THIS PAGE LEFT INTENTIONALLY BLANK

A Common Graphical Form

David Parrott [*]
Department of Computer Science, University College London
London, England.
Chris Clack
Department of Computer Science, University College London
London, England.

30th April 1991

Abstract

We present the Common Graphical Form, a low level, abstract machine independent structure which provides a basis for implementing graph reduction on distributed processors. A key feature of the structure is its ability to model disparate abstract machines in a uniform manner; this enables us to experiment with different abstract machines without having to recode major parts of the run-time system for each additional machine. Because we are dealing with a uniform data structure it is possible to build a suite of performance measurement tools to examine interprocessor data-flow and to apply these tools to different abstract machines in order to make relative comparisons between them at run-time. As a bonus to our design brief we exploit the unifying characteristics of the Common Graphical Form by using it as an intermediate language at compile-time.

1 Introduction

Graph reduction is a well established method for executing lazy, higher order, functional programs on sequential architectures and a number of abstract machines have been designed to execute programs using this technique (e.g. [1]). Recent research has been directed towards graph reduction on parallel architectures, adapting existing abstract machines to cope with the added complexities (e.g. [2]) and building new computational models with parallelism as a primary design factor ([3]).

Some study has been made towards theoretically comparing abstract machine designs [4, 5]. The research concentrates on the mathematical equivalence of abstract reduction mechanisms but does not encompass the wider environmental issues such as suspending and resuming tasks and the effects due to the way program state is represented in different machines. Our research programme includes the *practical* comparison of the behaviour of parallel abstract machines and, in this paper, we present the development of a data structure which greatly simplifies the experimental process. Much of the overhead experienced by distributed parallel architectures is due to interprocessor communication, hence the structure is designed to standardise the mechanisms for performing and measuring run-time communication, irrespective of the abstract machine employed.

Parallel architectures are amazingly diverse, ranging from tightly coupled, shared memory systems (e.g. the BBN Butterfly) to loosely coupled, distributed memory systems (e.g. the Intel iPSC). Understanding why abstract machines behave as they do in given circumstances should be an important factor influencing the choice or design of an abstract machine

[*]Supported by a SERC research studentship

for a particular environment. We shall be concentrating on distributed memory architectures which rely on message passing for interprocessor communications but the techniques described will also be applicable to those shared memory architectures which use some form of message passing. The data structure is based on the lambda calculus, common to most modern abstract reduction mechanisms, and is a graphical expression of functional programs, hence it is called the Common Graphical Form (or $\mathcal{CGF}$).

The *primary* use for $\mathcal{CGF}$ is to encapsulate programs at run-time in an abstract machine independent manner. However, because it identifies the common ground between various classes of abstract machine, we can also use the structure at other stages of implementation where there is a need for low level functional program description. For example, $\mathcal{CGF}$ can be used as a low-level intermediate language at compile-time. We shall discuss this secondary issue in the latter sections of the paper but it is worth noting at this point the existence of the dual modes of operation and to realise that they are distinct.

1.1 Machine Comparisons

It is possible to compare the raw performance statistics of abstract machines and to state which takes the least amount of time to execute a particular program [6, 7]. Unfortunately, this does not provide sufficient information to say *why* one machine runs faster than another. Also, distributed processing systems often communicate over data networks and so a degree of non-determinism is introduced due to network loadings, bottlenecks, and the reliability of the communications hardware. Non-determinism reduces the consistency of real-time measurements and makes it difficult to glean useful information. It is therefore more informative to monitor directly the internal workings of the abstract machines in order to investigate the effects of the host environment on the machines' efficiency. This provides a rationale for a machine's performance statistics and so helps to make predictions about how the machine would fare if the environment were altered.

Comparing the internal operations of different abstract machines is difficult because each machine uses its own, unique, data structures to denote programs. If the number of *reductions* is used as the performance metric then we have to account for the fact that there may be significant variation in the amount of work done by a single reduction (this can also be dependent on the way a program is compiled [6]). Moreover, the relationship between similar instructions on different machines may not be linear (e.g. instructions to select the next redex, and those to build graph structures).

We turn to parallel processing in order to speed up program execution, thence success depends heavily on the ability of a number of processing elements to cooperate efficiently. We believe, therefore, that interprocessor communication is an important factor when measuring distributed abstract machine behaviour and we use it as a basis for our study. A method of standardising interprocessor communications is required; $\mathcal{CGF}$ fulfils this requirement, making it possible to construct a monitoring and measurement package that can cope with all of the abstract machines likely to be studied. If $\mathcal{CGF}$ is properly designed then it will not be necessary to rebuild the mechanisms for each new abstract machine.

The peripheral advantages gained by using $\mathcal{CGF}$ are also applicable to *shared* memory systems. For instance, it is possible to build a standard set of tools to manipulate the data structure without committing the techniques to any specific abstract machine. The unification of communication measurement facilities made possible by the data structure is also relevant to a shared memory architecture, although it is of greatest utility when the primary method of interprocessor communications is message passing.

1.2 Organisation of the Paper

The remaining sections of this paper are organised as follows. Section 2 examines the depiction of functional programs at the lowest, abstract machine level. Using this as a starting point, we consider what is desired of a Common Graphical Form. Section 3 contains a complete description of $\mathcal{CGF}$ for use at run-time, giving justifications for the design decisions taken. Section 4 deals with the secondary use of $\mathcal{CGF}$ as a compile-time intermediate language, examining higher level characterisations of functional programs and presenting a full textual representation for the structure. Section 5 gives examples of uses for $\mathcal{CGF}$ and, finally, section 6 concludes with $\mathcal{CGF}$'s achievements.

2 Representing Functional Programs

$\mathcal{CGF}$'s primary goal is to describe lazy, higher order, functional programs at run-time in a manner that is not dependent on any one abstract machine so that interprocessor communications can be normalised.[1] To achieve this it is necessary to discover (a) the fundamental properties that are common to the many abstract models and (b) what information needs to be passed between remote processors. We shall examine both low and high level structures to obtain a complete picture. In this section we concentrate on the low level structures employed by some well documented abstract machines.

Graphs, Code, and Stacks

Implementations of lazy, higher order, functional languages typically have three distinct uses of memory:

heap space in which to build graphical structures,

code memory in which to place (abstract) machine code sequences, and

stacks on which evaluation is controlled (or, given that the result is not to be shared, the stack may be used by some compiled abstract machines to perform evaluation without having to access the heap).

Higher order functions and laziness call for a mechanism to implement *suspensions*; laziness also implies the ability to share the results of computations. The suspension mechanism is usually expressed using a *closure* which is described by a pair: ⟨function, environment⟩. This is precisely what every abstract machine needs to build. Both closures and shared values demand a more flexible storage medium than a stack, so a *heap* is required. The type of information stored in the heap is of consequence to $\mathcal{CGF}$'s design so a detailed study is made in the following sections.

2.1 Graphs

Table 1 lists a number of abstract machines, showing the *tuples* which make up the graphical run-time information on which each machine operates. The table contains a fairly small sample of abstract machines but it should be apparent that a trend is becoming established. The terminology varies from tuple to tuple (it is based on the sources referenced in the second column of the table), hence some explanation is necessary before we continue.

Function, code, and codeptr all reference some (abstract) machine code whilst Field_1 and head reference a graphical (i.e. interpretive) function definition. Field_2, tail, item, and arg are either unboxed data items or pointers, and Lal George's env is simply a collection of arguments (equivalent to arg*). Tim's frameptr is also a pointer to an environment but with a subtle difference: a Tim *frame* contains ⟨codeptr, frameptr⟩ pairs instead of single values or pointers. This is isomorphic to the ⟨code, arg*⟩ arrangement [15], as demonstrated in

[1]Note that $\mathcal{CGF}$ does not preclude the use of efficient, internal abstract machine representations (see section 5.1).

Machine	Reference	Node Description
Spineless G-Machine	[8]	$\langle$function, arg*, tag$\rangle$
Spineless Tagless G-Machine	[9]	$\langle$function, arg*$\rangle$
Tim	[10]	$\langle\langle$codeptr, frameptr$\rangle$*$\rangle$
$\langle \nu, G\rangle$-machine	[2]	$\langle$tag, code, link, arg*, tempspace$\rangle$
Lal George Abstract Machine	[11]	$\langle$tag, code, waitcount, notechain, env$\rangle$
Flagship	[12] [13]	$\langle$tag, item*$\rangle$
G-machine	[1]	$\langle$tag, field_1, $\text{field}_2\rangle$
Four-stroke Reduction Engine	[14]	$\langle$tag, head, tail$\rangle$

* indicates zero or more

Table 1: Some examples of graphical data.

figure 1 (just *slide* each function reference, f_n, from the head of a *vector application node*, backwards along the arrow pointing to the node, and pair it with the pointer), so no special data structure is required to build Tim's heap.

The other items in the tuples in table 1 are house-keeping information of one sort or another and these will be dealt with in due course. (There will be yet further house-keeping overheads depending upon specific implementation details; again, this will be dealt with later.)

The reason for the similarities between the abstract machines' data structures is that all are realisations of closures, thus if $\mathcal{CGF}$ is to be abstract machine independent then a general purpose closure mechanism must be defined. A closure is a function and a set of arguments which form an environment. Conveniently, a *vector application* node satisfies the basic format of a function applied to a variable number of arguments, which just leaves the problem of describing a (possibly hierarchical) environment. This issue is tackled in section 3.2.1.

Any remaining data such as further tags and dynamic links (see section 5.1.2) can be packaged as *extra* information with respect to the minimal closure definition. The *extras* do not effect the semantics of the graph but merely carry state information as required by the abstract machine. Separating semantically relevant data from overheads in this way allows a more accurate interpretation of the interprocessor communications to be made.

2.2 Code

A naïve scheme for program loading may dictate that every compiled function definition is loaded onto every processing element before execution commences (e.g. see [16]). This is clearly undesirable for large programs because a great deal of time and memory is wasted by duplicating code that will never be executed. An intelligent program loader, however, might partition the program code so that it is divided amongst the processing elements in the same manner that a graph is divided between the processors of an interpretive graph reduction engine. In this scenario, code sequences must be transferred between remote processing elements when tasks are migrated and hence a special addressing mode is required to uniquely name the sequences. By exploiting this addressing mode further, $\mathcal{CGF}$ can be used

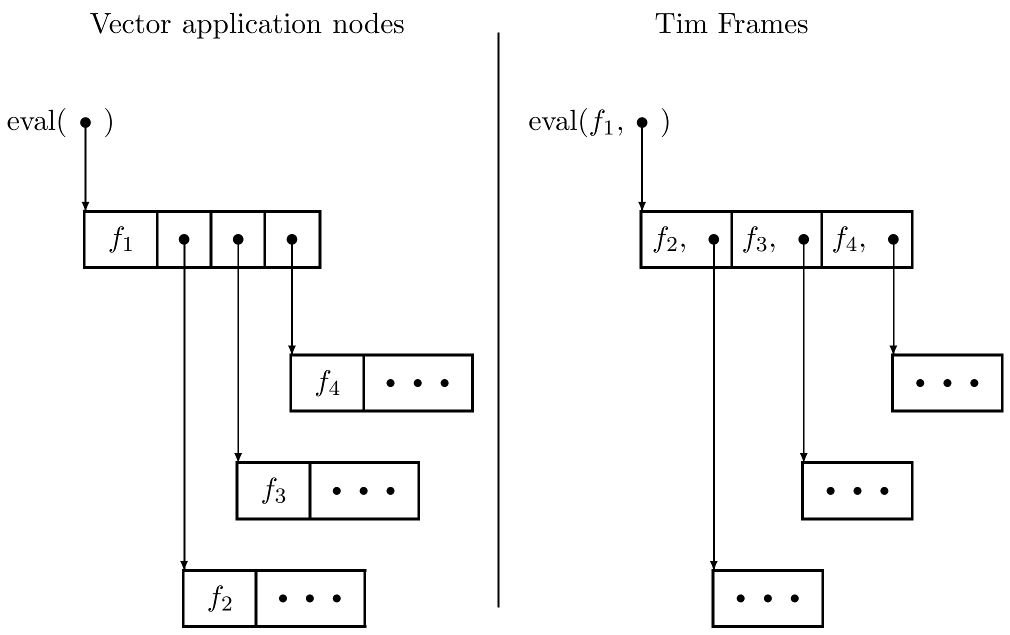


Figure 1: Tim's frames and vector applications are isomorphic.

by the loader itself for initial program distribution, again saving effort when experimenting with new abstract machines because the loader remains essentially unchanged from machine to machine.

Apart from insisting that separate code sequences are distinguishable, no structure is imposed on the code and no meaning is attributed to it. Each sequence may contain native machine code destined for a specific piece of hardware, abstract machine code that will be interpreted by an abstract machine, or some hybrid, depending upon the experimental whims of the implementer.

2.3 Stacks

Stacks are used extensively by abstract machines when evaluating programs. A stack is an efficient tool for expression evaluation and is an efficient medium for evaluating local, unshared, temporary values which do not incur heap updates [8, 9]. The terms *local* and *temporary* may give the misleading impression that it will not be necessary to move stacked information between non-local processors. There are cases, however, when stack based information will need to be moved—for example, state information required to resume a suspended task may reside on the stack; if the task is migrated then the stacked data must also be migrated. Passing arguments to a remote function may also benefit from the ability to transmit a segment of local stack. Therefore the tools for interprocessor communication are designed to cater for the transference of stacked data but, because stacks are an abstract machine mechanism and not a natural part of a program graph, no references are made to stacked items from the graph and hence no extension to the $\mathcal{CGF}$ addressing modes is necessary.

3 A Description of $\mathcal{CGF}$ for use at Run-Time

In this section we describe the Common Graphical Form as a mechanism for encapsulating functional programs at run-time. A program is represented in Common Graphical Form

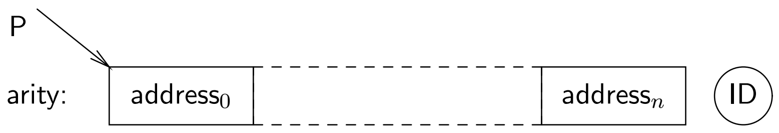


Figure 2: A node in Common Graphical Form.

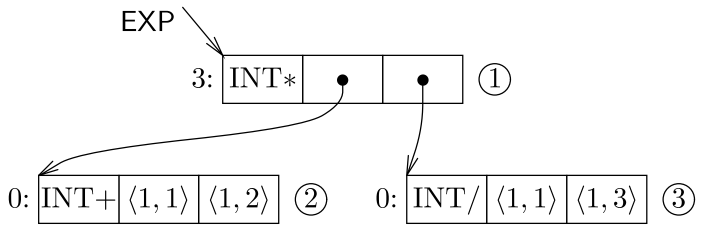


Figure 3: $\mathcal{CGF}$ structure for $\mathsf{EXP} = \lambda x\, y\, z.\, (x + y) \times (x/z)$

by a collection of vector application nodes. Figure 2 presents an illustration of a node; the arity coefficient indicates the number of variables bound by the node, P is a *label*, ID is a unique *identifier* depicting the abstract address of the node, and address is one of the addressing modes described in section 3.2 below—the addressing modes are the singularly most important feature of $\mathcal{CGF}$.

Identifiers and labels serve two distinct diagrammatic purposes: identifiers are fixed to their respective nodes, but labels may move from one node to another. This will be useful when we perform transformations on $\mathcal{CGF}$ (e.g. note the label movement in figure 6).

3.1 $\mathcal{CGF}$'s Relation to the Lambda Calculus

$\mathcal{CGF}$ is based on the typed lambda calculus, augmented with a number of primitive functions and constants. Mechanisms for bound variable abstraction, function application, and beta reduction are provided and thus $\mathcal{CGF}$ is computationally complete. $\mathcal{CGF}$ also provides for explicit expression-sharing (which can be used to implement named functions) and for the definition of recursive functions by the use of cyclic graph structures.

User functions are defined by assigning an arity to an expression node, the expression can then be applied to some arguments in the usual way. Nodes with zero arity are employed to portray bracketed sub-expressions. For example, the lambda expression $\lambda x\, y\, z.\, (x + y) \times (x/z)$ requires three nodes (see figure 3), each containing an operator applied to two arguments. The top node applies the multiplication operator to each of the two (zero arity) sub-expressions.

3.2 Addressing Modes

We provide a fixed number of addressing modes to describe various entities. It is a straightforward exercise to perform case analysis (or pattern matching) over the addressing modes in order to process the information within a node. The following sections describe the addressing modes in detail.

3.2.1 Bound variables

In [17] de Bruijn shows that variables bound within lambda expressions can be fully described by a pair of integers which provide access to *environments*. We use a modified version of his scheme, written $\langle d, i\rangle$, to reference the *ith* bound variable belonging to the node which occurs d bindings beyond the reference. (Binding levels are illustrated by following pointers backwards, counting the number of non-zero arity nodes visited.) We refer to the $\langle d, i\rangle$ construction as a *de Bruijn* address (Cousineau et al use a similar technique to overcome naming problems in their Categorical Abstract Machine [18]).

Note that all de Bruijn addresses in a lambda lifted program are of the form $\langle 1, i\rangle$ because variables must be bound by the innermost, non-zero arity node. There is no reason to suppose that all abstract machines will use lambda lifting however (e.g. [4]), so the full power of de Bruijn addressing is retained.

3.2.2 Inter-node references

To build a graph structure using $\mathcal{CGF}$ we must be able to make arbitrary references to nodes. Three addressing modes are provided for this purpose:

Abstract addresses are borrowed from [19] where graphs are described textually: each line of text symbolizes a unique node in the graph so that line numbers can be used to address the nodes. This abstracts away from machine level addressing and is ideal when graph segments are packaged up for interprocessor communication.

Real addresses are machine level pointers that identify memory locations within the concrete address space of the local processing element. (When graphs are migrated from one processor to another, some real addresses will be translated into remote addresses and others will become abstract addresses.)

Remote addresses refer to memory locations in remote address spaces. Hughes notes that remote pointers are rare in comparison to local pointers [20] and so by separating local and remote addressing into two distinct modes it is possible to build local indirections to remote addresses. Under non-strict evaluation, an indirection is followed only when a normal form is required, hence remote addresses are only ever encountered when a value *must* be obtained from a remote processing element.

Diagrammatically, a reference to a node may be shown (a) by an arrow physically pointing to the node, (b) by an identifier (which may be numeric), or (c) by a label. The graph to which a label, P, or an identifier, I, refers may be written $\vec{\mathsf{P}}$ or $\vec{\mathsf{I}}$, respectively.

3.2.3 Data objects

A single addressing mode is used to describe all data objects so that case analysis is not unduly slow. Structured data is built using boxed nodes and in the unevaluated state it is represented by the application of a constructor function to the requisite number of arguments. No constraints are placed on the range of structured data types available because new structures can be introduced by adding extra constructor primitives.

Data objects may or may not need to carry further information to distinguish different data types—this is dependent upon the underlying implementation. $\mathcal{CGF}$ possesses an explicit type mechanism for those implementations which need to perform type checking at run-time (a description of this mechanism is given in section 4.4) but it is not mandatory and need not be used when types are implied by context such as in strongly typed systems. Primitive functions are considered to be data, and may either be distinguished by context (i.e. by function application) or may be given a special tag. The choice of which primitive functions to employ is left to the discretion of the implementer.

4 A Description of $\mathcal{CGF}$ for use at Compile-Time

In this section we look at one way in which we can extend the compass of $\mathcal{CGF}$ beyond its use at run-time by demonstrating that, in order to satisfy those abstract machines which require

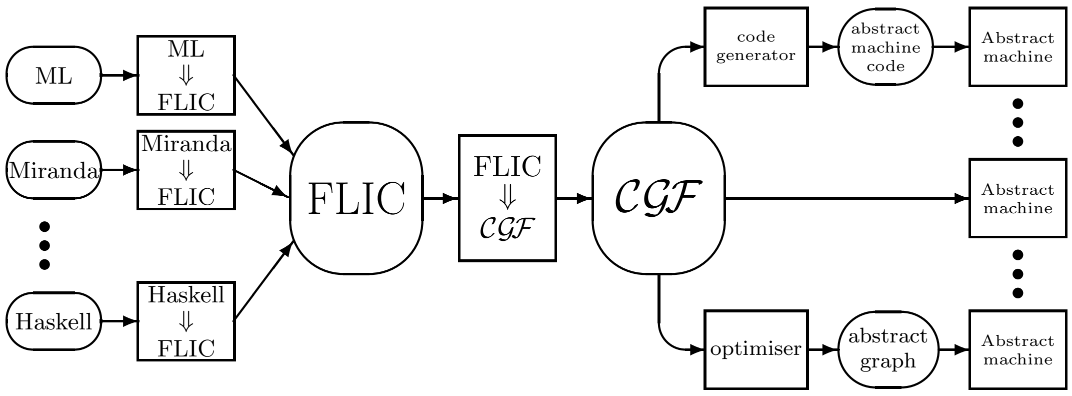


Figure 4: 𝒞𝒢ℱ can be used for compilation in conjunction with FLIC to provide a route from many high level languages to many abstract machines.

tagged data items at run-time, we have inadvertently developed a low level mechanism that can be used to implement FLIC at compile-time. 𝒞𝒢ℱ is thus promoted to the status of an intermediate language. Firstly, we consider 𝒞𝒢ℱ's relationship with FLIC and other intermediate languages to get an idea of where 𝒞𝒢ℱ fits into the overall framework. Then we present a *textual* description of 𝒞𝒢ℱ which is amenable to transmission using electronic mail, that can be used to inspect intermediate stages of compilation, and which is easily interpreted by parts of the compiler built using a functional programming language.

4.1 The Functional Language Intermediate Code

FLIC [21] was designed to sit one level below the high level programming languages to provide a reference point for compilers, allowing standardised mechanisms such as source to source compilation modules. Modularity implies that techniques can be freely exchanged and new languages can be easily accommodated. FLIC's design incorporates a powerful set of primitive functions which are capable of expressing many functional constructions, including complex data structures.

𝒞𝒢ℱ is a low level tool, designed primarily to cope with run-time and communications demands. Nevertheless, FLIC's family of primitives are a powerful compile-time asset and so we have adopted them as our primitive function set for compilation purposes. Therefore, 𝒞𝒢ℱ can provide a compile-time facility similar to FLIC but at a lower level.

FLIC programs retain much of the flavour of the lambda calculus; they are textually concise, are human readable, and are amenable to straightforward *pretty-print* formatting. 𝒞𝒢ℱ programs are less readable than the equivalent FLIC versions and textually less concise (see, for example, section 4.4) but, just as FLIC removes syntactic sugaring such as list comprehensions, 𝒞𝒢ℱ does much the same with syntax such as arbitrarily nested `let` and `letrec` definitions which complicate parsing and concrete representation. 𝒞𝒢ℱ's low level structure is well suited to rapid compilation because, in addition to maintaining a level of abstraction above the underlying abstract machine, it imposes little or no parsing overhead. This may be significant when there are many stages of compilation (for example, approximately 25 passes are claimed for the LML compiler [22]).

𝒞𝒢ℱ is designed with efficient machine readability in mind and so does not necessarily replace FLIC; it is, however, complementary. Just as FLIC provides a focal point for high level languages, 𝒞𝒢ℱ is able to provide a similar focus for compiler backends. Figure 4 demonstrates how 𝒞𝒢ℱ fits into the compilation scheme—note that source to source transformations can occur at either the FLIC or 𝒞𝒢ℱ stages.

An important feature of FLIC is its ability to annotate programs in order to influence the compiler's decisions (although not influencing the semantics of the program). $\mathcal{CGF}$'s *extras* field is ideal for implementing FLIC annotations because, like the annotations, it is designed to hold information which is not directly related to program semantics. Program transformations performed at the $\mathcal{CGF}$ level can then either take annotations into account given code to interpret the fields correctly, or they can simply ignore them, naïvly carrying them through to the next stage of compilation. (Some further work may be needed in this area to decide how best to maintain annotations when non-trivial transformations cause the program to be altered significantly.)

4.2 Another Graphical Language: GCODE

The functional language project at Birmingham and Warwick universities found it necessary to create a standard, printable format for representing graphs which they called GCODE [19]. The underlying mechanism is a binary graph but the representation is reasonably high level—for example, the original variable names are retained from the source as an aid to debugging. The standard is very tightly defined as it is intended for a specific environment and, more importantly, it does not attempt to address the problems of distributing graphs across separate address spaces.

This differs from $\mathcal{CGF}$'s design brief which requires a more general purpose mechanism. $\mathcal{CGF}$ is consequently more flexible at compile-time (i.e. conversion from GCODE to $\mathcal{CGF}$ is simple, but not vice versa) and is more powerful at run-time, catering for interprocessor communications and providing low level access to function parameters. $\mathcal{CGF}$'s abstract addressing mode (section 3.2.2) was inspired by one of the techniques described in [19].

4.3 DACTL

$\mathcal{CGF}$ differs from the DACTL intermediate language [23] because the latter is designed to encompass a range of declarative programming styles and so describes programs at a higher level of abstraction; low level issues are ignored. Our objective was to develop an intermediate language that could be used to describe programs at the abstract machine level, i.e. in terms of memory cells. For example, DACTL utilises local naming to describe sharing whereas $\mathcal{CGF}$ describes a graph exactly, using its abstract addressing mode to build shared nodes and cycles.

It is clear then that the objectives of the two formats are quite different. DACTL provides a metric for reasoning about different programming styles while, by limiting its scope to the applicative programming style, $\mathcal{CGF}$ provides the facility to measure and reason about the low level functionality of different abstract machines.

4.4 Representing $\mathcal{CGF}$ Textually

$\mathcal{CGF}$ is ideally suited to internal low level representations within a run-time system or a compiler but it is sometimes necessary to build a textual version so that the code may be inspected manually and so that it can be read in a straightforward manner by compiler modules written in a functional language. The following textual representation is simple and efficient to parse.

Each vector application node is represented by a single line of text terminated with a newline character. The first entry specifies the number of cells and is followed by a textual description for each cell. The arity of the node is given next, followed lastly by any (implementation specific) *extra* information. Addressing modes are specified by a two character code and data types by a three character code as follows:

Code	Addressing mode	Code	Data type
NO	no address (NULL)	INT	integer
AB	abstract	FLP	floating point
RE	real	PRM	primitive function
DA	data	STR	string data
DB	de Bruijn	CHR	character data
RT	remote	BLN	boolean data

Qualifiers, GN and SC, specify whether abstract addresses refer to graph nodes, or supercombinators. The textual representation for addresses is unambiguous and so no special delimiters are needed to separate cell definitions. Numeric and character constants are written in ascii format and are distinguished by context. Primitive functions are given in uppercase, e.g. CONS, TRUE, INT+, IS-NIL. The following is an example specification (equivalent to EXP in figure 3):

Abstract address	Textual description					
(line number)	cells	function	arguments		arity	extras
1	3	DA PRM INT∗	AB GN 2	AB GN 3	3	
2	3	DA PRM INT+	DB 1 1	DB 1 2	0	
3	3	DA PRM INT/	DB 1 1	DB 1 3	0	

5 Examples of Use

This section demonstrates some of the ways that $\mathcal{CGF}$ can be used both at run-time, as was originally intended, and during compilation. The given examples currently form part of an active programme of research in distributed implementations for functional programming at UCL.

5.1 Implementing Run-Time Systems

The primary design aim of $\mathcal{CGF}$ is to simplify the creation and maintenance of a run-time environment and to support a range of different abstract machines and system configurations. The abstract machines will be of many types, including compiled and interpreted machines, and loading and reduction strategies will demand varying degrees of interprocessor communication with assorted levels of granularity.

$\mathcal{CGF}$'s structures can be optimised for run-time use and tailored for compatibility with specific abstract machines—the specification is sufficiently flexible to enable special versions to be constructed and still take advantage of the library of tools. Our library contains routines to manipulate $\mathcal{CGF}$'s addressing modes, utilities to measure the data-flow between remote processing elements, programs to flatten graphs into a form suitable for transmission between processing elements and to reconstruct them at the other end, and so on. The library utilities are expressed in terms of $\mathcal{CGF}$ addressing modes and require minimal interfaces to accommodate the implementation details of the individual abstract machines.

Much careful consideration has been given to the design and we have avoided imposing overheads that could bias measurements in favour of one abstract machine over another; this is important because $\mathcal{CGF}$ will be used to make relative comparisons between machines. Many options are left available to the implementer to make the most efficient possible use of $\mathcal{CGF}$.

5.1.1 $\mathcal{CGF}$ graph reduction engine

An interpretive graph reduction engine has been built which uses binary $\mathcal{CGF}$ nodes (i.e. every node has two cells) as the basis for building graphs. The interpretive engine has the

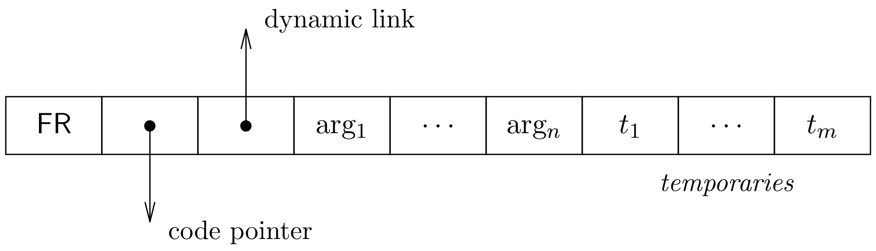


Figure 5: A $\langle \nu, G\rangle$-machine frame node.

advantage of simplicity, enabling experiments to be made with loading and task management strategies without the added complications resulting from compiled graph reduction. The simplicity of the machine allows new ideas to be rapidly prototyped and thus encourages experimentation—measurements can be taken from compiled abstract machines once the techniques are properly understood.

5.1.2 The $\langle \nu, G\rangle$-machine

The $\langle \nu, G\rangle$-machine [2] is a compiled abstract machine in which heap allocated graph nodes are treated as stack frames, thereby simplifying task suspension and removing the overhead of copying data between the heap and the stack. To emphasize their superior status the nodes are referred to as *frame nodes.* A frame node (shown in figure 5) consists of five parts:

1. A **tag** which distinguishes between *reducible* frame nodes (those which contain a function applied to the correct number of arguments), constant applications (a function applied to too few arguments), and constructor values (complex data items).

2. A **code pointer** identifying the compiled function that is to be applied to the arguments.

3. A **dynamic link** pointing backwards to the calling frame node (and hence to the preceding stack frame).

4. A list of **arguments**, $\mathrm{arg}_1 \dots \mathrm{arg}_n$, to which the function is applied.

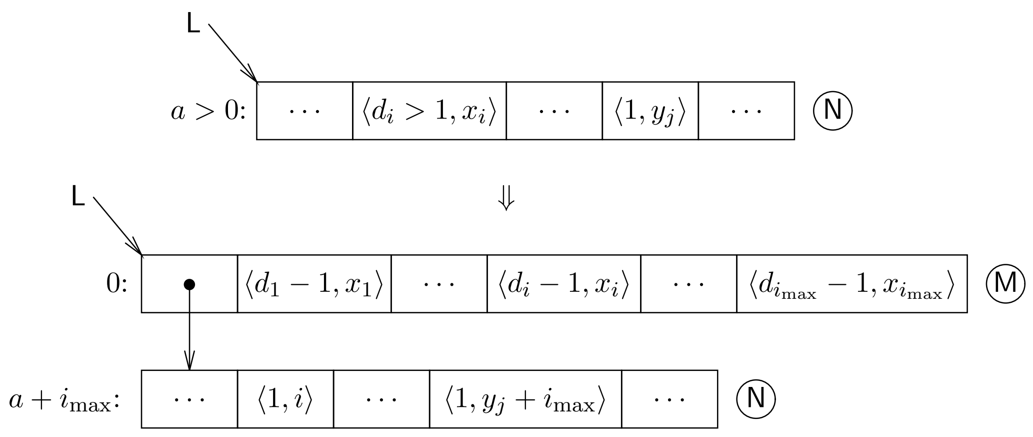
5. Cells, $\mathrm{t}_1 \dots \mathrm{t}_m$, for **temporary** calculations. The stack frame grows and shrinks at run-time as items are effectively pushed onto and popped from the stack. This space can either be allocated dynamically or, where it is possible to calculate an upper bound on its size, can be fixed.

It is a straightforward exercise to build frame nodes in $\mathcal{CGF}$: the tag and the dynamic link are included as *extra* information because they are implementation details whilst the function, arguments, and temporaries are naturally represented by a standard $\mathcal{CGF}$ vector application. If the implementation uses a fixed size temporary space then another *extra* item is required to specify the size of the currently active part of the stack.

The purely interpretive facets of $\mathcal{CGF}$ such as de Bruijn addressing are, of course, not required by the $\langle \nu, G\rangle$-machine run-time environment but all other aspects of the format are employed by a distributed implementation.

5.1.3 Partitioning and loading

Partitioning and loading programs onto distributed processors is a non-trivial task which has inspired a large amount of interest (e.g. [24]) and will continue to do so as parallel and distributed processing become ever more common. Using $\mathcal{CGF}$ as the underlying structure for



L
$a > 0$:
$\cdots$
$\langle d_i > 1, x_i \rangle$
$\cdots$
$\langle 1, y_j \rangle$
$\cdots$
N
$\Downarrow$
L
0:
$\langle d_1 - 1, x_1 \rangle$
$\cdots$
$\langle d_i - 1, x_i \rangle$
$\cdots$
$\langle d_{i_{\max}} - 1, x_{i_{\max}} \rangle$
M
$a + i_{\max}$:
$\cdots$
$\langle 1, i \rangle$
$\cdots$
$\langle 1, y_j + i_{\max} \rangle$
$\cdots$
N

Figure 6: Lambda lifting a sub-graph, N, which contains $i_{\max}$ distinct free variables.

this activity allows the production of a single, generic program partitioner/loader that can be interfaced with a number of abstract machines. The advantages are three fold: firstly the algorithms do not have to be re-coded when the target abstract machine is changed and hence the corollary that improvements to the algorithms are automatically available to every abstract machine. The third advantage is that with the data already in $\mathcal{CGF}$ format, it is ready for transmission to the distributed processing elements without further transformation.

5.2 Modular Compilation

$\mathcal{CGF}$ is intended to highlight and exploit the common ground in disparate abstract machines so that we can experiment with run-time strategies and compare the effects across the spectrum of abstract machines under test. As a consequence of being a low level common core representation, $\mathcal{CGF}$ proves to be an ideal medium for source to source compilation. The format is very close to the underlying graph structures that will be used at run-time and as such can be manipulated with efficiency and ease.

5.2.1 Program transformation

Three types of program transformation have been implemented as $\mathcal{CGF}$ to $\mathcal{CGF}$ modules; these are lambda lifting, sharing analysis, and time-complexity analysis. Figure 6 illustrates how neatly the lambda lifting algorithm [25] translates to $\mathcal{CGF}$'s graph structure[2] with an arbitrary number ($= i_{\max}$) of distinct free variables (this is the most general case). The code to lambda lift the whole graph consists of two nested depth-first graph traversals, the outer traversal to locate free expressions and the inner traversal to transform the whole subgraph including subsidiary nodes. Locating free variables is trivial because of the nature of de Bruijn addresses: an address $\langle d > 1, x \rangle$ depicts a free variable and is lifted by moving it outside the current binding, decrementing d to compensate for the reduced depth (a new node, M, is inserted into the graph to hold the lifted variables). Existing bound variables, $\langle 1, y \rangle$, are adjusted to account for the increased number of arguments. References which are buried many levels deep are simply *bubbled* upwards incrementally by the action of the outermost depth-first search. The algorithm requires the storage overhead of just a single marker which is used by both the inner and outer depth-first searches.

[2]full-laziness is achieved by modifying the algorithm slightly or by applying a second pass which recovers full-laziness upon completion of any other graph transformations.

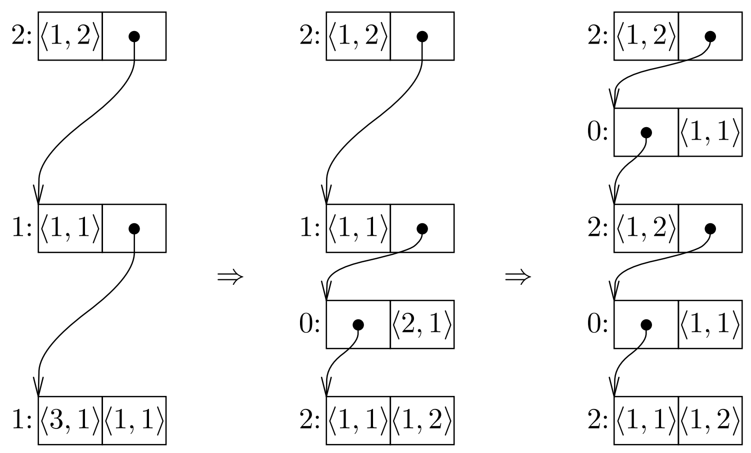


Figure 7: Lambda lifting $\lambda f\, g.g\,(\lambda h.h\,(\lambda x.f\, x))$ to give $\lambda f\, g.g\,((\lambda q\, h.h\,((\lambda p\, x.p\, x)\, q))\, f)$

Figure 7 provides a concrete example of lambda lifting using the algorithm given above. The lambda expression $\lambda f\, g.g\,(\lambda h.h\,(\lambda x.f\, x))$ contains three nested abstractions; the function variable f, which is applied at the innermost level, is free with respect to the two inner abstractions and is bound by the outermost abstraction. Two successive lift operations are needed to bubble the variable upwards, adding two extra nodes to the graph in the process.

5.2.2 Abstract machine backend

It is common for abstract machines to be specified by a set of transition rules and by a set of compilation rules (e.g. see [2, 10]) where the latter is usually defined over a small, convenient input language. $\mathcal{CGF}$ is ideal for use as input to a backend because the format so closely resembles the small languages employed in the specifications; the task of building a backend reduces to a simple implementation of the rules. As an example, we shall consider some of the compilation rules given in [2] for the $\langle \nu, G\rangle$-machine.

The $\langle \nu, G\rangle$-machine uses supercombinators and so all de Bruijn addresses will be of the form $\langle 1, i\rangle$. Advantage can therefore be taken of the constant *depth* by introducing a compiler pass which converts all de Bruijns into the form $\langle a, i\rangle$ where a is the *arity* of the enclosing user function; this gives the backend direct access to arity information without having to search the graph. The rule for compiling a parametric value, x, given an environment, ρ, and a current environment depth, n, is:

$$\mathcal{C}[\![\, x \,]\!]\rho\, n = \mathsf{PUSH}(n - \rho(x))$$

which becomes just:

$$\mathcal{C}'[\![\, \langle a, i\rangle \,]\!] n = \mathsf{PUSH}(n - a + i - 1)$$

thus reducing the environment function, ρ, to a closed form expression with respect to the values given by the de Bruijn address.

Consider the compilation scheme given for compiling function definitions:

$$\mathcal{F}[\![\, f\ x_1 \ldots x_n = e \,]\!] \;=\; \mathcal{R}[\![\, e \,]\!]\,[x_1 = n, \cdots, x_n = 1]\, n$$

which maps to:

$$\mathcal{F}'[\![\, n > 0 : \boxed{e_0}\boxed{\cdots}\boxed{e_m} \,]\!] \;=\; \mathcal{R}'[\![\boxed{e_0}\boxed{\cdots}\boxed{e_m}]\!]\, n$$

The body of the function, e, is given by the application of e_0 to arguments $e_1 \dots e_m$. The $\mathcal{R}$ scheme is responsible for compiling the application of a user function or primitive function (which may be a constructor) to a list of arguments. This is precisely what the $\mathcal{CGF}$ vector application node describes—we have just to examine the addressing mode of the function cell, e_0, to determine which $\mathcal{R}$ scheme case to invoke.

The other compilation schemes map just as easily onto $\mathcal{CGF}$'s structures and a compiler backend has been constructed for the $\langle\nu, G\rangle$-machine in a functional language. The functional code to manipulate the $\mathcal{CGF}$ bears a very close resemblance to the original set of compilation rules.

6 Conclusion

In this paper we have discussed two major advantages of using the Common Graphical Form: firstly, that it has encouraged the construction of shared libraries of utilities to manipulate functional data and, secondly, that it has provided a standard method to describe ideas in a low level, abstract machine independent manner. At UCL, $\mathcal{CGF}$ is employed as part of a distributed processing environment. It is helping to bind together various areas of our research including the development and performance measurement of distributed run-time strategies, the experimental comparison of abstract machines, and the construction of compilers for executing functional languages on distributed architectures.

$\mathcal{CGF}$ has been an invaluable tool for rapid prototyping at many stages of our system implementation, especially in constructing the compiler and its associated backends. Practice has shown that it is far more straightforward to manipulate the low level structure of $\mathcal{CGF}$ at compile time than to struggle with high level source languages, or intermediate forms such as FLIC. The format is simple to manipulate using both imperative and functional programming languages and so has provided the flexibility to choose whichever is most appropriate for each prototyping exercise. We have imperative and functional language library functions to translate efficiently between textual $\mathcal{CGF}$ and its internal representations and these form a simple interface between the otherwise incompatible programming styles.

As a mechanism for implementing run-time systems $\mathcal{CGF}$ has been extremely successful. During our continued research we expect it to save us a great deal of time and effort and to provide useful results when measuring abstract machine behaviour.

References

[1] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the Conference on Compiler Construction*, pages 58–69. ACM, June 1984.

[2] L. Augustsson and T. Johnsson. Parallel graph reduction with the $\langle\nu, G\rangle$-machine. In *Proceedings of FPCA Conference*, pages 202–213. ACM, 1989.

[3] G. Burn. Implementing lazy functional languages on parallel architectures. In P. Trealeven, editor, *Parallel Computers (Object-Oriented, Functional, Logic)*, Series in Parallel Computing, chapter 7, pages 101–139. Wiley, 1990.

[4] E. Meijer. Cleaning up the design space of function evaluating machines. Technical report, University of Nijmegen, Dept. of Informatics, Mar. 1989.

[5] E. Meijer. A taxonomy of lazy function evaluating machines. Technical report, University of Nijmegen, Dept. of Informatics, June 1989.

[6] I. Robertson. Hope+ on Flagship. In K. Davis and J. Hughes, editors, *Functional Programming Workshop, Glasgow 1989*, pages 296–307. Springer Verlag: Workshops in Computing, Aug. 1989.

[7] B. Goldberg and P. Hudak. *Alfalfa: distributed graph reduction on a hypercube multiprocessor*, volume 279 of *LNCS*, pages 94–113. Springer Verlag, Nov. 1986.

[8] G. Burn, S. Peyton Jones, and J. Robson. The Spineless G-Machine. In *Proceedings of Lisp and Functional Programming Conference*, pages 244–258. Snowbird, July 1988.

[9] S. Peyton Jones and J. Salkild. The Spineless Tagless G-Machine. In *Proceedings of FPCA Conference*, pages 184–201, 1989.

[10] J. Fairbairn and S. Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In *Proceedings of FPCA Conference*. ACM, Springer Verlag, 1987. LNCS 274.

[11] L. George. An abstract machine for parallel graph reduction. In *Proceedings of FPCA Conference*, pages 214–228, 1989.

[12] P. Watson and I. Watson. Evaluating functional programs on the Flagship machine. In *Proceedings of FPCA Conference*, pages 80–97. ACM, Springer, 1987. LNCS 274.

[13] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant. Flagship: A parallel architecture for declarative programming. Technical Report FS/MU/IW/017-88, Manchester Univ., Department of Comp. Sci., Mar. 1988.

[14] C. Clack and S. Peyton Jones. The four-stroke reduction engine. In *Proceedings of Lisp and Functional Programming Conference*, pages 220–232. ACM, Aug. 1986.

[15] S. Peyton Jones. The tag is dead—long live the packet. posting on fp electronic mailing list, Oct. 1987.

[16] P. Hudak and B. Goldberg. Experiments in diffused combinator reduction. In *Symposium on Lisp and Functional Programming*, pages 167–176. ACM, Aug. 1984.

[17] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[18] G. Cousineau, P. Curien, and M. Mauny. The categorical abstract machine. In *Proceedings of FPCA Conference*, pages 50–64. ACM, Springer Verlag, 1985. LNCS 201.

[19] M. Joy and T. Axford. A standard for a graph representation for functional programs. *ACM Sigplan Notices*, 23(1):75–82, 1988. University of Birmingham Internal Report CSR-87-1.

[20] J. Hughes. A distributed garbage collection algorithm. In *Proceedings of FPCA Conference*, pages 256–272. ACM, Springer Verlag, Sept. 1985. LNCS 201.

[21] S. Peyton Jones and M. Joy. FLIC – a Functional Language Intermediate Code. Internal Note 2048, University College London, Department of Computer Science, Aug. 1989.

[22] L. Augustsson and T. Johnsson. The Chalmers Lazy ML compiler. *The Computer Journal*, 32(2):127–141, 1989.

[23] J. Glauert, J. Kennaway, and M. Sleep. DACTL: A computational model and compiler target language based on graph reduction. *ICL Technical Journal*, 5(3), 1987.

[24] B. Goldberg. *Multiprocessor Execution of Functional Programs.* PhD thesis, Graduate School of Yale University, Apr. 1988. Research Report: YALEU/DCS/RR-618.

[25] S. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.