Astro-Gofer: Parallel Functional Programming with Co-ordinating Processes

Andrew Douglas, Niklas Röjemo, Colin Runciman and Alan Wood

Department of Computer Science*, University of York,
Heslington, York YO1 5DD

Abstract. This paper investigates the addition of operations for explicit parallelism to the lazy functional language, Gofer. For this purpose, we choose to use a co-ordination language, derived from Linda, based on logically shared associative memories, called *spaces*.

We describe the use of Gofer type classes in building an interface to spaces, which highlights and extends the traditional associative access mechanism. We then show how processes can be added to Gofer, and how we can combine spaces and processes to produce a parallel functional language, which we have called Astro-Gofer.

1 Introduction

In this paper, we describe a parallel lazy functional programming language based on the notion of co-ordination[4]. The functional language used is Gofer[8], a dialect of Haskell in which type classes may have multiple type parameters. The co-ordination language is based on shared associative memories called *spaces*, derived from Linda[4].

2 Simple Spaces in Gofer

We begin with a preliminary exploration of how we can include a kind of associative structure into Gofer. This associative structure will be used initially for sequential programming, but will motivate our parallel extension of Gofer in future sections.

We use Gofer type classes to enforce typed interaction with our associative structure. A type class associates a number of *methods* with one or more type parameters. An *instance* of a class declares a type to be a member of the class, and gives function definitions for the class methods. Although method names are overloaded and functions using methods can be polymorphic over the types declared as instances, correct method implementations are chosen for a particular type. The existence of a suitable method is guaranteed by compile time checks.

An association between two types of values, one of which (the *template* type) is used to retrieve the other (the *object* type), is represented by an instance of a

* For correspondence, `wood@minster.york.ac.uk`

class `Match`, with two type parameters. The method of the class, `match`, tells us how we can match objects of the required object-type and the template-type.

```
class Match o t where match :: o -> t -> Bool
```

One simple instance matches by equality between basic values of the same type:

```
instance Match Int Int where match a b = (a == b)
```

Generalising, templates of type `Maybe Int` can be used to specify *either* a specific integer, *or any integer*.

```
data Maybe a = Any | Just a

instance Match Int (Maybe Int) where
         match n Any = True
         match n (Just m) = (n == m)
```

As another example, we can match template-structures against object-structures component by component. For example, triples:

```
instance (Match a d, Match b e, Match c f) =>
         Match (a,b,c) (d,e,f) where

         match (a,b,c) (p,q,r) = (match a p) && (match b q)
                                              && (match c r)
```

This declaration states that `Match (a,b,c) (d,e,f)` is an instance of the class, for all types `a`, `b`, `c`, `d`, `e` and `f`, provided that (as specified to the left of the `=>`) `Match a d`, `Match b e` and `Match c f` are also instances of the class.

So now we can have templates such as `(Any :: Maybe Int, 10, 6)` which matches any integer triple whose second and third components are `10` and `6`, but whose first component is unconstrained apart from its type.

Finally, we could introduce more flexibility into matching, through constraints:

```
instance Match a (a -> bool) where
         match n f = f n
```

This extends the traditional Linda style of matching to include templates such as `((<99), 10, 6)`, which matches any integer triples whose second and third components are `10` and `6`, but whose first component is constrained to be less than `99`.

2.1 Simple space simulation

The `Match` class can be used to define computation in the context of several named *spaces*. Spaces are multisets containing values of a variety of object-types, together with operations to add new values and to retrieve values matching a given template.

Consider first a sequential implementation of uniformly-typed spaces, using *continuation-passing style*, a standard means of expressing sequences of operations in a purely functional language.

```
type Space o – multisets of values of object-type o
type Name    – names, eg integers
type Table o – associations of Names to Spaces
type Cont o = Table o -> Table o
```

First, there must be a means of generating a new (empty) space with a new name. The function is given a continuation to which is passed the identifier of the new space.

```
newspace :: (Name -> Cont o) -> Cont o
```

The simplest continuation is `stop`, which does nothing.

```
stop :: Cont o
stop = id
```

We can place things into a space using `put`:

```
put :: Name -> o -> Cont o -> Cont o
```

We also want to retrieve items from a space. It is this function, `get`, that relies on the class `Match o t` (being polymorphic only over those types which are instances of `Match`). Given a template of type `t`, a matching item of object-type `o` is retrieved from a named space. If no matching item is found, then the expression evaluates to $\perp$

```
get :: Match o t => Name -> t -> (o -> Cont o) -> Cont o
```

The $\perp$ semantics for a failed match might appear to be a little drastic. In section 4, when we use spaces shared between processes, the reason for our choice will become clear.

Finally, we include two operations which act *between* spaces. The first of these, `collect`, *moves objects matching a template from one space to another.* The continuation is passed the number of items moved – which may be zero.

```
collect :: Match o t =>
           Name -> Name -> t -> (Int -> Cont o) -> Cont o
```

The `copy_collect` operation behaves like `collect`, except that the items matching the template are *copied* rather than moved.

```
copy_collect :: Match o t =>
                Name -> Name -> t -> (Int -> Cont o) -> Cont o
```

The concept of *space*, together with the operations `put` and `get`, is derived from the *tuple space* concept in Linda [4]. The *collect* primitive is justified in [1], while the *copy_collect* primitive has been shown to be indispensable for maximising concurrency in certain algorithms [10]. Although no detailed semantics is given, it is probable that this sequential system will be deterministic.

```
single    :: Name -> Int -> Name -> Cont Int -> Cont Int
dolevels  :: Name -> Cont Int -> Int -> Name -> Cont Int
histogram :: Name -> Int -> (Name -> Cont (Int,Int,Int))
                         ->  Cont (Int, Int, Int)

single img level hist cont
   = collect img img (Any :: Maybe Int, Any :: Maybe Int, level) $
     \occurs -> put hist (level, occurs, 0) $
                cont

dolevels img cont 0     hist = cont ts
dolevels img cont (n+1) hist = single img n hist (dolevels n ts)

histogram img maxgrey cont = newspace $ \newts ->
                             dolevels img cont maxgrey newts
```

Fig. 1. Histogram program in Gofer extended with spaces.

2.2 An example space program

We give a brief example of how space operations can be used: producing a histogram of an image (see **Fig.1.**).

An image is represented by a space, containing triples; each triple represents a co-ordinate together with the grey level to which it is set in the image. For the moment, all spaces must hold objects of the same type, so a histogram of an image is also represented by a space of triples, where the first component represents a grey level, the second component represents the number of pixels set to that grey level in the image, and the third is always `0`.

The `histogram` function takes the name of the image space, and an integer representing the number of grey levels allowed in the image. It also takes a continuation, to which is passed the name of the histogram space. The function creates a new space identifier, which it passes to `dolevels`.

The `dolevels` function creates a histogram entry for each grey level. Given an integer `n`, it performs `single` on the values 0 to $\mathtt{n} - 1$, placing the results in space `hist`.

`single` is given a grey level value, the names of the image and the histogram spaces, and a continuation. The function will `collect` the pixels from the image space *into the image space*. This way, the pixels of the appropriate grey level value are counted, but not removed from the image. This value is then placed into the histogram space, tagged with the grey level value, and then the function behaves as the continuation.

An important deficiency of the above sequential system is that, although we can have multiple spaces, these all have to be of the same type. This will be remedied by hiding the implementation of spaces within the Gofer system. The result will be strongly and statically typed, through the use of Gofer type classes. In section 4.1 we will show that building a concurrent (parallel) function to create a histogram requires very little modification to the above program.

3 Processes and Gofer

We have considered spaces in a sequential programming context. However, spaces are intended for use in the context of concurrent processes, where they are used for communication and co-ordination.

Processes communicate through common access to named spaces. These named spaces constitute a *global state* which is hidden from the user by the Gofer system. Each *individual* process will have its own *world state*, which is acted upon by continuations – this enforces a single thread of control.

The type of a process is as follows:

```
type Process a = a -> a
```

A process is a state transformer, which, given a state of type `a` (the process's world state), returns a new state after performing some action. An action will affect the world state, and might also affect the global state.

Two functions are provided; the first provides a mechanism for spawning processes:

```
spawn :: Process a -> a -> Process b -> Process b
```

where `spawn P s Q` creates a new process which behaves as `P`, with initial state `s`, before continuing as the process `Q`.

The second function returns a process which terminates a computation:

```
stop :: Process ()
```

Operations for communication are provided in the next section.

4 Processes + Spaces = Astro-Gofer

This section combines spaces with processes to produce Astro-Gofer. With the introduction of processes, the *global state*, the named spaces, will be shared between *all* processes. So the implementation of spaces and their associated operations become hidden in the underlying Gofer system. This conveniently means that we can remove the typing restrictions on spaces, so that they can hold objects of a variety of types. However, Astro-Gofer is still strongly and statically typed, because of the `Match` type class.

In the place of the type `Cont t`, we use the `Process a` type to create a notion of threading. The implementation of spaces and the type `Name` is hidden from the user, as is the implementation of the space operations. The first of these is `newspace`, whose type becomes:

```
newspace :: (Name -> Process a) -> Process a
```

As before, the `put` function places an object into a space, and then continues. This implies that process communication is *asynchronous*.

```
put :: Name -> a -> Process b -> Process b
```

Objects are removed from a space by the operation `get`:

```
get :: Match o t =>
       Name -> t -> (o -> Process a) -> Process a
```

The operation `get` uses a template of type `t` to retrieve an object of type `o` from space. The operation will *block* (no result will be returned) until such an item can be found. This reflects the $\perp$ semantics given to `get` in section 2.1. As a result of adding processes, the semantics of `get` is now non-deterministic (that is, given many matching objects, it is not determined which will be chosen, and given several processes competing for an object, it is not determined which will get it).

The types given to `collect` and `copy_collect` are as follows:

```
type Rep a = Int

collect :: Match t u => Space -> Space ->
           u -> (Rep t -> Process a) -> Process a

copy_collect :: Match t u => Space -> Space ->
                u -> (Rep t -> Process a) -> Process a
```

The type `Rep a` is necessary because Gofer requires that both types of a two-type class must appear in the type of the function.

4.1 Example space program - revisited

We return to our histogram example, and show how we can now take advantage of processes (see **Fig.2.**).

```
single    :: Name -> Int -> Name -> Process a   -> Process a
dolevels  :: Name -> Process a -> Int -> Name   -> Process a
histogram :: Name -> Int -> (Name -> Process a) -> Process a

dolevels img cont 0     hist = cont ts
dolevels img cont (n+1) hist = spawn (single img n hist stop)
                                     (dolevels n ts)
```

Fig. 2. Concurrent histogram program in Astro-Gofer

The types of the functions have changed to reflect the addition of processes. On each recurse of the function `dolevels`, a new process is created to calculate the histogram entry for the grey level value. So, for n grey level values, n processes are spawned.

The histogram is known to be complete when n triples have appeared in the histogram space, which can be checked easily. However, a consumer might not need to wait for the histogram to be complete before it begins to consume the histogram values.

5 Conclusion

We have described an approach to providing explicit parallelism in a lazy functional language, using Linda-like co-ordination. This is quite different to other attempts at providing explicit parallelism within a functional setting, which use message passing. Our approach is highly asynchronous and decoupled, and in this respect, it is closest to Concurrent Haskell [5].

No other attempt to provide *space-based* parallelism within a lazy functional language has been undertaken, and the extendibility of the language (in its matching capabilities) and the inclusion of laziness within spaces is novel.

Various Linda-based systems have been undertaken in strict functional languages, such as Lisp [7], which has untyped tuple spaces, and ML[11]. The latter takes an approach which is rather static, with fixed algebraic types representing tuples and templates. This provides little typing information other than *this is a tuple* or *this is a template*. None of these languages has capabilities for extending matching.

5.1 Semantics

A two-part semantics for Astro-Gofer can be given. One part provides a functional semantics for individual processes, another deals with the interaction of processes.

An individual process can be seen as an interactive function, which produces a list of requests, and receives a list of replies to these requests. We could, then, give a semantics to processes which is of the kind used for describing IO [6].

The second level of the semantics of Astro-Gofer is given by the semantics for spaces. The semantics of Linda, as given in [2], can be used as a foundation for spaces. Difficulty comes when describing the two new operations, `collect` and `copy_collect` – a concurrent semantics is quite straight-forward, but when we consider truly parallel implementations such as [3], an intuitive semantics is much more difficult to describe.

A consequence of using continuations is that processes are *single threads of control which cannot interact*. The only way of performing space operations is as part of one of these threads. Consequently, no value expressions can perform space operations, and no space operations can be performed by the process of matching, or by items evaluated while in space. We can pass functions through space, and hence we can pass objects of type `Process a` – these can only be evaluated as part of a process.

Items within space may be evaluated only once but used many times (true laziness in space), or they may be copied and distributed as part of the underlying system, and evaluated many times (loss of laziness).

Finally, we note that the language Astro-Gofer has a model of processes and asynchronous communication which results in the following properties characteristic of a co-ordination language.

Spatial decoupling: a process knows little or nothing about other processes.

Temporal decoupling: communicating processes need not overlap temporally.

5.2 Implementation

The current implementation of Astro-Gofer is concurrent, being based on a version of Gofer extended with processes[12]. The scheduling strategy is simple – evaluate a process until a space operation is performed, and then swap. A space mechanism can be built quite simply.

We intend to build a parallel implementation to exploit networks of workstations. Recent developments in implementation technology [9] show that parallel implementations can be very efficient.

References

1. P. Butcher, A. Wood, and M. Atkins. Global synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505 – 516, 1994.
2. P. Ciancarini, K. Jenson, and D. Yankelevich. On the operational semantics of a co-ordination language. In *ECOOP'94*, LNCS 924. Springer Verlag, 1995.
3. A. Douglas, A. Rowstron, and A. Wood. Linda implementation revisited. In *Transputer and occam Developments*. IOS Press, 1995.
4. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages*, 7(1), January 1985.
5. A. Gordon, S. L. Peyton-Jones, and S. Finne. Concurrent Haskell. In *ACM Symposium on Principles of Programming Languages*. ACM, 1996.
6. A. D. Gordon. *Functional Programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.
7. S. Jagannathan. Customisation of first-class tuple spaces in a higher-order language. In *PARLE*, LNCS 506. Springer Verlag, 1991.
8. M. P. Jones. The implementation of the Gofer functional programming system. Technical Report YALEU/DCS/RR-1030, Yale University, 1994.
9. A. Rowstron and A. Wood. An efficient distributed tuple space implementation for networks of workstations. In *EuroPar'96*, LNCS. Springer Verlag, 1996.
10. A. Rowstron and A. Wood. Solving the Linda mutliple `rd` problem. In *Coordination Languages and Models*, LNCS 1061. Springer Verlag, 1996.
11. E. H. Siegel and E. C. Cooper. Implementing distributed Linda in Standard ML. Technical Report CMU-CS-91-151, Carnegie Mellon University, 1991.
12. M. Wallace. *Functional Programming and Embedded Systems*. PhD thesis, Department of Computer Science, 1995. Also Technical Report YCST95/04.

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