A Pure Meta-Interpreter for Flat GHC, A Concurrent Constraint Language

Kazunori Ueda

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

Abstract. This paper discusses the construction of a meta-interpreter of Flat GHC, one of the simplest and earliest concurrent constraint languages.
Meta-interpretation has a long history in logic programming, and has been applied extensively to building programming systems, adding functionalities, modifying operational semantics and evaluation strategies, and so on. Our objective, in contrast, is to design the pair of (i) a representation of programs suitable for code mobility and (ii) a pure interpreter (or virtual machine) of the represented code, bearing networked applications of concurrent constraint programming in mind. This is more challenging than it might seem; indeed, meta-interpreters of many programming languages achieved their objectives by adding small primitives into the languages and exploiting their functionalities. A meta-interpreter in a pure, simple concurrent language is useful because it is fully amenable to theoretical support including partial evaluation.
After a number of trials and errors, we have arrived at *treecode*, a ground-term representation of Flat GHC programs that can be easily interpreted, transmitted over the network, and converted back to the original syntax. The paper describes how the interpreter works, where the subtleties lie, and what its design implies. It also describes how the interpreter, given the treecode of a program, is partially evaluated to the original program by the unfold/fold transformation system for Flat GHC.

1 Introduction

1.1 Meta-Interpreter Technology

Meta-interpreter technology has enjoyed excellent affinity to logic programming since the seminal work by Bowen and Kowalski [5]. It provides us with a concise way of building programming systems on top of another. This is particularly useful for AI applications in which flexibility in designing and modifying inference mechanisms is of crucial importance. Interactive programming environments such as debuggers or visualizers are another example in which interpreters can play important rôles. Extensive survey of meta-interpretation in logic programming can be found in [11], Chapter 8.

Critics complain of performance degradation incurred by the interpreter technology, but the speed of system prototyping with interpreters and symbolic languages cannot be matched by any other methodologies. Hardwiring all design choices into a lower-level language such as C may be done, but at the latest possible stage and to the least extent. Indeed, due to Java and scripting languages, interpreter technologies – including bytecode interpreters and its optimization techniques such as just-in-time compilers – are now quite ubiquitous outside the world of symbolic languages. Java demonstrated that poor initial performance of non-optimized interpreters was acceptable once people believed that the language and the system design as a whole were the right way to go.

1.2 Concurrency and Logic Programming

The *raison d'être* and the challenge of symbolic languages are to construct highly sophisticated software which would be too complicated or unmanageable if written in other languages. Logic programming has found and addressed a number of such fields [4]. While many of those fields such as databases, constraints, machine learning, natural languages, etc., are more or less related to Artificial Intelligence, concurrency seems special in the sense that, although somewhat related to AI through agent technologies, its principal connection is to distributed and parallel computing.

Distributed and parallel computing is becoming extremely important because virtually all computers in the world are going to be interconnected. However, we have not yet agreed upon a standard formalism or a standard language to deal with concurrency. Due to the lack of appropriate tools with which to develop networked applications, computers communicate and cooperate much more poorly than they possibly can.

Concurrent logic programming was born in early 1980's from the process interpretation of logic programs [34]. Relational Language [7], the first concrete proposal of a concurrent logic language, was followed by a succession of proposals, namely Concurrent Prolog [20], PARLOG [8] and Guarded Horn Clauses (GHC) [27]. KL1 [29], the Kernel Language of the Fifth Generation Computer Systems (FGCS) project [22], was designed based on GHC by featuring (among others) *mapping* constructs for concurrent processes. To be precise, KL1 is based on Flat GHC [28], a subset of GHC that restricts guard goals to calls to test predicates.

The mathematical theory of these languages came later in the generalized setting of concurrent constraint programming (CCP) [18] based on Maher's logical interpretation of synchronization [12]. Grand challenges of concurrent logic/constraint programming are proposed in [32].

Although not as widely recognized as it used to be, Concurrent Prolog was the first simple high-level language that featured channel mobility exactly in the sense of π-calculus [15]. When the author proposed GHC as an alternative to Concurrent Prolog and PARLOG, the principal design guideline was to retain channel mobility and evolving process structures [22], because GHC was supposed to be the basis of KL1, a language in which to describe operating systems of Parallel Inference Machines as well as various knowledge-based systems. The

readers are referred to [22] for various researchers' personal perspectives of the FGCS project.

1.3 Meta-Interpretation and Concurrency

Another guideline of the design of GHC was the ability to describe its own meta-interpreter. Use of simple meta-interpreters as a core technology of system development was inspired by [5], and early work on Concurrent Prolog pursued this idea in building logic-based operating systems [21].

A key technology accompanying meta-interpretation turned out to be partial evaluation. Partial evaluation of a meta-interpreter with an additional "flavor" with respect to a user program will result in a user program with the additional "flavor" that runs almost as efficiently as the original user program [24].

This idea, though very elegant, has not become as popular as we had expected.

One reason is that before the booming of the Internet, a program ran either on a single processor or on parallel processors with a more or less uniform structure, where a hardwired approach was manageable and worked. However, software for distributed computing environments is much harder to build, configure and reconfigure, and run persistently. Such software would not be manageable without a coherent solution to the difficulties incurred by heterogeneous architectures, process and code mobility, and persistence.

Another reason is that the languages and the underlying theories were not mature enough to allow full development of the idea. Meta-interpreters of many programming languages achieved their objectives by adding small primitives into the language and exploiting their functionalities. Those primitives were often beyond the basic computational models of the languages. We believe that *pure* symbolic languages are the right way to go in the long run, because only with theoretical support we can expect a real breakthrough.

1.4 Goal of This Paper

In this paper, we discuss how we can construct a meta-interpreter of Flat GHC, one of the simplest and earliest concurrent constraint languages. Our objective is to design the pair of

1. a representation of programs suitable for code mobility and interpretation, and
2. a pure, simple interpreter of the represented code.

One of the motivations of the work is to use concurrent logic/constraint programming as a concise tool for networked applications. There are strong reasons to choose concurrent logic/constraint programming as a framework of distributed computing.

First, it features channel mobility, evolving process structures, and incomplete messages (messages with reply boxes), all essential for object-based concurrent programming.

Second, it is unlike most other concurrency frameworks in that data structures (lists, trees, arrays, etc.) come from the very beginning. This means that there is little gap between a theoretical model and a practical language. Actually, a lot of applications have been written in concurrent logic/constraint languages, notably in KL1 and Oz [23].

Third, it has been extremely stable for more than 15 years. After GHC was proposed, the main variation was whether to feature *atomic tell* (publication of bindings upon commitment) or *eventual tell* (publication after commitment). However, by now both concurrent logic programming and concurrent constraint programming seem to converge on *eventual tell*, the simpler alternative [22][26]. Indeed, concurrent constraint programming with *ask* and *eventual tell* can be thought of as an abstract model of Flat GHC.

Last, as opposed to other parallel programming languages, it achieves clear separation of concurrency (concerned with logical aspects of programs) and parallelism (concerned with physical mapping of processes). We regard this separation of concerns as the most important achievement of KL1 and its parallel implementation [29]. In other words, by using logical variables as communication channels we had achieved 100% network transparency within system-area networks (SAN). The fact that programs developed and tested on sequential machines ran at least correctly on parallel machines has benefited us enormously in the development of parallel software. We believe that this feature should be explored in distributed software as well.

Addressing networked applications using interpreters as a core technology is promising because flexibility to cope with heterogeneity is more important than performance. However, it is not obvious whether we can write a reasonably simple interpreter in a pure concurrent logic/constraint language such as Flat GHC. A meta-interpreter in a pure, simple concurrent language is fully amenable to theoretical support including partial evaluation and verification. Also, it can help *analytic approach* to language design [32], because meta-interpretation is considered an acid test of the expressive power of the language. The rôle of an interpreter technology in networked applications should be clear since an interpreter is just another name of a virtual machine.

2 Previous Work

Meta-interpreters of symbolic languages date back to a Lisp interpreter in Lisp around 1960 [13]. Prolog interpreters in Prolog were available and widely used in 1970's; an example is the interpreter of the *de facto* standard DEC-10 Prolog.

Meta-interpreters of Concurrent Prolog can be found in various papers. Figure 1 shows two versions, the first one in [20] and the second in [17].

Program (a) is very similar to a Prolog interpreter in Prolog, but it relies on the "large" built-in primitive, `clause`/2 (`clause` with two arguments), that performs synchronization, evaluation of clause guards, and committed choice. The only thing reified by the interpreter is parallel conjunction. Program (b) takes both a program and a goal as arguments, and reifies the unification of the

```
reduce(true).
reduce((A,B)) :- reduce(A?), reduce(B?).
reduce(A) :- A\=true, A\=(_,_) | clause(A?,B), reduce(B?).
```

(a) Without a program argument

```
reduce(Program,true).
reduce(Program,(A,B)) :-
    reduce(Program?, A?), reduce(Program?, B?).
reduce(Program,Goal) :-
    Goal\=true, Goal\=(A,B),
    clause(Goal?,Program?,Body) |
    reduce(Program?,Body?).
clause(Goal,[C|Cs],B) :-
    new_copy(C?,(H,G,B)), Goal=H, G | true.
clause(Goal,[C|Cs],B) :-
    clause(Goal,Cs?,B) | true.
```

(b) With an explicit program argument

Fig. 1. Meta-Interpreters of Concurrent Prolog

goal with clause heads and the evaluation of guards. Note, however, that most of the important operations are called from and performed in clause guards. In particular, `clause`/3 calls itself recursively from within a clause guard, forming a *nested (or deep) guard*.

While Concurrent Prolog employed read-only annotations as a synchronization primitive, GHC replaced it with the rule that no bindings (constraints) can be published from the guard (including the head) of a clause to the caller of the clause.

Figure 2 shows a GHC interpreter in GHC in [27]. Here it is assumed that a built-in predicate `clauses`/2 returns in a *frozen* form [16] a list of all clauses whose heads are potentially unifiable with the given goal. Each frozen clause is a ground term in which original variables are indicated by special constant symbols, and it is *melted* in the guard of the first clause of `resolve`/3 by `melt_new`/2. The goal `melt_new(C, (A :- G|B2))` creates a new term (say T) from a frozen term C by giving a new variable for each frozen variable in C, and tries to unify T with `(A :- G|B2)`. However, this unification cannot instantiate `A` because it occurs in the head of `resolve`/3.

The predicate `resolve`/3 tests the candidate clauses and returns the body of arbitrary one of the clauses whose guards have been successfully solved. This many-to-one arbitration is realized by the multi-level binary clause selection using the nested guard of the predicate `resolve`/3. It is essential that each candidate clause is melted after it has been brought into the guard of the first clause of `resolve`/3. If it were melted before passed into the guard, all variables

```
call(true  ) :- true | true.
call((A, B)) :- true | call(A), call(B).
call(A     ) :- clauses(A, Clauses) |
    resolve(A, Clauses, Body), call(Body).

resolve(A, [C|Cs], B) :- melt_new(C, (A :- G|B2)), call(G) | B=B2.
resolve(A, [C|Cs], B) :- resolve(A, Cs, B2) | B=B2.
```

Fig. 2. Meta-Interpreter of GHC

in it would be protected against instantiation from the guard. We must protect variables accessible from outside but allow local variables to be instantiated.

Again, this GHC meta-interpreter calls `resolve`/3 from within a guard recursively. However, our lesson is that, except for meta-interpreters, we can dispense with general nested guards. To put it more precisely, we can dispense with guard goals that may instantiate local variables; restricting guard goals to calls to *test* predicates is a more realistic choice. Test predicates are predicates defined in terms of clauses with no body goals. A nice property of test predicates is that they deterministically succeed or fail depending on their arguments. They are regarded as specifying conditions, as opposed to predicates for specifying concurrent processes. Test predicates defined using guarded clauses may call themselves recursively from guards, but unlike general nested guards, there is no need to maintain multiple layers of variable protection to implement synchronization. In this sense, languages with restriction to test predicates have been called *flat* languages. In most implementations of flat languages, test predicates are further restricted to predefined ones.

Later development of concurrent logic languages can be phrased as *devolution as evolution* [26][32] in the sense that it focused on high-performance, compiler-based implementation of flat languages. Strand [9], KL1 and Janus [19] all belong to this category. Accordingly, there was less work on meta-interpreters for the last 10 years. Huntbach [11] shows a meta-interpreter that implements *ask* using `match`/2, a special primitive discussed in detail in Sect. 3.3. Although using `match`/2 to implement *ask* is a natural idea, `match`/2 turns out to have properties not enjoyed by other goals definable in concurrent logic languages. This motivated us to design a meta-interpreter that does not use `match`/2.

Distributed computing based on concurrent constraint programming is not a new idea. The Oz group has done a lot of work in this direction [10]. However, code mobility in Oz is based on bytecode technology, and Oz has added to CCP a number of new constructs including ports (for many-to-one communication), cells (value containers that allow destructive update), computation space (encapsulated store, somewhat affected by nested guards of full GHC and KL1's *shoen*), and higher-order. This is in sharp contrast with the minimalist approach taken in this paper.

3 The Problem Statement

Now let us state the goal and the constraints of our problem precisely. Our goal is to design a binary Flat GHC predicate, say `exec`, that

- takes
  1. a multiset G of goals (represented as a list) to be executed and
  2. a ground representation of the program P to execute G, and
- behaves exactly like G running under the ordinary compiled code for P.

The predicate `exec`/2 is sometimes called a *universal* predicate because it can be tailored, at run time, to whatever predicate you like.

The only built-in primitives the `exec`/2 program is allowed to use are those definable using (a possible infinite number of) guarded clauses. Other primitives are considered extralogical and are ruled out. Observing this constraint will enable the resulting interpreter to run on KLIC [6], which is in our context considered as a (Flat) GHC-to-C compiler and its runtime system. Flat GHC and KLIC carefully rule out extralogical built-in primitives because they can potentially hamper efficient implementation and theoretical support.

A solution to the problem is not obvious because Flat GHC and KLIC do not have general nested guards, on which the interpreter of full GHC in Sect. 2 depends in a fundamental way.

Some remarks and discussions on our requirements are in order, which are (1) representation of code, (2) representation of runtime configuration, and (3) primitives for *ask* (matching) and *tell* (unification).

3.1 Representation of Code

Meta-interpreters vary in the representation of programs. Some retrieve programs from the internal database using primitives like `clause`/2. This is not suited to our goal of code mobility and persistence. Some use a list of clauses in which variables are represented using variables at the level of the interpreters. This is considered misuse of variables, as criticized by later work on meta-programming, because those variables are improperly scoped and awkward to handle. One solution is to use a higher-order construct as in Lambda Prolog [14], and another solution is to come up with a ground representation of variables. Although the higher-order approach gives us the most natural solution, the difference between the two solutions is not large when the programs to be represented have no nested scope, which is the case with Prolog and Flat GHC.

As we will see later, we have chosen to represent a variable in terms of a reserved unary constructor with an integer argument. This could be viewed as a de Bruijn notation as well.

3.2 Representation of Runtime Configuration

In a rule-based language where programs (rewrite rules) are given separately from expressions (goals), how to represent runtime configurations and how to

represent the programs are independent issues. The two alternatives for the representation of runtime configurations are

1. to reify logical variables and substitutions and handle them explicitly, and
2. not to reify them but use those at the interpreter level.

We adopt the latter, because

- an interpreted process must be *open-ended*, that is, it must be able to communicate with other native processes running in parallel with the interpreter,
- the reification approach would therefore require 'up' and 'down' predicates to move between the two levels of representation and (accordingly) a full-fledged meta-programming framework in the language, and
- explicit representation can cause performance degradation unless elaborate optimization is made.

3.3 Primitives for Matching/Ask and Unification/Tell

In the CCP terminology, Prolog and constraint logic languages in their basic forms are *tell*-only languages because unification or constraint solving is the attempt to publish bindings (constraints) to the binding environment (constraint store). In contrast, concurrent logic/constraint languages are *ask*+ *tell* languages which additionally feature matching (in algebraic terms) or the asking of whether a given constraint is entailed (in logical terms) by the current store. So how to implement *ask* and *tell* in an interpreter is a key design issue.

The Prolog and GHC versions of *tell* are unification over finite trees and can be written as $\texttt{unify}(G, H)$ or $G = H$. This has the following properties:

1. *Immediate* — It either succeeds or fails and does not suspend.
2. *Monotonic* — Its success/failure can depend on the current store; that is, $\texttt{unify}(G, H)$ that succeeds under some store can fail under a store augmented with additional constraints. However, if we consider failure as a over-constrained store, $\texttt{unify}(G, H)$ can be thought of as an operator that monotonically augments the current store.
3. *Deterministic* — The conjunction of all *tell*s generated in the course of program execution deterministically defines the current store.

Now we consider the properties of *ask*, which appears in concurrent logic languages as matching between a goal and a clause head. Let σ be the current store under which the *ask* is performed. We suppose $\texttt{match}(G, H)$

- *succeeds* when there exists a substitution θ such that $G\sigma = H\sigma\theta$,
- *suspends* when there is no such θ but $G\sigma$ and $H\sigma$ are unifiable, and
- *fails* when $G\sigma$ and $H\sigma$ are non-unifiable.

Clearly, $\texttt{match}(G, H)$ is not immediate. Furthermore, it is neither monotonic nor deterministic with respect to suspension behavior:

- `match(X,Y)` will succeed when `Y` is uninstantiated but may suspend when `Y` is instantiated. This behavior is opposite to that of ordinary CCP processes which can never be suspended by providing more constraints.
- `match(X,Y)` $\wedge$ `match(3,Y)` under the empty store succeeds if executed from left to right but suspends if executed from right to left.

When simulating matching between a goal G and a clause head H using `match`/2, H must have been renamed using fresh variables, and H is therefore immune to σ. If this convention is enforced, `match`/2 enjoys monotonicity, that is, if `match`/2 succeeds under σ, it succeeds under $\sigma\sigma'$ for any σ'. The convention guarantees determinism as well.

The lesson here is that the scope of the variables in H, the second argument of `match`/2, should be handled properly for `match`/2 to enjoy reasonable properties. As suggested by [12], the proper semantics of `match`(G, H) would be whether σ interpreted as an equality theory *implies* $G = \exists H$. Thus the second argument should specify an existential closure $\exists H$ rather than H. However, then, the second argument would lose the capability to *receive* matching terms from G. For instance, the recursive clause of `append`/3 in GHC is

```
append([A|X],Y,Z0) :- true | Z0=[A|Z], append(X,Y,Z).
```

while the CCP version of the above clause would be less structured:

```
append(X0,Y,Z0) :- ask(∃A,X(X0=[A|X])) |
    tell(X0=[A|X]), tell(Z0=[A|Z]), append(X,Y,Z).
```

To summarize, while implementing *tell* in an interpreter is straightforward, implementing *ask* without introducing new primitives is a major design issue.

4 A Treecode Representation

In this section, we discuss the design of our treecode representation of Flat GHC programs, which is interpreted by the treecode interpreter described in the Sect. 5.

4.1 Treecode

Treecode is intermediate code in the form of a first-order ground term which is quite close to the original source code. It is more abstract and "structured" than ordinary bytecode sequences that use forward branching to represent `if`...`then`...`else`. Trees are much more versatile than sequences and are much easier to represent and handle than directed graphs. Indeed, the booming of XML tells us that standard representation of tagged trees has been long-awaited by a great number of applications, and XML trees are little more than first-order ground terms.

Of course, the control flow of a program forms a directed graph in general and we must represent it somehow. Directed graphs could be created rather

easily by unification over rational terms, but we chose to dispense with circular structures by representing recursive calls (that form circularity) using explicit predicate names. When the interpreter encounters a predicate call, it obtains the code for the predicate using an appropriate lookup method. An optimizing interpreter may create a directed graph by "instantiating" each predicate call to its code before starting interpretation.

An alternative representation closer to source code is a set of rewrite rules. However, it turns out that a set (represented as a list) of rewrite rules is less suitable for interpretation. This is because GHC "bundles" predicate calls, synchronization and choice in a single construct, namely guarded clauses. While this bundling simplifies the syntax and the semantics of Flat GHC and captures the essence of concurrent logic programming, guards – even flat guards – can specify arbitrary complex conditions that may involve both conjunctive and disjunctive sets of multiple synchronization points. Programmers also find it sometimes cumbersome to describe everything using guarded clauses exactly for the reason why Prolog programmers find that the `(P -> Q ; R)` construct sometimes shortens their programs considerably.

As we will see soon, treecode still looks like a set of clauses, but the major difference from a set of clauses is that the former breaks a set of guards down to a tree of one-at-a-time conditional branching. In this sense, treecode can be regarded as *structured intermediate code*.

4.2 Treecode By Example

Now we are in a position to explain how treecode looks like. Throughout this section we use `append`/3 as an example. The treecode for `append`/3 is:

```
treecode(6,
  [c(1=[], b([<(2)= <(3)],[])),
   c(1=[>(4)|>(5)],
           b([<(3)=[<(4)|>(6)]],[append(5,2,6)]))])
```

The first argument, `6`, stands for the number of variables used in the treecode, and the second argument is the main part of the treecode.

The readers may be able to guess what it does basically, since it is quite similar to the original source code:

```
append(X, Y,Z ) :- X=[] | Y=Z.
append(X0,Y,Z0) :- X0=[A|X] | Z0=[A|Z], append(X,Y,Z).
```

In this simple example, the treecode still looks like a list of clauses, with heads (with mutually disjoint variables) omitted and variables represented by positive integers. The constructor `c`/2 forms a case branch by taking an *ask* and another treecode as arguments. The list of case branches forms a *casecode*.

The constructor `b`/2 forms a *bodycode* by taking a list of *tell*s and a list of calls to user-defined predicates. The former is understood by the interpreter, while the latter involves code lookup.

A *treecode* is either a casecode or a bodycode. Figure 3 shows the syntax of treecode.

⟨*treecode*⟩ ::= ⟨*casecode*⟩ | ⟨*bodycode*⟩
⟨*casecode*⟩ ::= list of ⟨*choice*⟩'s
⟨*choice*⟩ ::= `c(`⟨*ask*⟩`,` ⟨*treecode*⟩`)`
⟨*ask*⟩ ::= ⟨*reg*⟩ `=` ⟨*term*⟩ | ⟨*reg*⟩⟨*relop*⟩⟨*term*⟩
⟨*bodycode*⟩ ::= `b(`⟨*tells*⟩`,` ⟨*goals*⟩`)`
⟨*tells*⟩ ::= list of ⟨*tell*⟩'s
⟨*tell*⟩ ::= ⟨*annotatedreg*⟩ `=` ⟨*term*⟩ | ⟨*annotatedreg*⟩ `:=` ⟨*term*⟩
⟨*goals*⟩ ::= list of ⟨*goal*⟩'s
⟨*goal*⟩ ::= ⟨*pred*⟩`(`⟨*reg*⟩`,...)`
⟨*annotatedreg*⟩ ::= [⟨*annotation*⟩]⟨*reg*⟩
⟨*annotation*⟩ ::= `<` | `<<` | `>`
⟨*reg*⟩ ::= `1` | `2` | `3` | ...
⟨*term*⟩ ::= ⟨*functor*⟩`(`⟨*annotatedreg*⟩`, ...)`
⟨*relop*⟩ ::= `>` | `<` | `>=` | `=<` | `=:=` | `=\=`

Fig. 3. Syntax of Treecode

4.3 Representing and Managing Logical Variables

The unary constructors '`<`' and '`>`' have two purposes. First, they distinguish integer representation of variables from integer constants in the program to be interpreted. Second, they tell whether a variable has occurred before and whether it will occur later. *Initial mode*, denoted, '`>`', means the creation of a new variable, while *final mode*, denoted '`<`', means the final access to an already created variable. In `append`/3, each variable occurs exactly twice, which means that all accesses are either initial or final accesses. For variables that are read more than once, we use another reserved unary constructor, '`<<`', to indicate that they are accessed in *intermediate mode*, that is, they are neither the first nor the last occurrences.

The first occurrence of a variable in each case branch (`1` in the case of `append`/3) and the arguments of user-defined predicates are supposed to be final-mode. These are the only places where mode annotations are omitted for ease of interpretation.

Representing variables by positive integers suggests the use of arrays to represent them. We use a constructor `g`/n to represent goal records, where n is the number of variables in the treecode that works on the goal. The structure `g`/n can be regarded as a *register vector* as well.

Let a be the arity of the predicate represented by the treecode. The first ath arguments of `g`/n are the arguments of the original goal, while the remaining arguments are local variables of the original goal. Thus this structure can be regarded both (i) as a concretization of goals that makes housekeeping explicit and (ii) as an abstraction of implementation-level goal records. When the structure is created, the first ath arguments are initialized to the arguments of the original goal, while the remaining arguments are initialized to the constant `0`. The value of a is not recorded in the treecode itself. It is the responsibility of

the predicate `try`/3 to "apply" treecode to a goal record, as will be described in Sect. 5.

The distinction between initial, intermediate and final modes not only makes interpretation easier but also allows the reuse of the same register for different variables. For example, the code for `append`/3 could be written alternatively as:

```
[c(1=[], b([<(2)= <(3)],[])),
 c(1=[>(4)|>(1)],
         b([<(3)=[<(4)|>(3)]],[append(1,2,3)]))]
```

because

- Variable 1 in the second branch, holding the first argument of the caller, will not be accessed after its principal constructor has been known, and
- Variable 3 in the second branch, holding the third argument of the caller, will not be accessed after it has been instantiated to an non-empty list.

This is register allocation optimization which is optional in our treecode. Without it, different numbers represent different single-assignment variables and the code is more declarative. With it, the size of goal records can be reduced.

5 Structure of the Treecode Interpreter

This section describes, step by step, how our treecode interpreter works on a goal record. We focus on basic *ask* and *tell* operations. The actual interpreter handles arithmetic built-in predicates for comparison (guard) and assignment (body), but it is straightforward to include them.

The two main predicates of the interpreter are `exec`/2 and `try`/3. The predicate `exec`/2 takes a multiset G of goals and a program $\mathcal{E}$ for executing them. We call the program an *environment* because it associates each predicate name with its treecode. The goal `exec`$(G, \mathcal{E})$ resolves predicate names in G into their corresponding treecode, and invokes `try`/3 for each goal in G after preparing a goal record for the goal. The predicate `try`/3 takes a goal record, a treecode and an environment, and applies the treecode to the goal record. The more interesting aspects of the interpreter lie in `try`/3.

5.1 Deterministic and Nondeterministic Choice

When the treecode given to `try`/3 is *casecode*, it deterministically chooses one branch as follows: It picks up the first case branch of the form `c`(*Ask*, *Treecode*), where *Ask* is of the form $n = T$. This causes the interpreter to wait for the principal constructor of the nth argument, and when it is available, it is matched against the constructor of T. The n's in each case branch must be identical; thus casecode has exactly one synchronization point for all its top-level *ask*s and is therefore deterministic.

When some guard involves the asking of more than one symbol, it is compiled into nested casecode. For instance, the program

```
part(_,[],    S, L ) :- true | S=[], L=[].
part(A,[X|Xs],S0,L ) :- A>=X | S0=[X|S], part(A,Xs,S,L).
part(A,[X|Xs],S, L0) :- A< X | L0=[X|L], part(A,Xs,S,L).
```

can be compiled into:

```
[c(2=[], b([<(3)=[],<(4)=[]],[])),
 c(2=[>(5)|>(2)],
   [c(1>= <<(5), b([<(3)=[<(5)|>(3)]],[part(1,2,3,4)])),
    c(1<  <<(5), b([<(4)=[<(5)|>(4)]],[part(1,2,3,4)]))])]
```

Note that the matching of the second argument with `[X|Xs]` has been factored, as would be done by an optimizing compiler.

Nested casecode is still deterministic because it has at most one synchronization point (i.e., the variable on whose value the interpreter suspends) at any time. Our experience with Flat GHC/KL1 programming has shown that the majority of predicates are deterministic.

Nondeterministic predicates are those which contain *disjunctive* wait, namely wait for the instantiation of one of several variables. Some of the predicates people write are nondeterministic, but most of them involve binary choice only. For instance, the following stream merging program

```
merge([],Ys,Zs) :- true | Zs=Ys.
merge(Xs,[],Zs) :- true | Zs=Xs.
merge([X|Xs],Ys,Zs0) :- true | Zs0=[X|Zs], merge(Xs,Ys,Zs).
merge(Xs,[Y|Ys],Zs0) :- true | Zs0=[Y|Zs], merge(Xs,Ys,Zs).
```

has two disjunctive synchronization points, namely the principal constructor of the first argument and the principal constructor of the second argument.

In this paper we focus on binary nondeterministic choice, which is simpler to implement than general multiway choice. It can be expressed in terms of two nondeterministic branches in the interpreter. By extending our treecode in Fig. 3, the treecode for `merge`/3 can be written as follows:

```
treecode(4,
  (1->[c(1=[], b([<(2)= <(3)],[])),
       c(1=[>(4)|>(1)], b([<(3)=[<(4)|>(3)]],[merge(1,2,3)]))])
+ (2->[c(2=[], b([<(1)= <(3)],[])),
       c(2=[>(4)|>(2)], b([<(3)=[<(4)|>(3)]],[merge(1,2,3)]))]))
```

The extended syntax of treecode is:

$$\langle treecode \rangle ::= \langle casecode \rangle \mid \langle bodycode \rangle \mid \langle nondeterministiccode \rangle$$
$$\langle nondeterministiccode \rangle ::= \texttt{(}\langle reg \rangle \texttt{ -> } \langle treecode \rangle\texttt{) + (}\langle reg \rangle \texttt{ -> } \langle treecode \rangle\texttt{)}$$

where the form $(n_1 \rightarrow treecode_1) + (n_2 \rightarrow treecode_2)$ causes the goal to wait disjunctively upon variables n_1 and n_2.

5.2 Interpreting Casecode

The *ask* part of a casecode of the form $n = T$, where T is a non-variable term whose arguments are all *annotatedreg*s, is interpreted by the following piece of code:

```
try_one(A0,Rn=T,B,Cs,Env) :- true |
   setarg(Rn,A0,A0Rn,ARn,A), functor(A0Rn,A0RnF,A0RnN),
   functor(T,TF,TN), test_pf(A0RnF,A0RnN,TF,TN,Res),
   try_match(Res,T,A0Rn,ARn,A,B,Cs,Env).

test_pf(F1,A1,F2,A2,Res) :- F1=F2, A1=:=A2 | Res=yes(A1).
otherwise.
test_pf(F1,A1,F2,A2,Res) :- true | Res=no.

try_match(yes(N),T,A0Rn,ARn,A0,B,Cs,Env) :- true |
   ARn=0, getargs(1,N,T,A0Rn,A0,A), try(A,B,Env).
try_match(no,    T,A0Rn,ARn,A, B,Cs,Env) :- true |
   ARn=A0Rn, try(A,Cs,Env).

getargs(K,N,T,A0Rn,A0,A) :- K> N | A0=A.
getargs(K,N,T,A0Rn,A0,A) :- K=<N |
   arg(K,T,Tk), setarg(K,A0Rn,A0Rnk,0,A0Rn1),
   getputreg(Tk,A0,A0Rnk,A1),
   K1:=K+1, getargs(K1,N,T,A0Rn1,A1,A).

getputreg(<(Rk), A0,ARk,A) :- true | setarg(Rk,A0,ARk,0,A).
getputreg(<<(Rk),A0,ARk,A) :- true | setarg(Rk,A0,ARk,ARk,A).
getputreg(>(Rk), A0,ARk,A) :- true | setarg(Rk,A0,_,ARk,A).
```

This is almost a Prolog program with a cut in every clause. KL1's built-in predicate, `setarg`(I, T, X, X', T'), is like Prolog's `arg`(I, T, X) except that T' is bound to T with its Ith element replaced by X'. This is a declarative array update primitive and used extensively in the interpreter to read data from, and write data to, goal records.

The `try_one`/5 program first retrieves the `Rn`th variable in the goal record `A0`, binding it to `A0Rn`. Then it checks if `A0Rn` is instantiated and its principal constructor matches that of `T`, using `functor`/3 and `test_pf`/5. If the matching succeeds, the first clause of `try_match`/8 stores (by using `getargs`/6) the top-level arguments of `A0Rn` to the goal record `A0` according to the prescription template `T`. Then it executes the bodycode `B` under the updated goal record `A` and the environment `Env`. The first goal `AR0=0` binds the `Rn`th element in `A` to `0`; this is to explicitly discharge a pointer from the goal record to the top-level structure that has just been *ask*ed. The interpreter uses the constant `0` as a filler when some element of a goal record does not contain a meaningful value, that is, before a meaningful value is loaded or after a meaningful value is taken away.

5.3 Interpreting Bodycode

Bodycode performs *tell*s and the spawning of user-defined body goals:

```
try(A0,b(BU,BN),Env) :- true | tell(A0,BU,A), spawn(A,BN,Env).
```

The *tell*s are not only to instantiate variables passed from the caller; it is also used to prepare non-variable terms to be passed to user-defined body goals, and to unify two variables to create a shared variable between two body goals. How `tell`/3 manipulates data is quite similar to how `getargs`/6 gets data from a non-variable goal argument. A *tell* of the form $n = T$ manipulates the nth element of the goal record according to the template T:

```
tell(A0,[(Rn=T)|BU], A) :- true |
   getputreg(Rn,A0,A0Rn,A1), tell_one(T,A0Rn,A1,BU,A).
tell(A0,[],          A) :- true | A=A0.

tell_one(<(Rk),  A0Rn,A1,BU,A) :- true |
   getputreg(<(Rk),A1,A0Rn,A2), tell(A2,BU,A). /* load Rk */
tell_one(>(Rk),  A0Rn,A1,BU,A) :- true |
   getputreg(>(Rk),A1,A0Rn,A2), tell(A2,BU,A). /* store Rk */
tell_one(T,      A0Rn,A1,BU,A) :- integer(T) |
   A0Rn=T,  tell(A1,BU,A).
otherwise.
tell_one(T,      A0Rn,A1,BU,A) :- true |
   functor(T,F,N), new_functor(A0Rn0,F,N),
   putargs(1,N,T,A0Rn0,A0Rn,A1,A2), tell(A2,BU,A).

putargs(K,N,T,A0Rn0,A0Rn,A0,A) :- K> N | A0Rn0=A0Rn, A0=A.
putargs(K,N,T,A0Rn0,A0Rn,A0,A) :- K=<N |
   arg(K,T,Tk), setarg(K,A0Rn0,_,A0Rnk,A0Rn1),
   getputreg(Tk,A0,A0Rnk,A1),
   K1:=K+1, putargs(K1,N,T,A0Rn1,A0Rn,A1,A).
```

Note that the two functionalities of Prolog's `functor`/3 are provided by different KL1 built-ins, `functor`/3 and `new_functor`/3. While `functor`/3 suspends on the first argument and examines its principal constructor, `new_functor`/3 creates a new structure with a constructor specified by the second and the third arguments. The major difference between `new_functor`/3 and its Prolog counterpart is that the arguments of the structure are initialized to `0` rather than fresh, distinct variables. This is because we have found that initializing its elements to a filler constant and replacing them using `setarg`/5 shows much better affinity with a static mode system that plays various important rôles [30] in concurrent logic programming. As discussed in [31], strong moding is deeply concerned with the number of access paths (or references) to each variable (or its value). It prefers variables with exactly two occurrences to those with three or more occurrences by giving the former more generic, less-constrained modes. Our `setarg`/5 does not copy or discard the (direct or indirect) access paths to

the elements of an array, including the element to be removed and the element with which to fill in the blank.

Linearity analysis [33] for Mode Flat GHC is more directly concerned with the number of access paths. Under reasonable conditions, it enables us to implement `setarg`/5 as destructive update as long as the original structure is not shared.

Both mode and linearity systems encourage *resource-conscious programming*. Resource-conscious programming means to pay attention to the number of occurrences of each variable and to prefer variables with exactly two occurrences. This is not so restrictive as it might seem, and our static analyzer *klint* [33] and an automated debugger *kima* [2][3] support it by detecting – and even correcting – inadvertently too many or too few occurrences of the variables. Resource-conscious programs are easier to execute on a distributed platform because they can benefit more from compile-time garbage collection.

Finally, we show the definition of `spawn`/3 for spawning body goals according to the bodycode and the current goal record:

```
spawn(A, []     ,Env) :- true | true.
spawn(A0,[B0|BN],Env) :- true |
   functor(B0,F,N), setargs(1,N,B0,A0,B,A),
   exec_one(B,Env), spawn(A,BN,Env).

/* registers once read are cleared */
setargs(K,N,B0,A0,B,A) :- K> N | B=B0, A=A0.
setargs(K,N,B0,A0,B,A) :- K=<N |
   setarg(K,B0,Bk,ABk,B1), setarg(Bk,A0,ABk,0,A1),
   K1 := K+1, setargs(K1,N,B1,A1,B,A).
```

Note that concurrent execution of body goals is realized by the concurrent execution of `exec_one`'s.

5.4 Summary

Now we have almost finished the description of our interpreter. To be self-contained, here we show all the remaining predicates.

```
/* The interpreter's top-level */
exec([],Env)     :- true | true.
exec([G|Gs],Env) :- true | exec_one(G,Env), exec(Gs,Env).

exec_one(G,Env) :- true |
   retrieve(G,Env,TC), prepare_goalrec_body(G,TC,A,B),
   try(A,B,Env).

retrieve(G,Env,TC) :- true |
   functor(G,P,N), retrieve(P,N,Env,TC).
retrieve(P,N,[P/N-TC0|_],TC) :- true | TC=TC0.
otherwise.
```

```
retrieve(P,N,[_|Env],TC) :- true | retrieve(P,N,Env,TC).

prepare_goalrec_body(G0,treecode(N,B0),A,B) :- true |
   B=B0,
   functor(G0,_,Ng), new_functor(A0,g,N),
   transfer_args(1,Ng,G0,A0,_,A).

transfer_args(I,N,G0,A0,G,A) :- I> N | G=G0, A=A0.
transfer_args(I,N,G0,A0,G,A) :- I=<N |
   setarg(I,G0,Gi,0,G1), setarg(I,A0,_,Gi,A1),
   I1 := I+1, transfer_args(I1,N,G1,A1,G,A).

/* Simply a case branch based on the syntax of treecode */
try(A,[c(G,B)|Cs],Env)           :- true |
   try_one(A,G,B,Cs,Env).
try(A,(Rn1->Cs1)+(Rn2->Cs2),Env) :- true |
   try_two(A,Rn1,Cs1,Rn2,Cs2,Env).
try(A0,b(BU,BN),Env)             :- true |
   tell(A0,BU,A), spawn(A,BN,Env).

/* Binary disjunctive wait */
try_two(A0,Rn1,Cs1,Rn2,Cs2,Env) :- true |
   setarg(Rn1,A0,A0Rn1,ARn1,A1), setarg(Rn2,A1,A0Rn2,ARn2,A),
   try_two(A,A0Rn1,ARn1,A0Rn2,ARn2,Cs1,Cs2,Env).

try_two(A,A0Rn1,ARn1,A0Rn2,ARn2,Cs1,Cs2,Env) :- wait(A0Rn1) |
   ARn1=A0Rn1, ARn2=A0Rn2, append(Cs1,Cs2,Cs), try(A,Cs,Env).
try_two(A,A0Rn1,ARn1,A0Rn2,ARn2,Cs1,Cs2,Env) :- wait(A0Rn2) |
   ARn1=A0Rn1, ARn2=A0Rn2, append(Cs2,Cs1,Cs), try(A,Cs,Env).

append([],   Y,Z ) :- true | Y=Z.
append([A|X],Y,Z0) :- true | Z0=[A|Z], append(X,Y,Z).
```

The restrictions of the above interpreter and possible solutions to them are as follows:

1. Three unary constructors, '<', '>' and '<<', are reserved. This can be easily circumvented by wrapping non-variable as well as variable symbols by some constructors, but we did not do so for the readability of treecode.
2. Currently, the only built-in predicates provided (but not shown above) are those for arithmetics. However, other built-ins such as those used in the interpreter itself can be easily provided.
3. A nonlinear clause head, namely a head with repeated occurrences of a variable, cannot be compiled into treecode. Extending the interpreter to deal with nonlinear heads is straightforward and left as an exercise. However, the use of a nonlinear clause head to check the equality of arguments is discouraged, because it is the only construct that may take unbounded execution

time by comparing two terms of arbitrarily large sizes. For distributed and real-time applications, it is desirable that the execution time of every primitive language construct is bounded.
4. The only construct whose support requires non-straightforward hacking on the interpreter is non-binary disjunctive wait. Since n-ary disjunctive wait is essentially n-ary arbitration, this could be supported by implementing an n-ary arbiter which observes variables $x_1, \ldots, x_n$ and returns an arbitrary k such that x_k has been instantiated.

The interpreter is not self-applicable in its present form, but the discussions above indicate that we are quite close to a self-applicable meta-interpreter. Note that the `otherwise` construct to specify default cases can be expressed implicitly using casecode because the *ask* parts of its branches are tested both deterministically and sequentially.

6 Partial Evaluation

How can one be assured that interpreted treecode behaves exactly the same as its original code?

Instead of showing a translator from Flat GHC to treecode and its correctness, here we illustrate how the treecode for `append`/3 applied to our interpreter can be partially evaluated to its original Flat GHC code.

The rôle of partial evaluation in our framework is twofold. First, the receiver of treecode can figure out what Flat GHC code it represents. Second, although the interpreter itself is not directly amenable to static analysis because its behavior depends on the treecode given, the original code restored by partial evaluation is amenable to static analysis. In this way we can attach various kinds of type information (including mode and linearity) to the arguments of a goal whose behavior is determined by treecode.

For partial evaluation, we use unfold/fold transformation rules described in [28]. The rules consist of the following:

1. *Normalization* — executes unification goals in a guard and a body so that each clause reaches its unique normal form. A normal form should have no unification goals in guards, and all residual unification body goals should be to instantiate head variables of the clause.
2. *Immediate Execution* — deals with the unfolding of a non-unification body goal which does not involve synchronization. That is, the rule is applicable only when, for each clause C in the program and each goal g to be unfolded, either g is reducible using C or, for all σ, $g\sigma$ is irreducible using C.
3. *Case Splitting* — deals with the unfolding of non-unification body goals of a clause C which may promote *ask*s from the guards of clauses used for the unfolding to the guard of C. The clause C must not have unification body goals.
   To see how the Case Splitting of C works, consider a goal g that is about to be reduced using C. For g to generate some output, at least one more

reduction (of one of the body goals of C) is necessary because C has no unification body goals. Case splitting enumerates all the possibilities of the first such reduction.

4. *Folding* — which is essentially the same as the Tamaki-Sato folding rule [25].

The major difference from the Tamaki-Sato rule set is that unfolding is split into two incomparable rules, *Immediate Execution* and *Case Splitting*, to deal with synchronization.

Let $\mathcal{E}$ be the treecode for `append`/3:

```
[append/3-treecode(6,
   [c(1=[], b([<(2)= <(3)],[])),
    c(1=[>(4)|>(1)], b([<(3)=[<(4)|>(3)]],[append(1,2,3)]))])]
```

To show that `exec_one(append(X,Y,Z),`$\mathcal{E}$`)` behaves the same as `append(X,Y,Z)` under its standard definition, let us start with a clause

```
append(X,Y,Z) :- true | exec_one(append(X,Y,Z),E).
```

and start applying Immediate Execution to its body goal. Using `exec_one`/2 shown in Sect. 5.4, we obtain

```
append(X,Y,Z) :- true |
  retrieve(append(X,Y,Z),E,TC),
  prepare_goalrec_body(append(X,Y,Z),TC,A,B), try(A,B,E).
```

With two more applications of *Immediate Execution*, first to the goal `retrieve`/3 and the second to the primitive `functor`/3, we obtain

```
append(X,Y,Z) :- true |
  P=append, N=3, retrieve(P,N,E,TC),
  prepare_goalrec_body(append(X,Y,Z),TC,A,B), try(A,B,E).
```

which can be normalized to

```
append(X,Y,Z) :- true |
  retrieve(append,3,E,TC),
  prepare_goalrec_body(append(X,Y,Z),TC,A,B), try(A,B,E).
```

With several steps of *Immediate Execution* and *Normalization*, we arrive at

```
append(X,Y,Z) :- true |
  transfer_args(1,3,append(X,Y,Z),g(0,0,0,0,0,0),_,A),
  try(A,[c(1=[],b([<(2)= <(3)],[])),
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)]))],E).
```

where `transfer_args`/6 "loads" the arguments `X`, `Y`, `Z` to the goal record and returns the result to `A`. Further steps of *Immediate Execution* and *Normalization* lead us to

```
append(X,Y,Z) :- true |
  functor(X,A0RnF,A0RnN),
  test_pf(A0RnF,A0RnN,[],0,Res),
  try_match(Res,[],X,ARn,g(ARn,Y,Z,0,0,0),b([<(2)= <(3)],[]),
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)])),E).
```

This is the first point at which we can't apply *Immediate Execution* or *Normalization*.

We regard the primitive `functor`/3 as comprising clauses such as:

```
functor([],   F,N) :- true | F=[],  N=0.
functor([_|_],F,N) :- true | F='.', N=2.
functor(f(_), F,N) :- true | F=f,   N=1.
```

There is one such clause for each constructor available, but without loss of generality we can focus on the above three clauses, of which the third one is meant to be a representative of all constructors irrelevant to the current example.

Now we apply *Case Splitting* and obtain the following:

```
append([],Y,Z) :- true |
  A0RnF=[], A0RnN=0,
  test_pf(A0RnF,A0RnN,[],0,Res),
  try_match(Res,[],[],ARn,g(ARn,Y,Z,0,0,0),b([<(2)= <(3)],[]),
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)])),E).
append([H|T],Y,Z) :- true |
  A0RnF='.', A0RnN=2,
  test_pf(A0RnF,A0RnN,[],0,Res),
  try_match(Res,[],[H|T],ARn,g(ARn,Y,Z,0,0,0),b([<(2)= <(3)],[]),
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)])),E).
append(f(X),Y,Z) :- true |
  A0RnF=f, A0RnN=1,
  test_pf(A0RnF,A0RnN,[],0,Res),
  try_match(Res,[],f(X),ARn,g(ARn,Y,Z,0,0,0),b([<(2)= <(3)],[]),
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)])),E).
```

That is, we unfold `functor`/3 and promote its *ask*s to the guards of `append`/3. The *Case Splitting* rule dictates that we should unfold `test_pf`/5 and `try_match`/7 as well; however, unfolding `test_pf`/5 using its first clause, for instance, would promote two *ask*s, `A0RnF=[]` and `A0RnN=:=0`, which can never be satisfied because the two variables don't occur in the head of `append`/3. Clauses with unsatisfiable *ask*s are deleted finally. Note that clauses below the `otherwise` directive (such as the second clause of `test_pf`/5) implicitly perform all *ask*s in the clauses above the `otherwise`.

Now we come back to applying *Normalization* and *Immediate Execution*, which leads us via

```
append([],Y,Z) :- true |
  try_match(yes(0),[],[],ARn,g(ARn,Y,Z,0,0,0),b([<(2)= <(3)],[]),
```

```
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)])),E).
append([H|T],Y,Z) :- true |
  try_match(no,[],[H|T],ARn,g(ARn,Y,Z,0,0,0),b([<(2)= <(3)],[]),
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)])),E).
append(f(X),Y,Z) :- true |
  try_match(no,[],f(X),ARn,g(ARn,Y,Z,0,0,0),b([<(2)= <(3)],[]),
    c(1=[>(4)|>(1)],b([<(3)=[<(4)|>(3)]],[append(1,2,3)])),E).
```

to the following:

```
append([],Y,Z) :- true |
  getargs(1,0,[],[],g(0,Y,Z,0,0,0),A),
  try(A,b([<(2)= <(3)],[]),E).
append([H|T],Y,Z) :- true |
  getargs(1,2,[>(4)|>(1)],[H|T],g(0,Y,Z,0,0,0),A),
  try(A,b([<(3)=[<(4)|>(3)]],[append(1,2,3)]),E).
append(f(X),Y,Z) :- true | try(g(f(X),Y,Z,0,0,0),[],E).
```

Here, the rules as stated in [28] do not allow *Immediate Execution* of the third clause because we cannot form any unfolded clause to replace it. However, a close look at the reason why assures us that this clause *can* indeed be removed. The removal of a clause C whose body goal can never proceed changes the behavior of a goal g when there is another clause C' that can reduce g. However, the three clauses of `append`/3 above don't overlap with one another; that is, any goal that can be reduced using the third clause and then gets stuck will get stuck without it.

By steps of *Immediate Execution*, we can "load" necessary values to registers:

```
append([],Y,Z) :- true |
  try(g(0,Y,Z,0,0,0),b([<(2)= <(3)],[]),E).
append([H|T],Y,Z) :- true |
  try(g(T,Y,Z,H,0,0),b([<(3)=[<(4)|>(3)]],[append(1,2,3)]),E).
```

Now we have restored the guards of the original `append`/3, which is much more than halfway to our goal. It remains to restore the bodies, and this can be done by repetitive application of *Immediate Execution* and *Normalization*:

```
append([],Y,Z) :- true | Y=Z.
append([H|T],Y,Z) :- true |
  Z=[H|A0Rn], exec_one(append(T,Y,A0Rn),E).
```

Finally, we fold the body goal of the second clause using the clause we coined initially, and obtain the following:

```
append([],   Y,Z) :- true | Y=Z.
append([H|T],Y,Z) :- true | Z=[H|A0Rn], append(T,Y,A0Rn).
```

We anticipate that the significance of partial evaluation in our context is it enables us to use available tools for "just-in-time" static analysis. For faster execution, designing an optimizing compiler from treecode to machine code would be more appropriate than going back from treecode to Flat GHC source code.

7 Conclusions

We have described an interpreter of Flat GHC treecode in Flat GHC. The interpreter uses only *pure* built-in primitives, that is, those whose behavior can be defined using a set of guarded clauses (e.g., `functor`/3, `setarg`/5, etc.) or by simple source-to-source transformation (`otherwise`). The interpreter is only 39 clauses long (without arithmetics), and runs directly on KLIC.

Treecode is very close to source code but is designed so that it can be easily interpreted, transmitted over the network, and stored in files. The major differences from most bytecode representations are that it is more structured and, more importantly, that it is inherently concurrent.

The design of an interpreter involves decisions as to what are reified and what are not. To allow interpreted processes to freely communicate with non-interpreted, native processes, we made the following design choices:

- *Reified*: code, reduction, concurrency and nondeterminism; goal records, argument registers and temporary registers; control structures
- *Not reified*: logical variables and substitutions (constraints); heaps; representation of terms.

Although our initial objective was to have an 100% pure interpreter of Flat GHC, the outcome can be viewed also as a virtual machine working on register vectors. The three annotations, '`>`', '`<`', and '`<<`', are reminiscent of the distinction between `put` and `get` instructions in the Warren Abstract Machine [1].

Translation from source code to treecode is straightforward for most cases. For deterministic programs, its essence is to build a decision tree for clause selection. Some complication arises only when a predicate has both conjunctive and disjunctive synchronization points. The paper did not show a concrete translation algorithm, but instead illustrated how a treecode could be translated back to its source code using partial evaluation. Note that the source code could be restored because the interpreter was a *meta*-interpreter. Partial evaluation thus ensures the applicability of program analysis to interpreted code. Type analysis is important for an interpreted process to communicate with a native process running with no runtime type information. It is also important in building a stub and a skeleton of a (marshaled) logical stream laid between remote sites.

Our primary future work is to deploy those technologies to demonstrate that concurrent logic/constraint programming can act, possibly with minimal extensions, as a high-level and concise formalism for distributed programming. Another important direction is, starting with treecode, to develop an appropriate intermediate code representation for optimizing compilers. This is important for another application of concurrent languages, namely high-performance parallel computation.

Acknowledgment

Comments from anonymous referees were useful in improving the presentation of the paper.

References

1. Aït-Kaci, H., *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, MA, 1991.
2. Ajiro, Y., Ueda, K. and Cho, K., Error-Correcting Source Code. In *Proc. Fourth Int. Conf. on Principles and Practice of Constraint Programming (CP'98)*, LNCS 1520, Springer-Verlag, Berlin, 1998, pp. 40–54.
3. Ajiro, Y. and Ueda, K., Kima – an Automated Error Correction System for Concurrent Logic Programs. In *Proc. Fourth Int. Workshop on Automated Debugging (AADEBUG 2000)*, Ducassé, M. (ed.), 2000. http://www.irisa.fr/lande/ducasse/aadebug2000/proceedings.html
4. Apt, K. R., Marek, V. W., Truszczynski M., and Warren D. S. (eds.), *The Logic Programming Paradigm: A 25-Year Perspective*. Springer-Verlag, Berlin, 1999.
5. Bowen, K. A. and Kowalski, R. A., Amalgamating Language and Meta-Language in Logic Programming. In *Logic Programming,* Clark, K. L. and Tärnlund, S. Å. (eds.), Academic Press, London, pp. 153–172, 1982.
6. Chikayama, T., Fujise, T. and Sekita, D., A Portable and Efficient Implementation of KL1. In *Proc. 6th Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'94)*, LNCS 844, Springer-Verlag, Berlin, 1994, pp. 25–39.
7. Clark, K. L. and Gregory, S., A Relational Language for Parallel Programming. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA'81)*, ACM, 1981, pp. 171–178.
8. Clark, K. L. and Gregory, S., PARLOG: Parallel Programming in Logic. *ACM. Trans. Prog. Lang. Syst.*, Vol. 8, No. 1 (1986), pp. 1–49.
9. Foster, I. and Taylor, S., Strand: a Practical Parallel Programming Tool. In *Proc. 1989 North American Conf. on Logic Programming (NACLP'89)*, The MIT Press, Cambridge, MA, 1989, pp. 497–512.
10. Haridi, S., Van Roy, P., Brand, P. and Schulte, C., Programming Languages for Distributed Applications. *New Generation Computing*, Vol. 16, No. 3 (1998), pp. 223–261.
11. Huntbach, M. M., Ringwood, G. A., *Agent-Oriented Programming: From Prolog to Guarded Definite Clauses*. LNCS 1630, Springer-Verlag, Berlin, 1999.
12. Maher, M. J., Logic Semantics for a Class of Committed-Choice Programs. In *Proc. Fourth Int. Conf. on Logic Programming (ICLP'87)*, The MIT Press, Cambridge, MA, 1987, pp. 858–876.
13. McCarthy, J., *Lisp 1.5 Programmer's Manual*. MIT Press Cambridge, MA, 1962.
14. Miller, D. and Nadathur, G., Higher-order Logic Programming. In *Proc. Third Int. Conf. on Logic Programming (ICLP'86)*, LNCS 225, Springer-Verlag, Berlin, 1986, pp. 448–462.
15. Milner, R. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.
16. Nakashima, H., Ueda, K. and Tomura, S., What Is a Variable in Prolog? In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984 (FGCS'84)*, ICOT, Tokyo, 1984, pp. 327–332.
17. Safra, M. and Shapiro, E. Y., Meta Interpreters for Real, In *Information Processing 86*, Kugler, H.-J. (ed.), North-Holland, Amsterdam, pp. 271–278, 1986.
18. 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 (POPL'90)*, ACM Press, 1990, pp. 232–245.

19. Saraswat, V. A., Kahn, K. and Levy, J., Janus: A Step Towards Distributed Constraint Programming. In *Proc. 1990 North American Conference on Logic Programming (NACLP'90)*, The MIT Press, Cambridge, MA, 1990, pp. 431–446.
20. Shapiro, E. Y., Concurrent Prolog: A Progress Report. *IEEE Computer*, Vol. 19, No. 8 (1986), pp. 44–58.
21. Shapiro, E. Y. (ed.), *Concurrent Prolog: Collected Papers*, Volumes I+II. The MIT Press, Cambridge, MA, 1987.
22. Shapiro, E. Y., Warren, D. H. D., Fuchi, K., Kowalski, R. A., Furukawa, K., Ueda, K., Kahn, K. M., Chikayama, T. and Tick, E., The Fifth Generation Project: Personal Perspectives. *Comm. ACM*, Vol. 36, No. 3 (1993), pp. 46–103.
23. Smolka, G., The Oz Programming Model. In *Computer Science Today*, van Leeuwen, J. (ed.), LNCS 1000, Springer-Verlag, Berlin, 1995, pp. 324–343.
24. Takeuchi, A. and Furukawa, K., Partial Evaluation of Prolog Programs and Its Application to Meta Programming. In *Information Processing 86*, Kugler, H.-J. (ed.), North-Holland, Amsterdam, 1986, pp. 415–420.
25. Tamaki, H. and Sato, T., Unfold/Fold Transformation of Logic Programs. In *Proc. Second Int. Logic Programming Conf. (ICLP'84)*, Uppsala Univ., Sweden, 1984, pp. 127–138.
26. Tick, E. The Deevolution of Concurrent Logic Programming Languages. *J. Logic Programming*, Vol. 23, No. 2 (1995), pp. 89–123.
27. Ueda, K., Guarded Horn Clauses. ICOT Tech. Report TR-103, ICOT, Tokyo, 1985. Also in *Logic Programming '85*, Wada, E. (ed.), LNCS 221, Springer-Verlag, Berlin, 1986, pp. 168–179.
28. Ueda, K. and Furukawa, K., Transformation Rules for GHC Programs. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988 (FGCS'88)*, ICOT, Tokyo, 1988, pp. 582–591.
29. 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.
30. 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.
31. Ueda, K., Experiences with Strong Moding in Concurrent Logic/Constraint Programming. In *Proc. Int. Workshop on Parallel Symbolic Languages and Systems (PSLS'95)*, LNCS 1068, Springer-Verlag, Berlin, 1996, pp. 134–153.
32. Ueda, K., Concurrent Logic/Constraint Programming: The Next 10 Years. In [4], 1999, pp. 53–71.
33. Ueda, K., Linearity Analysis of Concurrent Logic Programs. In *Proc. Int. Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, Ito, T. and Yuasa, T. (eds.), World Scientific, Singapore, 2000, pp. 253–270.
34. van Emden, M. H. and de Lucena Filho, G. J., Predicate Logic as a Language for Parallel Programming. In *Logic Programming*, Clark, K. L. and Tärnlund, S. -Å. (eds.), Academic Press, London, 1982, pp. 189–198.