Marker-Passing Inference in the Scone Knowledge-Base System*

Scott E. Fahlman

Language Technologies Institute & Computer Science Department
Carnegie Mellon University
Pittsburgh PA 12517, U.S.A.
sef@cs.cmu.edu

Abstract. The Scone knowledge-base system, currently being developed at Carnegie Mellon University, implements search and inference operations using a set of marker-passing algorithms. These were originally designed for a massively parallel hardware architecture but now are implemented completely in software. The algorithms are fast, relatively simple, and they support efficient implementation of the most heavily used KB features. This paper describes these marker-passing algorithms, their strengths and limitations, and how they are used in Scone.

1 Introduction

Scone [1] is an open-source knowledge base (KB) system being developed in the Language Technologies Institute of Carnegie Mellon University. Scone is implemented in Common Lisp. It runs stand-alone or as a server process on a 32-bit or 64-bit Linux workstation. It can also run stand-alone under Windows.

Our goal is to make Scone a practical KB system that can be used as a component in a wide range of software applications. Therefore, we place primary emphasis on Scone's expressiveness, ease of use, scalability, and on the efficiency of the most commonly used operations for search and inference.

Scone differs from other knowledge-base systems in the way it implements search and inference. Scone uses marker-passing algorithms originally designed for a hypothetical massively parallel machine, the NETL machine [2]. These marker-passing algorithms cannot, by themselves, perform every kind of search and inference that can be handled by a general theorem-prover, and the Scone operations provide no guarantee of logical completeness. However, Scone's marker-passing algorithms are fast, and they can handle the most common search and inference operations needed for common-sense reasoning in a knowledge base. These include inheritance of properties, roles, and relations in a multiple-inheritance type hierarchy; default reasoning with exceptions;

* Development of Scone has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract number NBCHD030010. Thanks to Alicia Tribble and Benjamin Lambert for help in polishing this presentation.

J. Lang, F. Lin, and J. Wang (Eds.): KSEM 2006, LNAI 4092, pp. 114 – 126, 2006.

detecting type violations; search based on set intersection; and maintaining multiple, overlapping world-views at once in the same KB.

To handle more complex reasoning tasks, we can build a general reasoner or theorem-prover on top of Scone's basic inference machinery. If we do that, Scone's marker-passing operations play a supporting role, providing a fast way to perform many of the low-level steps required by the higher-level reasoning system.

This paper describes Scone's marker-passing algorithms in greater detail, along with their application in a variety of KB tasks. It describes some of the strengths and limitations of the marker-passing approach and presents timing measurements for typical search and inference operations in a Scone knowledge base with 10^6 elements.

2 Marker-Passing Operations in Parallel Hardware

The idea for the massively parallel NETL architecture was born about 1974, while I was pondering two problems that seem to lie at the very heart of AI:

- *In any knowledge base, the amount of knowledge virtually present is very much greater than the amount of knowledge explicitly present. The extra knowledge is the result of query-time inference, which can require a lot of computation. And yet, we humans routinely perform this kind of inference quickly and in a way that seems almost effortless. We somehow do this in a knowledge base with millions of items (at least), using millisecond-speed "hardware". We're not even aware that inference is going on unless someone points this out. We don't have the same sense of mental effort that we feel when adding numbers or doing a logic puzzle.*
- *We humans also have a remarkable ability that is central to all recognition tasks: we begin with a set of observed features, a set of expectations, and a vast collection of stored descriptions; the problem is to find the stored description that best matches these features and expectations. This core operation, involving search and matching, is essentially the same whether we are talking about visual recognition, speech, or recognizing what task someone is working on after observing a few actions. Again, this is a computationally demanding task that we humans do frequently, quickly, and with no sense of mental effort.*

I came to believe that these two mysterious human abilities were related, and that they could only be explained by making effective use of the brain’s massive parallelism. So if we want an AI system that can hold vast amounts of symbolic knowledge and that can do these kinds of search and inference tasks in real time, we must develop an appropriate parallel architecture and figure out how to use it.

The NETL architecture was developed to satisfy these needs. NETL was inspired in part by the work of M. Ross Quillian [3]. He proposed storing knowledge in the form of a *semantic network*, a sort of active memory with nodes representing concepts and links representing the relations between them. In his "spreading activation" model, markers flowed in parallel through all the links of the network, looking for the shortest paths between one concept node and another. NETL is similar in structure, but it uses several distinct markers, and they flow only through certain types of links under the precise control of an external, serial *control computer*. This change allows NETL to use marker propagation for more complex forms of inference.

Every node in NETL is represented by a very simple processing element that has storage for some number of *marker-bits* – typically between 16 and 32. There are a few permanently-set bits that tell the node what kind it is: an individual-node, a type-node, or some more exotic type. Each node also has a *tie-point* to which any number of links can be attached – I will say more about that below.

These nodes are all connected to the control computer by a common bus, and they can respond in parallel to simple commands like the following:

- *All nodes: turn off marker 4.*
- *All nodes with markers 1 and 2 on and marker 3 off: turn on marker 4.*
- *All nodes with marker 4 on: queue up in serial order and report your identities to the control computer.*

Every link in NETL is also a simple hardware element, and also receives its commands from the common bus. A link has several *wires*, each of which can be tied to the tie-point of any node in the knowledge base. This is a private, non-shared connection. We may think of it as physically connecting the wire to the tie-point, though in practice the connection would be established via a switching network. A generic link (in the current Scone model) has five wires: A and B (for the two concepts the link is relating), C (used in trinary relations), PARENT (what kind of link am I?), and CONTEXT (whose purpose we will describe in section 7). Some special link-types are built-in and have special meaning to the inference machinery: is-a, eq, cancel, map, and split. We will describe these below.

In addition, each link has a built-in node, complete with marker memory and a tie-point. This node, referred to as the link's *handle node*, represents the statement itself. Other links can connect to this handle node, providing meta-information about the statement: where the information came from, how certain we are, etc.

Links can sense and alter the marker-state of the nodes attached to their various wires, so they can respond, in parallel, to commands like the following:

- *All is-a links: if the node on your A-wire has marker bit 3 on and the node on your B-wire has marker 3 off, mark the B-node with marker 3.*
- *If any link took action in the previous cycle, report that on the common bus.*

The effect of the first operation is to propagate all 3-markers one level up the is-a hierarchy; the effect of the second operation is to test whether any new nodes were marked, in which case the upward-propagation step should be repeated until all superior nodes have been marked.

This massively parallel architecture can perform certain operations very fast, even as the knowledge base grows to millions of nodes and links. For example, suppose we want to mark all the gray mammals that live in Africa. We can put marker 1 on the "mammal" node and, in a few cycles, mark all the subtypes and instances of "mammal", even if there are thousands of these because of downward branching. Similarly, we can mark all the gray things with marker 2 and all the African residents with marker 3. Then, in a single cycle, we intersect these three sets by telling every node with markers 1, 2, and 3 to turn on marker 4. The 4-marked set can then be used for other operations, or its members can be reported, one by one, to the controller.

Note that only the query-time or search-time operations are fast. It may be a slow operation to add new knowledge to the KB, since this requires connecting together new nodes and links. But that seems like a good trade-off for most KB applications.

3 Pseudo-Parallel Marker-Passing Operations in Software

In the early 1980's I tried to find an economical way to build a parallel NETL machine big enough for research on “common sense” reasoning and natural language understanding. The initial goal was to create a machine that could directly implement (or efficiently simulate) 10^6 parallel NETL elements, each representing an entity or statement. I finally came to the conclusion that achieving this with the technology of that time would be impractical, especially in a university environment with limited funding. So I set this goal aside and turned my attention to other research challenges.

Daniel Hillis at MIT, and later at Thinking Machines Corporation, did make a serious attempt to implement a marker-passing machine of this type. The result was the Connection Machine [4]. But that machine took years to develop and ultimately was so expensive that few AI researchers had regular access to one. Other researchers continued to explore the parallel marker-passing approach as well. Among the most prominent were Dan Moldovan and his colleagues [5, 6, 7] and James Hendler [8, 9].

In 2000 I began to think again about the need for a practical, large-scale knowledge-base system, both for AI research and for a number of practical applications. I realized that readily available computers were now 10,000 times faster than they were in the early 1980's, and their memories were 10,000 times larger. So I began to think about implementing a NETL-like system purely in software, running on a standard workstation with enough main memory to hold the desired KB. It took some time to get this project funded and under way. Scone is the result.

I decided that Scone would retain NETL's marker-passing model, but implemented in carefully optimized software. I refer to this as the *pseudo-parallel* layer of Scone. Scone also includes a considerable body of conventional (*i.e.* less performance-critical) software built on top of this pseudo-parallel base.

It may seem like a strange decision to organize the system in this way. Even if we accept that marker passing is a good way to implement a symbolic knowledge base in parallel hardware, why would we emulate this parallel model on a serial machine? There are several reasons:

- The pseudo-parallel layer is a small, relatively simple body of code that can be carefully tuned for maximum performance.
- By basing Scone's inference on marker-passing operations, rather than on some form of resolution theorem-proving, we lock Scone into a certain part of the design space: inference is fast, following pointer chains only to relevant items in memory. There is no need for global pattern-matching in the inner loops of the program.
- Because Scone's inference algorithms are not trying to guarantee logical completeness, the damage caused if some subtle inconsistency sneaks into the KB is localized. If we say that John is a male and later assert that he is someone's mother, the "John" description may become confused, but Scone is unlikely to conclude from this that 1+1 = 3.

- Because the pseudo-parallel operations of Scone remain close to the parallel NETL model, we can easily re-implement these performance-critical parts of Scone on a data-parallel machine or on a cluster of processors. So we can develop and popularize Scone on affordable hardware and later use essentially the same model to handle much larger knowledge bases.

The software implementation of Scone's marker-passing machinery is fairly straightforward. Each knowledge-base element is implemented as a multi-word data structure in memory. One word in this structure, the *bits-word*, holds the element's marker bits, so one or two full-word Boolean operations can test the status of several markers at once: "*Does element E have all of markers 3, 4, and 7 and not marker 8?"* The link wires are implemented as pointers, and each node has back-pointers to all the links that connect to it – a separate back-pointer list for each type of wire.

For each marker M, we maintain a two-way linked list of all the elements marked with M. We call this M's *marker chain*. This chain is essential because a common operation is to scan all the elements marked with M, looking for elements that also have certain additional markers. Finally, for each marker M, we maintain a *count* of the number of nodes marked with M.

So to mark element E with marker M, we set bit M in the bits-word of E, we add element E to M's marker chain, and we increment the count of M-marked elements. To remove marker M from E, we do the opposite: clear the M bit, splice E out of M's marker chain, and decrement M's counter.

In Lisp, the frequent addition and removal of list cells from the marker chain would lead to excessive garbage collection, so we pre-allocate space for the forward and backward pointers of each marker chain in the data structure representing each element. In a system with 32 markers, this increases the size of each element by 64 pointers – 512 bytes on an implementation with 64-bit addresses. This pre-allocation is a time-space tradeoff: it makes the marking/unmarking operations much faster, at the cost of making the element data structures much larger.

As a rule, the entire active KB should be kept in main memory, since inference and search are much slower if parts of the knowledge base must be paged in from secondary storage. In a 64-bit implementation each Scone element, with its associated strings and data structures, requires about 2000 bytes of memory. So a KB with 10^6 elements, all loaded and potentially active at once, requires a machine with slightly more than 2G bytes of main memory.

4 Basic Marker Operations in Scone

Scone supports multiple inheritance through the is-a hierarchy. That is, a type or individual node may have any number of is-a links connecting it to superior (more general) types. At the top is the most general type, named "thing"; at the bottom are individual nodes. By the rules of inheritance, when we want to know some property of an individual, we must look at the individual's node and at all the type-nodes above it in the hierarchy; a property or relation could be connected to any one of these. For example, when we say that Clyde is an elephant, we connect the A-wire of an is-a link to the "Clyde" node and the B-wire to the "elephant" node. If we later ask what color Clyde is, the answer "gray" is actually inherited from the "elephant" node.

The most important pseudo-parallel operation in Scone is the *upscan*, which is used whenever we have a query about the class membership, properties, or relations of some node N. We mark N with marker M, and then propagate M to all of the type-nodes above N in the is-a hierarchy. Because we allow multiple inheritance (or upward branching) in the is-a hierarchy, an upscan might mark a large number of nodes. (Think about the number of classes of which a typical person is a member, or the number of unary predicates that may be true of that person.) The basic algorithm for an upscan is simple:

1. *Mark the starting node N with marker M.*
2. *For every element E1 newly marked with M, examine each link L that is connected to E1 by its A-wire. (A is considered to be the lower-end of the is-a link.)*
3. *If L is an active IS-A link, examine the element E2 that is attached to the B-wire of link L. (We will define "active" later, but it is a simple bit-test.)*
4. *If E2 is not already marked with M, mark E2 now.*
5. *Steps 2-4 have propagated marker M one level up the is-a hierarchy. If any new nodes have been marked with M, return to step 2 and continue iterating until we reach quiescence – that is, no new elements have been marked with M. Then stop.*

In the pseudo-parallel implementation, an upscan takes time proportional to the number of superior nodes that actually must be marked, times the average fan-out of the is-a hierarchy in the upward direction.

A *downscan* is similar to an upscan, but we cross is-a links downward, in the B-to-A direction. If we mark a type-node N with marker M and downscan, we mark all the subtypes and instances of type N. A downscan from a node high in the network, such as "physical object", might mark a large number of nodes.

In addition to is-a links, Scone has *eq-links*, which indicate that (in a given context) two nodes refer to the same entity. For example, we might have an eq-link from "George W. Bush" to "president of the U.S.". We want any marker placed on one side of an eq-link to flow to the other side, so during both upscans and downscans the markers cross active eq-links in both directions.

Upscans are used in several ways in Scone. If we want to know whether Clyde is a member of some type T, we simply upscan from Clyde and see whether the T node is marked. If we want to ask about some relation, for example to mark the set of all things that Clyde "fears" (directly or by inheritance), the algorithm is as follows:

1. *Mark the "Clyde" node with M1 and upscan to mark all its superiors.*
2. *If any active "fears" link has M1 on its A-wire, put M2 on its B-node.*
3. *Downscan all M2 markers.*

The effect of this is to put M2 on every individual or type node representing someone that Clyde fears. If there is a "fears" link from "elephant" to "mouse", we will end up with M2 on "mouse" and on "Mickey Mouse".

A similar operation can be used to detect type violations in the network. Suppose that under "person" we have two subclasses, "child" and "adult", and that these classes are disjoint. We represent this by attaching a *split-link* to these type-nodes. (A single split-link can connect to any number of nodes.) Suppose that John is known to be a child, and that someone tries to assert that John is an airline pilot, which would imply (via is-a links) that he is an adult. Before asserting this new item, we check whether doing so would violate any splits. We place marker M on both “John” and on “airline pilot”, and upscan. In this case, both “child” and “adult” are marked with M. We then ask every split-link that has M on *one* of its connected nodes to check whether M is present on *more than one* of these nodes; if so, that split is violated and something is wrong. In this case, we would report that John cannot be both a child and adult, and ask the user which assertion is incorrect. (If the user really wants to assert both, the split-link can be over-ridden for John only using a cancel-link.)

The most common operation in recognition is to find the intersection of several types. We saw this in section 2, where we were looking for gray mammals in Africa. In the software version, the operation is basically the same: mark each set with a different marker, and then tell every node that has collected all of the specified markers to label itself as a winner. However, in the pseudo-parallel software version, we cannot perform the intersection in a single cycle. Instead we must scan the marker chain of one of these markers, checking each node to see whether it has the other markers. This is more efficient if we scan the marker chain with the smallest number of marked nodes; the marker counts tell us which one this is.

So in Scone the time required to intersect *n* sets is the time required to mark the members of each set, plus the time required to visit and test all the members of the smallest of the marked sets. This is slower than on the parallel machine, but it still is fast enough for most applications.

5 Default Reasoning with Exceptions

The ability of one marker to block the passage of another is used in Scone to implement a cancellation mechanism. Consider the fragment of network shown in figure 1. The dashed links in the diagram are *cancel-links*. The unlabeled arrows are is-a links. This fragment says that a bird is a flying-thing and that both canaries and penguins are birds, but a penguin is *not* a flying-thing. Tweety is a flying thing; Fred would normally be one, but he is a non-flying exception to the general rule – perhaps he is afraid of heights.

So far, we have spoken of the propagation of single markers in Scone, but in fact markers are allocated in pairs: a positive marker M and an associated *cancel-marker*, designated ~M. During an upscan, we propagate M upward from some node such as "Max", as described above. But at each step, if marker M is on the A-wire of a cancel link, we mark the B-node with ~M. During the upscan, we do not allow an M marker to cross a link that already has a ~M marker, nor do we allow an M marker to enter any node marked with ~M.

In figure 1, we also see a link L stating that birds eat worms. This statement is inherited by canaries and (as a default) by all other birds, but it is cancelled for penguins – they only eat fish. This cancellation is implemented by the cancel-link running from the "penguin" statement L (connected to L's handle-node).

Now, if we ask what Fred eats, we follow the algorithm described above for tracing relations. This marks "worm" and any subtypes or individual "worms". But if we ask what Max likes to eat, link L is cancelled (and thus rendered inactive) before any other markers can cross it. So Max only eats fish, not worms.

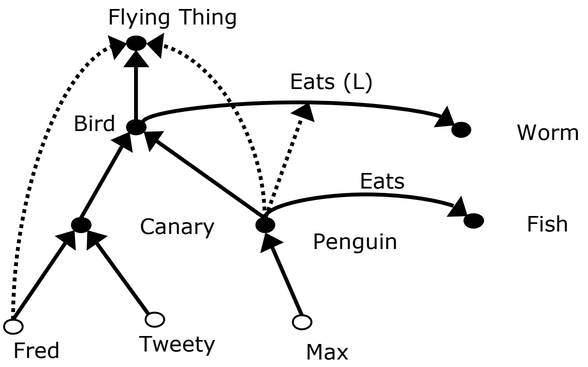

Fig. 1. Example of Cancellation

This kind of default reasoning with cancellation is a complex and controversial topic in the knowledge representation community. The examples I have presented above are straightforward and efficient, but it is possible to create networks where it is unclear whether a conclusion should be allowed or not: some paths supporting the conclusion are cancelled while others are not, and there is no clear reason to prefer one interpretation over the other. Also cancellation is incompatible with some notions of sound logical inference: it is possible to deduce a conclusion by doing a certain amount of work, and then to withdraw it when additional processing discovers that the conclusion should be cancelled.

It is beyond the scope of this paper to discuss the cancellation problem and possible solutions in greater depth. A good overview, with references to the technical literature on this subject, can be found in [10]. I will just say this: for an application-oriented system like Scone, it is impossible to live without some form of default reasoning with exceptions. The real world is full of flightless birds and white elephants. So our general approach in Scone is to try to detect all the ambiguous cases as new elements are added to the network, and to consult the user in cases where the desired meaning is not clear. Then we set up the KB network so that the runtime operations described above will yield the desired results.

The more general point is this: it is possible to use Scone-like marker passing whether or not you want to implement some form of cancellation. If you do allow cancellation, the use of cancel-markers can make this reasonably efficient at runtime.

6 Virtual Copy Semantics

In dealing with complex descriptions, Scone attempts to implement *virtual copy semantics*. Suppose we create a type-node representing a typical "family". This node serves as the container for a number of individual roles, such as "father" and "mother" and a number of type-roles, such as "child". (The type-role node stands for the typical member of a set. The set may be empty.) There are also some statements describing relations between these roles. For example, we might want to say that the mother (by default) loves the children.

The nodes and links at the top of figure 2 illustrate a somewhat simplified family for explanatory purposes: one mother, one child, and love. The dotted links signify that for every copy of “family”, there will be one mother and one child.

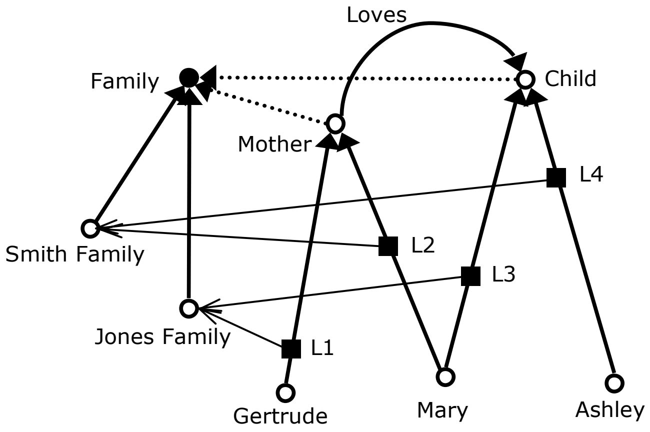

Fig. 2. Family Relations

"Smith Family" has an is-a link to family, so we want this instance to behave as if we had made a private copy of the entire "family" description. "Smith Family" has its own mother, its own child, and the mother loves the child. But we don't want to make an actual copy of all this structure; we want to create a *virtual copy* that behaves like a real copy but is implemented by inheritance and marker-passing. In general, we don't create an actual copy of an inherited node or link unless we have something to say about that specific individual; otherwise, these elements are only virtually present.

We see in figure 2 that Mary is the mother in "Smith Family". The 3-wire link L2 is a *map-link* that states this. “Smith Family” is said to be the *owner* of this map-link. The link L4 indicates that “Ashley” is the “child” in the “Smith Family”. Figure 2 also shows a second instance of “family”, the “Jones Family”. Mary is the child in this family (link L3) and Gertrude is the mother (link L1). So Mary appears in two different copies of the "family" description, playing a different role in each.

The ability of one marker to gate the passage of another is used to good advantage in implementing virtual copies in Scone. Suppose we want to mark the nodes representing all the people that Mary loves. If we do a simple upscan, not crossing any map-links, we find nothing. But the two map-links, L2 and L3, indicate that Mary plays some role in two different descriptions; we must consider each of these descriptions separately to see if a "loves" relation is present in either of them.

Looking first at L2, we place a description marker, M_D, on its owner, "Smith Family", and upscan M_D. This *activates* the "Smith Family" description. Now we ask again whether Mary loves anyone, using the algorithm described in the previous section: upscan from Mary using M1, cross the relation link with M2, then downscan M2 on the other side. But this time, we treat any map-link as an eq-link if it is tied to an owner that is activated with M_D. So in this case, any marker arriving at one end of L2 is passed on to the other end. The same is true of L4, but L1 and L3 are inactive. So the M1 marker on Mary reaches the "mother" node, we cross the "loves" link with M2, and the resulting M2 marker on "child" propagates down into "Ashley".

Then we clear all these markers and consider the description tied to L3. We mark the owner, "Jones Family" with M_D, and repeat the procedure above. But this time L1 and L3 are active, while L2 and L4 are dormant. The M1 marker on Mary reaches the "daughter" node, but can go no further, since there is no "loves" link going in the right direction out of “daughter” – only one coming in. The final result is that, according to this piece of the network, Mary loves Ashley, but nobody else.

Note that if we had tried to look at both the "Smith Family" and "Jones Family" descriptions at the same time, it would have led to confusion. With L1, L2, L3, and L4 all active at once, we could have deduced that Gertrude loves Ashley (possible, but not supported by this network) and that Mary loves herself.

This illustrates a fundamental limitation of marker-passing: in situations where an entity plays different roles in many different virtual copies, we cannot look at all of these descriptions at once without confusion. We must look at them one by one. Or, to put it another way, each instance of a description can be viewed as a set of variable bindings: in the Smith family, "Mary" is bound to "mother" and "Britney" to "daughter". A marker-passing system cannot look at many distinct sets of variable bindings at once without confusion, though some more complex (and expensive) parallel architectures can do this. On the other hand, the marker-passing machinery makes it reasonably fast to activate and explore each of these descriptions.

7 Multiple Contexts

An important feature of Scone is its multiple-context mechanism, which allows the user to represent several different *contexts* (or states of the world) in the KB at the same time. A context is just an individual node in the KB that serves as the container for some collection of knowledge. Every link (statement) in Scone has a *context-wire* that is connected to one of these context-nodes. If that context-node is marked as active, the link is active; if not, the link is effectively turned off. Similarly, every node has a context wire that indicates the context in which its referent exists. Most of our general knowledge about the world is tied to a single large context called “general”.

Contexts are connected into a hierarchy with is-a links, just like any other nodes in the Scone KB. We *activate* a context C by marking it with a special context-marker M_C and upscanning M_C to mark all the nodes above C in the hierarchy. All of the nodes and links in the M_C-marked contexts become active; all others are dormant and take no part in Scone’s search and inference.

Suppose we want to create the “Harry Potter World” context, which is very similar to the real world, but in which a few people are wizards with special powers. We simply create a new individual node, “HPW” and connect an is-a link from this node to “general”. At this point, “HPW” acts as an exact clone of “general”: if we activate “HPW”, M_C propagates upward to “general”, and all the general knowledge is turned on. But now we can add some new nodes and links in the “HPW” context that are seen only when the "HPW" context is active. For example, we can assert that, in this context, a broom is a vehicle. If we return to “general”, these new elements are invisible. Similarly, we can use cancel-links in the “HPW” context to turn off some specific "general" knowledge that would otherwise be active.

So in Scone, because markers can affect the behavior of other markers, we have a very powerful, lightweight, and efficient way to represent many distinct contexts in the same KB. A context can inherit all of the knowledge in some other context without the need for any actual copying – the copies are virtual. We can easily activate any context C and reason about what is true there without disturbing the contents of any other context (except for descendants of C).

Because it is inexpensive to create and populate a new context, Scone uses contexts in many ways. For example, an action or event creates two contexts, one representing the world before the event and the other representing the world after the event. If I drive from home to the airport, both I and my car are at home in the *before* context and at the airport in the *after* context. Both of these contexts inherit from “general”, so all my general knowledge is present in both contexts; only a few specific things have changed as a result of the "drive" action. Contexts are also used to represent (and isolate) some person’s beliefs, desires, things that are true only in certain historical periods or certain places, “what if” scenarios, and so on.

The use of multiple contexts in a KB is not a new idea. Logicians sometimes include a state term S_N in each assertion to indicate the state in which that assertion is considered to be valid. Extra formulas or rules of inference can be added to implement inheritance among these states. But reasoning with these state terms can greatly increase the amount of work that must be done by the inference system, so this mechanism is seldom used in large-scale knowledge bases.

While I am not claiming that the use of multiple contexts is a novel contribution of Scone, I will suggest that Scone's marker-passing machinery makes it efficient to use many lightweight contexts organized in a hierarchy, making this an extremely useful representational technique in Scone.

8 Performance Measurements

There are no widely accepted benchmarks for the speed of inference in a knowledge-base, and I have found very few published performance figures for the kinds of inference that are the primary focus of Scone. In order to give the reader some general idea of Scone's speed, I have run a few tests on a "synthetic" Scone knowledge base of 1,018,894 elements. We do not yet have any "real" and meaningful Scone knowledge bases of this size, so I created a synthetic KB by combining and several smaller KBs and then creating a lot of additional types and instances. I believe that the result is a fairly realistic KB in terms of its structure, though the content is not meaningful.[1] The actual KB used in these tests can be obtained from the author.

The timings given here are for the current (March 2006) version of Scone running under Steel Bank Common Lisp and Red Hat Linux. The machine is a generic workstation with a single 64-bit AMD Opteron 146 processor, rated at 2.0 GHz, and

[1] Of course, the complexity and structure of a "real" knowledge base will vary greatly, depending on the domain it is describing, so there probably is no such thing as a "typical" KB structure.

with 8G bytes of main memory. The machine was purchased in March 2005 for $3100. The times reported include garbage collection and (for the load tests) file I/O. For accuracy, the shorter times reported here are the result of executing the operation N times and then dividing the total time by N. In the intersection test we have set things up so that only one element is in the final intersection set.

Operation	**Time**
Time to create/load 1,018,894 elements, with full type-checking	240 sec
Time per element added	236 µsec
Time to create/load 1,018,894 elements, no checking	193 sec
Time per element added	189 µsec
Downscan "thing". (Marks every node in KB, then frees marker.)	10.2 sec
Time per element marked	10 µsec
Look up an inherited property of a typical individual	.93 msec
Test whether an indv can be of a given type	.60 msec
Mark, then intersect, two sets with 10K members, one winner	49.70 msec
Mark, then intersect, three sets with 10K members, one winner	83.40 msec

9 Conclusions

Our main conclusions are these:

- *Marker passing, even on a serial machine, appears to be a good implementation technology for a knowledge-base system with goals similar to Scone's: a primary emphasis on speed, scalability, and expressiveness, with relatively less emphasis on formal guarantees of logical completeness and consistency.*
- *Marker passing algorithms in Scone support multiple inheritance with cancellation, detection of type violations, reasoning with virtual copies of complex descriptions, and multiple contexts with large amounts of shared information. Statements in Scone are first-class entities in the knowledge base, so we can make statements about statements. These features give Scone great expressive power.*
- *Programs implementing deeper, more complex forms of reasoning can be built on top of Scone's low-level pseudo-parallel machinery.*
- *The marker-passing operations in Scone can be used with or without an exception mechanism. However, if you want to use such a mechanism, marker passing provides a way to implement this facility that is efficient at query-time.*
- *For a Scone KB of 10^6 elements (nodes and/or links) running on an inexpensive workstation, speed is adequate for many applications.*
- *These marker-passing algorithms were originally designed for a massively parallel machine, and they can easily be adapted to run on most parallel machines. Some machines are better suited for KB use than others: KB operations stress memory bandwidth and communication, and make little use of floating-point arithmetic.*

In closing, I offer an informal conjecture: There are many kinds of parallelism. Parallel marker-passing, as found in NETL and as simulated in Scone, is a very simple form of parallelism that is easy to implement, and that is very fast for certain operations. But the simplicity of this model brings with it some limitations. The processing elements are Boolean, with no arithmetic capabilities and hardly any memory. Searches and inferences that require reasoning about many simultaneous variable bindings cannot be handled in parallel by marker-passing; serial case-by-case reasoning, with a good deal of book-keeping, is required. A parallel machine with a full processor at every node could look at many such cases at once, but a marker-passing machine cannot do this.

I noted in section 2 that we humans have the ability to perform certain kinds of search and inference with almost magical ease. But when asked to solve a logical or mathematical puzzle or to prove a theorem, we experience that as hard mental work. We may require a pencil and paper. We may have to take a class to learn how to do this. Some people never learn these higher mental skills, though they usually survive anyway.

My conjecture is that the set of operations that marker-passing can handle is (more or less) co-extensive with the set of operations that are very easy for people. I am not suggesting that the human brain is implemented as a marker-passing machine; I am simply suggesting that there is a cognitively important class of computation that can be handled very well by both the human brain and by marker-passing, and another class that is difficult for both architectures. In Scone, we attempt to separate the marker-passing operations from the rest, and we give them special attention. I suspect that this separation may eventually help us to replicate some of that human magic.

References

1. Fahlman, S.E.: The Scone Knowledge Base (home page), http://www.cs.cmu.edu/~sef/ scone/
2. Fahlman, S. E.: *NETL: A System for Representing and Using Real-World Knowledge*, MIT Press, Cambridge MA (1979)
3. Quillian, M.R.: Semantic Memory. In: Minsky, M.L. (ed.): *Semantic Information Processing*, MIT Press (1968)
4. Hillis, W.D.: The Connection Machine, MIT Press, Cambridge MA, (1985)
5. Moldovan, D.I., Lee, W., Lin, C.: SNAP: A Marker-Propagation Architecture for Knowledge Processing. IEEE Trans. Parallel Distrib. Syst. 3(4): 397-410 (1992)
6. Kim, J.T., Moldovan, D. I.: Classification and Retrieval of Knowledge on Parallel Marker Passing Architecture. IEEE Trans. Knowl. Data Eng. 5(5): 753-761 (1993)
7. Harabagiu, S. M., Moldovan, D. I.: Parallel System for Text Inference Using Marker Propagations. IEEE Trans. Parallel Distrib. Syst. 9(8): 729-747 (1998)
8. Hendler, J.A.: Integrating Marker-passing and Problem Solving: A spreading activation approach to improved choice in planning, Lawrence Erlbaum, Mahwah NJ (1987)
9. Hendler, J.A.: Marker-passing over microfeatures: Towards a hybrid symbolic/ connectionist model, *Cognitive Science,* 13(1), p. 79-106 (1989)
10. Brachman, R. J., Levesque, H. J.: Knowledge Representation and Reasoning, Morgan Kaufmann, San Francisco, chapters 10 and 11 (2004).