# Program Transformations for Static Process Networks

Stuart Cox Shell-Ying Huang* Paul Kelly Junxian Liu
Frank Taylor

Department of Computing, Imperial College, London SW7 2BZ, UK
Tel: +44 71 589 5111 x5028, Fax: +44 71 581 8024
email: phjk@doc.ic.ac.uk

## Introduction

An important recent idea in software technology for distributed-memory multicomputers is the use of annotations to control data partitioning and placement, relying on a compiler to infer the necessary process placement and communications. We have been developing a declarative language for controlling data partitioning and placement in functional programs, called Caliban. It is interesting because the annotation language builds on the power of the functional language, allowing the user to reuse code developed for the computational part of the problem when expressing how it should be distributed across a parallel computer. The key to doing this is partial evaluation, that is symbolically executing the program until the annotation is in its primitive form, and the partitioning is clear.

While this work was developed primarily in the context of parallel functional programming, there is the interesting prospect of applying it to the problem of controlling the distribution of data in traditional languages.

## The FAST Project

The FAST project was set up to produce a highly optimised compiler for a parallel functional language. We are targeting a distributed memory multicomputer, and consequently have to tackle the problems of controlling process partitioning, placement, and communications. The solution we adopt is to provide the programmer with the means to solve them, with as high a level of abstraction as possible. Our approach is to allow the programmer to specify placement of expressions. The compiler can then examine the code and work out where the computations to evaluate these expressions need to be placed. This forms a compile-time static process network — values are calculated on processors and sent to other processors that need it.

The types of data that can be placed are *streams* (head strict lazy lists — sequences of normal form elements modelling a communications link). This means that when the compiler extracts the computation placement from the program and a programmer written annotation we end up with something that looks just like a process network in the classic style.

## The language

Our source language is a variant of Haskell (a lazy, higher order and polymorphic functional language [HWe90]) with the annotation language, called Caliban, built on top. The version of Caliban we are implementing is a restriction of the language described in [Kel89]. Annotations are introduced using a new keyword, **moreover**, to the language. Annotations are of type `Placement`, a standard Haskell constructed data type. In this lies the power of our approach. The annotations are written in the source language, and so we can use program transformations to manipulate these annotations in a controlled way. We also have access to the complete power of the Haskell notation to construct annota-

*School of Applied Science, Nan Yang Technological University, Nan Yang Avenue, Singapore 2263.

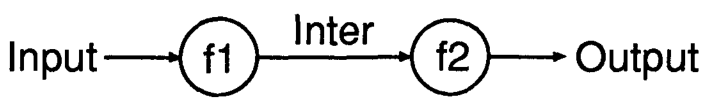

Figure 1: A simple pipeline process network.

tions. This allows the programmer to build up libraries of commonly used process network structures that can be used as easily as normal higher order functions. Standard topologies include pipelines, fans, local-neighbourhood operations, etc.

To build a pipeline program that applies two stream functions to the input stream, as in figure 1, we would write:

```
pipe :: [Stream] -> Placement
pipe [ ] = NoPlace
pipe [a] = Node a
pipe (a:(b:xs))
 = (Node a) And (Arc a b) And (pipe (b:xs))

main :: Stream -> Stream
main input
 = result
   moreover
    (pipe [input, inter, result])
   where
    result = f2 inter
    inter = f1 input
```

## The Transformations

We need to produce a compile time representation of the static process network so that we can produce code that constructs it at run-time. The first phase of this transformation is to use *partial evaluation* to collect annotations into one static annotation. This technique is called *simplification*. The process of simplification removes any calls to *network forming operators* (NFOs) such as **pipe**, to leave one simple annotation representing a static network. This is done by partially evaluating the annotation to *annotation normal form* — all `Placement` constructors are showing and each of the annotated expressions is evaluated to WHNF. This may cause some of the program to be evaluated. It is therefore important that sharing is maintained between the annotation and the program code. In some cases placed expressions are introduced during the simplification that are not already explicit in the original program. This sharing is achieved by viewing **where** definitions as naming nodes in the program graph and using term-graph rewriting[BvEG+87] as the evaluation mechanism.

The annotation (after simplification) is in annotation normal form and looks like this:

```
(Node input) And (Arc input inter) And
(Node inter) And (Arc inter result) And
(Node result)
```

The NFO, **pipe**, allows us to change the number or type of functions in the pipeline with only a minimal adjustment in the code. The simplifier is then responsible for the expansion of the NFO to normal form.

This annotation now tells us which data is to be computed separately. The next stage in our compiler is to produce information about the placement of the computation of that data. With the annotation we know where the data is placed and we can use this to extract information about what code needs to be where. This stage is called *network extraction*. Each placed stream is transformed into a function whose argument specifies which other placed streams it references. This is done by a form of lambda-lifting, where we lift references to placed streams out of each `Node` annotated expression. We can then replace the **moreover** annotation with a call to a system primitive called **procnet**. This special primitive actually implements the run-time parallelism. Here is the example after network extraction:

```
main :: Stream -> Stream
main input
 = procnet [result', inter']
           [((0,1),(1,1)),
            ((1,1),(2,1)),
            ((2,1),(0,1))]
   where
    result' [inter] = f2 inter
    inter' [input] = f1 input
```

Notice how the streams `inter` and `input` have been lifted out of the expressions `result` and `inter` respectively to produce functions that capture the work that each node needs to perform.

`Procnet` takes two arguments. The first is a list of functions that represent the job that each node in the process network should perform. The second is a list of connections that describes the communications that are required between processes.

We can now compile this standard functional program like any other and rely on the run-time implementation of procnet to place the processes that represent result and inter and connect them up.

## Composing placements

The central problem with distributed-memory programs is to minimise data movement when program modules are composed. We examine here a slightly more involved example to illustrate how this can be addressed using Caliban.

### A parallel local-neighbourhood operator

As our example we employ a 1-dimensional finite difference operator, in turn an instance of a 1-dimensional local-neighbourhood operation. This can be captured by a higher-order function ApplyRowLno:

```
ApplyRowLno f <... $v_i$ ... >
 = <... (f $v_{i-1}$ $v_i$ $v_{i+1}$) ... >
```

Here angle brackets denote a vector; we will not give a full definition of the function here. We will assume that the first and the last elements are connected in a ring to avoid boundary conditions which complicate what follows.

A typical local neighbourhood operation is a low-pass filtering operation:

```
ApplyRowLno smooth InitVector
where
smooth $v_{i-1}$ vi $v_{i+1}$ = ($v_{i-1}$ + vi + $v_{i+1}$) / 3
```

Commonly we are interested in re-applying such an operator repeatedly. A neat way to do this is use the function map3 smooth:

```
map3 smooth [...ai...][...bi...][...ci...]
 = [... ai+bi+ci/3 ...]
```

We can use this with the ApplyRowLno functional to construct a vector of stream processing functions:

```
ApplyRowLno (map3 smooth) InitStreams
```

Here, the input is a vector of streams, allowing the operator to be applied to a sequence of vectors of elements. Typically we would in fact apply the difference operator repeatedly to the same input:

```
iterates
 = ApplyRowLno (map3 smooth)
               (join InitVector iterates)
```

Here the function join builds a vector of streams, each beginning with an element from InitVector, followed by elements from iterates when they become available:

```
join <...vi...><...vsi...>
 = <...vi:vsi...>
```

We can use Caliban to express the requirement that each of the resulting streams is computed on a separate processing element, communicating with its nearest neighbours, as follows:

```
ParallelRowLno InitVector
 = iterates
   moreover
    All (ApplyRowLno nbours iterates)
   where
    iterates
     = ApplyRowLno
        (map3 smooth)
        (join InitVector iterates)
     All = reduce (And)
     nbours vim1 vi vip1
      = Node vi And Arc vi vim1
        And Arc vi vip1
```

Following a very common pattern we use the functional ApplyRowLno to construct its annotation.

### A parallel reduction operator

Similarly, a reduction operator implemented using a binary tree can be written as follows:

```
ParallelReduce op < v > = v
ParallelReduce op v
 = result
   moreover
    Node result And Arc result v1
    And Arc result v2
   where
    result =  op v1 v2
    (v1, v2) = split v
```

Given a particular vector, the definition of ParallelReduce can be unfolded (split has to be invoked) to expose the moreover clauses which are then combined to yield a single representation of the process network.

### Composing process networks

To repeat the finite-difference operator until convergence is achieved, we need to sum the results at each step:

```
solve InitVector
 = takeWhen iterates
           (map (< epsilon) residuals)
   where
    iterates = ParallelRowLno InitVector
    residuals = ParallelReduce
                 (map2 (+)) iterates
```

where takeWhen returns the first iterate for which the comparison is true.

The process network for this computation is a composition of the chain of PEs computing the smoothing functions, together with the tree performing the summation. To construct a static process network, the compiler must transform the program so that it has a single moreover annotation in normal form. This can be done by transforming ParallelRowLno and ParallelReduce so that they return a pair (result, annotation), carrying the data structure describing how the computation is to be distributed. These are then combined to yield the overall annotation:

```
solve InitVector
 = takeWhen iterates
           (map (< epsilon) residuals)
   moreover
    ann1 And ann2
   where
    (iterates, ann1)
     = ParallelRowLno' InitVector
    (residuals, ann2)
     = ParallelReduce' (map2 (+)) iterates
```

Partial evaluation and network extraction can then be used to construct the static assignment as before.

## Conclusion

The notation provides a clean and simple method of partitioning the program into a static process network. Libraries of NFOs can be built up that allow easy construction of these process networks. The approach is very much like that of "skeletons" (as advocated, for example, by Cole [Col89]), except here, we allow the programmer to write new skeletons of their own instead of relying on a collection provided with the system.

The transformation techniques of simplification and network extraction provide the programmer with a high level view, whilst allowing the system to implement the program efficiently, exploiting neighbour-to-neighbour communications and uncompromised sequential compilation technology for each of the constituent processes to achieve high sequential speed within each node.

A prototype highly optimised sequential compiler has been developed, and a complete parallel system will be available shortly. [CHK+92]

The work was funded in part by the UK Science and Engineering Research Council and the Department of Trade and Industry, under grant number GR/F 35081. The equipment was funded under grant number GR/G 31079.

## References

[BvEG+87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. 1987. In [dBNT87, pages 141–158].

[CHK+92] Stuart Cox, Shell-Ying Huang, Paul Kelly, Junxian Liu, and Frank Taylor. An implementation of static functional process networks. In *PARLE'92*, LNCS 605, pages 497–512. Springer Verlag, 1992.

[Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman/MIT Press, 1989.

[dBNT87] J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors. PARLE, *Parallel Architectures and Languages Europe*, volume I. Springer Verlag, June 1987. LNCS 258.

[HWe90] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

[Kel89] Paul H.J. Kelly. *Functional Programming for Loosely-coupled Multiprocessors.* Pitman/MIT Press, 1989.