EuLisp Threads: A Concurrency Toolbox

NEIL BERRINGTON  
University of Southampton, UK  
(N.Berrington@southampton.ac.uk)

PETER BROADBERY  
University of Bath, UK  
(pab@maths.bath.ac.uk)

DAVID DE ROURE  
University of Southampton, UK  
(D.C.DeRoure@southampton.ac.uk)

JULIAN PADGET  
University of Bath, UK  
(jap@maths.bath.ac.uk)

Keywords: Concurrency, objects, threads, channels, Linda, futures, communicating sequential processes

Abstract. Many current high level languages have been designed with support for concurrency in mind, providing constructs for the programmer to build explicit parallelism into a program. The EuLisp threads mechanism, in conjunction with locks, and a generic event waiting operation provides a set of primitive tools with which such concurrency abstractions can be constructed. The object system (TELOS) provides a powerful approach to building and controlling these abstractions. We provide a synopsis of this ‘concurrency toolbox’, and demonstrate the construction of a number of established abstractions using the facilities of EuLisp: pcall, futures, stack groups, channels, CSP and Linda.

1. Introduction

Programs for modern computer systems must frequently address issues relating to concurrency. These programs are becoming more commonplace, especially at applications programming level, as the availability of multiple processors and networked computer systems increases. Recent high-level programming languages offer concurrency features to support the programmer in this task.

EuLisp was designed with concurrency features from the outset. In the Lisp tradition, the intention was not to enforce a particular concurrency model, but rather to provide the primitive tools and means of combination so that programmers (and library-writers) could build the appropriate abstractions for the task at hand.

Programming with multiple threads of control within a single address space is emerging as an important and increasingly common-place paradigm.
in modern computing. We note two reasons:

- Solutions to a large class of programming problems often fall naturally into the thread model, particularly programs which respond to asynchronous events. Many programs associated with user interaction and with networks fall into this class, hence this programming style is becoming more common as these application areas expand.

- Programs written with threads can be executed on uniprocessor or multiprocessor hardware (perhaps with distributed shared memory). In the multiprocessor case, the threads can execute on any available processor. This promotes portability with the potential for making good use of the available resources, and provides a degree of scalability. In this way, threads provide a means to harness the power of modern computing platforms.

It is for these reasons that EuLisp adopted a threads mechanism as its primitive concurrency model. The importance and utility of a thread model is demonstrated by the adoption of threads in recent operating systems, for example MACH, DCE[19], Chorus[20]. This trend will facilitate the implementation of EuLisp systems on new platforms.

In contrast with many other concurrent lisp languages[21][7], EuLisp attempts to provide a model of concurrency which can be extended by efficient implementations of concurrency abstractions which are not explicitly supported by the kernel, such as futures and Linda. The thread model is quite general, and does not require pre-emptive scheduling or true multiprocessing, which may be infeasible on some systems. Even without this form of scheduling, the thread model can be used to create useful abstractions that use separate threads as a problem-solving aid, rather than a means of obtaining concurrency.

The EuLisp object system, TELOS[1] can be used in order to provide a degree of reflectiveness and extensibility in the thread model — one can effectively provide a protocol for a particular operation, and a simple implementation, and allow the user to extend or specialise it. This specialisation allows the new abstractions to be more of an integrated part of the existing kernel, rather than an additional library with independent ideas about how to create and manipulate threads — we illustrate this later in the Linda section.

In this paper we give an overview of the EuLisp thread model (sections 2–4) and give reasons why the particular primitives have been chosen. In subsequent sections we discuss several different parallel abstractions and their realisation in EuLisp. Finally (section 12), we compare these with existing languages.