Optimizing KLIC Generic Objects by Static Analysis

Kazunori Ueda and Ryoji Tsuchiyama[*]

Department of Information and Computer Science, Waseda University
4-1, Okubo 3-chome, Shinjuku-ku, Tokyo 169-8555, Japan
ueda@ueda.info.waseda.ac.jp

Abstract

The KLIC system has achieved both high portability and extensibility by employing C as an intermediate language and featuring generic objects that allow users to define new classes of data. It is also efficient for an untyped and unmoded language with fine-grained concurrency, but its flexibility incurs runtime overhead that could be reduced by static analysis. This paper studies how constraint-based static analysis and abstract interpretation can be used to reduce dynamic data checking and to optimize loops. We applied the proposed technique to the optimization of floating-point numbers and their arrays. The optimized KL1 programs turned out to be only 34%–70% slower than the comparable C programs.

1 Introduction

The KLIC system [4] compiles KL1 [8] programs into efficient C programs, leaving low-level details of optimization to the underlying C compilers. Thus KLIC achieves high portability, and at the same time it is quite efficient as a symbolic processing language with tagged data representation. However, it is our belief that *parallel symbolic processing is fully justified only with static program analysis*, because the (costly) effort of parallelization is easily cancelled out by insufficient effort to improve single-processor performance. Also, real-life parallel symbolic applications (such as machine learning) may involve a lot of numerical computation. We anticipate that future symbolic languages should provide certain support of high-performance computing.

We developed a constraint-based mode system for GHC programs [9] and adapted it to deal with the features of KL1. The implementation of our mode analyzer is called *klint* [13]. The mode system deals with the direction of information flow and checks if every piece of communication is cooperative. A well-moded program is *unification-safe*, that is, it will not cause the failure of unification body goals (except due to occur-check). In the terminology of Concurrent Constraint Programming [7], this means that a *tell* operation will not make the constraint store inconsistent.

Constraint-based analysis can be used also for type analysis and linearity analysis, both partly supported by the current *klint* system [13]. Our implementation technique builds upon these three analyses, plus abstract interpretation of the instantiation states of variables, as described in Section 4.

In this paper, we study how static program analysis improves the performance of the current KLIC implementation. Both for pedagogical and practical reasons, we take simple numerical computation as an example to demonstrate the technique.

2 Number Representation in KLIC

KLIC employs 32-bit representation for scalar data. To accommodate tag bits, each integer is represented using the most significant 28 bits of a word, accompanied by a

[*]Currently with Mitsubishi Electric Co. Ltd.

tag “0010”. Floating-point numbers (64-bit long in KLIC) are implemented using *data objects*, one of the three kinds of generic objects [4]. A data object is represented as a pointer to a record containing the value (or a reference to the value) and a pointer to the table of methods available to the data. Vectors (one-dimensional arrays) are represented as data objects as well.

Although data objects are a natural and flexible means of defining new classes of data, accessing and operating on data objects involve the checking of tag values and the dereferencing of a couple of pointers. Another problem is that, since data objects are basically pointers to records, floating-point numbers not accessed any more become garbage. Integers do not have the garbage problem, but still involve the manipulation of tags—checking, removing (for arithmetic operations), and/or attaching.

Arrays that allow constant-time access and update are essential in many efficient algorithms. KLIC provides one-dimensional arrays called *vectors*. Vectors in a single-assignment language such as KL1 are necessarily immutable at the language level, so KLIC implements them as *multi-version data structures* in which old element values are preserved in an association list while the latest values are stored in random-access arrays [6]. In many programs, however, old element values thus preserved are not accessed later, in which case the management of an association list turns out to be an overhead. Also, KL1 vectors can store any values including uninstantiated variables, which means that before operating on a vector element, one must check if it is already instantiated and has the intended data type.

It is highly desirable to be able to identify a class of programs which, with static analysis, can be executed without tag operations for the checking of types and the availability of data. In addition to the compile-time techniques, we have designed and implemented an array class for *instantiated* numbers, which is intended to be used in combination with static analysis.

3 Constraint-Based Static Analysis

One of the novelties of our optimization technique is that it is largely based on *constraint-based* static analysis, which has a lot of advantages:

Simplicity. The mode system has been designed so that it is accessible to programmers. In other words, its purpose is not only for compilers to analyze programs but also for programmers to understand their programs better.

Efficiency. Mode analysis is a constraint satisfaction problem with many simple mode constraints, and can be solved efficiently by unification over feature graphs [1].

Modularity. Thanks to its incremental nature, it is naturally amenable to separate analysis of large programs.

Generality. It allows simple and general formulations of various interesting applications including the diagnosis of non-well-moded programs [5] and automated debugging [2].

We have designed three constraint-based systems for static analysis: mode, type, and linearity systems. Of these, the mode system provides the most fundamental information in the sense that it is referred to by type and linearity systems. Because mode analysis has been described in detail in the literature [9, 10, 11], we outline our type analysis and linearity analysis below.

3.1 Type Analysis

There may be a number of ways to introduce a type system into concurrent logic programming, but we chose to have a type system very similar in structure to the mode system. That is, a type tells what function symbols (including constant symbols) can occur at what positions in data structures. To mention a “position”, we define the notion of a path as a sequence of pairs, of

(HBF$_\tau$) $\tau(p) = F_i$, for a function symbol in F_i occurring at p in h or B.

(HBV$_\tau$) $\forall q \in P_{Term}(\tau(pq) = \tau(p'q))$, for a variable occurring both at p and p' in h or B.

(GV$_\tau$) $\forall q \in P_{Term}(m(p'q) = in \Rightarrow \tau(pq) = \tau(p'q))$, for a variable occurring both at p in h and at p' in G.

(BU$_\tau$) $\forall q \in P_{Term}(\tau(\langle =_k, 1\rangle q) = \tau(\langle =_k, 2\rangle q))$, for a unification body goal $=_k$.

Figure 1: Type constraints imposed by a program clause h `:-` G `|` B or a goal clause `:-` B

the form $\langle symbol, arg\rangle$, of predicate/function symbols and argument positions. Let P_{Atom} be the set of paths for specifying positions in a goal, and P_{Term} the set of paths for specifying positions in a term. For example, in a goal `p(f(a,b),C)`, the symbol `b` occurs at $\langle \texttt{p}, 1\rangle\langle \texttt{f}, 2\rangle \in P_{Atom}$. The set *Fun* of function symbols are assumed to be appropriately classified into mutually disjoint subsets $F_1, \ldots, F_n$. For instance, all integers may form one subset while all floating-point numbers may form another subset.

Formally, a type is a function from P_{Atom} to the set $\{F_1, \ldots, F_n\}$. Like principal modes, principal types can be computed by unification over feature graphs. A program and/or a goal clause is said to have a type τ if it satisfies all the typing constraints summarized in Figure 1. The choice of a family of sets $F_1, \ldots, F_n$ is somewhat arbitrary. This is another reason why moding is more fundamental than typing in concurrent logic programming.

3.2 Linearity Analysis

The purpose of linearity analysis [14] is to distinguish between data structures possibly referenced by two ore more pointers and those referenced by only one pointer. We call the former *shared* data and the latter *nonshared* data. Nonshared data structures can be recycled as soon as they are read by the sole reader (compile-time garbage collection).

Sharing of a data structure is caused by *nonlinear* variables defined as follows: An occurrence of a GHC variable is called a *channel occurrence* unless it is the second or subsequent occurrences in a clause head or an occurrence in a guard. A variable in a program clause or a goal clause is called *linear* if it has two or less channel occurrences in the clause, and *nonlinear* if it may have three or more channel occurrences. Mode analysis guarantees that exactly one of the channel occurrences of a variable is the writer occurrence and all the others are reader occurrences, so communication with a linear variable is one-to-one (or one-to-zero), while communication with a nonlinear variable is one-to-many. Readers are encouraged to see that surprisingly many of the variables in existing concurrent logic programs are linear.

To distinguish between nonshared data and shared data in structural operational semantics, we extend the semantics and attach either of the annotations 1 or ω to every function symbol f occurring in the bodies of program clauses and goal clauses. Suppose a substitution operation $\{v \leftarrow t\}$ is implemented by pointer assignment. Then $f^\omega(\ldots)$ means that the structure $f(\ldots)$ is possibly shared and $f^1(\ldots)$ means that $f(\ldots)$ is never shared. The annotations are managed as follows:

1. Annotations in program clauses and initial goal clauses are given according to how the structures are implemented. For instance, suppose two processes `p` and `q` in a goal clause

```
:- p([1,2,3],X), q([1,2,3],Y).
```

   share a single list `[1,2,3]` in an *ac-*

(BF$_\lambda$) $\lambda(p) = \mathit{shared}$, for a function symbol f^ω occurring at p in B,

(LV$_\lambda$) For a linear variable with two occurrences at p_1 and p_2,
$\forall q \in P_{\mathit{Term}}(m(p_1 q) = \mathit{in} \wedge \lambda(p_1 q) = \mathit{shared} \Rightarrow \lambda(p_2 q) = \mathit{shared})$
(when p_1 is a head path)
$\forall q \in P_{\mathit{Term}}(m(p_1 q) = \mathit{out} \wedge \lambda(p_1 q) = \mathit{shared} \Rightarrow \lambda(p_2 q) = \mathit{shared})$
(when p_1 is a body path),

(NV$_\lambda$) For a nonlinear variable occurring at p (and elsewhere),
$\forall q \in P_{\mathit{Term}}(m(pq) = \mathit{out} \Rightarrow \lambda(pq) = \mathit{shared})$ (when p is a head path)
$\forall q \in P_{\mathit{Term}}(m(pq) = \mathit{in} \Rightarrow \lambda(pq) = \mathit{shared})$ (when p is a body path)

(BU$_\lambda$) For a unification body goal $=_k$,
$\forall q \in P_{\mathit{Term}}(\langle =_k, 1\rangle q = \mathit{shared} \Leftrightarrow \langle =_k, 2\rangle q = \mathit{shared})$

Figure 2: Linearity constraints imposed by a program clause h `:-` G `|` B or a goal clause `:-` B

tual implementation. Then the eight function symbols in the *textual* representation of the goal clause must have the annotation ω. If two separate lists are created at runtime, the annotations can be either 1 or ω. All function symbols occurring in a term with a principal function symbol with ω must have the annotation ω (closure condition).

2. In the extended operational semantics, the annotations are handled as follows. Suppose an assignment $\{v \leftarrow t\}$ takes place in a goal reduction. This happens either by the execution of a unification goal $v = t$ or by the reduction of a non-unification goal $p(\ldots t \ldots)$ using a clause of the form $p(\ldots v \ldots)$ `:-` G `|` B (renamed using fresh variables).

   - When v is nonlinear, all the annotations in t are first changed to ω in order to indicate that multiple readers have (direct or indirect) access to all the subterms in t. Then the other occurrences of v (if any) are replaced by (modified) t.
   - When v is linear, the other occurrence of v (if any) is replaced by t without changes of annotations.

The annotation can be viewed as modeling a 1-bit reference counter [3], though it is to be compiled away.

Linearity of a well-moded GHC program can be characterized by a linearity function $\lambda : P_{\mathit{Atom}} \to \{\mathit{nonshared}, \mathit{shared}\}$. A program clause h `:-` G `|` B or a goal clause `:-` B is said to have a linearity λ if it satisfies all the linearity constraints shown in Figure 2. The linearity constraints can be trivially satisfied by a function that always returns *shared*, but such a linearity function provides no useful information. The purpose of linearity analysis is to find which paths can have the value *nonshared*.

The main result about linearity analysis is the *Subject Reduction Theorem*, which guarantees that when a program $\mathcal{P}$ and a goal clause G satisfies λ and G is reduced to G' using $\mathcal{P}$, then G' also satisfies λ. It follows from this theorem that all data structures occurring at nonshared paths have an annotation 1.

4 Abstract Interpretation

Constraint-based program analysis nicely captures implementation-independent properties of programs, and provides basic information for optimization. However, it cannot capture time-dependent or implementation-dependent properties. An example of such properties is the instantiation state of the arguments of goals. It depends on both when it is observed (possibilities include when it is created, when it starts execution, and when it is finished) and how the goals are scheduled.

The analysis of instantiation states enables a compiler to form a *thread*, namely a sequence of (fine-grain) goals that can be executed sequentially without suspension checking (except upon entry into the thread). The form of a thread we are particularly interested in is a sequence of built-in body goals possibly followed by a tail-recursive call. The analysis can be done using an abstract domain $\{bound, unbound\}$ and proceeds as follows:

1. reorder body goals in each program clause using mode information to form a potential thread, and

2. perform abstract execution of the `main` program (or the top-level program of the program module being analyzed) according to the obtained control flow until the instantiation state of goal arguments reaches a fixpoint.

If the abstract interpretation guarantees the sequential execution of reordered goals to determine all the necessary values of input arguments, we have succeeded in forming a thread and can proceed to loop optimization described in the next section.

5 Loop Optimization

One of the most important applications of constraint-based analysis and abstract interpretation is the optimization of loops programmed as simple or mutual tail recursion.

Since tail-recursive calls usually have more statically available information about their arguments than the initial call to the predicate, it is reasonable to have two entry points for each predicate, one for external calls and the other for the tail-recursive loop. The purpose is to eliminate

1. tag operations and

2. general procedures for accessing generic objects

from the loop and to cache KL1 data using C variables while looping.

Abstract interpretation may not guarantee that all the data examined in clause guards in a loop have been instantiated and fully dereferenced when the loop is entered. In this case, a compiler may insert a synchronization code outside the loop to guarantee that all the data examined inside the loop have concrete values and thus to reduce the number of synchronization operations performed at runtime. It is not always possible to move a synchronization point out of the loop because it may block the execution of unification body goals whose results are otherwise observable from other concurrent processes. However, in many cases we can prove that this won't block the publication of any output data.

6 Number Arrays

Loop optimization is effective for both scalar and vector computation. However, vectors in KLIC are not as efficient as they could be to represent homogeneous arrays of instantiated data (Section 2). To achieve performance competitive with programs written in procedural languages, we have implemented arrays of 64-bit floating point numbers and arrays of 32-bit integers as KLIC's data objects. They are supposed to be used with static analysis and have the following features:

1. All the elements of an array must be of the same type (64-bit floating-point numbers or 32-bit integers).

2. The values to be stored into arrays must be instantiated.

3. Linearity analysis must guarantee that multi-version control can be safely omitted.

4. Allocated on a special area not under management of the garbage collector.

Table 1: Optimization of floating-point numbers and loops

without opt.	with opt.	C
32.6 msec. (12.4)	4.47 msec. (1.70)	2.63 msec. (1.00)

Table 2: Arrays vs. vectors

KLIC vector	double array	double array with loop opt.	C
8.66 sec. (15.0)	6.54 sec. (11.4)	0.772 sec. (1.34)	0.576 sec. (1.00)

This is to avoid copying of large arrays by garbage collection. In an implementation on a shared-memory parallel computer we are currently working on, arrays may be allocated on shared memory.

5. No bound check of index values. It would be more reasonable to separate array bound checks from access operations so that static analysis may eliminate checks that are known to succeed.

6. Split and join operations with no copying. Suppose we want to let processes access and update different parts of an array concurrently and without interference. This can be done by splitting the original array and giving the resultant subarrays to the processes. The subarrays finally returned by the processes can be rejoined without copying, as long as they have been updated in place.

7 Experiments

To demonstrate the effect of our optimization techniques, we took two examples, one to compute $\sum_{k=1}^{10000}(1/k^2)$ and the other to compute the product of two 100×100 matrices, and compared the performance of

- the C code generated by KLIC (version 3.002),
- optimized, hand-compiled intermediate code (in C) we have designed, and
- programs directly written in C.

The results are shown in Tables 1 and 2. The measurements were done using Sun Ultra Enterprise 4000 (MPU: 168MHz), and the numbers shown are the execution times of the main loops. All the C programs were compiled using `gcc -O2 -fomit-frame-pointer`. The results show that loop optimization and array types were both effective, and the optimized KL1 programs turned out to be only 34%–70% slower than the comparable C programs. One of the remaining sources of overhead is the polling of external events in each iteration, without which a thread executing a tight loop might run indefinitely.

8 Conclusions

We have shown that static analysis can make the performance of KLIC, an implementation of a "pure" concurrent logic language, quite close to C for numerical computation. The most important future work is to build an optimizing KLIC compiler that makes use of the output of the static analyzer. Another important direction is to extend our array objects to allow parallel processing on shared-memory parallel computers.

A key feature of concurrent logic languages is (and should be) that parallelization can be achieved with very low additional programming effort. We hope our research will open up a new approach to high-performance computing and new application of concurrent logic programming.

References

[1] Aït-Kaci, H. and Nasr, R., LOGIN: A Logic Programming Language with Built-In Inheritance. *J. Logic Programming*, Vol. 3, No. 3 (1986), pp. 185–215.

[2] Ajiro, Y., Ueda, K. and Cho, K., Error-correcting Source Code. To be presented at the *Fourth Int. Conf. on Principles and Practice of Constraint Programming (CP98)*, Pisa, Italy, October 1998.

[3] Chikayama, T. and Kimura, Y., Multiple Reference Management in Flat GHC. In *Logic Programming: Proc. of the Fourth Int. Conf (ICLP'87)*, The MIT Press, 1987, pp. 276–293.

[4] Chikayama, T., Fujise, T., and Sekita, D., A Portable and Efficient Implementation of KL1. In *Proc. PLILP'94*, LNCS 844, Springer, 1994, pp. 25–39.

[5] Cho, K. and Ueda, K., Diagnosing Non-Well-Moded Concurrent Logic Programs. In *Proc. 1996 Joint Int. Conf. and Symp. on Logic Programming (JICSLP'96)*, The MIT Press, 1996, pp. 215–229.

[6] Eriksson, L.-H. and Rayner, M., Incorporating Mutable Arrays into Logic Programming. In *Proc. Second Int. Logic Programming Conf.*, Uppsala Univ., Sweden, 1984, pp. 101–114.

[7] Saraswat, V. A. and Rinard, M., Concurrent Constraint Programming (Extended Abstract). In *Conf. Record of the Seventeenth Annual ACM Symp. on Principles of Programming Languages*, ACM, 1990, pp. 232–245.

[8] Ueda, K. and Chikayama, T., Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (1990), pp. 494–500.

[9] Ueda, K. and Morita, M., Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, Vol. 13, No. 1 (1994), pp. 3–43.

[10] Ueda, K., I/O Mode Analysis in Concurrent Logic Programming. In *Proc. Int. Workshop on Theory and Practice of Parallel Programming*, LNCS 907, Springer, 1995, pp. 356–368.

[11] Ueda, K., Experiences with Strong Moding in Concurrent Logic/Constraint Programming. In *Proc. Int. Workshop on Parallel Symbolic Languages and Systems*, LNCS 1068, Springer, 1996, pp. 134–153.

[12] Ueda, K. and Cho, K. *kima* — Analyzer of Ill-moded KL1 Programs. Available from `http://www.icot.or.jp/AITEC/FGCS/funding/itaku-H8-index-E.html`, 1997.

[13] Ueda, K., *klint* — Static Analyzer for KL1 Programs. Available from `http://www.icot.or.jp/AITEC/FGCS/funding/itaku-H9-index-E.html`, 1998.

[14] Ueda, K., Linearity Analysis of Concurrent Logic Programs. Presented at the 11th Annual Summer United Workshops on Parallel, Distributed and Cooperative Processing (SWoPP'98), IPSJ SIGPRO, August 1998. Full paper in preparation.