Parallelising a Large Functional Program Or: Keeping LOLITA Busy

Hans-Wolfgang Loidl[1], Richard Morgan[2], Phil Trinder[3], Sanjay Poria[2], Chris Cooper[2], Simon Peyton Jones[1], Roberto Garigliano[2]

[1] Department of Computing Science, University of Glasgow, Glasgow, Scotland, U.K.
[2] Department of Computer Science, University of Durham Durham, England, U.K.
[3] The Computing Department, The Open University, Milton Keynes, England, U.K.

Abstract. A parallel version of the LOLITA natural language engineering system is under construction. We believe that, at 47,000 lines of Haskell, LOLITA is the largest non-strict parallel functional program ever. In this paper we report on the ongoing parallelisation of LOLITA, which has the following interesting features common to real world applications of lazy languages:

- the code was not specifically designed for parallelism;
- laziness is essential for the efficiency in LOLITA;
- LOLITA interfaces to data structures outside the Haskell heap, using a foreign language interface;
- LOLITA was not written by those most closely involved in the parallelisation.

Our expectations in parallelising the program were to achieve moderate speedups with small changes in the code. To date speedups of up to 2.4 have been achieved for LOLITA running under a realistic simulation of our 4 processor shared-memory target machine. We are currently tuning the program on the Sun SPARCserver target machine, where wall-clock speedup is limited by physical memory availability and a sequential foreign-language front end.

In the process of parallelising LOLITA we made changes to both our parallel software engineering techniques and our implementation technology. The most intriguing aspect, however, is that the parallelism is achieved with a very small number of changes to, and without requiring an understanding of most of the application. We attribute the ease of parallelisation primarily to the top-down approach enabled by evaluation strategies, our new mechanism for introducing and controlling parallelism in non-strict languages.

1 Introduction

Despite the advantages of the functional programming model for parallel computation, there are few large non-strict parallel functional programs. We believe that this due to the scarcity of robust parallel functional language implementations as well as the lacking support for high level languages describing some behavioural aspects of the program. We have recently constructed GUM, a parallel runtime system for Haskell, based on the Glasgow Haskell Compiler [THM+96]. In order to facilitate parallel program design we have developed evaluation strategies, describing the dynamic behaviour of lazy parallel programs. GUM is portable, and available on both shared- and distributed-memory architectures, including the CM5 [Dav96], Sun SPARCServer shared-memory multiprocessor and networks of Suns and Alphas. It is freely available and has users and developers worldwide. GUM is also integrated with the GranSim parameterisable simulator and its visualisation tools [HLP95].

The superset of Haskell used to program parallel machines is called Glasgow Parallel Haskell, GpH.

The LOLITA natural language engineering system [MSS94] has been developed at Durham University over several years. It has not originally been written with a parallel execution of the code in mind. The team's interest in parallelism is partly as a means of reducing runtime, and partly also as a means to increase functionality within an acceptable response-time. The groups at Glasgow and Durham have cooperated to parallelise LOLITA, and we believe that the result is the largest non-strict parallel functional program ever constructed. Such a large program tests the scalability both of our parallel software engineering techniques and of the underlying implementation technology, i.e. the runtime system, simulator and the visualisation tools. It also makes excellent use of our integrated engineering environment for parallel program development and performance evaluation. We focus on this particular aspect in the accompanying paper [LT97].

In addition to its size, LOLITA has several features of interest as a parallel program:

- As LOLITA already exists, we are parallelising code written without consideration of parallelism. The fundamental reason that this approach is feasible is that Haskell is declarative and leaves the evaluation order of expressions in the program unspecified. As a result the program is not fixed to a sequential evaluation order. In contrast, it would be extremely difficult to take a large body of existing imperative code and construct a parallel version.
- Laziness is essential for the efficiency in LOLITA: the program explores forests of alternative sentence parses and meanings, and investigating just one additional tree is prohibitively expensive.
- Typically for a large piece of software LOLITA was written by a group, and not by those most closely involved with introducing parallelism. Fortunately it transpires that the parallelisation only requires an understanding of the program's global structure and of one particular sub-module.
- Like many large programs LOLITA is not written entirely in a single language. Most of the program is written in Haskell, but for example the parser is written in C. There is also a large, conceptually persistent data structure, which has to be pre-loaded and shared by the processors and accessed via a foreign language interface.

The structure of the remainder of the paper is as follows. Section 2 briefly describes LOLITA. Section 3 describes both parallelism in GpH, and our new mechanism for introducing and controlling parallelism, *evaluation strategies*. Section 4 describes the parallelisation to date, and ongoing work. Section 5 discusses the lessons learnt from parallelising LOLITA and addresses runtime-system issues for parallel functional languages.

2 Introducing LOLITA

2.1 Introduction

The LOLITA (Large-scale Object-based Linguistic Interactor Translator and Analyser) system is a state of the art natural language processing system, able to grammatically parse, semantically and pragmatically analyse, reason about, and answer queries on complex texts, such as articles from the financial pages of quality newspapers. Written in the pure, lazy functional programming language Haskell, it consists of over 47,000 lines of source code [GW] (excluding comments).

The system has been under development at the Laboratory of Natural Language Engineering, University of Durham since 1986. The project currently involves a team

of approximately 20 developers working simultaneously on different components of the system. In June 1993 the LOLITA system was demonstrated to the Royal Society in London. Research in the group follows a pragmatic approach to NLP; it is the production of a robust and useful working system that is of primary interest. This pragmatic view has spawned a new field of Natural Language Research termed Natural Language Engineering (NLE) which we feel best describes the adopted methodology.

2.2 The LOLITA System

Many Natural Language Processing (NLP) systems have been built to solve specific problems. These systems are restricted, either in the particular task they perform or the domain in which they work. The aim with LOLITA is to produce a general, domain-independent knowledge representation and reasoning system.

Conceptually, the LOLITA system can be thought of as consisting of three major processes:

1. Analysis – consists of mapping text into some logical representation of its meaning.
2. Inference – the process of deriving inferences from the logical representation of some text.
3. Generation – is the conversion of information represented in a logical form into text.

Each of these processes interacts with the heart of LOLITA: the knowledge base, which is a type of **Sem**antic **Net**work called SemNet. The analysis phase mainly writes the logical form of text into SemNet[4], the inference component reads knowledge from and possibly writes inferences to SemNet and the generation module traverses SemNet in order to verbalise knowledge.

2.3 SemNet Basics

In common with semantic networks, SemNet is a graph based knowledge representation, where concepts and relationships are represented by nodes and arcs respectively, with knowledge being elicited by graph traversal. The power of SemNet lies with the efficient inference procedures it is designed for; namely inheritance[LG88]. In addition it supports many other forms of reasoning, e.g. analogy [LG94], epistemic reasoning, time and location [BG] and standard logical connective reasoning.

Currently, SemNet comprises approximately 100,000 nodes and is compatible with WordNet [Mil90]. There are three types of nodes: entities, events (assertions) and actions (roles), and three main types of directed arcs: subject, object and action which can be read/traversed in either direction. Only event nodes can have a subject, object or action arc, and only action nodes can be a action for an event node.

Figure 1 shows an example of the assertion of an event (or statement) in SemNet.

Events generally (depending on the action) have a *subject* and *object* arc (often abbreviated `sub_` and `obj_` resp.) which specify the particular entities related by the action. The inverse of the `spec_` arc is called `generalisation` (abbreviated `gen_`) which is interpreted as set inclusion. Set membership is represented by an *instance* (abbreviated `inst_`) arc, e.g. Sanjay is an *instance* of all humans. The reverse of an *instance* arc is called a *universal*.

[4] in actual fact the analysis phase also reads knowledge about word meanings from SemNet

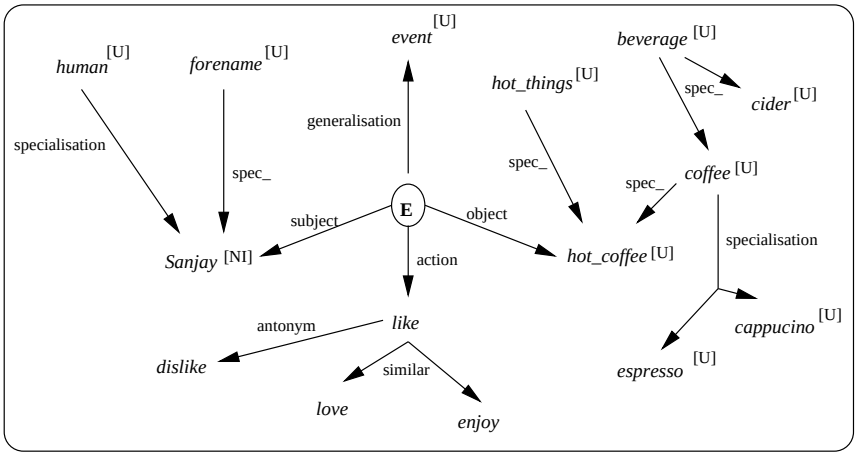

Fig. 1. A portion of SemNet around an event **E**, expressing the statement "*Sanjay likes hot coffee*".

2.4 Analysis

The front end analysis phase of LOLITA was previously described as the process of transforming natural language into its logical meaning represented as newly created events and entities in SemNet. In actual fact this stage consists of a number of abstracted sub-processes, which are of special interest for the parallelisation of the system.

Morphology and Parsing. Morphology is the initial preparation of input text before it can be parsed. It consists of using punctuation and spaces in the input to separate it into grammatical units, and replaces short hand words by their longer versions, e.g. *I'll* to *I will*. In addition there is also a facility for recovering misspelt words [Par94] and guessing unknown ones.

The LOLITA parser provides large-scale coverage, i.e. it is able to deal with full and serious text (the Brown corpus and WordNet both having been used to develop it), such as newspaper articles. The grammar rules (over 1500 of them) are expressed in a BNF notation. It uses a top-down Tomita-style approach and incorporates deterministic operators for improved efficiency [EGM93], building a parse forest of all possible parses from the input. The grammar uses a set of features and penalties to order and hence select the most likely parses of the input. The LOLITA grammar was built with the aim of dealing with erroneous and incomplete input (e.g. real-life speech and fragments of NL utterances).

The parser produces the best parse or a list of possible parses representing the deep grammatical structure of the input, with all word features extracted, errors (structural or feature caused) printed, missing parts inferred and un-parsable parts isolated.

Semantic and Pragmatic Analysis. The task of semantic analysis is to map the deep grammatical representation of the input provided by the parsing component onto nodes in SemNet (as shown in Figure 2).

To do this, the network has to be checked for the existence of nodes that already represent the concepts in the input, and decisions have to be taken on when to

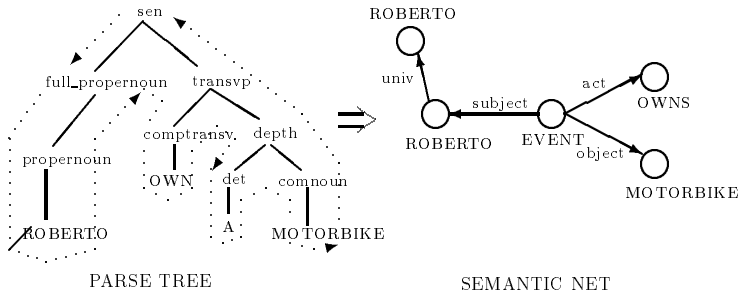

Fig. 2. An simplified example of semantic analysis. The input (a parse tree) is transformed into a section of SemNet.

generate new nodes and how to connect them to the rest of SemNet. Amongst other things, this involves anaphora resolution and making deictic references absolute; "tomorrow" will be expanded into the date after the utterance event, "I" will be resolved into a reference of the speaker, etc.

Further semantic and pragmatic analysis ensure that, after a new or modified portion of the SemNet has been built on the basis of previous stages, this portion is consistent with the existing network. Examples are sentences like "I saw a tree fly over the house", where no obvious syntactical and semantic rules are violated, or "I bought one of those Japanese TV's made by Phillips", where it is highly unlikely and undesirable to extend the coverage of the semantic representation to world knowledge (stating that Phillips is not a Japanese manufacturer of consumer electronics). If pragmatic analysis (located at the intersection between semantic, linguistic and world knowledge) cannot resolve a conflict between new and existing information in SemNet, a low level of belief is attached to the new portion of the semantic network resulting from the input.

Another way of deciding on the acceptability of input is source control. It takes into account from whom the information came and the way in which it was provided, e.g. a reliable source, an unknown source, part of a chat or a factual news report. A model of source control is incorporated into LOLITA [BG92].

2.5 Generation

The LOLITA generator [Smi96] was, like the rest of the LOLITA system, developed without any specific restrictions imposed by a particular application and is thus very flexible. It is capable of generating NL utterances from SemNet and is widely used as an interface to LOLITA and as a debugging tool. Its input consists of a chunk of SemNet, and its output is a NL utterance of a complexity greater than or equal to a sentence, depending on the parameters such as the particular application, e.g. storytelling or query, the context, the required style i.e. colourful or simple, and the previous dialogue where applicable.

2.6 LOLITA Applications

Built around the core LOLITA systems are the various applications shown in Figure 3.

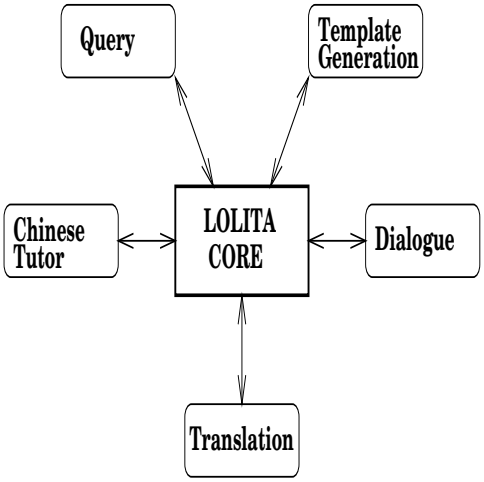

Fig. 3. A diagram showing the various applications built around the LOLITA core.

An example of one of the applications is template generation. This involves the identification of relevant information contained within ordinary text such as newspaper articles. The relevant information is presented using a template. The template contains various slots which act as field headings, whose bodies are filled in according to the content of the original text. For example a suitable template for *meetings* might contain the slots: *participants*, *when* and *where*. LOLITA identifies the information taken from some input text to fill these slots.

2.7 Summary

An overview of the LOLITA NLP system was presented. Input to the system consists of a natural language text to be processed. This text is analysed, i.e. is parsed and a representation of its meaning is created as new events asserted into SemNet(the knowledge representation at the heart of LOLITA). In order to do this the analysis must handle the inherent ambiguity in natural language, e.g. parsing ambiguities, word sense ambiguity and anaphora resolution being the most common.

Once the unambiguous logical form of the input is represented in SemNet inference procedures may operate upon the knowledge. These procedures range from basic inheritance to more complex forms such as induction and analogy.

The back-end of LOLITA consists of the generation module which traverses SemNet and verbalises the knowledge contained within it as NL utterances.

This core system has many applications built upon it. They range from language tutoring and machine translation through to dialogue. The range of applications illustrates the versatility of the core system.

3 GPH and Evaluation Strategies

3.1 GPH — A Parallel Extension of Haskell

The essence of the problem facing the parallel programmer is that, in addition to specifying *what* value the program should compute, explicitly parallel programs must also specify *how* the machine should organise the computation. There are many aspects to the parallel execution of a program: threads are created, execute on a

processor, transfer data to and from remote processors, and synchronise with other threads. Managing all of these aspects on top of constructing a correct and efficient algorithm is what makes parallel programming so hard. One extreme is to rely on the compiler and runtime system to manage the parallel execution without any programmer input. Unfortunately, this purely implicit approach is not yet fruitful for the large-scale functional programs we are interested in.

The approach used in GpH is less radical: the runtime system manages most of the parallel execution, only requiring the programmer to indicate those values that might usefully be evaluated by parallel threads and, since our basic execution model is a lazy one, perhaps also the extent to which those values should be evaluated. We term these programmer-specified aspects the program's *dynamic behaviour*.

Parallelism is introduced in GpH by the `par` combinator, which takes two arguments that are to be evaluated in parallel. The expression `p `par` e` (here we use Haskell's infix operator notation) has the same value as `e`, and is not strict in its first argument, i.e. $\perp$ `` `par` e`` has the value of `e`. Its dynamic behaviour is to indicate that `p` could be evaluated by a new parallel thread, with the parent thread continuing evaluation of `e`. We say that `p` has been *sparked*, and a thread may subsequently be created to evaluate it if a processor becomes idle. Since the thread is not necessarily created, `p` is similar to a *lazy future* [MKH91].

Since control of sequencing can be important in a parallel language [Roe91], we introduce a sequential composition operator, `seq`. If `e1` is not $\perp$, the expression `e1 `seq` e2` has the value of `e2`; otherwise it is $\perp$. The corresponding dynamic behaviour is to evaluate `e1` to weak head normal form (WHNF) before returning `e2`.

Even with the simple parallel programming model provided by `par` and `seq` we find that more and more code is inserted in order to obtain better parallel performance. In realistic programs the algorithm can become entirely obscured by the dynamic-behaviour code.

3.2 Evaluation Strategies

Evaluation strategies use lazy higher-order functions to separate the two concerns of specifying the algorithm and specifying the program's dynamic behaviour. A function definition is split into two parts, the algorithm and the strategy, with values defined in the former being manipulated in the latter. The algorithmic code is consequently uncluttered by details relating only to the parallel behaviour. In fact the driving philosophy behind evaluation strategies is that *it should be possible to understand the semantics of a function without considering its dynamic behaviour.*

Because evaluation strategies are written using the same language as the algorithm, they have several other desirable properties. Strategies are powerful: simpler strategies can be composed, or passed as arguments to form more elaborate strategies. Strategies are extensible: the user can define new application-specific strategies. Strategies can be defined over all types in the language. Strategies are type safe: the normal type system applies to strategic code. Strategies have a clear semantics, which is precisely that used by the algorithmic language.

This section gives an abridged description of evaluation strategies. A complete description and discussion of strategies can be found in [THLP98].

3.3 Strategy Type

A strategy is a function that specifies the dynamic behaviour required when computing a value of a given type. A strategy makes no contribution towards the value being computed by the algorithmic component of the function: it is evaluated purely for effect, and hence it returns just the nullary tuple `()`.

```
type Strategy a = a -> ()
```

3.4 Strategies Controlling Evaluation Degree

The simplest strategies introduce no parallelism: they specify only the evaluation degree. The simplest strategy is termed `r0` and performs no reduction at all. Perhaps surprisingly, this strategy proves very useful, e.g. when evaluating a pair we may want to evaluate only the first element but not the second.

```
r0 :: Strategy a
r0 _ = ()
```

Because reduction to WHNF is the default evaluation degree in GPH, a strategy to reduce a value of any type to WHNF is easily defined:

```
rwhnf :: Strategy a
rwhnf x = x `seq` ()
```

Many expressions can also be reduced to *normal form* (NF), i.e. a form that contains *no* redexes, by the `rnf` strategy. The `rnf` strategy can be defined over built-in or datatypes, but not over function types or any type incorporating a function type as few reduction engines support the reduction of inner redexes within functions. Rather than defining a new `rnfX` strategy for each data type `X`, it is better to have a single overloaded `rnf` strategy that works on any data type. The obvious solution is to use a Haskell type class, `NFData`, to overload the `rnf` operation. Because NF and WHNF coincide for built-in types such as integers and booleans, the default method for `rnf` is `rwhnf`.

```
class NFData a where
  rnf :: Strategy a
  rnf = rwhnf
```

For each data type an instance of `NFData` must be declared that specifies how to reduce a value of that type to normal form. Such an instance relies on its element types, if any, being in class `NFData`. Consider lists and pairs for example.

```
instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x `seq` rnf xs

instance (NFData a, NFData b) => NFData (a,b) where
  rnf (x,y) = rnf x `seq` rnf y
```

3.5 Combining Strategies

Because evaluation strategies are just normal higher-order functions, they can be combined using the full power of the language, e.g. passed as parameters or composed using the function composition operator. Elements of a strategy are combined by sequential or parallel composition (`seq` or `par`). Many useful strategies are higher-order, for example, `seqList` below is a strategy that sequentially applies a strategy to every element of a list. The strategy `seqList r0` evaluates just the spine of a list, and `seqList rwhnf` evaluates every element of a list to WHNF. There are analogous functions for every datatype, indeed in later versions of Haskell (1.3 and later [PH96]) constructor classes can be defined that work on arbitrary datatypes. The strategic examples in this paper are presented in Haskell 1.2 for pragmatic reasons: they are extracted from programs run on our efficient parallel implementation of Haskell 1.2 [THM+96].

```
seqList :: Strategy a -> Strategy [a]
seqList strat []     = ()
seqList strat (x:xs) = strat x `seq` (seqList strat xs)
```

3.6 Data-oriented Parallelism

A strategy can specify parallelism and sequencing as well as evaluation degree. Strategies specifying data-oriented parallelism describe the dynamic behaviour in terms of some data structure. For example `parList` is similar to `seqList`, except that it applies the strategy to every element of a list in parallel.

```
parList :: Strategy a -> Strategy [a]
parList strat []     = ()
parList strat (x:xs) = strat x `par` (parList strat xs)
```

Data-oriented strategies are applied by the `using` function which applies the strategy to the data structure `x` before returning it. The `using` function is defined to have a lower precedence than any other operator because it acts as a separator between algorithmic and behavioural code.

```
using :: a -> Strategy a -> a
using x s = s x `seq` x
```

A parallel map is a useful example of data-oriented parallelism; for example the `parMap` function defined below applies its function argument to every element of a list in parallel. Note how the algorithmic code `map f xs` is cleanly separated from the strategy. The `strat` parameter determines the dynamic behaviour of each element of the result list, and hence `parMap` is parametric in some of its dynamic behaviour.

```
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap strat f xs = map f xs `using` parList strat
```

3.7 Control-oriented parallelism

In control-oriented parallelism subexpressions of a function are selected for parallel evaluation. A control-oriented strategy is typically a sequence of strategy applications composed with `par` and `seq` that specifies which subexpressions of a function are to be evaluated in parallel, and in what order. The sequence is loosely termed a strategy, and is invoked by either the `demanding` or the `sparking` function. The Haskell `flip` function simply reorders a binary function's parameters.

```
demanding, sparking :: a -> () -> a

demanding = flip seq
sparking = flip  par
```

A parallel Fibonacci function with (divide-and-conquer) control-oriented parallelism can be written as follows. Several variants of a divide-and-conquer strategy for a linear system solver are discussed in the accompanying paper [LT97].

```
pfib n
  | n <= 1    = 1
  | otherwise = (n1+n2+1) `demanding` strategy
    where
      n1 = pfib (n-1)
      n2 = pfib (n-2)
      strategy = rnf n1 `par` rnf n2
```

3.8 Additional Dynamic Behaviour

Strategies can control other aspects of dynamic behaviour, thereby avoiding cluttering the algorithmic code with them. A simple example is a thresholding mechanism that controls thread granularity. In `pfib` for example, granularity is improved for many machines if threads are not created when the argument is small. This application of thresholding turns out to be very important in the parallelisation of LOLITA, and we discuss it in detail in Section 4.3.

```
pfibT n
  | n <= 1    = 1
  | otherwise = (n1+n2+1) `demanding` strategy
    where
      n1 = pfibT (n-1)
      n2 = pfibT (n-2)
      strategy = if n > 10
                   then rnf n1 `par` rnf n2
                   else ()
```

4 Parallelising LOLITA

4.1 Sequential Profiling

As a preparation for parallelising such a large program we performed sequential profiling of the code. This did not reveal a particular hotspot in the program although the syntactic parsing stage is the biggest component in the top level structure with about 20% of the execution time. This stage makes heavy use of C-functions, called from within Haskell, to optimise the time consuming parsing process. These foreign language calls complicate the parallelisation of the parsing stage. The Haskell part of the parsing, however, can be parallelised without major recoding.

4.2 Top Level Pipeline

Without a dominating hotspot in the sequential execution of the program a pipeline approach is a promising way to achieve enough parallelism for a 4 processor shared-memory machine like the Sun SPARCServer we are aiming at. The structure of a pipeline parallel version is shown in Figure 4. Each stage discussed in Section 2 is executed by a separate thread, which are linked to form a pipeline.

The key step in parallelising this structure is to define strategies on the complex intermediate data structures (e.g. parse trees) which are used to communicate between these stages. In order to avoid naming the data structures we use a *strategic function application* operator, which comes in a sequential `$|` and in a parallel `$||` variant. An expression of the form `f $|| s $ x` applies the strategy `s` to the input value `x` and in parallel applies the function `f` to `x`. Thus, the strategy is mainly used to specify the evaluation degree whereas the `$||` operator specifies the pipeline parallelism. The definitions of these new combinators are as follows:

```
infixl 6 $|, $||      -- strategic function application (sequential, parallel)
infixl 9 .|, .||      -- strategic function composition (sequential, parallel)

($|), ($||) :: (a -> b) -> Strategy a -> a -> b
(.|), (.||) :: (b -> c) -> Strategy b -> (a -> b) -> (a -> c)

($|) f s x  = f x `demanding` s x
($||) f s x = f x `sparking`  s x

(.|) f s g = \ x -> let  gx = g x
                   in   f gx `demanding` s gx
(.||) f s g = \ x -> let  gx = g x
                    in   f gx `sparking` s gx
```

This *data-oriented approach* of defining parallelism on the intermediate data structures simplifies the top-down parallelisation of this very large system, since it is possible to define the parts of the data structure that should be evaluated in parallel without considering the algorithms that produce the data structures.

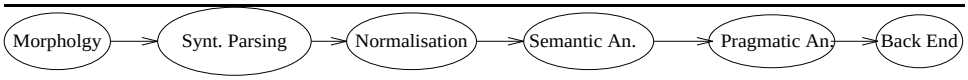

Fig. 4. Overall Pipeline Structure of LOLITA

The code of the top level function `wholeTextAnalysis` in Figure 5 clearly shows how the algorithm is separated from the dynamic behaviour in each stage by using the `$||` operator. In a first parallel version we achieved the same separation with an explicit pipeline strategy. However, this required to name every intermediate value in the pipeline which did not reflect the structure of the code in quite the same way. This experience was our main motivation for developing the strategic function application operator.

Note that this code uses a `parMap` in the parsing stage to describe data parallelism over the whole input by processing sentences in the input text in parallel. At the moment we cannot use this source of parallelism because the C code in this stage is not re-entrant. Changing the C code to exploit this form of parallelism is ongoing work. The strategies in the individual stages of Figure 5 will be discussed in the subsequent sections.

On first glance the activity profile in Figure 6 seems to exhibit very poor parallelism. However, part of the sequential front end of this algorithm is the C part of the parsing, which cannot be parallelised in this fashion because of its strictness. Furthermore, the consistent parallelism towards the end of the compilation yields a fairly good utilisation for a four processor machine. Considering the minimal changes in the code to achieve parallelism this is a reasonable first step in the parallelisation process.

4.3 Parallel Parsing

One major source of parallelism in the time consuming syntactic parsing phase is the merging of possible parse trees in order to build a parse tree for a whole sentence. One complication in the parsing of natural languages is their ambiguity. Because of this ambiguity the parsing stage produces not just one but a list of possible parse trees. Internally, however, the result is represented as a single tree, which at some points contains alternatives ("or-nodes") representing different possible parses of the subtrees. A lazy function is used to convert this single tree into a list of possible parse trees. In each or-node this function has to merge the list of parse trees produced by the recursive calls. In merging these lists the possible parse trees have to be sorted based on some simple syntactic criteria of the likelihood of a parse. In this stage, the laziness of Haskell is crucial. In order to produce one parse tree in an or-node it is only necessary to evaluate the first element in the lists produced by all alternatives.

From a parallelism point of view this behaviour explains why we cannot force the evaluation of parts of the parse forest without risking to introduce a high degree of redundant work. Furthermore, within the parsing process the merging of lists triggers the evaluation of sublists, in particular the evaluation of the quality of possible parses. Although the merging itself is very cheap it triggers work that can be usefully done in parallel.

```
wholeTextAnalysis opts inp global =
  result
  where
    -- (1) Morphology
    (g2, sgml) = prepareSGML inp global
    sentences  = selectEntitiesToAnalyse global sgml

    -- (2) Parsing
    rawParseForest = parMap rnf (heuristic_parse global) sentences

    -- (3)-(5) Analysis
    anlys = stateMap_TimeOut (parse2prag opts) rawParseForest global2

    -- (6) Back End
    result = back_end anlys opts

-- Pick the parse tree with the best score from the results of
-- the semantic and pragmatic analysis.  This is done speculatively!

parse2prag opts parse_forest global =
 pickBestAnalysis global  $|| evalScores  $
 take (getParsesToAnalyse global)         $
 map analyse parse_forest
 where
   analyse pt =   mergePragSentences opts $ evalAnalysis
   evalAnalysis = stateMap_TimeOut analyseSemPrag pt global
   evalScores =   parList (parPair rwhnf (parTriple rnf rwhnf rwhnf))

-- Pipeline the semantic and pragmatic analyses
analyseSemPrag parse global =
 prag_transform             $|| rnf   $
 pragm                      $|| rnf   $
 sem_transform              $|| rnf   $
 sem (g,[])                 $|| rnf   $
 addTextrefs global         $|  rwhnf $
 subtrTrace global parse

back_end inp opts =
 mkWholeTextAnalysis     $|  parTriple rwhnf (parList rwhnf) rwhnf $
 optQueryResponse opts   $|| rnf $
 traceSemWhole           $|| rnf $
 addTitleTextrefs        $|| rnf $
 unifyBySurfaceString    $|| rnf $
 storeCategoriseInf      $|| rnf $
 unifySameEvents opts    $|  parPair rwhnf (parList (parPair rwhnf rwhnf)) $
 unpackTrees             $|  parPair rwhnf (parList rwhnf)  $
 inp
```

Fig. 5. The top level function of LOLITA

In order to improve the granularity of the threads produced by this parallel tree traversal we apply a thresholding strategy shown in Figure 7 to the "span" in the tree. This value, which is attached to each node in the tree, specifies the number of leaves in the current subtree. The threshold for generating a parallel process in order to merge all possible subtrees is specified as a percentage of leaves that have to be reached from the current node. This is very cheap because it only involves the checking of the provided span value.

One strength of strategies is their reusability for different algorithmic code that has the same dynamic behaviour. We were able to exploit this feature with `mergeStrategy` in Figure 7 by applying the same polymorphic thresholding strategy to two lists of different types within the syntactic parsing stage.

We have performed a series of measurements under GranSim in order to determine the best value for the span in this strategy. We have used a setup that models the 4 processor shared-memory Sun SPARCServer available at Durham.

Figure 8 shows the activity and granularity profiles for a span threshold of 50%. During the syntactic parsing stage we achieve a quite good utilisation of the system. However, almost 100 blocked threads and a high number of runnable threads are generated, too. These impose significant runtime overhead in the system. The histogram of threads sorted by thread size in the granularity profile at the right hand side of Figure 8 reveals that most of the threads are very fine-grained: 3,422

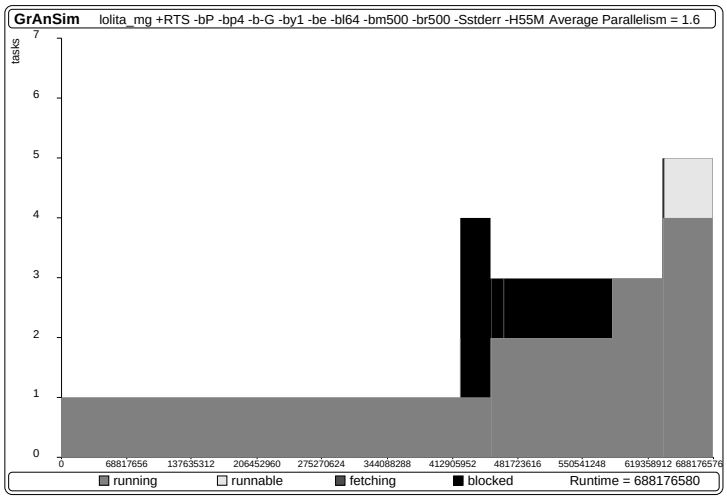

Fig. 6. Activity profile of LOLITA with pipeline parallelism

```
mergeStrategy :: (NFData a, NFData b) =>
                 (ParseForest,FeatureForests) -> Span -> MergeStrategy a b
mergeStrategy (pf,ff) span
  | totalSpan == 0             = MStrat serialMerge
  | percentSpanned >= minSpan  = MStrat parallelMerge
  | otherwise                  = MStrat serialMerge
    where
      percentSpanned = (span * 100) `div` totalSpan
      totalSpan = forestSpan pf
      minSpan = getParsingParPercent (forestGlobal pf)

      parallelMerge :: (NFData a, NFData b) =>
                       [(a,b)]  -> [(a,b)] -> Strategy [(a,b)]
      parallelMerge as bs _
       = fstPairFstList bs `par`
         fstPairFstList as `seq`
         ()

      serialMerge :: (NFData a, NFData b) =>
                     [(a,b)]  -> [(a,b)] -> Strategy [(a,b)]
      serialMerge as bs
       = r0
```

Fig. 7. A granularity control strategy used in the parsing stage

of the 5,122 threads (67%) are shorter than 2,000 cycles. This leads to a bad ratio of computation versus parallelism overhead.

In comparison, when increasing the span threshold to 90% the number of blocked and runnable threads is reduced significantly (at most 36), and the number of small threads drops drastically, as shown in Figure 9. Now, only 67 of the 165 threads are shorter than 2,000 cycles (40%). Corresponding to this drop in the total number of threads, especially fine-grained threads, the runtime drops from 754,687,749 cycles in the previous version to 526,842,092 cycles in this version.

As a result of these measurements and considering the low amount of parallelism that is required to fully utilise the 4 processor shared-memory machine, we use span thresholds around 90% for GUM executions of LOLITA.

4.4 Parallel Semantic Analysis

Another source of parallelism can be used to improve the quality of the analysis by applying the semantic and pragmatic analyses in a data-parallel fashion on different

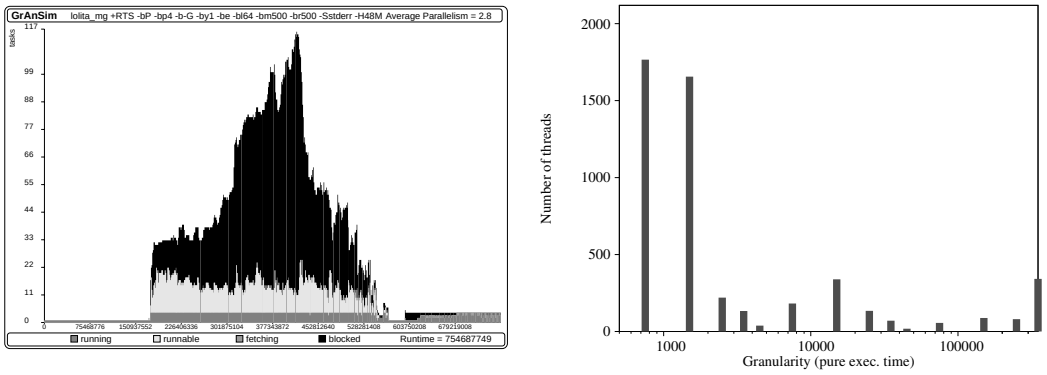

Fig. 8. Activity and granularity profiles of LOLITA with a span threshold of 50%

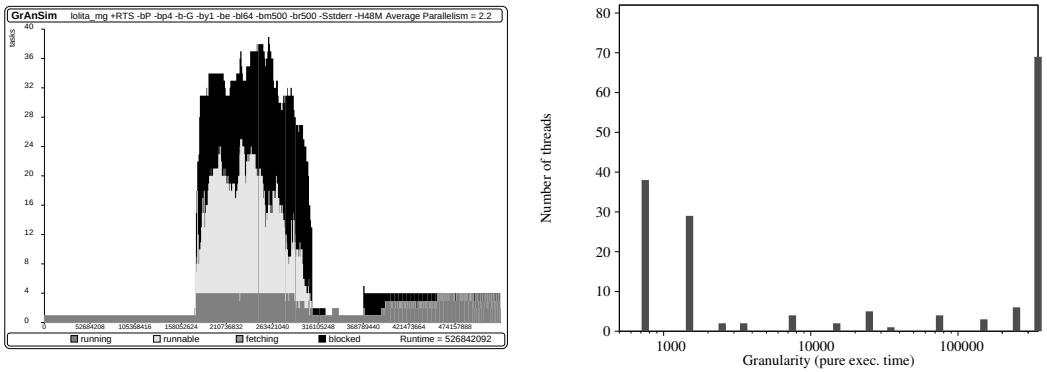

Fig. 9. Activity and granularity profiles of LOLITA with a span threshold of 90%

possible parse trees for the same sentence. Because of the complexity of these analyses the sequential system always picks the first parse tree, which may cause the analysis to fail, although it would succeed for a different parse tree. Such parallelism would not increase the performance of the system but it might improve the quality of the result.

This additional data parallelism is defined by the strategy `evalScores` in the function `parse2prag` (see Figure 5). The parse forest `rawParseForest` contains all possible parses of a sentence. The semantic and pragmatic analyses are then applied to a predefined number (specified in `global`) of these parses. The data parallel strategy `evalScores` is applied to the list of these results and demands only the score of each analysis (the first element in the triple) in order to avoid unnecessary computation at this stage. This score is used in `pickBestAnalysis` to decide which of the parses to choose as the result of the whole text analysis.

So far we have not systematically measured the improvements in the quality of the result by analysing several possible parse trees. However, considering that about 70% of all sentences that are analysed have several possible parse trees, the possibility to analyse several of them without large additional costs is very attractive

from a natural language engineering point of view.

4.5 Overall Parallel Structure

Figure 10 summarises the overall parallel structure arising when all of the sources of parallelism described above are used. The syntactic parsing stage is internally parallelised using the granularity control strategy shown in Figure 7. Note that the analyses may add nodes to the semantic net as described in Section 2.4. This creates an additional dependence between different instances of the analysis (indicated as vertical arcs). Lazy evaluation ensures that this does not completely sequentialise the analyses, however.

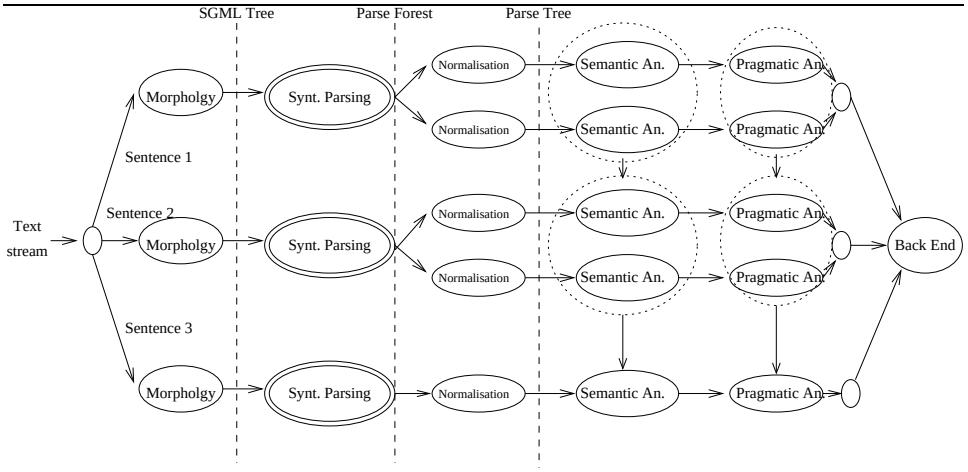

Fig. 10. Detailed Structure of LOLITA

It should be emphasised that specifying the strategies that describe this parallel behaviour entailed understanding and modifying only one of about three hundred modules in LOLITA and three of the thirty six functions in that module. So far, the only module we have parallelised is the syntactic parsing stage. If it proves necessary to expose more parallelism we could parallelise other sub-algorithms, which also contain significant sources of parallelism. In fact, the most tedious part of the code changes was adding instances of `NFData` for intermediate data structures, which are spread over several dozen modules. However, in the meantime this process has been partially automated [Win97].

4.6 Sun SPARCserver Implementation

We are currently tuning the performance of LOLITA on the Sun SPARCServer. A realistic simulation showed an average parallelism between 2.5 and 3.1, using just the pipeline parallelism and parallel parsing. The actual speedup, however, does not exceed 2.4. Our measurements with varying span values indicate that this is partly caused by fine-grained parallelism in the parsing stage. Since LOLITA was originally written without any consideration for parallel execution and contains a sequential front end of about 10–15% (the C part of the syntactic parsing stage), we are pleased with this amount of parallelism. In particular the gain for a set of rather small changes is quite remarkable.

However, the wall-clock speedups obtained to date do not quite match the simulation results. As shown in Figure 11 a two processors execution on small inputs achieves an average parallelism of 1.4. We use a high span value to bound the amount of parallelism in the parsing phase. This also bounds the total heap residency in the system, which proves to be very important. With more processors the available physical memory is insufficient and heavy swapping causes a drastic degradation in performance. The reason for this is that GUM, which is designed to support distributed-memory architectures uniformly, loads a copy of the entire code, and a separate local heap, onto each processor. LOLITA is a very large program, incorporating large static data segments (totalling 16Mb), and requires 100Mb of virtual memory in total in its sequential incarnation. Clearly a great deal of this data could be shared, an opportunity we are exploring. We are also making the C parsing functions re-entrant which will allow the analysis to be performed in a data-parallel fashion over a set of input sentences.

One difference of the GUM activity profile in Figure 11 to the GranSim results is a larger degree of fetching in the former. This is probably caused by the rather expensive but generic communication routines used by PVM, on which GUM is based. In contrast, GranSim measures mainly the hardware costs for performing communication. Together with the fine granularity of the generated threads this increased overhead leads to a significantly smaller utilisation in the parsing stage. However, the later pipeline stages in the computation are still an effective source of parallelism.

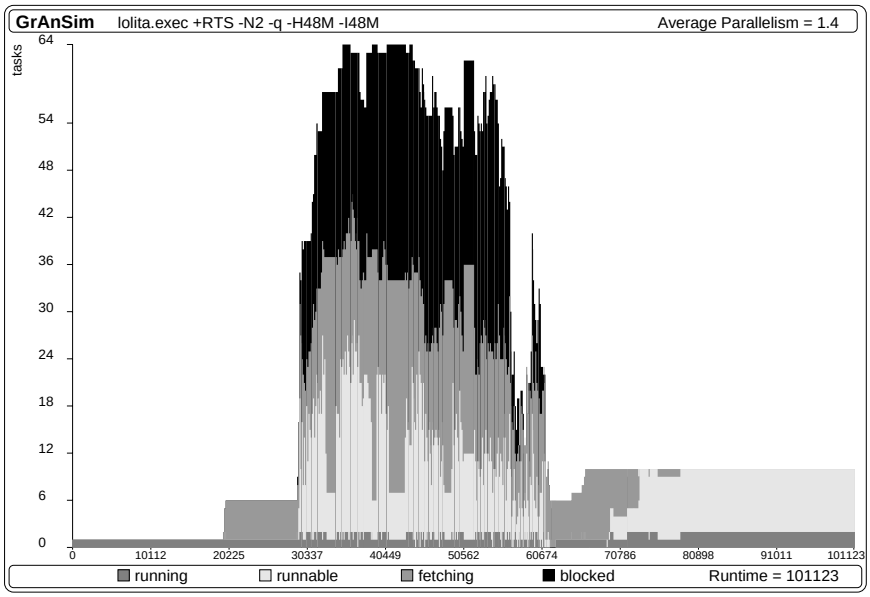

Fig. 11. Activity profile of LOLITA run under GUM with 2 processors

5 Discussion

At the current stage of parallelising LOLITA we use top level pipeline parallelism as well as divide-and-conquer parallelism in the most expensive stage of the pipeline. Additional speculative parallelism can be used in the analysis stage in order to improve the quality of the result. We are currently working on changes in the C code to provide global data parallelism over the whole program (the changes in the Haskell code are in place already).

The current parallel version of LOLITA achieves a parallelism between 2.5 and 3.1 under GranSim emulating a 4-processor shared-memory machine. The corresponding speedup, however, does not exceed 2.4. This is partly due to overhead caused by very fine-grained parallelism and partly due to strategies that perform speculative computations (although we avoided speculation on potentially expensive components). The thresholding strategy in the parsing stage proved to be very effective in increasing the average granularity of the generated threads.

We have yet to obtain significant wall-clock speedups on our target machine. This, however, is not due to a lack of parallelism but due to the very high memory consumption of the application, which exceeds the available main memory in the current setting. We are working to obtain better results by making the C parsing code re-entrant to exploit overall data parallelism.

Parallelising LOLITA has taught us a great deal about large-scale parallel functional programming. The most intriguing aspect is that the parallelism is achieved using a very small number of changes to the application. We have been able to use a top-down approach of the parallelisation to an extent, which would be very difficult in a strict language. All of the parallelism has been specified by evaluation strategies acting on the data structures passed between modules. As a result, the parallelism has been introduced without changing, and indeed without understanding most of the program. This abstraction is crucial when working on an application of this size. For example, introducing top-level parallelism entailed changing just one out of around three hundred modules. Considering this small amount of code changes in a large application, the fact that it was not designed to be executed in parallel, and the presence of inherently sequential foreign language calls, we are pleased with the achieved degree of parallelism.

Our experiences with LOLITA and several other medium-scale programs are being distilled into some guidelines for engineering large parallel functional programs [LT97]. Another concrete outcome of parallelising LOLITA has been the introduction of strategic function application to support pipeline parallelism. This extension to our strategy module has in the meantime proven useful for several other parallel programs like the parallelising compiler Naira [Jun97].

The parallelisation of LOLITA pointed out the importance of several runtime-system issues. The choice of one fixed heap size for each processor proved to be too inflexible and we had to slightly change the runtime-system. In a shared-memory setting we would like to have heaps that can grow dynamically if necessary. In general either program annotations or runtime-system options for tuning the data-locality of the program would be useful. Uniform access to an external data-structure from every processor is important because the semantic net is constructed outside the Haskell heap. For a distributed memory usage of such data-structures the communication mechanism in GUM would have to treat them specially. In the current version, the presence of such a conceptually persistent data structure required recoding of the C routines for reading and sharing this data structure. In fact, recoding just this part of the C code required more changes than the actual parallelisation of the Haskell program itself. General support of persistent data in a distributed setting would simplify the program and its parallelisation considerably.

References

[BG] S. Baring-Gould. Semnet: The knowledge representation of LOLITA. PhD thesis, University of Durham, forthcoming.

[BG92] A. F. Bokma and R. Garigliano. Uncertainty management through source control: A heuristic approach. In *International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*. Mallorca, Spain, July 1992.

[Dav96] K. Davis. MPP Parallel Haskell. In *IFL'96 — Intl Workshop on the Implementation of Functional Languages*, pages 49–54, Bad Godesberg, Germany, September 1996. Draft Proceedings.

[EGM93] N. Ellis, R. Garigliano, and R. Morgan. A new transformation into deterministically parsableform for natural language grammars. In *Proceedings of 3rd International Workshop on Parsing Technologies*, Tilburg, Netherlands, 1993.

[GW] A. Gill and P. Wadler. Real world applications of functional programs. <URL:`http://www.dcs.gla.ac.uk/fp/realworld.html`>

[HLP95] K. Hammond, H-W. Loidl, and A. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *HPFC'95 — High Performance Functional Computing*, pages 208–221, Denver, Colorado, April 10–12, 1995. <URL:`http://www.dcs.gla.ac.uk/fp/software/gransim/`>

[Jun97] S. Junaidu. NAIRA: A Parallel Compiler for Message Passing Multiprocessors. In *IFL'97 — Intl. Workshop on the Implementation of Functional Languages*, September 10–12, St. Andrews, Scotland, 1997.

[LT97] H-W. Loidl and P. Trinder. Engineering Large Parallel Functional Programs. In *IFL'97 — Intl. Workshop on the Implementation of Functional Languages*, September 10–12, St. Andrews, Scotland, 1997.

[LG88] D. Long and R. Garigliano. Inheritance hierarchies. Technical Report 4/88, Department of Computer Science, University of Durham, 1988.

[LG94] D. Long and R. Garigliano. *Reasoning by Analogy and Causality: A model and application*. Artificial Intelligence. Ellis Horwood, 1994.

[Mil90] G.A. Miller. Wordnet: An online lexical database. *International Journal of Lexicography*, 1990.

[MKH91] E. Mohr, D.A. Kranz, and R.H. Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[MSS94] R.G. Morgan, M.H. Smith, and S. Short. Translation by Meaning and Style in Lolita. In *Intl. BCS Conf. — Machine Translation Ten Years On*, Cranfield University, November 1994.

[Par94] B. Parker. Spelling correction in the nlp system LOLITA: Dictionary organisation and search algorithms. Master's thesis, Department of Computer Science, University of Durham, 1994.

[PH96] J.C. Peterson, K. Hammond, et al. *Haskell 1.3 — A Non-Strict, Purely Functional Language*, May 1996.

[Roe91] P. Roe. *Parallel Programming using Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, February 1991.

[Smi96] M.H. Smith. *Natural Language Generation in the LOLITA System: An Engineering Approach*. PhD thesis, Department of Computer Science, University of Durham, 1996.

[THLP98] P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), Jan. 1998. <URL:`http://www.dcs.gla.ac.uk/ hwloidl/publications/Strategies/strategies.html`>

[THM+96] P. Trinder, K. Hammond, J.S. Mattson Jr., A.S Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel implementation of Haskell. In *PLDI'96 — Programming Languages Design and Implementation*, pages 79–88, Philadelphia, USA, May 1996.

[Win97] N. Winstanley. A Type-Sensitive Preprocessor for Haskell. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, September 15–17, 1997.

This article was processed using the LaTeX macro package with LLNCS style