7 Parallel Execution Models of Declarative Languages

7.1 Imperative and declarative languages

One of the major divisions in programming languages is that between the imperative and declarative style of programming.

An imperative (or procedural) language instructs the computer to perform a set of operations. The family of imperative languages comes from the early machine code and by successive stages of abstraction it broke away from the hardware. Imperative languages in their current state ensure high productivity and portability. However, even in their most sophistcated form they reflect the algorithmic steps in reaching a solution.

These languages still assume a linear address space (which is reasonable for most of the computers today) thus, their natural data structure is the array which is a direct mapping of data onto the memory.

The concurrency in imperative languages means that disjoint sequences of instructions proceed in parallel. The demand for such an execution has arisen before the real hardware concurrency appeared. The first form of concurrency was when logically separate tasks shared the cycles of a single CPU.

The declarative programming languages are based on a clear mathematical foundation. This allows the programmer to move away from the algorithmic description towards the description of the problem. Roughly saying: the programmer concentrates on what to do instead of how to do it. These languages are based either on the lambda calculus (the calculus of functions) or on predicate logic[1]. The first set is called functional languages, and its most widespread instance is LISP, while the latter group is the logic languages with its famous representative Prolog.

The natural data structure of these languages is the list (or compound terms) which is an indirect mapping of data onto the linear address space. This is not well suited to nowaday's computers and the processor-memory interface may form a bottleneck (the so called von Neumann bottleneck.)

Although the existing implementations of declarative languages are less effective than the imperative ones, there is a demand for languages based on the formal rigour of mathematics. This is due to the growing complexity of software systems and their change towards intelligent or expert systems. Declarative languages allow a less regular, less imperative programming style which fits better to these applications. Moreover declarative languages also give the possibility to formal verification of programs thanks to their pure mathematical foundations.

Another very important feature of declarative languages which is the subject of this study that they can supply implicit parallelism. There is an intensive research in the field of exploitation the inherent parallelism of declarative languages, since an effective parallel implementation could overcome some aspects of the weakness of declarative systems.

In this study the most important parallel execution models of logic and functional languages will be surveyed. The focus is set rather to the arising problems and their solutions than the details of a certain implementation.

7.2 Logic programming

Logic programs are a set of Horn clauses in the form:

```
Ai :- B1, B2, ...Bm.
```

and a query:

$:- Q_1, Q_2, \ldots Q_n.$

All the expressions above as A_i, B_i, Q_i are atomic formulas, i.e. predicates in the following form: $p(t_1,t_2,..t_k)$, where p is the name of the predicate, t_i can be constants, variables or compound terms.

The A_i is called as the **head** of a clause while $B_1,...B_m$ is the **body**. Each B_i in the body is called as a **goal**. Clauses of the same name and arity are forming a **predicate** together.

[1]Recently there are research efforts to combine the two branches of declarative languages.

The execution can be considered as a transformation of the query. The set of goals remaining to be executed is called the **resolvent**. Thus, the first resolvent is the query. The transformation runs in the following cycle:

while the resolvent is non-empty
  Goal selection: select **any** of the goals in the resolvent: G
  if there exists at least one clause whose head unifies with G then *Resolution:*
    Clause selection: select **any** of these clauses
    Unification: compute the most general unifier
    Inference: replace G with the body of the selected clause in the resolvent and apply the new variable bindings resulted from the unification in the resolvent
  else *Backtrack*:
    if there exists a previous resolvent that has untried alternatives then recall this previous resolvent and do *Goal selection*
    else exit with **Failure**
if the resolvent became empty, then exit with **Success**

When the resolution was successful, the result is the binding of the variables in the query.

The execution can be graphically notated as a tree graph, the so called search-tree. Nodes in the graph represent the current resolvent, and the outgoing arcs lead to different alternatives of the currently selected goal.

At this point the duality of semantics should be mentioned: the logical programs can be interpreted either in declarative or procedural way. The clause: A:-B_1,B_2,B_3. can be read in declarative sense as: if B_1, B_2 and B_3 are true then A is true. On the other side, it can be explained as all the predicates are procedure definitions and goals in the body are procedure calls where the unification is the parameter passing.

7.2.1 Prolog

Prolog can be considered as the most successful implementation of logic programming languages. However, it has a slightly different execution strategy than that of pure logic programming: in the *Goal selection* phase it takes the first (leftmost) goal in the resolvent, and in the *Clause selection* phase it takes the first among the candidate clauses, according to their order in the Prolog source. This means the sequential transversal of the search tree in depth-first, left-to-right order.

Since Prolog was designed for practical applications, it contains several built-in functions which do not fit into the pure logical programming scheme, e.g. input, output instructions, database handling and cut.

7.2.1.1 The source of parallelism

As we have seen, there are two restrictions in Prolog execution in comparison with that of logical programs: the goal to be selected is strict and the clause to be selected is also strict. Thus, in Prolog we have designated a strict sequential execution order of inherently parallel activities. However, while maintaining the Prolog semantics, this sequential selection can be abandoned and the inherent parallelism of logical programs can be exploited.

Usually, two types of parallelism are distinguished in Prolog programs according to the choices listed above: the AND-parallelism and the OR-parallelism [ChassinCodo94]. In the first case one may select multiple goals in the resolvent and execute the resolution step in parallel. In the second case one may select not one but all the possibly clauses to be unified. However, since there are many techniques to exploit the AND- or OR-parallelism, a more sophisticated categorisation should be done [Kacsuk93]:

- OR-parallelism: alternate clauses of a predicate represent alternative resolutions to a given goal and these are processed in parallel
- Independent AND-parallelism: several goals which do not contain shared variables are processed in parallel on the same resolution
- Dependent AND-parallelism: several goals which contain shared variables are processed in parallel on the same resolution
- Pipeline AND-parallelism: several goals are processed in pipeline on alternative resolutions

- Search parallelism: unit clauses are processed in parallel (a special case of OR-parallelism)
- Path parallelism: multiple bindings are processed in parallel (a special case of OR-parallelism)

7.2.1.2 Sequential execution

Before introducing the parallel execution models, the most efficient sequential scheme should be mentioned. When the Prolog was introduced in the early 70s, it was an interpreted language, i.e. the source code was analysed run-time. (Many researchers were doubtful at that time whether Prolog can be compiled at all.) In the mid 70s the first compiled Prolog appeared. The Prolog source code was compiled to the code of an abstract machine. This code was then interpreted or compiled to the language of the target machine. It means, that instead of compiling the source to the machine code (e.g. as in the case of C), an intermediate level was introduced between the highly abstracted source code and the physical machine code. This level is called the abstract machine, and it has its own abstract machine language.

D.H.D.Warren introduced his abstract machine in 1983 and it can be still considered the most effective sequential execution scheme [Warren83]. It became de facto standard and it is referred usually as the WAM (Warren's Abstract Machine).

Since it forms the basis of many parallel Prolog implementations, it must have been mentioned, however, it is out of the scope of this study to describe it in details. A tutorial introduction to WAM can be found in [Ait-Kaci91].

7.2.2 OR-parallel logic computational models

7.2.2.1 A simple example

Let us consider the following simple Prolog program:

```
:-fruit(X, sweet, red).
fruit(X,Y,Z):- taste(X,Y), colour(X,Z).
taste(apple, sweet).
taste(lemon, acid).
taste(strawberry, sweet).
colour(lemon, yellow).
colour(apple, green).
colour(strawberry, red).
```

Listing 7-1

The execution is depicted in Figure 7-1.

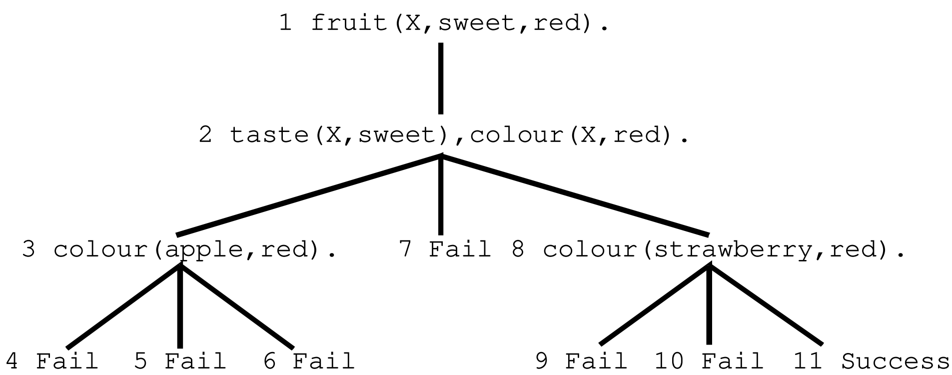


Figure 7-1 The search tree of the fruit-classification program.

During the sequential execution the search-tree would be traversed in a depth-first, left to right order as the nodes are numbered. However, we can see, that there is a potential parallelism to be exploited in nodes 2, 3 and 8.

A typical OR-parallel Prolog system traverses the tree in the following way [Karlsson92]. The tree is traversed (i.e. the compiled Prolog code is interpreted) by processes, usually called workers. There is a limited number of workers. One worker starts the execution in the same way as in the sequential case, while other workers are idle. When the active worker reaches node 2, three alternative branches are created. The active worker will continue on the leftmost alternative, while the remaining alternatives could be offered to idle workers. Thus, the three alternative branches are traversed in parallel. The second worker fails at unifying taste(X,sweet) with taste(lemon,acid) and becomes idle. At this point it can seek for work again. This procedure repeats until a solution is (or all the solutions are) found. This is the simplest and most common form of OR-parallel Prolog systems.

To put it in a more precise way: OR-parallelism consists of the simultaneous development of several resolvents that would be computed succesively by backtracking in a sequential execution.

7.2.2.2 Common problems and issues

Let us consider the following situation in the previous example. In node 2 the first worker bound X to apple, and continued the execution while another worker bound the same X to strawberry at the same time. Obviously, this is impossible due to the single assignment property of variables. Thus, an efficient binding scheme for handling multiple bindings must be introduced. Another crucial question is how an idle worker can get work. This second point refers to the problem of scheduling. The most important problems and issues at implementing an OR-parallel Prolog system are:

- Handling of multiple bindings. The issues are:
    1. Efficient memory management. Declarative languages suffer from huge memory requirement.
    2. Efficient dereferencing of terms. At unifications variables must be dereferenced, i.e. the pointers representing bindings must be traced. This operation should be bounded in time.
- Assigning work to workers, i.e. scheduling.
    3. Controlling the granularity, i.e. the ratio of effective work and communication, control, etc.
    4. Minimizing overhead when creating or switching tasks (workers)
    5. Handling side-effect predicates. This issue is related to maintaining sequential order, i.e. while the search tree is traversed in a different order, the system should behave as if it was traversed sequentially

The different OR-parallel Prolog implementations so far showed efforts to meet the more possible conditions listed above. However, it was shown that the following three issues:

- task creation in constant time
- access to variables (dereferencing) in constant time
- task switching in constant time

can not be satisfied at the same time.

In a sequential implementation the most important data areas are the following: stack, heap, trail[2]. The stack contains information about the current state of execution: environments for clause activation (~the state of the machine at entering a body) and choicepoints (~the state of the machine at reaching an alternative branch). The heap holds the compound terms and the persistent bindings. While the stack can be considered as volatile due to entering and exiting clause bodies and backtracking when frames (either environment or choicepoint) are created or discarded, the heap is persistent. The third data area, the trail is used to restore information at backtracking.

[2]Usually the WAM is treated as a machine with four stacks: the environment stack, the choicepoint stack, the term stack and the trail stack. However, the first two stacks are usually implemented in the same memory region forming a stack, the term stack is called heap, and the trail stack is called trail.

Thus, the central question at designing an OR-parallel system is how to share this information efficiently and safely among multiple workers.

7.2.2.3 Copying of stacks

In this scheme each worker has its own instances of the stacks and the workers have not access to each others stacks. The share of work is obtained by copying the appropriate portions of stack (and, thus, the computational state at an OR-branch) from one workers' memory location to another.

A common problem in this scheme is that the worker getting a piece of work has to restore its stack to their state corresponding to the time when the actual OR-node was created. One solution is the Kabu-Wake model, where each binding has a logical date thus, it can be detected whether a binding was made after the creation of the choicepoint and must be undone. The other solution makes this un-binding by means of the trail stack.

The following example which follows the second solution is taken from the Muse model [Karlsson92]. Its name comes from the *multi-sequential* approach suggesting that it assumes independent workers which traverse the search-tree in the usual sequential way. It also assumes that there is a global address space shareable by all workers.

7.2.2.3.1 The basic execution model

The result of a stack copying operation can be seen in Figure 7-2.

1. When the execution starts, one worker P processes the query. The other workers are idle until P creates a choicepoint.
2. One of the idle workers, Q requests for work.
3. P shares its local choicepoints with Q thus, allowing Q to grab a piece of work:
   a) for each local choicepoint P creates a shareable frame in the shared memory area.
   b) P moves information describing the remaining alternatives from the local choicepoints to the shareable frames.
   c) each local choicepoint points to the corresponding shareable frame in the shared area.
   d) P copies its state (its stacks) to Q. At this point both P and Q have the same state and share P's choicepoints.
4. From this point P and Q work together. P continues its current task while Q simulates a failure to get the next alternative from the bottommost choicepoint. This is the step which corresponds to the restoration of the stacks.
5. When a worker gets a task, it processes that task exactly as a standard sequential Prolog engine.
6. When a worker P creates local choicepoints and there are idle workers, P makes its choicepoints shareable. (Step 3.)
7. When all work is processed, all workers become idle and the execution terminates.

7.2.2.3.2 Incremental copying

In step 3d. worker P copies all its stack content to the local space of worker Q. However, it can be observed that some of the frames may have been already shared. Thus, P overwrites superfluously some of the stack frames of Q with the same content. Obviously, this overhead should be eliminated. The solution called incremental stack copying was introduced in Muse.

The basic idea is to keep the stack frames of Q identical with that of P and to copy only the differing frames from P to Q. This is obtained in the following way.

Assume that worker Q is going to share the local choicepoints of P (Figure 7-3). First Q backtracks to the youngest choicepoint *n* shared with P. Then P makes its local choicepoints shareable and copies only the differing parts. The differing parts are the top stack frames corresponding to choicepoints younger than *n* and the local modifications in the common part since the creation of *n*. It is easy to identify the frames younger than *n*. The younger choicepoints and environment frames are simply on the top of *n*, while the value of heap pointer and trail pointer at the moment of creation of *n* is saved in the choicepoint frame *n*. The local modifications in the common part can be traced by the trail frames younger than *n*. At this moment P and Q have identical states.

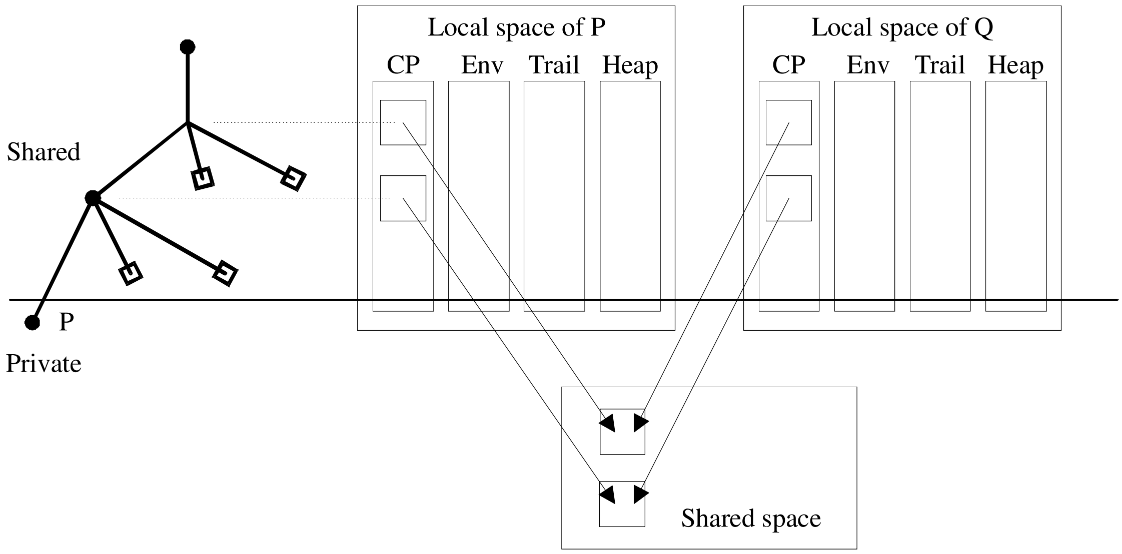


Figure 7-2. The stack copying scheme. The first 2 choicepoints of worker P became shared with Q. The rectangular leaves of the tree are the alternatives to be grabbed by another workers.

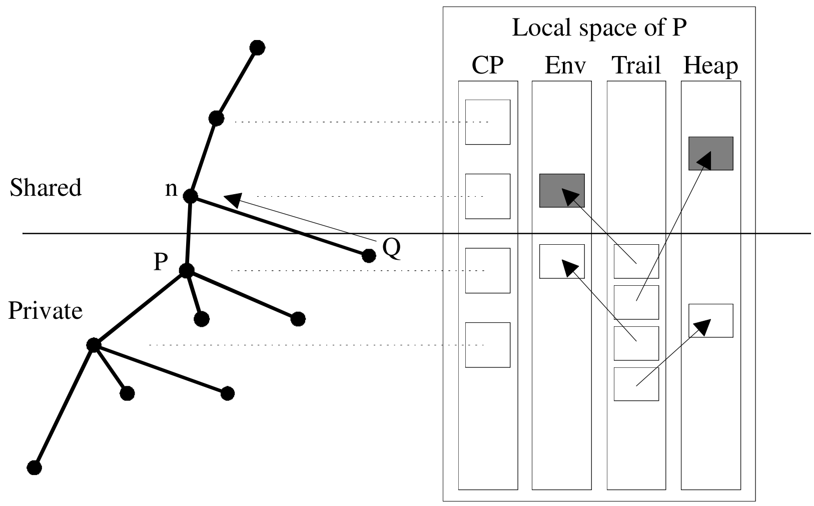


Figure 7-3. The incremental copying. Only frames in private part of P will be copied and the shared part will be updated according to the modifications done by P. The modifications can be traced upon the trail.

7.2.2.4 Sharing of stacks

The second approach to the problem scraps the idea of completely disjoint private stacks. Instead, parts of the stacks are shared and additional data structures are used to handle multiple bindings. There are several such additional data structures proposed as: binding arrays, hash windows, directory trees, version vectors. The idea is common: the shared parts of the stacks represent the pieces of work to be shared. The bindings which could cause conflicts are stored in private data structures. The term representation is the same as that in the WAM except that conditional variables never reference to a value but they hold the information how to access the value in the context of the given worker.

In the binding array scheme each worker has its own private array to register the variable bindings. Variables neither reference to values nor hold the UNBOUND value, but they contain an integer which is the index in the array for the given variable. Thus, a variable X in the common part may contain the index i therefore, worker P will find its binding value at the ith cell of P's binding array, while worker Q will find the value in the ith cell of Q's binding array. Obviously, different workers may see different values for the same variable (physically at the same location in a shared stack). The introduction of the binding array requires only one extra step at dereferencing thus, the variable access still remains a fast, constant time operation.

However, the situation is not the same at task switching. At task switching a worker is moving from one branch of a tree to another one. To do this it must rebuild its binding array corresponding to the new branch in the tree. It is obtained in two phases. In the first phase the worker moves back in the tree by backtracking thus, deinstalling bindings based on the trail stack. Then it moves forward, and installs bindings based on the other worker's trail. However, the trail contains not only the variables and addresses as in standard WAM, but their binding values as well. Hence, it is called Forward stack. The Forward Stack must be accessible to any worker. Obviously, the task switching is not a constant time operation, it depends on the distance in the tree (the nodes to be traversed) to be covered while moving from one point to another. Thus, the scheduling strategy must be sensible to this parameter.

There is a very similar solution to binding arrays called version vectors. In this binding scheme there is a vector for each OR-shared variable having as many elements as many processors (workers) there are. The binding is performed by putting the value into the vector element corresponding to a given worker. Obviously, the vectors must be accessed by all the workers, and there must be synchronisation among them when creating a new vector. This solution has the same problems and benefits as the binding array method has.

In an alternative solution, bindings of a shared variable are stored in a hash table, called hash windows. There is a hash window assigned to each node in the graph. When an OR-shared variable must be modified, its location is computed by hashing the variable address. All the bindings computed from the root to the current node are valid for the worker computing this branch. Thus, hash windows are linked, and whenever a variable fails to be found in the hash window of the current node, the worker has to search for it in a hash window of a higher level; possibly, the search must continue up to the hash window of the root.

This solution uses the memory sparingly, and the cost of task switching is very low, since only a new hash table must be created and linked to the ancestor. However, the access and binding of variables may involve the search of the whole hash structure up to the root.

7.2.2.5 Recomputing of stacks

The recomputation of stacks is an alternative to copying of stacks. At copying stacks a very fast connection or shared memory was assumed among machines of the same architecture. However, it can not be assumed in every case. For example, a cluster of workstation machines (which became wide-spread recently) can contain machines of different type (different data sizes, different endianness, etc.), and they are connected with a relative slow network. In this case, obviously, the copying of stacks is either impossible or not worth due to the slow communication.

Instead of copying the state, it can be recomputed as well. It is trivial, that when the computation is deterministic, i.e. there is no backtracking. However, the computation to a given node can be made deterministic by 'prompting' to the worker which way to choose at a branch. When a worker P explores the tree, it keeps track about its decisions at the choicepoints. If it shares some of its works with other worker Q, simply tells them how to reach the given state where the appropriate piece of work can be grabbed. This can be achieved by the so called *oracles*. This information is relative small to transfer, and can be transferred independently from the hardware. (An oracle is usually a sequence of integers.) This oracle will guide worker Q to build up its stack exactly the same way as they are in worker P.

The same optimization can be applied to the recomputation scheme, as to the stack-copying: the incremental recomputation. It means that when worker Q is about to grab some work from P, it backtracks to the last choicepoint shared with P. Then, the new node can be reached by following the oracles.

7.2.2.6 Closed binding environments

The problem of OR-shared variables arises because a variable slot in a worker's environment can reference to any other environment in the current path (the path from the root to the current node.) The introduced solutions so far tackled with the problem of multiple bindings, however, they still allow a worker to search for a variable through several levels in the tree (Figure 7-4.)

Apart from this overhead, these solutions assume a single address space or at least a part of the memory must be shared. Basically, these solutions are just modifications of the standard sequential WAM machine and can not be applied in a distributed environment because of the high cost of remote accesses.

The closed environment scheme is not an adaptation of the standard 4-stack model. A binding environment seen by any worker is always restricted to one or two frames. Bindings are organized so that all the necessary information needed for unification is present in the two input frames. *A closed environment is a set of frames E such that no pointers or links originating in E dereference to slots in frames that are not in E [Conery87].*

At the closed environment scheme two environment frames must always be taken into account. One can consider an environment closed with respect to another environment. One is the environment of the goal in the resolvent (the goal environment, GE), the other is the environment of the called clause (clause environment, CE.) So, in Figure 7-5 the last frame is closed with respect to the previous.

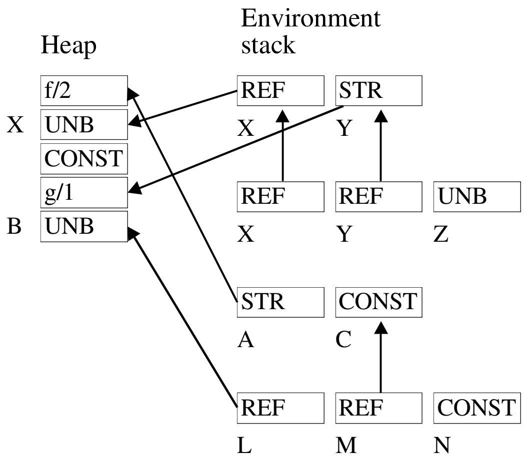


Figure 7-4. This snapshot is from Listing 7-2 after unifying the head of t. The environments can refer to any other environment; the variables on the heap can even be considered as a kind of global data.

The unification is a two-phase unification, when the first phase sets up references between the unified terms, and the second phase ensures that the clause environment is closed with respect to the goal environment. Then entering the body of the clause it will call other clauses and acts the role of the goal. Thus, the role of a clause changes as we go down by the chain of calls (Figure 7-6).

Similarly, when the execution of a clause is finished, it returns the results to the caller goal, and the so-called back-unification must take place. Whereas, the unification ensured that the clause environment is closed with respect to the goal environment, the back unification ensures, that the goal environment is closed with respect to the clause environment. In Figure 7-5. the environment of q/3 is not closed with respect to the environment of t/3, however, it will become so after the back-unification.

The following features of the closed environment scheme break some WAM rules:

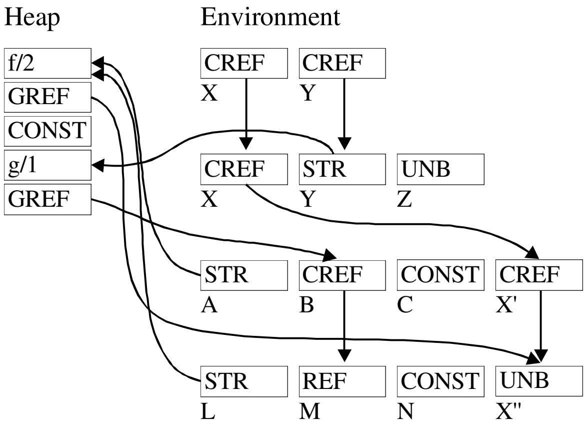


Figure 7-5. The same snapshot as in Figure 7-4 in a closed environment scheme. Note that the last frame (the environment of clause t) holds all the necessary information, so there are no references out of the environment.

1. Variables are never put on the heap.
2. The directions of the pointers representing bindings do not follow the younger-older relationship.

The environment closing procedure (2nd phase of unification) has the following steps:

1. Direct the pointers representing bindings that all must point from GE towards CE.
2. Extend the CE with unbound variables for each unbound variable in GE referenced from CE.

The environment closing mechanism will be demonstrated by an example. The example is taken from the LOGFLOW system [Kacsuk92].

Let us consider the following simple Prolog program fragment:

```
:-p( X, Y ).
p( X, Y ) :- q( f( X, 2 ), Y, 7), r( Z, X ), s( Z, Y ).
q( A, g( B ), C ) :- t( A, B, C).
t( L, M, N):- ....
r(...
s(...
```

Listing 7-2

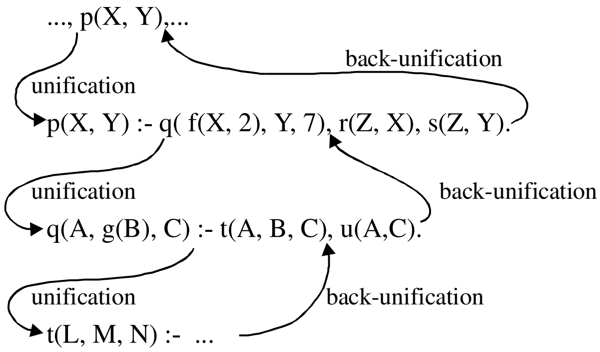


Figure 7-6. Each clause has its own environment. The role of the environment (whether it is a GTE or CTE) changes during the successive calls.

Let us focus on the unification of the head of q/3 (see Figure 7-7.) The unification has the following steps. The incoming environment (goal environment, GE) has 3 slots, X, Y, Z respectively, each of them unbound. The clause environment (CE) has also 3 slots, A, B, C, which are also unbound, since they are newly generated.

After the first step of unification, A is bound to the strucure f/2, Y is bound to structure g/1, C is bound to constant 2. X, B, Z remain unbound. Now, the goal is to separate the CE from GE, i.e. close CE with respect to GE. This is the second step of unification.

1. The direction of pointers between GE and CE need not be altered, since during the unification they were set into the appropriate direction. Note, that this direction is opposite to that of WAM convension. (In this certain case there are no such pointers.)
2. However, CE is still referencing to GE via the f/2 structure. Therefore, CE must be extended with a new unbound variable which will act as X, and set X in GE and the first argument of the f/2 structure to point to the new X'.

At this point, CE is closed with respect to GE, i.e. no references point from CE to GE. Now, entering the body of q/3, it will call other goals and will act as a goal environment. So, the environment of q/3 must be altered to a goal environment which is simply swapping the reference tags from CREF to GREF.

At back unification the situation is opposite: the GE must be closed with respect to the incoming CE (Figure 7-8.) The incoming frame has GREF tags, however, it is a clause environment with respect to the environment of p/2. First the tags must be swapped to CREF tags. The GTE is copied, and the back-unification is performed on the copy. This is in the all-solution nature of LOGFLOW.

1. The direction of pointers must be changed. This involves that the non-empty terms must be copied from CE to GE.
2. The GE is still referencing to CE via the g/1 structure. GE must be extended with a new unbound variable, and all the variables pointing to B must be set to point to B'.

At this point GE is closed with respect to CE. Note, that a given environment is CE first, and then GE, since firstly it is called, then it calls other clauses. The role of an environment changes in a similar way at backward unification.

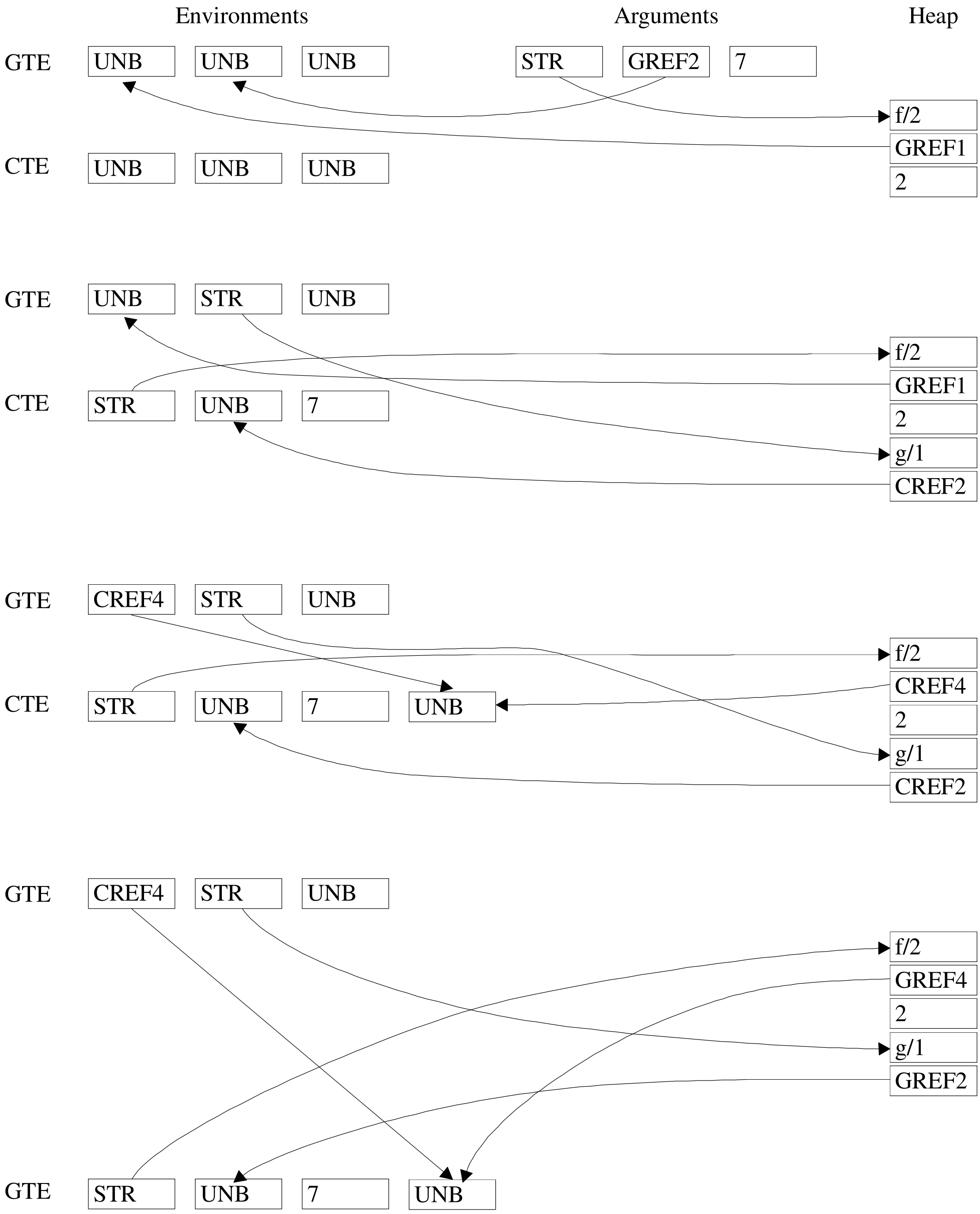


Figure 7-7. Steps of the unification.

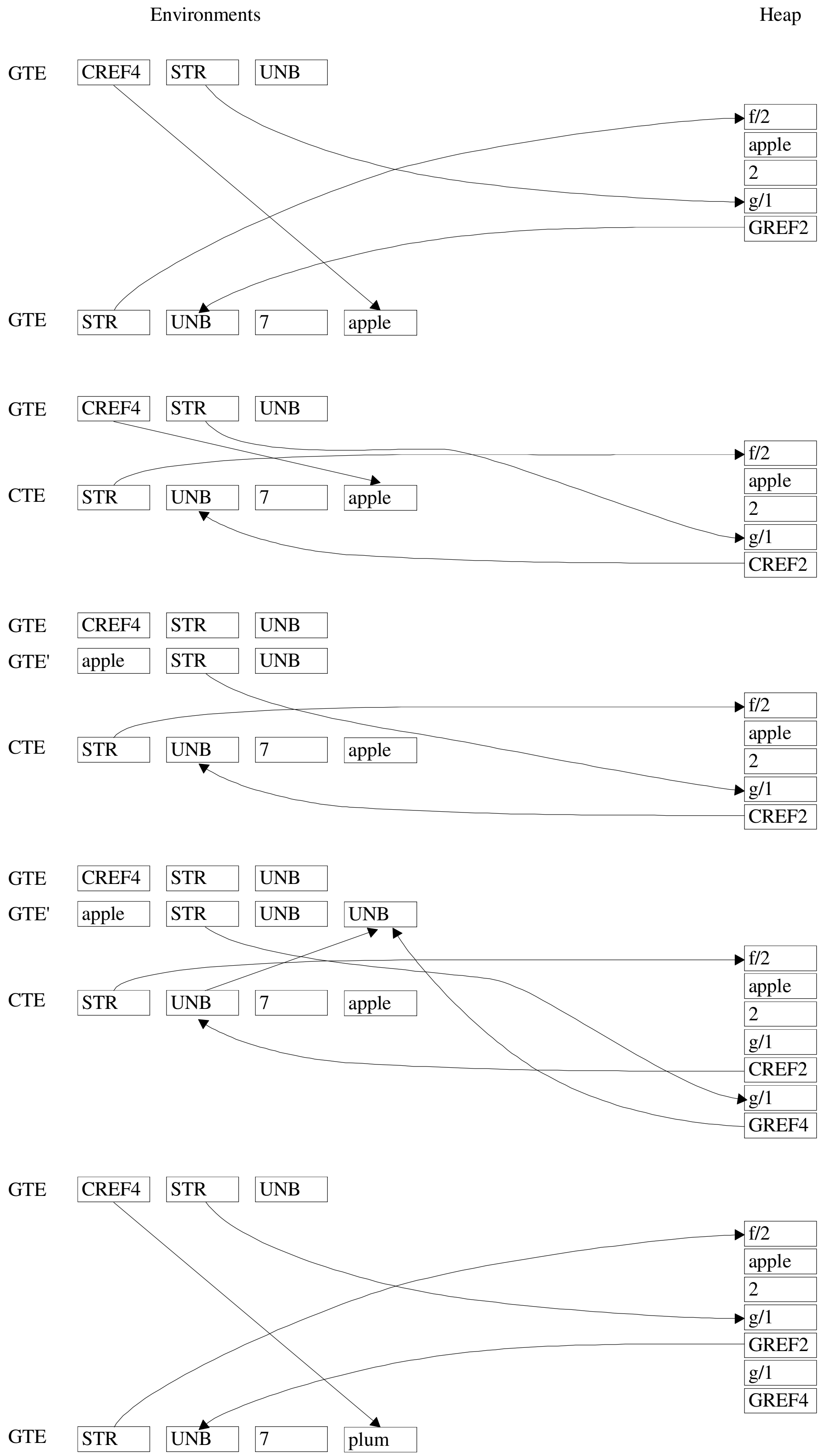


Figure 7-8. The steps of back-unification.

Then the saved GTE is ready to receive another solutions to the given goal (the last stage in Figure 7-8.)

The closed environment is an essential technique in distributed environment. For exmaple, the LOGFLOW sytem is implemented on a transputer network. If it was not a closed environment system, the referenced terms could be find on a different transputer, since the environment on a given transputer could refer to another environment, possibly on a different processing element, far away. To fetch the necessary data, it would require huge communication overhead and delay.

However, the closed environment scheme has a serious drawback: it intensively copies the terms, which is a very expensive operation regarding both the time and memory consumption.

7.2.2.7 Scheduling

All the OR-parallel Prolog systems have some kind of a scheduler. Some of them treat the scheduler as an other persona of a worker, some of them implement the scheduler as an independent process. The scheduler has several tasks. First, it matches the idle workers with an appropriate piece of work, then, it is responsible for the correct execution of side-effect predicates (maintaining the sequential semanics.) In this section different scheduling strategies will be introduced focusing on the first point.

The nodes in the search tree can be divided into two groups: shared (public) and nonshared (private) ones. In the same way, the search tree can be divided into public and private regions. Nodes in the public region are accessible to each worker in the subtree, whereas private nodes are accessible only to the worker which created them.

Basically, there are two types of scheduling strategies: the top-most and the bottom-most ones. In the case of top-most strategy one of the oldest private node of a selected worker will be shared. In such a way, quite large amount of work can be grabbed (it is relatively at a higher level in the tree, therefore the subtree represents a big chunk) thus, the granularity can be kept at coarse level. However, the transitiion between the two nodes (the one where the worker became idle and the new one) in the tree can present considerable overhead.

On the other side, bottom-most scheduling tries to share many nodes of a busy worker and allows idle workers to grab a piece of work near to their previous position. In this case the granularity is much finer. The difference between the two strategies is depicted in Figure 7-9.

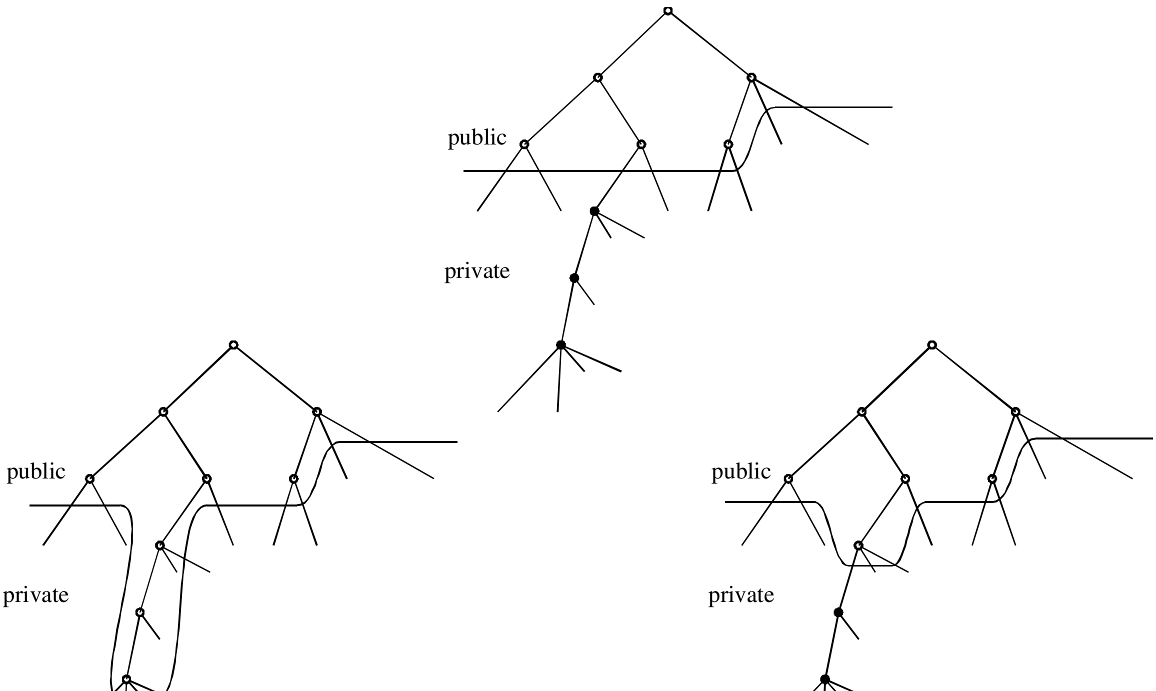


Figure 7-9. Bottom-most and top-most scheduling strategies.

7.2.2.7.1 The Muse strategy

The Muse [Karlsson92] tries to get the next piece of work, possibly the closest to the worker's current position in the tree.

1. First, it tries to find the bottommost live node (a node with untried alternatives) in the current branch. If there is such a node, the positioning to the new node is simple backtracking.
2. If there is not available work in the current branch, the idle worker must be matched with the nearest worker with the maximum load. (This is a heuristic which is hard to be supported efficiently.) The load is measured by the number of private unexplored alternatives. The worker first tries to find workers in its subtree and select those which are not closer to other idle worker. Then, within this subset, it selects one with the maximum load.
3. If there is no such a worker, the idle worker tries to find one outside the current tree. It first determines the set of busy workers that are not closer to any other idle worker. Then it chooses the nearest node which has a worker with maximum load in its subtree. The idle worker backtracks to this node N, and the situation is the same as in case 2.
4. If none of the conditions above are met, the worker remains idle.

The entire mechanism is supported by global data structures.

7.2.2.7.2 The Aurora strategies

The Aurora [Carlsson90] follows the top-most scheduling strategy and has several schedulers implemented and tested.

The Manchester scheduler assigns the newly generated piece of work to the worker closest in the search tree. To achieve it, global data structures are used. The main metrics is the cost of migraiton.

The Argonne scheduler uses local data for the decision. An idle worker in the public region can decide whether it will grab a piece of work from the current node or to move along an arc of the tree to a nearby node and repeat the decision procedure.

The Wavefront scheduler is based on the borderline between the public and private parts. New work can be found only at the youngest choicepoints in the public area thus, connecting these nodes results the so called wavefront. In other scheduling strategies, when a worker moves from one node to another, it must go through node by node a large part of the tree performing backtrack and forward movement. In the Wavefront scheduler the wavefront provides both the direct way of finding work and the information necessary for updating the binding array. It is obtained by means of a pseudo, so called join node. Workers can move along the wavefront via these join nodes.

7.2.3 AND-parallel logic computational models

7.2.3.1 Basic concepts

Let us extend the simple Prolog program in the following way, amounting a *really meaningful* program:

```
:-fruit(X, sweet, red, juicy).
fruit(X,Y,Z):- taste(X,Y), colour(X,Z), juicy(X).
taste(apple, sweet).
taste(lemon, acid).
taste(strawberry, sweet).
taste(cherry, sour).
colour(lemon, yellow).
colour(apple, green).
colour(strawberry, red).
colour(cherry, red).
juicy(lemon).
juicy(strawberry).
```

Listing 7-3

Whenever the resolvent contains more than one goal, an AND-parallel system tries to execute these goals (or some of them) in parallel. In this case taste(X,sweet), colour(X, red) and juicy(X) would be executed in parallel.

Precisely: in AND-parallel Prolog systems processes execute different goals of the same resolvent with a common instantiations of variables.

At this point no simple and spectacular execution scheme can be shown, like in the case of OR-parallel systems, since there are more AND-parallel execution models according to the greater number of solutions to problems and issues.

7.2.3.2 Problems and issues

Let us consider the fruit-classification program again. When taste(X,sweet), colour(X,red) and juicy(X) are executed in parallel, the first one will produce a set of bindings as {X=apple}, {X=strawberry}, the second goal will produce {X=strawberry}, {X=cherry}, while the third amounts {X=lemon}, {X=strawberry}. X represents the same fruit in the three cases, since the variables of the three goals are in the same environment. This is the binding conflict[3]. Obviously, the only solution in this case would be {X=strawberry}. However, it is not hard to conceive that in common cases the selection of matching binding sets, i.e. ensuring the compatibility of the binding sets, is more complex.

To tackle with this problem, there are several possible solutions proposed:

1. Only those goals may be executed in parallel where the binding conflict can be excluded. This is the **independent (or restricted) AND-parallelism**. It does not mean simply that the goals have not shared unbound variables, since, e.g. a(X), b(Y) have not such a variable, they may be dependent if X is bound to Z and Y is bound to f(Z). In this case obtaining the compatible binding set after the execution of the goals is simply a cross product of the resulted binding sets. The central problem in this approach is how the dependency among subgoals can be detected.
   a) The dependency can be annotated by the programmer in the program source. This is *the explicit goal ordering*.
   b) The dependency is tested by the system, thus the Prolog program to be executed does not differ from any other standard Prolog sources. This is the *implicit goal ordering*. The dependency check can be done fully run-time, fully compile-time, or partially run-, and compile-time.
      i) Dynamic dependency check: the data-flow dependency among subgoals is detected at run-time
      ii) Static: the dependency analysis is done at compile-time.
      iii) Hybrid: this approach tries to combine the benefits of the static and dynamic analyses.
2. All the goals can be executed in parallel. In this case a consistency check must be executed. This is the **dependent (or full) AND-parallelism**.
   a) The consistency check is static.
   b) The consistency check is dynamic.

7.2.3.3 Independent AND-parallelism

This section concerns 1.b in the previous listing. All these independent AND-parallel schemes have two basic procedures: forward execution and backward execution. In forward execution the proper execution order of goals must be obtained according to the data dependency graph. The data-dependency graph describes the potential dependencies between goals. The nodes represent goals and arcs between them show the dependant reletionship. Only those goals can be executed in parallel which have no direct arc between their nodes. This data-dependency graph can be created dynamically, at run-time, statically, at compile-time or partially run-time and partially ompile-time.

[3]At first sight this problem may be quite similar to that of multiple bindings in OR-parallel systems. However, the problem is entirely different since, in the case of OR-parallel systems different bindings were established in different binding environments whereas, in case of AND-parallel systems they are in the same environment. Thus, in the previous case different bindings of the same variable might be correct, while in the latter case they exclude each other.

Upon a failure the backward execution procedure selects the goals which must be re-executed and establishes their order thus, performing the backtrace.

7.2.3.3.1 A dynamic approach: Lin's bit vector

During the forward execution phase the data dependency graph is created dynamically by a token-passing scheme [LinKumar88]. In a clause there are as many tokens as many variables in the environment[4] are. A token of variable V is a list of goals in left-to-right order which goals will be the consumers of V (the F-list). Initially, each goal G gets a token for variable Vi, if Vi appears first in G in a left-to-right order. A goal G is said to be executable if it holds all the tokens for its variables which are not bound to a ground term. The steps of the forward execution are:

1. Process P_0 is created for the head literal and executes unification.
2. If the unification is successful, procedure S_0 is executed for generating and distributing the tokens.
3. If literal P_i receives a token for each uninstantiated variables, it turns from the initial GATHER state into EXECUTING state and starts executing the goal.
4. If in EXECUTING state P_i has generated bindings for some variable V
5. the binding environment is updated for all the literals which are consuming V
6. S_0 is executed to create tokens for the newly generated variables
7. If P_i succeeds, its state turns from EXECUTING into SOLVED

A simple example for the forward execution can be seen on Figure 7-10.

The backtrack is realised by the backward execution procedure. It is supported by the data structure called B-list[5]. The B-list contains the list of goals in right-to-left order which can offer alternative bindings to the variables whenever a failure occurs. Obviously, those goals which do not contain any common variable with the failed goal, will not be backtracked contrary to standard Prolog. The backward execution has the following steps:

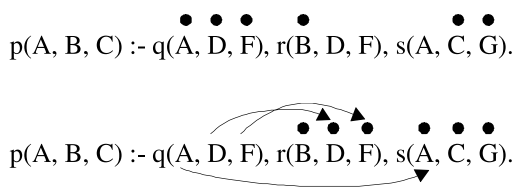


Figure 7-10. The token-passing scheme to create the data-dependency graph dynamically. As r and s got all the tokens for their variables, they can be executed in parallel.

1. Each time goal P_i consumes binding from P_k, P_k is added to B-list(P_i)
2. When Pi fails, the first goal in B-list(P_i) will be selected as a backtrack candidate (P_j), i.e. P_j will try another alternative from its choicepoint.
3. If P_j is the head, then the clause is failed
4. The tail of B-list(P_i) will be merged with B-list(P_j). If P_j can not generate the proper binding to P_i, other different bindings will be chased from both lists of generators.

There are other states introduced in backward execution:

When P_j is selected, it changes its state from SOLVED to EXECUTING. However, since P_j hopefully will produce new bindings for its variables, all goals which consumed the variables must turn from SOLVED to GATHER and wait for the tokens to re-calculate. This is obtained by a CANCEL message. This message must be sent recursively to each consumer of variables of P_j.

A RESET message should be sent to a goal instead of a CANCEL message to keep the old bindings. This situation is depicted in Figure 7-11.

---

[4]The number of variables in the scope of the clause may increase during the execution. For example, a(X,Y):- b(X), c(Y) originally has 2 variables. However, b(f(A)) binds X to the f structure and introduces the new A variable.

[5] Before the reader gets confused: the F-lists (forward) belong to variables and contain all the goals in ascending order which will consume the variable. The B-list (back) belongs to goals and contains all the goals in descending order which can offer an alternaive solution upon a failure.

The whole token-passing scheme is implemented by bit-vectors, that is, where the name of the procedure comes from.

7.2.3.3.2 A static approach: SDDA

In case of static data-dependency analysis (SDDA) the whole test is done in compile-time [ChangDesp85]. The analysis is based on worst-case assumptions and mode declarations, i.e. annotations from the programmer. These annotations for the arguments are in the following form:

clause(g,c,i), where

g tells that the given argumentum is a ground term (or a variable bound to a ground term)

i: independent, i.e. variable in this position is independent from fariables in other positions

c: coupled term: may contain variables which are bound to other variables

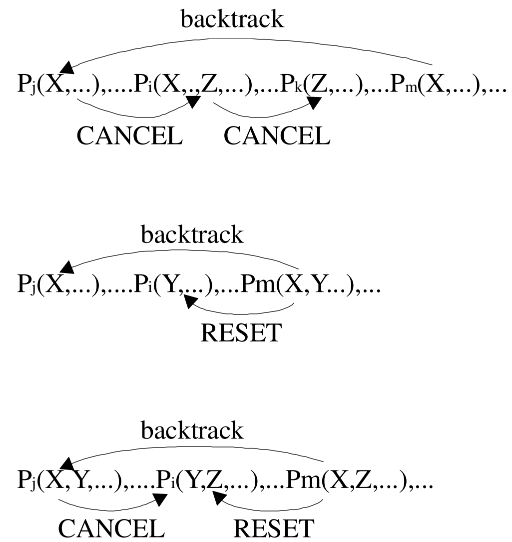


Figure 7-11. Different cases of backtracking.

Variables are classified into 3 sets:

G: {V V is bound to a ground term}

C: {V V is a member of EC, EC is an equivalence class which contains variables that may be coupled together}

I : { V V is neither in G, nor in C}

First the SDDA examines each clause, and each goal within the caluse. As a result, there are two sets for the clause: the activation and the exit modes. Upon this information the data-dependency graph can be generated. There are two lists for each goal: the preceding goals and the backtrack list. Every element in the backtrack list represents different types of backtracking. There are 3 types of them:

Shallow backtracking: the goal was called in forward execution and failed. It backtracks to the closest predecessor in the data-dependency graph. If this predecessor goal has other dependent goals, they must be cancelled.

The predecessor goal can not give another alternative, it must backtrack itself. Whenever it cancels the computation of all its dependent goals, the search space may remain unexplored. In this case the siblings will backtrace sequentially.

As it can be seen, in the SDDA scheme there is no need for any run-time check however, some amount of parallelism remain unexploited due to the worst case assumptions.

7.2.3.3.3 A hybrid approach: RAP

As it was stated above, determining the independence at compile-time is not too difficult but too restrictive due to the worst case assumptions. There is a need for some run-time checks however, most of the work should be done at compile-time.

In the RAP Prolog programs are compiled into Conditional Graph Expressions (CGE) [Degroot84]. This graph is a result of the statis analysis and only simple checks must be performed at run-time. The graph expressions consist of the following terms:

G	Execute goal G
(SEQ $E_1,E_2,...E_n$)	Execute sequentially $E_1..E_n$
(PAR $E_1,E_2,...E_n$)	Execute $E_1...E_n$ in parallel
(GPAR ($V_1,...V_k$) $E_1,E_2,...E_n$)	If all $V_1...V_k$ are ground then execute $E_1...E_n$ in parallel, else execute it sequentially
(IPAR ($V_1,...V_k$) ($X_1...X_m$) $E_1,E_2,...E_n$)	If all $V_1...V_k$ are independent from all $X_1...X_m$ then execute $E_1...E_n$ in parallel, else execute it sequentally
(IF C E_1 E_2)	If C is true execute E_1 else execute E_2

For example, a quicksort program would have been compiled into the following expressions (those knowing Occam will find it familiar):

```
qsort(L, SL):-                        (SEQ
      partition(L, L1, L2),                 partition(L, L1, L2)
      qsort(L1,SL1),                        (IPAR (L,L2)
      qsort(L2, SL2),                             qsort(L1,SL1)
      append(SL1, SL2, SL).                       qsort(L2, SL2))
                                            append(SL1, SL2, SL))
```

The operations, remaining to be performed at run-time, are these simple independence and groundness tests, which still may be of complexity $O(n^2)$ where n is the number of variables to be tested. These expressions do not define a strict execution order.

Thus, the forward execution is based on the CGEs and actually governed by the current binding values of variables. The backward execution may be more complex.

1. The failure, occured in a block of sequential goals, and the latest choicepoint are in the same block: conventional sequential backtrack must be performed.
2. The failure occured in a block of parallel goals. Since these goals are independent, the failure can not be corrected by backtracking a goal in the same block. Thus, a goal in the previous sequential block must backtrack. However, all the processes in the parallel block must abandon their work since, the preceding goal may result new bindings in their arguments.
3. The failure occured in a sequential block, and the latest choicepoint are in a parallel block. If the latest choicepoint is in goal G_i within the block then
   a) Goals G_j $j < i$ must not be affected by the backtrack according to the sequential scheme.
   b) Goal G_i takes the next alternative from its choicepoint and resumes the work.
   c) Goals G_k $k > i$ must abandon their work and restart according to the sequential scheme.

This approach tries to find a reasonable compromise and to combine the benefits of the static and dynamic aproaches. However, the run-time tests sometimes may be quite complex, and in some cases this hybrid scheme fails to find the exploitable parallelism.

7.3 Functional programming

Functional languages have many similarities, mostly they differ in syntax and a very little in semantics. Functional programs are based on Church's λ-calculus which is a calculus of functions. This gives a pure mathematical foundation to all the functional languages.

Functional languages are implemented in such a way, that the high-level functional program is deduced to λ-calculus (or - due to efficiency reasons - to variants of the λ-calculus like SK-combinators, director string terms, supercombinators), then these λ-expressions are either interpreted or compiled into machine-level instructions. Since λ-calculus is a simple, powerful and extremely well-studied language, it both allows to translate the high level languages into it and to implement a highly effective translation into machine code. This study focuses on a specific area of the latter stage, the parallel implementations of the λ-calculus.

7.3.1 A short introduction to Λ-calculus

In case of logic programming languages, the implementation issues are understandable without a precise knowledge about logic and Horn-clauses. Unfortunately, this is not true regarding the functional languages and λ-calculus.

7.3.1.1 λ-expressions

A λ-expression may consist of constants, variables, other λ-expressions and λ-abstractions. A λ-expression is denoted by upper-case letters.

- Constants may be numbers (0, 1, 2...), logical constants (TRUE, FALSE), and characters ('a','b',...).
- Variables are denoted by lower-case letters.
- 'A λ-expression may contain other λ-expressions' means that simple juxtaposition of λ-expressions are for expressing the application of a function, i.e. F E where both F and E are λ-expressions is also a λ-expression and denotes 'apply function F on argument E.'
- For practical reasons λ-calculus contains some built-in (or primitive) functions[6] (mainly arithmetical, logical ones.) λ-abstraction is the way how new (non-built-in) functions can be created. It is in the form of λ<variable>.<λ-expression >, where λ says that a definition follows, <variable> stands for the parameter of the function, '.' precedes the function body, and <λ-expression > is the function itself (like a procedure definition.) Note: the newly generated function has no name. An example: λx. + x 1 is a function of x which results in x+1.

Example:
(λx.λy. + x ((λx. - x 3) y)) 5 6 is a λ-expression as a juxtaposition of three λ-expressions: the λ-abstraction, 5 and 6. The λ-abstraction is a function of x with the body of λy. + x ((λx. - x 3) y) which is a λ-abstraction itself. It is a function of y and the body is + x ((λx. - x 3) y). It generates the sum of x and the value of the second argument. The second argument is an application of the function (λx. - x 3) to the argument y. The function is a λ-abstraction which defines the function f(x)=x-3.

7.3.1.2 Bound and free variables

One can not evaluate the λ-expression (λx. + x y) 4, since the value for x is 4, however, the value of y is unknown in this scope, it depends on the environment. The reason is that x occurs bound in this λ-expression, while y is not bound by any λ, i.e. it occurs free. An occurrence of a variable is bound if there is an enclosing λ-abstraction which binds it, and is free otherwise. Note: the terms 'bound' and 'free' refer to a specific occurrence of the variable. It may vary in different scopes of an expression.

[6]The λ-calculus in its purest form does not contain built-in functions.

Examples:

$\lambda x.\lambda y.$ + x (($\lambda x.$ - x 3) y): both x and y are bound in all occurrences.

$\lambda y.$ + x (($\lambda x.$ - x 3) y): the first occurrence of x is free, all other occurrences of x and y are bound.

+ x (($\lambda x.$ - x 3) y): the first occurrence of x and y are free, the inner x is bound.

($\lambda x.$ - x 3) y: y is free, x is bound.

- x 3: x is free

7.3.1.3 Reduction, abstraction, conversion

A functional program can be considered as an expression, and the program execution is the evaluation of the expression. Evaluation proceeds by successively selecting a reducible expression (redex) and **reducing** it. The reducible expression means that all the arguments of proper types to a given function are present. The reduction is denoted by an arrow '$\rightarrow$'.

Example:

+ (* 3 4) (- 8 3) is not a redex, since function '+' expects two numbers as input. It reduces:

$\rightarrow$ + 12 (- 8 3)

$\rightarrow$ +12 5

$\rightarrow$ 17

As it was mentioned earlier, the simple juxtaposition of a λ-abstraction and the argument denotes the application of the function to the argument. The rule of application: the result of applying a λ-abstraction to an argument is an instance of the body of the λ-abstraction in which free occurrences of the formal parameter in the body are replaced with copies of the argument. This operation is called as **β-reduction** and denoted by an arrow '$\rightarrow$' as the simple reduction.

Example:

($\lambda x.\lambda y.$ + x (($\lambda x.$ - x 3) y)) 5 6

$\rightarrow$ $\lambda y.$ + 5 (($\lambda x.$ - x 3) y) 6 (the inner occurrence of x is not free)

$\rightarrow$ + 5 (($\lambda x.$ - x 3) 6)

$\rightarrow$ + 5 (- 6 3)

$\rightarrow$ + 5 3

$\rightarrow$ 8

The β-reduction can be applied backwards yielding the **β-abstraction**. It is denoted by a backward arrow '$\leftarrow$'.

Example:

+ 12 5 $\leftarrow$ ($\lambda x.$ + x 5) 12

The β-reduction and β-abstraction are called together as **β-conversion**.

If all occurrences of the formal parameter in a λ-expression are replaced by another variable, the two expressions ought to be equal as long as the replacement is done consistently. This simple renaming is called **α-conversion**.

Example:

($\lambda x.$ + x 5) $\leftrightarrow_{\alpha}$ ($\lambda y.$ + y 5)

There are certain expressions, which are equivalent, i.e. they represent the same function although, they are literally different. The **η-conversion** is a rule for expressing their equivalence.

Example:

$(\lambda x.\ +\ 1\ x) \leftrightarrow_{\eta} (+\ 1)$

The application of built-in functions is also a form of conversion and are called as **δ-conversion**. In summary:

- Name changing: α-conversion allows to change the name of the formal parameter in a λ-abstraction.
- Function application: β-reduction allows to apply a λ-abstraction to an argument
- Eliminating redundant λ-abstractions: η-conversion can sometimes eliminate a λ-abstraction.
- Applying built-in functions.

7.3.1.4 Normal forms

If there are no more redexes in an expression, the reduction can not proceed, and the expression is said to be in **normal form**. Thus, the aim of the evaluation is reaching the normal form.

Not every expression has a normal form, for example:

$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \rightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \rightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$, etc.

An expression with no normal form is denoted by ⊥ ('bottom'.)

If an expression contains more redexes, any of them can be chosen for reduction. This results in several alternative reduction paths. Some of them may reach the normal form, some of them will not. This will be explained in detail in .

However, in some cases there is no need (or no possibility) to reduce all the expressions to their normal form. The **weak head normal form (WHNF)** is a λ-expression containing *no top-level* redex, i.e. $F\ E_1\ E_2\ E_3..E_n$ ($n \geq 0$) is in WHNF if either F is a variable or data object, or F is a λ-abstraction or primitive (built-in) function and $F\ E_1\ E_2\ E_3..E_m$ is not a redex for any $m \leq n$.

A λ-expression is in **head normal form (HNF)** if it is a λ-abstraction of form

$\lambda x_1.\lambda x_2....\lambda x_n.M_1\ M_2\ ...M_m$ ($m,n \geq 0$) and M_1 is either a variable or data object or $M_1\ M_2\ ...M_p$ is not a redex for any $p \leq m$.

Everything in HNF is in WHNF but not vice versa.

7.3.1.5 Strictness

A very important property of a function is whether it really needs a certain argument or not. This question is especially important at optimizing compilation and also at controlling parallelism.

If a function f needs an argument x which evaluation will not terminate, then, obviously the application of f to the argument will fail to terminate.

A function f is **strict** if and only if

$f\ \perp = \perp.$

This can be extended to multiple arguments: f is strict in its second argument if and only if

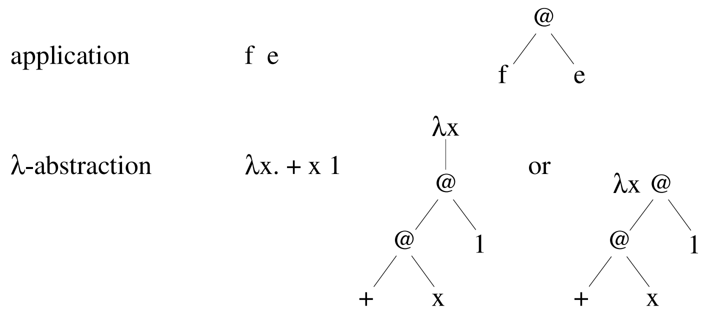
$f\ x\ \perp\ z = \perp.$

It means that in such a situation the application of f could terminate only in that case if the ⊥ argument was eventualy not needed, i.e. f was not strict.

7.3.1.6 Expression representation: abstract syntax trees

The expressions to be evaluated can be depicted as a tree and, in fact at program level they are really represented and treated as graphs.

The leaves are constant values, built-in functions or variables. The application is denoted as an @ node in the graph, and has two descendants: the *rator* (the function to be applied) and the *rand* (the argument). The elements of a syntax tree are depicted in Figure 7-12.. An example can be seen in Figure 7-13.



Variables, numbers, built-ins are always leaves.

Figure 7-12. Elements of the syntax tree.

One might wonder why the application has only one arc to the argument, since there are functions with multiple arguments. This technique is called *currying*, and it can be shown, that all functions can be treated as a series of single argument function where the arguments are substituted in successive steps.

Example: + 5 6. '+' is a function with two arguments. However, it can be seen as (+ 5) 6, that is (+ 5) is a new function, which adds 5 to its only argument. By such a 'virtual' function the addition and other functions with multiple arguments can be considered as a single argument function.

λx.λy. (+ y x) x 4

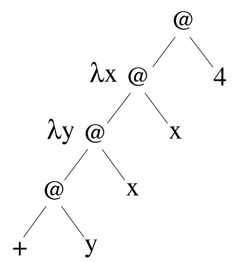


Figure 7-13 A simple syntax tree.

7.3.1.7 Towards implementation: combinators, supercombinators, SK-combinators and director strings

Although λ-expressions are powerful enough to express all the features of functional languages, for efficiency reasons, mainly for the ease of compilation, there are extensions introduced to λ-calculus.

A **combinator** is a λ-expression which contains no occurrences of a free variable. This means that the value of the expression depends only on the arguments and not on the free variables (environment).

A **supercombinator** $S of arity n (n≥0) is a λ-expression of the form

$\lambda x_1.\lambda x_2....\lambda x_n.E$,

where E is not a λ-abstraction, $S has no free variables and any λ-abstraction in E is a supercombinator. It simply means that neither $S nor any of the λ-abstractions within E do not contain any free variables. Every supercombinator is a combinator but not vice versa.

One of the most frequent and quite complex operation is constructing an instance of a λ-body. The usage of supercombinators allows to compile this operation into a fixed sequence of instructions thus gaining speed. It also gives the possibility to simultaneously instantiate multiple arguments.

λ-expressions can be easily converted into supercombinators by means of *λ-lifting*.

The **SK-combinators** are a limited set of supercombinators: S, K, I, where I=λx.x, K=λx.λy.x and S=λx.λy.λz.((x z)(yz)). Surprisingly, all the programs may be converted into these supercombinators, and even I can be expressed by S and K. They have their own very special theory which is completely out of the scope of this study.

The **director string terms** are a result of compilation, and they can be considered as signposts in the expression graph. At substituting the bound variables, the incoming argument follows these signposts, thus reaching the position of bound variables. This mechanism is depicted in Figure 7-14.

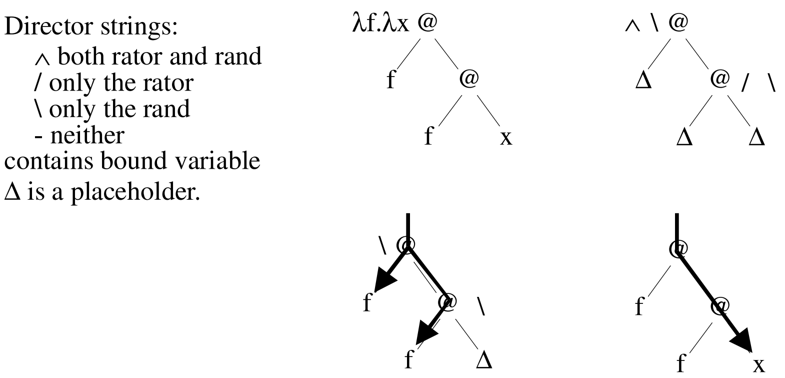


Figure 7-14. Elements of director string terms. The upper left abstract tree is translated into the upper right graph. The lower row shows how arguments are directed by the strings.

7.3.2 Reduction machines

Machines for interpreting program expressions based on the λ-calculus are known generically as reduction machines.

In the design stage of a reduction machine there are two most important issues: the evaluation strategy and the argument handling. The first is concerned with the selection of the redexes to be evaluated, while the latter is regarding how the arguments are substituted into the body. These, and other issuses basically affect the efficiency of implementation and the amount of exploitable parallelism which is the subject of this study.

7.3.2.1 Evaluation strategies

7.3.2.1.1 Eager and lazy evaluation

The imperative languages evaluate their arguments before the call to the procedure, although the argument probably will not be consumed in the procedure body. This is the *call by value* scheme. However, in case of functional languages it is possible to postpone the evaluation until the given argument is really needed. This is the *call by need* semantics. Some functional languages follow the first, **eager evaluation** scheme, while some of them are based on the latter, **lazy evaluation**. The lazy evaluation gives a greater freedom of data representation (and some researchers consider it as a basic feature of functional languages), however the price for it is the execution speed.

A lazy evaluation has two important features:

- Arguments to functions should be evaluated only when their value is needed, not when the function is applied.
- Arguments should be evaluated at most once.

7.3.2.1.2 Selecting the next redex

Selecting the next redex is a critical point both from theoretical and practical point of view even in a sequential implementation, and it basically influences the way the parallelism may be exploited in a parallel implementation.

The evaluation of an expression consists of successively reducing redexes until the expression is in normal form. If there are more redexes at a certain point of reduction, any of them can be chosen. However, not every expression has a normal form, and some reduction sequences may reach the normal form, some others do not.

According to the Church-Rosser Theorem,

- at most one normal form exists to a given expression,
- if there is a normal form, then it can be reached by the normal order reduction.

The **normal order** reduction specifies that the leftmost outermost redex should be reduced first.

This means in summary, that other reduction order than the normal order either will not terminate or will give the same normal form as the normal order reduction would. There is no way to reduce an expression to different normal forms by different reduction orders.

Thus, one possible selection scheme could be the normal order reduction which guarantees that the normal form will be reached if it exists. However, it does not guarentee that the normal form will be reached in the fewest possible number of reductions. This reduction order is *strictly sequential* and has *a call-by-need semantics*, since arguments are not evaluated before they are really needed in the body of a function.

The other scheme is the **applicative** reduction, where each argument must be reduced to normal form before they are passed to the function. This has a *call-by-value* semantics. Since all the arguments must be evaluated before the actual call, these arguments could be reduced in parallel, thus the applicative evaluation offers an opportunity for parallelism.

The lazy and eager evaluations represent the extremes of a range of reduction strategies as well as the normal order and applicative reductions do. The relation between them can be depicted in .

There are other reduction schemes and orders in this range, however, as it can be seen in the case of the previously introduced two strategies, usually there is a relation between the reduction order and the laziness or eagerness of reduction.

7.3.2.2 Argument handling: string reduction and graph reduction

At substituting the argument as a formal parameter into the body, there are two possibilities:

The complete argument is copied into the body for every occurrence of the bound variable. This is called **string reduction**, since it can be considered as if each bound variable was substituted by the argument string. However, the argument might be very large, and thus, copying is a waste of space. Second, the argument may contain redexes, which will be reduced as many times, as many copies were made.

Instead of copying the whole argument, only a pointer to it is substituted to the bound variable. It is called as **graph reduction**. It allows to share the arguments, and there may be several pointers to the same expression. This is the reason why the expression tree may become a graph. In this scheme any reductions within the argument are performed only once, and benefit all recipients of the result of the argument[7]. However, after a reduction step graph fragments may remain in the storage space (it can not be removed, since other expressions might reference to it), therefore a garbage collection procedure must be introduced. Recently there are garbage collection techniques which solve the problem on-the-fly thus, the only drawback of graph reduction can be eliminated.

The graph reduction better fits the lazy evaluation, while the string reduction can be viable in an eager reduction scheme, since there are not large structures to be copied. However, even in this case there are some drawbacks, and string reduction is not a real alternative to a sophisticated graph reduction. In the following sections the parallel graph reduction will be examined.

7.3.3 Parallel graph reduction

One of the features of functional languages is that they are not inherently sequential. At a moment there may be a number of redexes, and they could be reduced simultaneously. The parallel execution of functional languages may be possible without adding any new language constructs.

1. Graph reduction is an inherently parallel activity. At any moment the graph may contain a number of redexes and it is very natural to reduce them simultaneously.
2. Graph reduction is an inherently distributed activity. A reduction is a local transformation of the graph and no shared bottleneck (e.g. environment) need be consulted to perform a reduction.
3. All communication is mediated through the graph.
4. The entire state of the computation is well defined - it is the current state of the graph.

[7]In fact, even in a graph reduction scheme there is some amount of copying, since function definitions, e.g. λ-abstractions can be considered as templates, and since in different occurences they may be applied to different arguments, each time a new copy, i.e. a new instance must be made.

7.3.3.1 Models for parallel reduction

Sequential reduction machines take a pointer to the root of the expression (program) to be evaluated then, according to an evaluation strategy it performs successive reduction steps until the expression is in WHNF.

The simplest form of parallel execution is a generalization of this execution scheme. There is a number of evaluator tasks (or agents) working simultaneously on the graph. If a task requires a value which is a result of computation, it may trigger another task to evaluate a certain subgraph. The newly started task may also create new tasks, etc. If a task needs the value, it may take it if it is already present, or the task has to wait until its child task completed the evaluation. Each evaluator task is active until the assigned expression is in WHNF.

There may several problems and questions arise:

- when should a new task be started?
- how to synchronise the tasks?
- how to assign an expression to a task?

In the next sections these questions will be answered.

7.3.3.2 Task control

7.3.3.2.1 Conservative parallelism

One strategy may be that a new task is created only if its value is really needed. For example it is sure that in (+ x y) both values will be required, that is, evaluation of x may be assigned to a task, while the evaluation of y to another. However, in (IF c THEN x ELSE y) it is unpredictable whether x or y will be eventually needed, so, according to the rule of conservative parallelism, they will not be evaluated in parallel.

Recall that the property of a function that it needs an argument is called strictness. It is easy to detect whether a built-in function is strict or not, therefore its arguments may be evaluated in parallel or not, but it is not obvious for the other functions. There are techniques for obtaining the strictness of a function. By abstract interpretation, i.e. by program text analysis, the strictness of a function can be determined. Upon this information the syntax tree can be annotated thus, during the execution there is a prompt for the tasks whether to start an evaluation or not.

Conservative parallelism is safe in a way that, no work is done in vain, but in some cases it fails to exploit the available parallelism.

7.3.3.2.2 Speculative parallelism

Speculative parallelism is the opposite of conservative parallelism: tasks may be activated even if their result turns out to be irrelevant. The extreme speculative strategy is to start a task for every node in the graph, or to regard any redex in the gaph for reduction. Of course, conservative and speculative parallelism represent the extremes of a wide range again. There are (and usually these are implemented in practical applications) combinations of these strategies which have some conservative feature as well as some speculative.

The speculative strategy increases the opportunity of parallelism. However, large amount of resources may be allocated for evaluator tasks of such type which result will be discarded. Recall, that the reduction of some expressions will never terminate. In some cases it may cause serious shortage of resources, and the evaluation of the really required arguments is delayed.

The speculative parallelism can be controlled to find an optimal performance. Tasks can be divided into two groups: vital tasks and specultaive task. The results of the vital tasks are known to be needed, while it is uncertain whether the results of speculative tasks will be needed or not. Vital tasks have a higher priority, and speculative tasks are triggered only if there are resources in excess. Depending on the certain situation this control could lead to conservative parallelism where there is no speculative work at all, or, in other cases could give rise to greater amount of speculation.

A speculative task may promote to vital when it is discovered that its result is needed. On the other side a speculative task can be discarded when its result turns out to be unneccessary.

7.3.3.2.3 Amount of parallelism

It must be kept in sight that the aim is a better performance and not the parallel execution. Parallel execution is the way to achieve the goal. Thus, it is a wrong conclusion from the previous sections that the more parallelism appears in execution the better execution strategy was found.

The major source of parallelism is the algorithmic parallelism, i.e. the problem described by the programmer inherently contains parallel activities. If so, sufficient conservative parallelism should be found. However, it strongly depends on the system and the strictnes analyzer. It is a real risk that conservative parallelism fails to exploit all the potential parallelism in the program.

On the other side, too much parallelism is also dangerous. It can cause resource-management problems, and as it was described regarding the speculative parallelism, the overall performance may dramatically drop.

7.3.3.2.4 Granularity

In the previous section the amount of parallelism, i.e. the number of tasks was investigated. However, it is also a very important question that what is the size of a task.

Obviously spawning a new task is a costly operation, it represents an overhead to the system. If the amount of work done by the task is comparable to this overhead, then a considerable fraction of resources is wasted for system administration. This overhead begin to dominate when the grain of parallelism is too fine. However, on the other side coarse-grain parallelism may cause too little concurrency, and some of the agents may remain idle.

7.3.3.2.5 Task interaction

It may happen that two tasks try to evaluate the same piece of the graph, since it is a basic feature of syntax graphs that nodes may be shared. Since reduction steps are atomic operations, i.e. the operation can not be aborted and before and after the operation the graph is in consistent state, the multiple access will not cause any harm to the computation, but the repeated evaluation of the same graph is a waste of work.

Thus, a blocking mechanism was introduced. When a task starts to evaluate an expression, it first goes down to the left-most bottom-most leaf, which should be a function. (This is called as unwinding the spine.) While it moves down, it marks all the nodes. These marks block the other task whenever it tries to access the same expression. When the first task finished the evaluation, the root node of the expression will be replaced by the result, and the marks will be removed. At this moment the second task can proceed, however, instead of a whole expression it will find only the result. Recall, that by the blocking mechanism one of the ingredients of lazy evaluation is restored into the parallel reduction: everything is evaluated at most once.

The blocking mechanism can be optimized further. If an expression is in WHNF already, the blocking of a task is in vain, since the graph will not be altered, i.e. tasks have read only access to it. This suggests to introduce a new kind of application node: the WHNF application which will not be marked whenever a task is working on it. Hence, if an expression contains a redex, only one task will have access to it, however, if an expression is known to be in WHNF, many tasks may have simultaneous access to it.

7.3.4 A case study: CTDNet2

CTDNet2 is a reduction mechanism for embedded systems as a successor of the CTDNet. Although the aims of the two projects were different, CTDNet2 inherited some of the basic features of CTDNet such as process-based implementation scheme, director string notation, evaluation based on applicative reduction, augmentation of the applicative framework with lazy evaluation subsystem.

The most important new features of CTDNet2 are:

- optimised graph reduction
- garbage collection
- structured data (lists)
- proper handling of conditionals
- recursion is handled directly

7.3.4.1 Process structure

The process structure of CTDNet2 differs from the usual process strucure in other reduction mechanisms as it was introduced in the previous sections. While in the ususal schemes tasks are traversing the graph, in CTDNet2 the garph itself is a network of tasks.

As it was shown earlier, high level functional languages can be compiled into a lower level language where the program is represented as a binary tree called λ-graph. Programs in CTDNet2 are represented as a set of stored, named definitions (graphs) and a principal application. While definitions remain unchanged, the principal application will be overwritten by a value, which is the result of the program.

In CTDNet2 the λ-graph is a process graph, i.e. each application node in the graph is a process. Each process has a link to an ancestor and to a left (rator) and a right (rand) children. A child process is a root of a subtree representing an expression. An example can be seen in .

There are two types of nodes: active and passive. The passive graphs are the definitions. They may be activated by request of an active process. The activation means creating a copy of the definition graph, i.e. instantiating it.

Nodes are uniquely named. Uppon instantiating the active process gets the name of its template graph suffixing with a digit.

In CTDNet2 the director string terms have been implemented, i.e. a string expression is assigned to each application node leading the incoming arguments. The place of bound variables is marked with placeholders.

7.3.4.2 Garbage collection

Since one of the basic features of a graph reduction system is sharing the objects, it is a central problem how to detect when an object is not referenced anymore by other processes and may be removed. This is the goal of the garbage collection. Usual techniques are pre-emptive, i.e. the computation is halted while the garbage collection takes place. There is an on-the-fly garbage collection technique called weighted reference counting which allows to keep the memory garbage free without interrupting the execution. This garbage collection has been implemented in CTDNet2.

In this scheme all newly created pointer is assigned a weight so that the sum of the weights of the pointers is equal to a counter in the shared object. If a pointer is copied, it shares its weight with the newly created pointer thus, the sum of weights remain constant, the shared object need not be modified.

On the other side, when a pointer is destroyed, the shared object decrements its counter by the weight of the destroyed pointer. The count decreases monotonically to zero. Zero counter means that the object is not referenced by any other processes and may be safely purged.

It also may be easily detected when only one process points to an object: the weight of the pointer equals to the counter. This feature allows to optimize the instantiation process which will be discussed in .

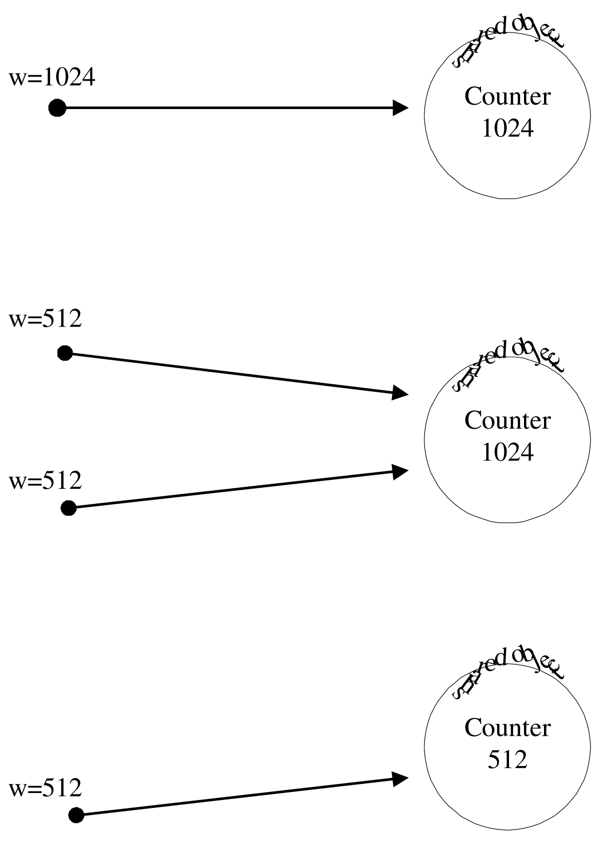


Figure 7-15. Initially one process points to the object. Upon copying the pointer, the weight is shared between the original and the new pointer. When a pointer is destroyed, the counter is decremented.

7.3.4.3 Reduction rules

As it was emphasized in the introduction, although in λ-calculus a redex is any reducible expression, it can basically influence the power and efficiency of the system which of the redexes will be reduced next. Thus, a redex is only potentially reducible, it will be reduced when it conforms to the strategy, i.e. it becomes strategically reducible. So, a graph may be in strategically normal form, although it still contains redexes.

As it was stated earlier, applicative reduction order better fits to parallel execution. However, applicative reduction order may result in expression which are in normal form and still contain free variables. This may lead to complexification, since the value of such a variable depends on the context[8]. In CTDNet2 a slightly lazier selection strategy is used which allows to postpone the evaluation of a redex until all of its free variables are bound. It is a form of context-free applicative reduction (CFAR.)

7.3.4.3.1 Context-free applicative reduction

A context-free expression is a λ-expression containing no free variables and which is not an abstraction. A context-free expression is a combinator, however, not all combinators are context-free. In director string terms, a context-free expression is one having an empty null director in the root node. This means that all the variables in the expression have been conveyed to their place.

In CTDNet2 a restricted form of CFAR, the innermost context-free applicative reduction (ICFAR) has been implemented.

7.3.4.3.2 The ICFAR strategy

A redex is strategically reducible in an ICFAR strategy if and only if it is context-free and both the rator and the rand are strategically normal under an ICFAR strategy.

The benefit of ICFAR strategy in CTDNet2 is that the criteria for s-reducibilty can be determined locally by examining each node in the graph as:

- a node is ICFAR-normal if it is normal.
- a node is ICFAR-normal if and only if its rator and rand are ICFAR-normal and it is not ICFAR-reducible.

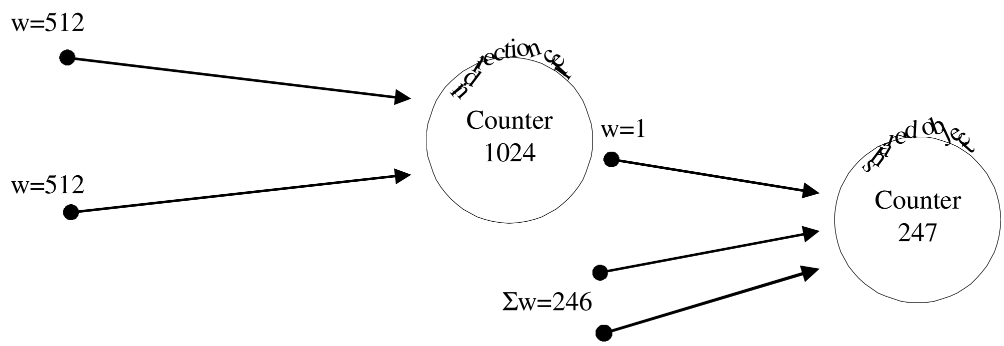


Figure 7-16. There may be cases when the weight can not be shared, e.g. it is 1. In this case an indirection cell is introduced.

7.3.4.4 Reduction cycle

Recall, that there are two types of nodes (processes) in the system: active and passive ones. Expression definitions are stored as passive processes. However, only active processes may reduce. Active processes pointing to a definition (passive process) are activating it, enabling to evaluate the redexes it contains. For the sake of efficiency, the principal application is pre-activated by the compiler.

Thus, the reduction starts with the pre-activated root which may activate other processes if they are not ICFAR-normal. Activation means to generate a copy (instance) of the passive template. While the

[8]This problem arises also in other implementations. A number of other reduction machines use some kind of combinators to eliminate the free variables.

definition graphs are shared, these copies are initially unshared. Expressions containing no bound variables are never copied. The instantiation mechanism is lazy.

There are two types of reduction: β-reduction and δ-reduction. The latter is the application of primitives (built-in functions.) Arguments are presented in an ordered sequence to the primitive. There is a counter assigned to the primitive counting the already consumed arguments. When the counter reaches the arity, the primitive is said to be ready and may be substituted by the result. This is the currying introduced in .

The β-reduction will be introduced by a series of examples which covers the most cases.

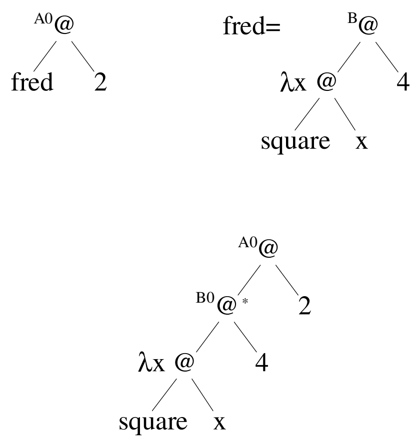


Figure 7-17. Initially process A0 is active pointing to a passive definition. By copying the definition graph it becomes active.

7.3.4.4.1 Unshared active rator

The fact that the rator is unshared means that no copies needed, the arguments may be substituted directly into the body. The first argument (3) dissipates through the graph following the director strings. The argument wavefront is called the β-wave. The β-reducing process (in this case the second node) sends its right child (argument) and parent pointer to its left child (abstraction body.) Then the process vanishes. This yields the second graph in Figure 7-18.

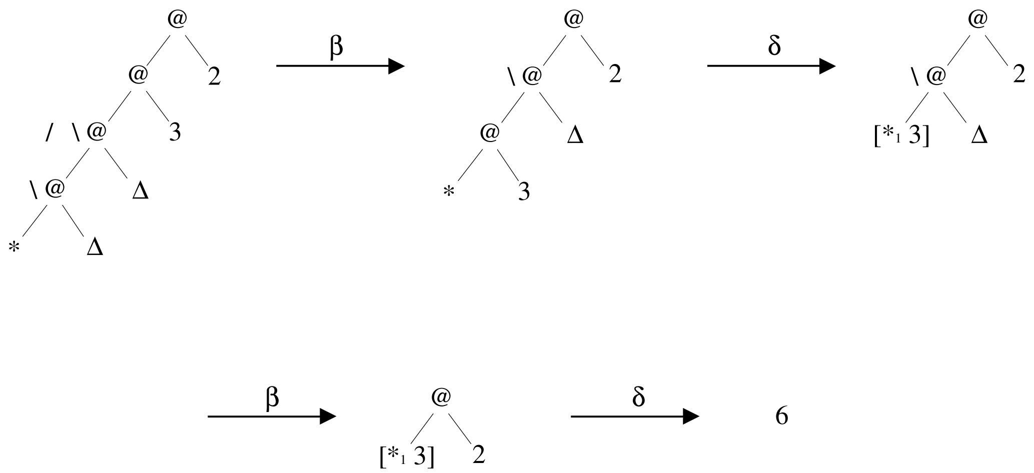


Figure 7-18. Reduction steps of an unshared active rator.

The next step is a δ-reduction: applying the multiplication to argument 3. After the reduction the completeness (counter) of multiplication becomes 1 indicating that one argument has been processed. The result can be considered as a new function of arity 1 which multiplies its argument by 3.

Next a β-reduction follows and finally the result is obtained by a δ-reduction. The whole graph is substituted by a constant value which is the result of evaluation.

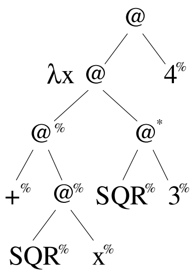


Figure 7-20

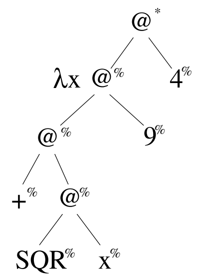


Figure 7-19

Note, that ICFAR-normality is lost by all nodes traversed by the argument during the β-reduction. However after the reduction step, the normality appears on the leaves and spreads upwards. The ICFAR-normality wavefront is depicted in Figure 7-20 and Figure 7-19. The ICFAR-normal nodes are marked with '%' while the next redex is marked with '*'.

7.3.4.4.2 Passive rator

In this case the pre-activated principal application points to a definition, i.e. to a passive graph (Figure 7-21.) The passive graph must be activated thus, a copy is made of it. This is called instantiation. While the definition is named as 'B', its instance is named as 'B0'. The newly instantiated process is unshared so, the reduction proceeds as in the case of unshared active rator.

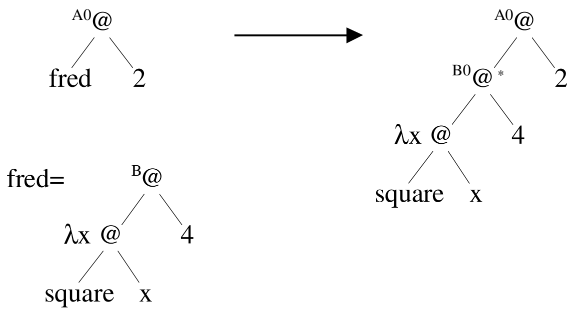


Figure 7-21. Activation of a passive rator.

Since the expression denoted as fred points to another definition, this activation or instantiation will repeated during the execution. The instantiation is fully lazy.

7.3.4.4.3 Unshared active rand

In Figure 7-22. the P1 denotes an active graph which is initially unshared. The unshared active rand (P1) and the pointer to it is given an initial weight, W. This weighted pointer is distributed to the rator and substituted into the bound variables during the β-reduction. When the pointer is duplicated (it goes through a node with a director string '∧') the weight is divided between the template and the new pointer. In this example the unshared active rand became a shared active rator.

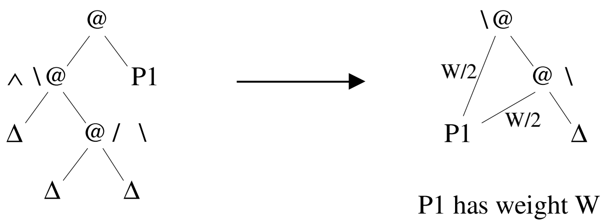


Figure 7-22. The initialy unshared active rand (P1) becomes shared active rator by substituting the argument.

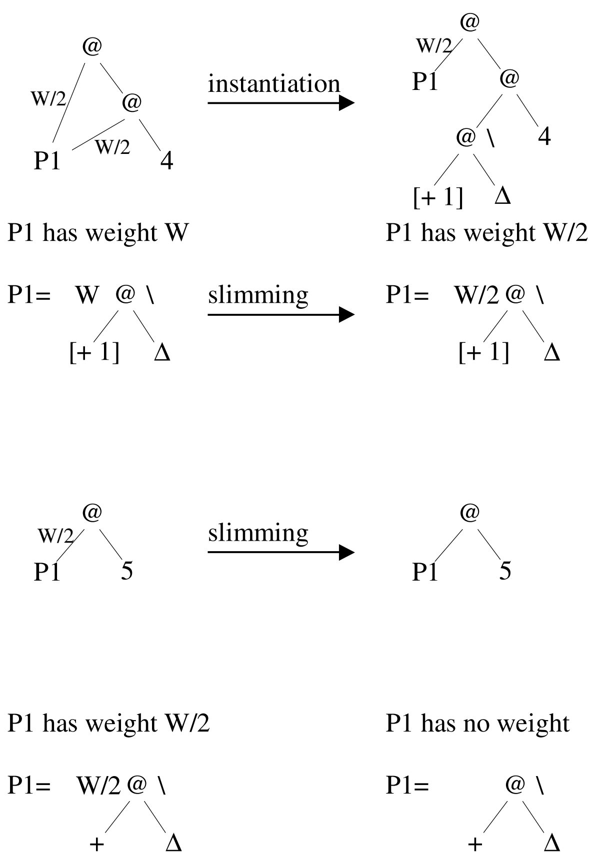
7.3.4.4.4 Shared active rator

In Figure 7-23. the reduction of a shared active rator is depicted. The first stage may be the result of substituting an unshared active rand as it was shown in the previous section.

The eligible redex has a shared rator. In general, a shared graph must be copied before use, since the bound variables of different instances may be substituted by different values. The mechanism is similar to the activation of a passive graph.

The instantiation process can be seen in the first stage of Figure 7-23. First the shared rator slims, i.e. it decrements its counter by the weight of the pointer from the root of the selected redex. If the resulting weight is non-zero, it means, that the rator is used by another expression thus, it must be copied.

However, if there is only one pointer to the rator as it is the case in the second stage, after the slimming the weight of P1 is 0, so, there is no need to instantiate it, arguments may be distributed directly to the rator. Furthermore, after the evaluation, the P1 may be discarded and purged from memory.



@
W/2
@
W/2
P1
4
instantiation
@
W/2
P1
@
@ \
4
[+ 1]
Δ
P1 has weight W
P1 has weight W/2
P1= W @ \
[+ 1] Δ
slimming
P1= W/2 @ \
[+ 1] Δ
@
W/2
P1
5
slimming
@
P1
5
P1 has weight W/2
P1 has no weight
P1= W/2 @ \
+ Δ
P1= @ \
+ Δ

Figure 7-23