,] \\ \qquad\quad \mathsf{then}\, a \\ \qquad\quad \mathsf{else}\ \mathsf{let}\, x_1 = tl(x) \\ \qquad\qquad\qquad\ \ x_2 = hd(x) \\ \qquad\qquad\quad \mathsf{in}\ F[x_1; -x_2 + a] \end{array}$$

The current term to be compacted is $F[x_1; -x_2 + a]$. Because the first argument is computed, the inlining of F will show new ready sub-terms to be executed in parallel with the current ready sub-terms. After the inlining, the following application has to be compacted.

$$(\lambda[x;a].\,\mathsf{if}\, x = [\,]\,\mathsf{then}\, a\, \mathsf{else}\, F[tl(x); -(hd(x)) + a])([x_1; -x_2 + a])$$

A simple renamming rule is used for simplifying the application. We get the new term:

$$(\lambda a.\,\mathsf{if}\, \overline{x_1 = [\,]}\,\mathsf{then}\, a\,\mathsf{else}\, F[tl(x_1); -(hd(x_1)) + a])(\overline{-x_2} + a)$$

We can move-up the ready sub-terms. The resulting program is given below, on the left. In this term, the sub-expression $(\lambda a.a)(x_3 + a)$ can be simplified in $x_3 + a$. After the first step of compaction on the false branch, we have the term given on the right.

$$\begin{array}{l} \mathsf{let}\, x_3 = -x_2 \\ \mathsf{in}\ \ \mathsf{if}\, x_1 = [\,] \\ \quad \mathsf{then}\, (\lambda a.a)(x_3 + a) \\ \quad \mathsf{else}\, (\lambda a.F[\overline{tl(x_1)}; -(\overline{hd(x_1)}) + a])(\overline{x_3 + a}) \end{array} \qquad \begin{array}{l} \mathsf{let}\, x_3 = -x_2 \\ \mathsf{in}\ \ \mathsf{if}\, x_1 = [\,] \\ \quad \mathsf{then}\, x_3 + a \\ \quad \mathsf{else}\ \mathsf{let}\, x_3 = x_3 + a \\ \qquad\qquad\quad x_2 = tl(x_1) \\ \qquad\qquad\quad x_4 = hd(x_1) \\ \qquad\quad \mathsf{in}\ (\lambda a.F[x_2; -x_4 + a])x_3 \end{array}$$

The next expression to be compacted is $(\lambda a.F[x_2; -x_4 + a])x_3$. By the renaming rule, this term is simplified in $F[x_2; -x_4+x_3]$. We already saw a very similar term. It is equal, modulo a renaming of its free variables, to the term $F[x1; -x2 + a]$. Compaction does not depend on names so it will produce the same result for this input, hence the process enters in an infinite loop. To stop compaction, we give a name — here *ff* — to the term and replace it by a call. The final program is the following one.

```
let ff = Fix_ff( λ[x1; x2; a].
                 let x3 = -x2
                 in  if x1 = []
                     then x3 + a
                     else let x3 = x3 + a
                              x2 = tl(x1)
                              x4 = hd(x1)
                          in ff[x2; x4; x3])
in λ [x; a]. if x = []
             then a
             else let x1 = tl(x)
                      x2 = hd(x)
                  in ff[x1; x2; a]
```

This final program cannot be compacted anymore. It is the *Software Pipelining* version of the initial program, i.e. another inlining followed by the compaction process would produce the same result since we have inlined F as soon as possible.

Note that the recursive call to *ff* can be implemented here by a direct branch and variable names can be seen as register names. Then, register allocation can be done on the fly with compaction (but it is not necessary). In this case, move instructions between x_1 and x_2, x_2 and x_4, a and x_3 can be eliminated.

This example raises the following questions. In this process, some transformations were done on programs like renaming, moving instructions, reducing *identity* calls. What are the legal transformations a *reasonable* compiler can do? Why would the last program be better than the first one? Does the compaction process always terminate and how can this be formalized? Answers to these questions will be developed in the following sections.

4 Formalization

We present a complete formalization of compaction. The first part deals with the presentation of a first-order recursive language, named $\mathcal{F}$. It is a low level language (an intermediate language). We then present the machine model that will allow discussing about "a program is better than an other one for a given machine". The definition should apply to *sequential* machines as well as *superscalar* or *vliw* machines. The next part deals with the semantical model of fine grain parallelization. It presents the notion of *dependences* in $\mathcal{F}$. We shall see why dependences are defined intentionally by an equivalence relation named *structural equivalence.* The last part is the definition of *software pipelining* and the transformation system to produce it.

4.1 The $\mathcal{F}$ language

The $\mathcal{F}$ language is a recursive language with an ML-like syntax, but with a granularity close to the one of a machine.

Definition 1 The $\mathcal{F}$ language. The definition of the language is the following.

$$\begin{aligned} a &::= i \mid \mathsf{true} \mid \mathsf{false} \mid x \mid \mathsf{let}\, x = a \,\mathsf{in}\, a \mid a[a; ...; a] \mid \lambda\vec{x}.a \\ &\mid op[a; ...; a] \mid \mathsf{Fix}_f(a) \mid \mathsf{if}\, a \,\mathsf{then}\, a \,\mathsf{else}\, a \\ op &::= \mathsf{add\_int} \mid \ldots \end{aligned}$$

Primitives (called op) are those of the architecture. Terms range over a. $\mathcal{F}$ manages scalars (i), variables (x), defines local values (let), n-ary functions $\lambda[x_1; ...; x_n].a$, recursive functions $\mathsf{Fix}_f(a)$, and n-ary primitives (op) and applications $a[a_1; ...; a_n]$. All memory accesses are done by explicit primitives and there are no *side-effects* in the sense that the language has the SSA (Static Single Assignment) property [3]. In the paper, we note $\vec{a}$ instead of $[a_1; ...; a_n]$ and $\vec{a}|_i$ for $[a_1; ...; a_{i-1}; a_{i+1}; ...; a_n]$.

$\mathcal{F}$ is a typed language *à la ML* and we suppose the existence of a typing function — named *type* — returning for each expression of $\mathcal{F}$, its type taken from the type language below for which a classical typing algorithm can be chosen [4].

$$t ::= \tau \mid t \rightarrow t \mid [t; ...; t]$$

τ denotes scalar types (int, bool,...). $t \rightarrow t$ is the function typeand $[t; ...; t]$ is the product type. Information needed about a type is its complexity, that is, the number of arrows on the left.

Definition 2 Type complexity. We define the function $|.|$, returning the type complexity of its argument.

$$|t_1 \rightarrow t_2| = 1 + |t_1| \quad |[t_1; ...; t_n]| = max_i(|t_i|) \quad |\tau| = 0$$

For instance, the type complexity of (int $\rightarrow$ int) $\rightarrow$ int is 2 whereas the type complexity of int $\rightarrow$ (int $\rightarrow$ int) is 1. This gives the functionality order. We need a preliminary definition before defining the ready sub-terms of an expression.

Definition 3 Cost. The cost of a term a is defined by the function $||.||$, returning an element from $I\!N \cup \{\infty\}$.

$$||\mathsf{if}\, a_1 \,\mathsf{then}\, a_2 \,\mathsf{else}\, a_3|| = \textstyle\sum_{i=1}^{3}(||a_i||) \qquad ||op[a_1; ...; a_n]|| = 1 + \textstyle\sum_{i=1}^{n}(||a_i||)$$

$$||\lambda\vec{x}.a|| = 1 \qquad ||\mathsf{let}\, x = a_1 \,\mathsf{in}\, a_2|| = ||a_1|| + ||a_2||$$

$$||x|| = ||i|| = 0 \qquad ||a([b_1; ...; b_n])|| = \infty \qquad ||\mathsf{Fix}_f(a)|| = ||a||$$

The cost is a coarse approximation of the execution time of an expression. All primitives are assumed to be executed in one cycle and the execution time of an expression containing a call is infinite. The cost of an abstraction is 1 as we consider the cost of constructing a closure to be elementary [4]. Now we can define the ready sub-terms of an expression.

Definition 4 Ready sub-terms. Let a be a term of $\mathcal{F}$. $s \prec a$ means that the sub-term s of a is ready in a.

$$\frac{}{s \prec s} \quad \frac{s \prec a_i}{s \prec op(\vec{a})} \quad \frac{s \prec a \quad x_i \not\prec s \quad \|s\| < \infty}{s \prec \lambda \vec{x}.a} \quad \frac{s \prec a_1}{s \prec \mathsf{let}\, x = a_1 \,\mathsf{in}\, a_2} \quad \frac{s \prec \lambda f.a}{s \prec \mathsf{Fix}_f(a)}$$

$$\frac{s \prec a_1}{s \prec \mathsf{if}\, a_1 \,\mathsf{then}\, a_2 \,\mathsf{else}\, a_3} \quad \frac{s \prec a_i \quad \|s\| < \infty}{s \prec \mathsf{if}\, a \,\mathsf{then}\, a_1 \,\mathsf{else}\, a_2} \quad \frac{s \prec a_2 \quad x \not\prec s}{s \prec \mathsf{let}\, x = a_1 \,\mathsf{in}\, a_2} \quad \frac{s \prec a_i}{s \prec a_0(\vec{a})}$$

A ready sub-term is a sub-term that can be computed immediately. For example, $1 + x \prec \lambda y.(1 + x) * y$. The notion of ready sub-terms is an extension of free variables (hence, a free variable is always ready). For this reason, we will use the classical notation $FV(a)$ containing the set of free variables of a, i.e, $x \in FV(a)$ iff $x \prec a$. Ready sub-terms of special interest are those whose cost is 1, that is elementary computations.

Definition 5 Ready sub-terms substitution. Let t be such that $t \prec a$. The substitution of all occurrences of t in a by b, noted $a[t\backslash b]$, is defined by:

$$\begin{array}{rcll}
t[t\backslash b] &=& b & \\
x[t\backslash b] &=& x & \text{if } x \neq t \\
(op[a_1; ...; a_n])[t\backslash b] &=& op[[a_1[t\backslash b]; ...; a_n[t\backslash b]] & \\
(a_0[a_1; ...; a_n])[t\backslash b] &=& (a_0[t\backslash b])([a_1[t\backslash b]; ...; a_n[t\backslash b]]) & \\
(\lambda x.a)[t\backslash b] &=& \lambda x.a & \text{if } x \prec t \\
(\lambda x.a)[t\backslash b] &=& \lambda z.(a[x\backslash z][t\backslash b]) & \text{else. } z \not\prec a \wedge z \not\prec b \\
(\mathsf{let}\, x = a_1 \,\mathsf{in}\, a_2)[t\backslash b] &=& \mathsf{let}\, x = a_1[t\backslash b] \,\mathsf{in}\, a_2 & \text{if } x \prec t \\
(\mathsf{let}\, x = a_1 \,\mathsf{in}\, a_2)[t\backslash b] &=& \mathsf{let}\, z = a_1[t\backslash b] \,\mathsf{in}\, a_2[z\backslash x][t\backslash b] & \text{else. } z \not\prec a_2 \wedge z \not\prec b \\
\mathsf{Fix}_f(a)[t\backslash b] &=& \mathsf{Fix}_f(a) & \text{if } f \prec t \\
\mathsf{Fix}_f(a)[t\backslash b] &=& \mathsf{Fix}_g(a[f\backslash g][t\backslash b]) & \text{else. } g \not\prec a \wedge g \not\prec b \\
(\mathsf{if}\, a_1 \,\mathsf{then}\, a_2 \,\mathsf{else}\, a_3)[t\backslash b] &=& \mathsf{if}\, a_1[t\backslash b] \,\mathsf{then}\, a_2[t\backslash b] \,\mathsf{else}\, a_3[t\backslash b] &
\end{array}$$

The substitution of ready sub-terms is very similar to the classical substitution of free variables. For example, $(\lambda x.(1 + y) * x)[1 + y\backslash y] = \lambda x.y * x$. Thus substitution makes it possible to extract computations from a term.

[4] A multicycle computation $op(x)$ can be treated, replacing it by $op_1(op_2(...(op_n(x))...))$.

4.2 Operational semantics

The operational semantics has two aims. As usual, it defines the values and the execution order. But, it also express how a fine-grain parallel machine handles a given program. It leads to a measure on programs.

The language has a call-by-value evaluation in a left-to-right order. A machine can be seen as a function, selecting a subset of elementary ready sub-terms and executing them. All sub-terms cannot be selected. A fine grain parallel machine executing some in-line code does not see the entire control structure of the program and is limited by *control-dependences* [12]: an instruction following a function call is *unreachable*. To define machines, we need a preliminary definition to explain what *unreachable* means.

Definition 6 Occurrences and depth. The set of occurrences (access paths), $\mathcal{O}(a)$ of a term a contains words on integers (the empty word is Λ). It is defined as usual by:

$\Lambda \in \mathcal{O}(a)$
For all construction C of the language, $i.o_i \in \mathcal{O}(C(a_1, ..., a_n))$ if $o_i \in \mathcal{O}(a_i)$.

We define the Depth of an occurrence in a given term as follows:

$\mathcal{D}_a(\Lambda) = 0$
$\mathcal{D}_{a[a_1;...;a_n]}(1.o) = \sum_{i=1}^{n} ||a_i|| + \mathcal{D}_a(o)$
For all other constructions C, $\mathcal{D}_{C(a_1,...,a_n)}(i.o) = \mathcal{D}_{a_i}(o) + \sum_{j<i} ||a_j||$.

The *depth* of an occurrence is the number of instructions that are executed before it, in a left-to-right evaluation order. Thus, the depth of $(1+y)$ in $(\lambda x.(1+y) * x)(2+z)$ is 2 because the argument is executed before the body of the function. The depth of a sub-term on the right of a function call always equals ∞. It means that the computation is unreachable by the machine.

When possible, the occurrence will be omitted and the notation $\mathcal{D}_a(t)$ will be used instead of $\mathcal{D}_a(o)$ if o is the occurrence of the sub-term t in a.

A machine can be seen as a sub-relation of $\prec$. For example, a machine is able to forbid speculative execution. Therefore, instructions under conditionals or abstractions are never ready.

Definition 7 Machine $\mathcal{M}$. A machine $\mathcal{M}$ is a function from terms from $\mathcal{F}$ to set of terms verifying the following constraint: there exists $k_1, k_2 \in \mathbb{N}$ such that for all a with $\mathcal{M}(a) = \{t_1, ..., t_n\}$, we have $n \leq k_1$, $t_i \prec a$ and $\mathcal{D}_a(t_i) \leq k_2$ and there exists at most one t_i such that $||t_i|| = \infty$.

The number of ready sub-terms (k_1) gives the number of parallel units of the machine and the depth gives the window size (k_2) — the number of instructions that can be fetched every cycle. This window contains at most a function call (an unconditional jump) which is on the right of the others[5]. A *super-scalar* machine also matches this description. A *sequential* machine is the special case where $k_1 = 1$ and $k_2 = 1$. A *vliw* machine is a special case of a machine.

Example 1 VLIW machine. A vliw machine is a maximal machine $\mathcal{M}$ such that: if $t \in \mathcal{M}(a)$ and $\mathcal{D}_a(t) = n$ then $\forall t', \mathcal{D}_a(t') < n \Rightarrow t' \in \mathcal{M}(a)$

Here, the maximal prefix of independent instructions from a window of sequential instructions can be executed in parallel. We call it a VLIW machine, even if it is not a real one — a Multiflow-like machine [14] — because no scheduling is done by the machine.

Pipelined machines enter in this description if the compiler transforms a pipelined instruction $op(x)$ in $\mathsf{nop}(...(\mathsf{nop}(op(x)))...)$.

Example 2 Pipelined machine. A pipelined machine is a machine $\mathcal{M}$ such that: if $t \in \mathcal{M}(a)$ and $t \neq \mathsf{nop}(y)$ then $\mathsf{nop}(x) \prec a \Rightarrow \mathsf{nop}(x) \in \mathcal{M}(a)$.

All nop operations are necessarily executed in parallel with a non-nop operation. A machine will be given by its state automaton that reads only a finite part of the term. Let us see now the definition of the operational semantics of the $\mathcal{F}$ language given in a structural way [16].

The operational semantics is straightforward. Like a classical fine-grain parallel machine, the machine selects a subset of ready sub-terms that can be executed in parallel, executes them and substitutes selected ready sub-terms by their values. Nonetheless, because of the representation of programs, a term of $\mathcal{F}$ contains some *noise*, that is, syntactical constructions that do not generate any computation (in fact, they correspond to no assembly instructions). For example, the let represents *dags* and the expression $\mathsf{let}\, x = 1\, \mathsf{in}\, x + x$ has the same execution time as $1 + 1$. To reduce this noise, we define an equality.

Definition 8 ϵ-transitions. We define the equality $=_{\mathcal{M}}$. $C(a)$ denotes a term containing the sub-term a.

$$\begin{aligned}
C(\mathsf{let}\, x = x\, \mathsf{in}\, a) &=_{\mathcal{M}} C(a)\\
C(\mathsf{let}\, x = i\, \mathsf{in}\, a) &=_{\mathcal{M}} C(a[x \backslash i])\\
C(\mathsf{let}\, x = \lambda y.a_1\, \mathsf{in}\, a_2) &=_{\mathcal{M}} C(a_2[x \backslash \lambda y.a_1])\\
C(\mathsf{let}\, x = \mathsf{Fix}_f(\lambda y.a_1)\, \mathsf{in}\, a_2) &=_{\mathcal{M}} C(a_2[x \backslash \mathsf{Fix}_f(\lambda y.a_1)])\\
C(\,\mathsf{if\, true\, then}\, a_2\, \mathsf{else}\, a_3) &=_{\mathcal{M}} C(a_2)\\
C(\,\mathsf{if\, false\, then}\, a_2\, \mathsf{else}\, a_3) &=_{\mathcal{M}} C(a_3)
\end{aligned}$$

$$\begin{aligned}
C((\lambda \vec{x}.a)[b_1; ...; x_i; ...; b_n]) &=_{\mathcal{M}} C((\lambda \vec{x}|_i.a)[b_1; ...; ; ...; b_n])\\
&\quad \text{if } x_i \in \mathcal{M}((\lambda \vec{x}.a)[b_1; ...; x_i; ...; b_n])\\
&\quad \wedge type(x_i)| = 0\\
C(\mathsf{Fix}_f(\lambda \vec{x}.a)) &=_{\mathcal{M}} C(\lambda \vec{x}.a[f \backslash \mathsf{Fix}_f(\lambda \vec{x}.a)])\\
&\quad \text{if } \mathsf{Fix}_f(\lambda \vec{x}.a) \in \mathcal{M}(\mathsf{Fix}_f(\lambda \vec{x}.a))
\end{aligned}$$

[5] According to the definition, the depth of a function call is finite if it is not preceeded by any other call.

The constraint on the complexity of the type for the reduction of applications will be explained further. The equality says which constructions are invisible for the machine $\mathcal{M}$. Here, only reachable sub-expressions are reduced: the equality is not applied in all contextes. This is due to the inlining rule for recursions. Indeed, without the use of Mm, the following infinite reduction could occur: $F[x;y], ..., F[x;op(y)], ..., (\lambda y.F[x;op(y)])(op(y)), ...$
with $F = \mathsf{Fix}_f(\lambda[x;y].f[x;op(y)])$. In our definition, empty computations are erased when necessary.

Definition 9 Operational semantics. The operational semantics of a program on a machine $\mathcal{M}$ is given by the relation $\Downarrow$. $a \Downarrow b$ means that a evaluates to b. Terms are considered modulo $=_{\mathcal{M}}$. The inference rules are:

$$\frac{\mathcal{M}(a) = \{t_1, ..., t_n\} \quad t_i \Downarrow v_i \quad a[t_1\backslash v_1]...[t_n\backslash v_n] \Downarrow b}{a \Downarrow b}$$

Axioms are of the following form:

$x \Downarrow x$
$\lambda x.a \Downarrow \lambda x.a$
$\mathsf{Fix}_f(\lambda x.a) \Downarrow \mathsf{Fix}_f(\lambda x.a)$
$i_1 + i_2 \Downarrow i_3$ where i_3 is the sum of i_1 and i_2.

The execution time $|a|_{\mathcal{M}}$ of a is the size of the proof of $a \Downarrow v$.

For simplicity, we only give the axiom for integers. With the definition of the execution time, we may compare programs.

4.3 Dependence semantics

We used some program transformations like renaming of inlining in the compaction of the second example. What reasonable transformations should be incorporated in a compacting compiler? In the classical framework, only transformations preserving *dependences* are legal. There is a dependence between two instructions if one instruction modifies a value read by another. These dependences are decomposed into *true-dependences* — or *data-dependences* — when the second instruction reads the value produced by the first, and *false dependences* — *anti-dependences* and *output-dependences* — when the second modifies a value read or written by the first one [2].

Because $\mathcal{F}$ is a functional language, dependences, in the usual sense as *def-use* links are directly given in the text by variables and composition of computations. Thus, x of $\mathsf{let}\ x = a\ \mathsf{in}\ b$ depends on a — and of all its sub-terms — and every subterm of b containing x depends on it. The transitive closure of this relation gives the classical notion of dependence graph. It is useless to define another notion of dependences of a term in $\mathcal{F}$ and if it would exist, it would be as complex as the term itself. Our notion of dependences is too rigid here for what we need since no syntactical transformation of the term is allowed.

We prefer to define dependences as an equivalence between "programs having the same dependences". This equivalence will be named *structural equivalence.* How does this equivalence interact with β-reduction[6]? On one hand, two really different programs can be extensionally (beta) equivalent and it is clearly infeasible (in a compiler) to allow all kinds of β-reductions. Moreover, we would have some coherency problems since $1+2$ is not equivalent to 3 but the translation in pure λ-calculus would yield the equivalence. On the other hand, can we limit the equivalence to the syntactic equality? Of course, not. It is reasonable to allow instruction moves, inlining, duplication[7] which are the minimal transformations needed for compaction.

The solution proposed here is to limit β-reductions by the complexity of the type: only β-reductions where the type of the argument is simple (the complexity is 0) are allowed.

Definition 10 Structural equivalence. Let a and b, two terms of $\mathcal{F}$. We say that a and b are structurally equivalent, noted $a \sim b$, when $lim_{n\to\infty}a^n = lim_{n\to\infty}b^n$ where a^n is defined by $a \to a^2 \to \ldots \to a^n$ and by the rewriting relation $\to$ applied to all contexts. $\to$ is defined by:

1. $\mathsf{let}\, x = a_1 \,\mathsf{in}\, a_2 \to a_2[x\backslash a_1]$
2. $\mathsf{if}\, a \,\mathsf{then}\, \vec{b} \,\mathsf{else}\, \vec{c} \to \overrightarrow{\mathsf{if}\, a \,\mathsf{then}\, b \,\mathsf{else}\, c}$
3. $\mathsf{if\ true\ then}\, a_1 \,\mathsf{else}\, a_2 \to a_1$
4. $\mathsf{if\ false\ then}\, a_1 \,\mathsf{else}\, a_2 \to a_2$
5. $(\lambda\vec{x}.a)\vec{b} \to (\lambda\vec{x}|_i.a[x_i\backslash b_i])\vec{b}|_i$ if $|type(b_i)| = 0$
6. $\mathsf{Fix}_f(a) \to a[f\backslash \mathsf{Fix}_f(a)]$

The first rule allows to duplicate a computation. The second rule is equivalent for the test. The next two rules show that conditions can be simplified. The next rule is a β-reduction when the type of the argument b_i is simple.

4.4 Compaction

We saw in the examples that compaction can be obtained by selecting and moving up ready sub-terms. We first define instruction move by two simple rules and then define how to select a subset of the ready sub-terms.

Definition 11 Move-up. Instruction moves are defined by the relation $a \xrightarrow{t} b$ where t is ready in a, such that:

(MOVE) $a \xrightarrow{t} \mathsf{let}\, x = t \,\mathsf{in}\, a[t\backslash x]$ if $x \not\in a$
(TEST) $a \xrightarrow{t} \mathsf{if}\, t \,\mathsf{then}\, a[t\backslash \mathsf{true}] \,\mathsf{else}\, a[t\backslash \mathsf{false}]$
if $\mathsf{if}\, t \,\mathsf{then}\, a_1 \,\mathsf{else}\, b_1$ is a sub-term of a

[6] β-reduction is the following rule $(\lambda x.a_1)a_2 \to_\beta a_1[x\backslash a_2]$.

[7] Duplication is useful when it is cheaper to recompute a value than to communicate it.

Instruction moves are simply inverse β-reductions. Now, to select a subset of the ready sub-terms that can be executed in parallel on the machine, the compaction process needs a *strategy*. For example, the classical *List-scheduling* [9] strategy will select ready sub-terms which are on the longest dependence path.

Definition 12 Strategy. A strategy $\mathcal{C}$ is a function $\mathcal{C}$ between terms and set of terms such that $\mathcal{C}(a) = \{t_1, ..., t_n\}$ and for all t_i, $t_i \prec a$

A strategy is not very different from a machine except that — for the moment — there is no constraint on the position of ready sub-terms in the term. It is the reason why we can deal with $=_{\mathcal{C}}$, replacing $\mathcal{M}$ by $\mathcal{C}$.

Let us see now the compaction system. It is very similar to the operational semantics: at every step, some computations are selected and moved-up in the program. The rest of the program is then compacted.

Definition 13 Compaction. Let ρ be a renaming. Terms are considered modulo $=_{\mathcal{C}}$. The general compaction is defined by the predicates $e \models_{\mathcal{C}} a \Rightarrow_l b$ given in the figure 1, meaning that in the environment e, the term a is compacted to b, using the strategy $\mathcal{C}$ and moving up the selected ready sub-terms in l. An environment is a set $[f_1 \backslash \lambda \vec{v_1}.a_1]...[f_n \backslash \lambda \vec{v_n}.a_n]$ used to record terms.

The (SELECT) rule selects a subset of ready sub-terms from a. It records the current term a in the environment e and compacts a with the selected ready sub-terms. The resulting program may be recursive. Of course, if the compaction of a does not use f, the term $(\mathsf{Fix}_f(\lambda v.b))v$ simplifies to b. The (EQUIV) rule stops compaction when the current term is equivalent modulo a renaming ρ to a recorded term. In this case, the compaction of the current term is a call to the name of the recorded term. The (LAMBDA) rule is compositional rule. The (PRIM), (APP), (CONST) and (VAR) rules are the axioms. The (MOVE) rule moves up a selected ready-sub term and the (TEST) moves up a selected test.

Now, do the resulting program always exists, that is, does the system terminate? What are the conditions under which the compacted program is better than the initial one?

For the termination problem, without constraints on $\mathcal{C}$, the answer is clearly no. Indeed, consider $F = \mathsf{Fix}_f(\lambda[x;y].f[op_1(x);op_2(op_3(y))])$ and $F([x;y])$. Let us see the suite of terms that we shall have to compact. It is,

$$F[x;y], \ldots, F[x;op_2(y)], \ldots, F[x;op_2(op_3(y))], \ldots, F[x;op_2(op_3(op_2(y)))], \ldots$$

$F[x;y]$ is first inlined and the two ready sub-terms $op_1(x)$ and $op_3(y)$ are substituted. Then, the current term to compact is $F[x;op_2(y)]$, etc. This is a very classical problem of iterative compaction methods [1]: the problem is that ready sub-terms are selected deeper and deeper. In the case of tail recursive functions, i.e. loops, we shall see that compaction is simpler than in the general case and that simple and satisfying conditions over $\mathcal{C}$ can be taken. We shall come back on general recursive function at the end of the paper to show the problems.

$$(\text{SELECT})\ \frac{\mathcal{C}(a) = l \quad \vec{v} = FV(a) \quad e[f\backslash\lambda\vec{v}a] \models_{\mathcal{C}} a \Rightarrow_l b}{e \models_{\mathcal{C}} a \Rightarrow (\mathsf{Fix}_f(\lambda\vec{v}b))\vec{v}}$$

$$(\text{EQUIV})\ \frac{b = \rho(a) \quad \vec{v}' = \rho(\vec{v})}{e[f\backslash\lambda\vec{v}a] \models_{\mathcal{C}} b \Rightarrow f(\vec{v}')} \qquad (\text{LAMBDA})\ \frac{e \models_{\mathcal{C}} a \Rightarrow a'}{e \models_{\mathcal{C}} \lambda\vec{x}a \Rightarrow \lambda\vec{x}a'}$$

$$(\text{MOVE})\ \frac{a \xrightarrow{t} \mathsf{let}\, x = t\, \mathsf{in}\, a_1 \quad e \models_{\mathcal{C}} t \Rightarrow t' \quad e \models_{\mathcal{C}} a_1 \Rightarrow_l b_1}{e \models_{\mathcal{C}} a \Rightarrow_{\{t\}\cup l} \mathsf{let}\, x = t'\, \mathsf{in}\, b_1}$$

$$(\text{PRIM})\ e \models_{\mathcal{C}} op(\vec{x}) \Rightarrow op(\vec{x}) \qquad (\text{CONST})\ e \models_{\mathcal{C}} i \Rightarrow i$$

$$(\text{APP})\ e \models_{\mathcal{C}} y(\vec{x}) \Rightarrow y(\vec{x}) \qquad (\text{VAR})\ e \models_{\mathcal{C}} x \Rightarrow x$$

$$(\text{TEST})\ \frac{a \xrightarrow{t} \mathsf{if}\, t\, \mathsf{then}\, a_1\, \mathsf{else}\, a_2 \quad e \models_{\mathcal{C}} t \Rightarrow t' \quad e \models_{\mathcal{C}} a_1 \Rightarrow_l b_1 \quad e \models_{\mathcal{C}} a_2 \Rightarrow_l b_2}{e \models_{\mathcal{C}} a \Rightarrow_{\{t\}\cup l} \mathsf{if}\, t'\, \mathsf{then}\, b_1\, \mathsf{else}\, b_2}$$

Fig. 1. The compaction system

Constraint 1 (Terminal recursions) *A term a is terminal if every sub-term $\mathsf{Fix}_f(b)$ of a is such that f is in a terminal position in b, noted $\mathcal{T}(f, b)$.*

$\mathcal{T}(f,\ \mathsf{if}\, a\, \mathsf{then}\, a_1\, \mathsf{else}\, a_2)$ *if* $f \notin FV(a)$, $\mathcal{T}(f, a_1)$ *and* $\mathcal{T}(f, a_2)$

$\mathcal{T}(f, f(a))$ *if* $f \notin FV(a)$
$\mathcal{T}(f, \mathsf{let}\, x = a\, \mathsf{in}\, b)$ *if* $f \notin FV(a)$ *and* $\mathcal{T}(f, b)$
$\mathcal{T}(f, \lambda x.a)$ *if* $\mathcal{T}(f, a)$
$\mathcal{T}(f, a(b))$ *if* $\mathcal{T}(f, a)$ *and* $f \notin FV(b)$
$\mathcal{T}(f, a)$ *if* $f \notin FV(a)$ *for the other constructions*

The definition of tail-recursive functions says that recursive calls to f are never followed by a computation. Now, we define a constraint on the strategy that must be bounded to guaranty termination of the compaction system.

Constraint 2 (Bounded strategy) *Let $|a|$ be the number of ready sub-terms of a. A strategy $\mathcal{C}$ bounded by k is a strategy such that if $|a| > k$ then for all t, $t \in \mathcal{C}(a)$ iff $t \in \mathcal{C}(b)$ where $a \rightarrow b$.*

A bounded strategy limits the number of selected ready sub-terms. The implication $t \in \mathcal{C}(a) \Rightarrow t \in \mathcal{C}(b)$ is reasonable: it means, for example, that instructions inside a loop iteration have a priority over the ones belonging to the next iterations. The other constraint is quite unusual: it means that even if a sub-term is ready after an inlining, for example, it cannot be selected if the size of the term is greater than a certain limit. This condition is not so strange: consider a term $F(a)$ where F stands for a recursive function. An argument for termination in iterative methods can be that the recursion is inlined when the size of the argument is bounded (usually, an instruction from an iteration can be moved up when the scheduling time of the previous iteration is less than a certain limit).

How does this constraints act with the classical *List-scheduling* strategy? This strategy selects ready sub-terms which are on the longest path. The constraint means that only bounded paths are considered [8].

Proposition 1 (Termination) *The compaction system* $\models_{\mathcal{C}}$ *terminates with a bounded strategy.*

Proposition 2 (Correction) *The compaction system preserves the structural equivalence.*

We have to prove that this system is useful, that is, it decreases the execution time of programs. This cannot be done for general machines because scheduling is NP-hard [9] and cannot be achieved by a greedy algorithm. Nonetheless, the speed-up can be guaranteed for VLIW machines and a strategy which always select at least the nodes selected by the machine.

Proposition 3 (Speed-up) *Let* $\mathcal{M}$ *be a vliw machine and* $\mathcal{C}$ *such that* $\forall a, \exists c$, *such that* $\mathcal{M}(a) \subseteq \mathcal{C}(a) \subseteq \mathcal{M}(c)$. *Then if* $\models_{\mathcal{C}} a \Rightarrow b$ *then* $|a|_{\mathcal{M}} \geq |b|_{\mathcal{M}}$

The result applies for VLIW machines because the depth of instructions in the program never increase: an instruction that is selected by the machine in the initial program is still selected in the second one (except if it has been executed before).

We can ask now how the resulting program compares with the classical software pipelining principle. Software pipelining for recursive programs can be seen as a finite representation of programs, infinitely inlined and compacted. It is a fixpoint for compaction: the compaction of the inlined version programs yields the same program. Software pipelining is defined from the notion of optimality.

Definition 14 Software pipelining. Let a be a term of $\mathcal{F}$. A program a is optimal for $\mathcal{C}$ if $\models_{\mathcal{C}} b \Rightarrow a$ where $a \rightarrow b$. A program a' is a *software pipelining* of a by $\mathcal{C}$ when $a \sim a'$ and a' is optimal.

Proposition 4 (Optimality) $\models_{\mathcal{C}}$ *constructs a software pipelining for tail-recursive functions.*

[8] Proofs can be found in [17].

The proposed system is a greedy system and is not well suited to general recursion. Indeed, consider the very simple expression $F = \mathsf{Fix}_f(\lambda\vec{x}.A(\vec{x}, f(op(\vec{x}))))$ where A denotes an $\mathcal{F}$ expression. We have the list of terms to compact:
$F(\vec{x_1}), A(\vec{x_1}, F(\vec{x_2})), A(\vec{x_1}, A(\vec{x_2}, F(\vec{x_3}))), \ldots$
The compaction process only reads the beginning of the term $(op(x_i))$. Here the term grows after the recursive call. Even if compaction is stopped, using conditions over $\mathcal{C}$, we do not have any re-rolling rule for the production of the software pipelining. It is unclear how a greedy algorithm, where software pipelining is obtained using what has been done in previous steps and where terms are inlined *a priori*, can be obtained for non-tail recursive functions.

5 Conclusion and future work

We have presented a method to directly compact the first-order part of functional programs. Compaction is described as a set of program transformations. An operational semantics has been proposed to model the behavior of a fine grain parallel machine that executes a program. A notion of dependences has been given here by the ***structural equivalence***: it defines only legal — but minimal — program transformations a parallelizing compiler needs to have. This paper can be seen as a formalization of classical compaction techniques, but also as an extension to the general class of recursive programs. By applying compaction to *terms* instead of control flow based representation, the compiler can take benefit of semantical informations available in the source program (for dependence analysis, for example). Because the representation is more general than dependence based representations — this representation can be obtained with sharing — it can be used to represent dependences of a large class of languages.

Some extensions can be done. First, a better strategy will improve convergence since iteration methods are known to be slow and to produce large resulting code. Secondly, imperative features could be treated with a slight modification of de *ready sub-terms* definition. Thirdly, it could be interesting (and useful) to take into account semantical properties of primitives in order to increase the set of the structurally equivalent terms and, thus, to improve compaction. This could be done by the addition of other simplifying rules (commutative rules,...). It is certainly for this kind of extension that the functional representation of programs is better than the classical control-flow representation. Thirdly, non-tail first-order recursion must be treated and, finally, the compaction of general functional program can be studied.

Acknowledgments

Many thanks to my PhD. advisor Jean-Jacques Lévy and to careful readers Paul Caspi from VERIMAG, Thérèse Hardin and Bernard Serpette from INRIA Rocquencourt. Thanks also to the usefull remarks from the referees.

References

1. Alexander Aiken. *Compaction-based Parallelization.* PhD thesis, Cornell University, 1988.
2. R. Cytron and J. Ferrante. What's in a name ? In *International Conference on Parallel Processing,* pages 19–27, August 1987.
3. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignement form. In *Conference on Principles of Programming Langages*, 1989.
4. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Conference on Principles of Programming Languages*, 1982.
5. Digital. *Alpha Architecture Handbook.* Digital, 1992.
6. Kemal Ebcioglu. A compilation technique for software pipelining of loops with conditional jumps. In *Annual Workshop on Microprogramming,* pages 69–79, December 1987.
7. J. R. Ellis. *Bulldog-A Compiler for VLIW Architectures.* MIT Press, 1985. Ph.D dissertation.
8. J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Symposium on Compiler construction.* SIGPLAN Notices, June 1984. Volume 19, Number 6.
9. M. R. Garey and D. S. Johnson. *Computers and Intractability - A guide to the Theory of NP-completeness.* Freeman, New-York, 1979.
10. Carl A. Gunter. *Semantics of programming languages: structures and techniques.* The MIT press, Cambridge, Mass., London, 1992.
11. IBM. IBM Risc System/6000 technology. Technical Report SA23-2619, IBM, 1990. Copies can be obtained from the local IBM Branch office.
12. M. S. Lam and R. P. Whilson. Limits of control flow on parallelism. *ACM Sigarch. Computer Architecture News*, 20(4), 1992.
13. Monica S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Conference on Programming Language, Design and Implementation,* pages 318–328, Atlanta, Georgia, June 22-24 1988.
14. Multiflow Computer Inc. Trace/300 series. Technical report, Multiflow Computer Inc, Brandford, Connecticut, 1987.
15. Alex Nicolau. Percolation scheduling: A parallel compilation technique. Technical report, Cornell University, 1985.
16. Gordon D. Plotkin. *A structural approach to operational semantics.* Daimi FN-19. University of Aarhus, Computer Science Department Aarhus University Ny Munkegade DK 8000 C Danemark, September 1981.
17. Marc Pouzet. The program compaction revisited: the functional framework. Technical Report Spectre-94-11, Verimag, Grenoble, France, December 1994. Available by anonymous ftp on imag.fr in pub/SPECTRE.
18. R. F. Touzeau. A fortran compiler for the FPS-164 scientific computer. In *Symposium on Compiler Construction,* pages 48–57, June 1984.