Optimizing Sisal Programs: A Formal Approach

I. Attali[1] and D. Caromel[1] and R. Guider[1] and A. L. Wendelborn[2]

[1] INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis,
BP 93, 06902 Sophia Antipolis Cedex - France
{ia, caromel}@sophia.inria.fr
[2] Dept. Computer Science, University of Adelaide 5005 - Australia
andrew@cs.adelaide.edu.au

Abstract. We formally describe optimization techniques for the compilation of the language Sisal 2.0. More precisely, we translate Sisal programs into data-flow IF1 graphs and optimize these graphs. An interactive visualization environment for IF1 graphs is also provided.

1 Introduction

Software engineering of parallel programming is becoming an important issue in computer science; one focus has been to design compiler schemes that will provide efficient parallel code running on various parallel architectures. Sometimes complementary, sometimes orthogonal, the increasing importance of tools and environments dedicated to parallel computing is another significant aspect (see [9, 1, 18] for recent developments).

The trade-off between the programmer's and compiler's part in designing parallel programs has been debated at least for a decade. One solution to ensure portability and reusability is a well-specified language with a high-level of abstraction; moreover, a compiler for such language must automatically exploit the parallelism of programs, with a clean model of execution, independent, as much as possible, of the architecture of the computer.

Our goal is to investigate this trade-off, namely on the Sisal programming language [5], a strongly typed, applicative, single assignment language in use on a variety of parallel processors (see [5] for a description of Sisal and associated research). We intend to establish a base for the definition and implementation of validated compilers for Sisal, using formal specifications and verifiable transformations. We have observed indeed that the description of Sisal and its implementation call for a more formal approach. The semantics of Sisal is typically given in English and although the various authors have attempted to be precise, we have found ambiguities in the extant definitions of Sisal. We have already given in [3] a formal definition of the dynamic semantics of a significant part of the language Sisal 2.0 in Natural Semantics [11], using the Typol formalism [7], within the Centaur system [6], a generic tool for designing languages. The Sisal 1.2 compiler[1], (OSC, Optimizing Sisal Compiler, OSC [10]), has proved remarkably successful through exploitation of a variety of sophisticated optimizations.

[1] At the moment, there is no compiler for the version 2.0 of the language.

Unfortunately, the most precise definition of the optimizations currently resides in the compiler itself, making it difficult to extend either the language or the catalogue of optimizations. Two intermediate formats IF1 [14] and IF2 [17] are used in Sisal 1.2 implementation. IF1 is a textual description of data-flow graphs which allows traditional code optimizations and IF2 allows optimizations to be applied in terms of abstract storage.

We give a syntactic definition of the IF1 format in Section 2 and describe an interactive visualization environment for IF1 graphs in Section 3. Section 4 describes the translation scheme from Sisal to IF1 and points out some deficiencies in the IF1 formalism, as it was originally defined for Sisal 1.2. In Section 5, we give an example of transformation, the elimination of common subexpressions. Finally, Section 6 concludes with some directions for future work.

2 The IF1 intermediate format

IF1 is a hierarchical, single assignment language defining data-flow graphs. An IF1 program consists of one or more acyclic graphs made up of simple and compound nodes, edges, and types. Nodes correspond to computations: simple nodes represent arithmetic operations and structure manipulation. Compound nodes represent structured expressions, using subgraphs for the components of conditionals and loops. Edges show the transmission of data between nodes; types are associated with the data transmitted on edges. An IF1 program expresses data dependencies, with control left implicit; for example, an iteration is represented as a compound node with subgraphs describing generation of index values, the body of the loop, and the packaging of results. However, the control relationships between these components are left unspecified, and can be

```
file -> GRAPH_TYPE * ...;
type -> INT INT PARAMS PRAGMA;
global_graph -> GRAPH_ELT_S STR INT PRAGMA;
imported_graph -> STR INT PRAGMA;
local_graph -> GRAPH_ELT_S STR INT PRAGMA;
subgraph -> INT GRAPH_ELT_S PRAGMA;
graph_elt_s -> GRAPH_ELT * ...;
snode -> NODE_LABEL INT PRAGMA;
edge -> INT INT INT INT INT PRAGMA;
cnode -> GRAPH_LIST INT INT INT
        ASSOC_LIST PRAGMA;
literal -> INT INT INT VALUE PRAGMA;
params -> INT * ...;
int -> implemented as INTEGER;
pragma -> PRAGM * ...;
pragm -> STR VAL;
str -> implemented as STRING;
no_val -> implemented as SINGLETON;
error_value -> implemented as STRING;
graph_list -> GRAPH * ...;
assoc_list -> INT * ...;
```

```
FILE ::= file;
GRAPH_TYPE ::= GRAPH type;
GRAPH ::= global_graph imported_graph
          local_graph subgraph;

GRAPH_ELT_S ::= graph_elt_s;
GRAPH_ELT ::= snode cnode literal edge;

PARAMS ::= params;
INT ::= int;
PRAGMA ::= pragma;
PRAGM ::= pragm;
STR ::= str;
VAL ::= no_val id INT STR CHAR complex;
VALUE ::= error_value VAL;
GRAPH_LIST ::= graph_list;
ASSOC_LIST ::= assoc_list;
```

Figure 1. Abstract Syntax for IF1

interpreted as, for example, a data flow machine graph, or a loop structure for a conventional multiprocessor.

We specify syntactic aspects — concrete and abstract syntax of the IF1 language. From this specification, one can derive a parser that transforms the textual form of a program into a structural representation, an abstract syntax tree (AST), well-typed with respect to an abstract syntax[2] (see Figure 1). From syntactic specifications, we derive a structure editor in order to parse IF1 graphs (coming from OSC or our own translation from Sisal), edit and visualize them in a textual representation, as shown in Figure 2.

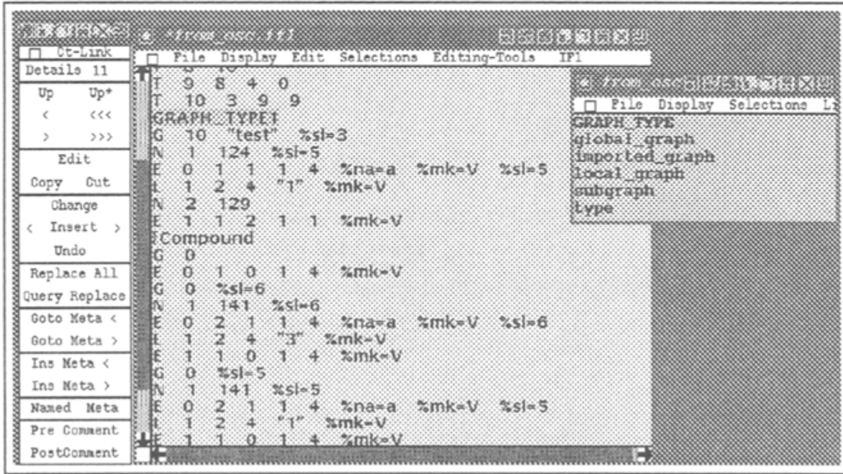

Figure 2. Textual Presentation of IF1 Graphs

3 Graphical Visualization of IF1 Graphs

This textual representation of an IF1 program describing a graph is not appropriate for understanding of the complete graph structure. Using the Centaur system and a specific package for graph display [12], we provide an automatic graphical representation of IF1 graphs (see Figure 3).

An IF1 graph is a hierarchical graph made up with subgraphs, each component with its own inputs and outputs denoted by ports. Two types of relations are represented: (1) a graph hierarchy representing the program structure, and (2) data dependencies (between nodes of the graphs).
Instead of visualizing the whole IF1 graph in a single window, since the number of nodes can be huge, we separate the relations as follows:

- the graph itself appears in a main window which is a synthesized view of the hierarchy of the program (a loop in a selection, etc);
- data dependencies appear in separate windows.

[2] Lower-case (resp. upper-case) identifiers define operators (resp. sorts).

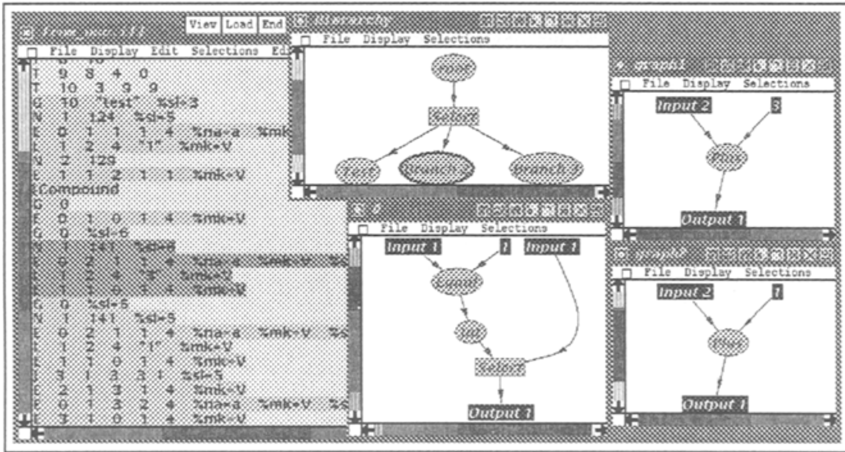

Figure 3. Graphical Visualization of IF1 graphs

We use different colors, fonts, and shapes to distinguish nodes and edges (compound nodes have a rectangular shape and graphs are denoted with oval shapes). The graph server takes care of the layout of the graph according to some specific heuristic (planarity for instance). This visualization is interactive since the graph server reacts to selection or move of a node.

We provide a zoom mechanism which expands on demand graph nodes with dependencies. Then, we have (in a main window named `Hierarchy`) a synthesized view of the graph, and we can also visualize data dependencies on a given graph in a new window, when zooming. In Figure 3, the `root` graph in the `Hierarchy` window is zoomed in the 0 window.

Because the whole graph (hierarchy and dependencies) is complex, we maintain the coordination of multiple views of graph objects, with a selection mechanism: clicking in windows displaying data dependencies shows the corresponding node in the hierarchical graph and highlights the textual fragment in the IF1 source, as illustrated in Figure 3 with a dark selection in the textual presentation as well as on the `Branch2` node. We think that this type of representation is mandatory in the context of a large number of nodes in the data-flow graph.

4 Translating Sisal to IF1

In this section, we outline the methods used in translating Sisal 2.0 to IF1, especially ordinary expressions and loops; we also consider possible extensions to IF1 in order to reflect the powerful array operations of Sisal 2.0.

To translate expressions and function calls, we write Typol inference rules which take as input Sisal expressions and construct an IF1 AST. IF1 expressions, and associated subgraph edges and ports, can be produced from the operands and operators of the expression. Each Sisal construction can be thought of as an

operator $Oper(N_1, \ldots, N_n)$; the corresponding IF1 graph is obtained by translating each N_i and connecting every resulting graph to a node that denotes *Oper*. This scheme, applied over a Sisal abstract syntax tree, results in a complete IF1 graph. Figure 4 shows the Typol rule that realizes the translation of Sisal binary expressions:

```
expression(label1, port1 |- Expr1 -> graph_elts1, label2, val1, port2, type_val1) &
expression(label2, port2 |- Expr2 -> graph_elts2, label3, val2, port3, type_val2) &
operator(label3 |- Oper -> graph_elt, label4) &
create_edge(label3, int 1, int 0, val1, type_val1 -> edge1) &
create_edge(label3, int 2, int 0, val2, type_val2 -> edge2) &
appendtree(graph_elts1, graph_elts2 -> graph_elts3) &
appendtree(graph_elts3, graph_elt_s[graph_elt,edge1,edge2]-> graph_elts4)
--------------
label1, port1 |- binary(Expr1, Oper, Expr2) -> graph_elts4, label3, label4, port3;
```

Figure 4. Translating Sisal expressions into IF1

The IF1 graph produced for a Sisal binary expression given as the subject `binary(Expr1, Oper, Expr2)`, is the composition (predicate `appendtree`) of the subgraphs respectively produced from `Expr1` and `Expr2` with recursive calls to the predicate `expression`. From the operator `Oper`, the predicate `operator` produces a node and the calls to the predicate `create_edge` build two edges, which will augment the IF1 graph. It is also necessary to maintain specific information such as the current node label (`label1`), or the current available port number (`port1`), in a left-to-right manner.

4.1 Translating loops

In translating loops, one problem that must be addressed is the classification of loops in two categories: parallel or sequential. Sisal 2.0's loop syntax is quite powerful, allowing expression of a variety of loop structures over index ranges and array structures. Unlike Sisal 1.2, it is not apparent from the loop syntax in Sisal 2.0 whether or not it is a parallel loop. This must be determined from analysis of variable usage within the loop—if the value at one iteration is dependent in any way upon the value at a previous iteration, then the loop is sequential, and iterations must be executed sequentially, otherwise all the executions of the loop body are notionally independent, and can be executed in parallel. This choice affects the IF1 produced from a loop, as IF1 provides three compound nodes with which loops can be expressed: for sequential loops with post-test (*LoopA*) and pre-test termination conditions (*LoopB*), and the *ForAll* node for parallel loops. Let us focus on distribution control loops (iteration control loops are easily translated into *LoopA* or *LoopB* nodes, depending on the presence of a pre- or post-test). The syntax for such loops is the following:

```
for in-exp-list                 % top part
    [decl-def-part]
do [decl-def-part]              % body part
returns return-clause           % returns part
```

A loop expression has three parts: the *top* part establishes indices (`in-exp-list`) and initialization of the loop (`decl-def-part` is evaluated only once, so it does not affect the dependency analysis); the *body* part defines the actions to be carried out in a distributed manner (`decl-def-part`); the *returns* part packages the produced values. In the body part, carried values are transmitted from one iteration to the following. We study dependencies between variable definition and usage to detect if this loop can be evaluated in a sequential or in a parallel manner. If the body part contains references of one variable which is defined in the previous iteration of the body part, then the loop must be executed in a sequential manner and will be translated into a *LoopB* node. Otherwise, the loop is a parallel one and can be translated into a *ForAll* node. Such an analysis of dependencies is facilitated by the functional nature of Sisal: side-effects and aliasing problems are banished by definition.

4.2 Discussion

IF1 is a powerful intermediate form capable of expressing data dependencies in expressions, iterative control flow, and recursive and higher-order functions, as well as standard data structures; there are, however, some aspects in which it lack expressiveness and this leads to somewhat clumsy translations of Sisal 2.0. This is principally a consequence of a much more sophisticated model of array structures employed in Sisal 2.0, whereby arrays are multidimensional values, can be constructed monolithically, and sophisticated sub-array selection operations can be expressed. We firstly examined how translation could be achieved with existing nodes. However, we discovered that this option leads to a loss of efficiency and abstraction in terms of evaluation policy for those parts of the language. This allowed us to identify those aspects of IF1 which needed modification and provided insight into how to do it. A forthcoming document will describe in detail our proposed extensions to IF1, especially concerning arrays.

5 Optimizing IF1 graphs

Optimizations on the IF1 graph constitute an early phase of the OSC compiler [10] (inline expansion, common subexpression elimination, loop invariant removal, dead code elimination, see [15] for an overview). To demonstrate our approach using IF1, we formally describe common subexpression elimination.
The principle of this optimization is to eliminate redundant computations. Redundant computation occurs in an IF1 graph when the same sub-expression is computed more than once. To perform this transformation, for each node (i.e. sub-expression), we search for a sub-graph which matches the one whose root is the node currently visited (although the structure is a DAG, it is similar to a tree when traversed from nodes to their predecessors). When such a sub-graph is detected, all its "clients" (nodes using its results) are connected to the visited node in the same way they were to the removed node.

Visualization of IF1 graphs uses a representation with an unordered list of nodes and edges but this is inadequate for the transformation. We define an alternative representation which uncovers actual data dependencies between graph nodes and encodes nodes predecessors. This representation permits to identify any two equal subgraphs and to restructure the whole graph. This alternative structure (named PGRAPH) comprises a list of nodes, and for each of them a list of predecessors (see definition Figure 5); the graph is then traversed from output to input: a subexpression can be identified with a node and all its predecessors, with the transitive closure of the relation "is predecessor of".

```
pgraph -> GRAPH_ELT_S NODE_S                            PGRAPH  ::= pgraph ;
        - - GRAPH_ELT_S are only cnodes and snodes
node_s -> NODE * ...;                                   NODE  ::= node;
node -> INT PRED_S;                                     - -  <node, list of predecessors>
pred_s -> PRED * ...;                                   PRED_S ::= pred_s;
pred -> INT GRAPH_ELT;                                  PRED ::= pred GRAPH_ELT;
     - - a predecessor can be a literal or an output port number + a predecessor node.
```

Figure 5. Representation of IF1 graphs by predecessors

The common subexpression elimination can be expressed in three phases: change the representation, detect and eliminate common subexpressions, and convert back to the primary representation (for visualization purpose).
Modularity makes it possible to reuse various sets (especially graph traversals) for the specification of other transformations.

The first step of the transformation is made of two traversals of the list of nodes and edges composing the primary graph representation (GRAPH) to build the PGRAPH structure. Information contained in the node description (label for the node's operation, name, etc) is needed during the detection of redundant computations, and when returning to the primary format. A first traversal constructs the list of predecessors from the edge description for each node. At this step, information on nodes themselves has not been retained. So a second traversal is needed to construct from a list describing predecessors with a pair of integers (node number, input port), a new predecessor list containing full informations on nodes.

Equality on graphs is expressed in the two following rules: equality is first checked on nodes (predicate node_equal) and then recursively on their predecessors (predicate pred_equal). A termination rule applies when nodes with no predecessors have been reached.

```
graph_equal(pgraph |- literal(_, _, _, P4, _), literal(_, _, _, P4, _) -> bool true ;

pred(node1 |- pgraph -> pred1) &                        - predecessors of node1
pred(node2 |- pgraph -> pred2) &                        - predecessors of node2
node_equal(|- node1, node2 -> bool true) &              - equality of nodes
pred_equal(pgraph |- pred1, pred2 -> bool true)         - equality of predecessors
-----------
graph_equal(pgraph |- node1, node2 -> bool true) ;
                provided diff(node1, literal);
```

After identification of redundant computation, the next step in the transformation consists in their elimination. Given a redundant subgraph (w.r.t. an original subgraph), identified by its root node, its elimination comprises a search for all nodes whose predecessors contain this root. When detected, such a node is replaced by the original subgraph's root properly connected (input ports are connected to the appropriate output ports). Returning to the primary IF1 format is done in a straightforward manner with a traversal on the PGRAPH structure. Due to lack of space, we do not detail the whole specification of graph manipulation in either form. They are composed of 150 rules or axioms and make extensive use of pattern-matching and unification.

6 Conclusion and Future Work

From the specifications, using the Centaur system, we have derived a program development environment for Sisal, illustrated in Figure 6. This environment provides a sequential execution of Sisal programs, the translation of Sisal programs into IF1 graphs, their textual and graphical presentations, and transformations on IF1 format. We first intend to complete the formal specification of the suite of transformations as used in the OSC compiler, contributing not only to a formal definition of transformation techniques for parallelization of Sisal programs, but also to a wide availability. Moreover, thanks to formal specifications, we intend to prove the correctness of the transformations, using proof assistant systems. Experiments have been done in this sense to specify semantic definitions and program transformations, and to study and prove their properties [4].

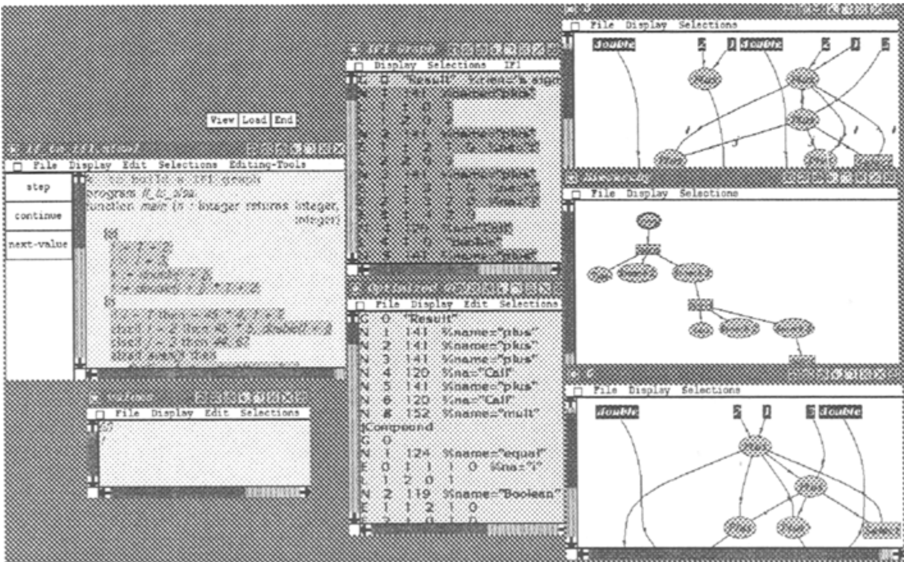

Figure 6. A Graphical Interactive Environment for Sisal and IF1.

Finally, our environment could be used for other high-level languages (logic, data-flow, object-oriented [16, 2]), or imperative languages where array operations are critical (e.g. Fortran, where parallelization is an active research area [13, 8]). Once a comprehensive set of transformation have been formally specified in a modular manner, it should be possible to take advantage of it for other paradigms, possibly through common intermediate formats.

References

1. "Parallel and Distributed Technology - Systems and Applications", Agha G. editor, 3 (4), 1995.
2. Attali I., Caromel D., Ehmety S. O., Lippi S. Semantic-based visualization for parallel object-oriented programming, To appear in OOPSLA'96, ACM Press, Sigplan Notices, San Jose, CA, 1996.
3. Attali I., Caromel D. and Wendelborn A. "A Formal Semantics and an Interactive Environment for Sisal", pp 231-258, in [18].
4. Bertot Y. and Fraer R. "Reasoning with Executable Specifications", Proc. of TAPSOFT, LNCS 915, Aarhus, Denmark, 1995.
5. Böhm A. P. W., Cann D.C., Feo J.T., Oldehoeft R.R., "Sisal Reference Manual (language version 2.0)" Draft Report, 1992.
6. Borras P. et al., "CENTAUR: the system", Third Annual Symposium on Software Development Environments, Boston, 1988.
7. Despeyroux T. "Typol: a formalism to implement Natural Semantics" INRIA research report 94, 1988.
8. Detert U. and Gerndt M., "TOP^2 - Tool Suite for the Development and Testing of Parallel Applications", CONPAR'94, Linz, Austria, LNCS 854, 1994.
9. Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing, Dongarra J.J. & Tourancheau B. eds, SIAM, Townsend, 1994.
10. Feo J.T., Cann D.C., Oldehoeft R.R., "A Report on the Sisal Language Project" Journal of Parallel and Distributed Computing, 1990.
11. Kahn G. "Natural Semantics", Proc. of STACS, Passau, Germany, LNCS 247, 1987.
12. Le Hors A., "Graph: A Directed Graph Displaying Server", in *GIPE 2 ESPRIT project, 4th Review Report*, Workpackage 4, 1992.
13. Maslov V., "Lazy Array Data-Flow Dependence Analysis" Proc. 21st ACM SIGPLAN-SIGACT POPL, Portland, Oregon, 1994.
14. Skedzielewski S. and Glauert J. "IF1 - An intermediate form for applicative languages" Manual M-170, Lawrence Livermore National Laboratory, Livermore, 1985.
15. Skedzielewski S. and Welcome M. "Data-flow graph optimization in IF1" Proc. of FPCA'85, LNCS 201, 1985.
16. "Programming Languages for Parallel Processing", Skillicorn D. B. & Talia D. eds, IEEE Computer Society Press, 1995.
17. Welcome M.L., Szymanski B.K., Yates R.K., Ranelletti J. E. "An applicative language intermediate form explicit memory management" Manual M-195, Lawrence Livermore National Laboratory, Livermore, 1986.
18. "Tools and Environments for Parallel and Distributed Systems", Zaky A. & Lewis T. eds, Kluwer Academic Publishers, 1996.