Farms, Pipes, Streams and Reforestation
Type-Directed Parallelisation

David Castro, Kevin Hammond and Susmit Sarkar
University of St Andrews, UK

kevin@kevinhammond.net

IFIP Working Group 2.11 Meeting, Bloomington, Indiana, 23/8/16
RePhrase Project: Refactoring Parallel Heterogeneous Software – a Software Engineering Approach (ICT-644235), 2015-2018, €3.6M budget

8 Partners, 6 European countries
UK, Spain, Italy, Austria, Hungary, Israel

Coordinated by Kevin Hammond, St Andrews
ParaFormance Project: Parallel Patterns for Heterogeneous Multicore Systems (ICT-288570), 2015-2018, £537K budget

Goal: Formation of High-Growth Company of Scale by 2023

Coordinated by Kevin Hammond, St Andrews
The Problem

• We need to choose the best parallel *abstractions*
  • *Algorithmic skeletons* [Cole 1989] implement patterns

• We need a formal way to reason about parallel *structure*
  ▪ Correctness of transformations
  ▪ Reasoning about performance
Example Skeleton: Parallel Task Farms

- Task Farms use a fixed number of workers
  - Each worker applies the same operation \( f \)
  - \( f \) is applied to each of the inputs in a stream.
Example Skeleton: Parallel Pipeline

- Parallel pipelines compose two operations ($f$ and $g$)
  - over the elements of an input stream
  - $f$ and $g$ are run in parallel
Example
Image Merge

Image merging composes two operations, merge and mark

\[
\text{imgMerge} : \text{List}(\text{Img} \times \text{Img}) \rightarrow \text{List Img} \\
\text{imgMerge} = \text{map} \ (\text{merge} \circ \text{mark})
\]

Possible implementations include:

\[
\text{imgMerge}_1 = \text{farm} \ n \ (\text{fun} \ (\text{merge} \circ \text{mark})) \\
\text{imgMerge}_2 = \text{farm} \ n \ (\text{fun} \ \text{mark}) \parallel \text{farm} \ m \ (\text{fun} \ \text{merge}) \\
\text{imgMerge}_3 = \text{farm} \ n \ (\text{fun} \ \text{merge}) \parallel \text{fun} \ \text{mark} \\
\ldots
\]
Choosing an Implementation

Decorate the function type with $\text{IM}(n,m)$

\[
\text{imgMerge : List(Img x Img)} \xrightarrow{\text{IM}(n,m)} \text{List Img}
\]

\[
\text{imgMerge = map (merge \circ \text{mark})}
\]

where

\[
\text{IM}(n, m) = \text{FARM } n \left( \text{FUN } A \right) \ || \ \text{FARM } m \left( \text{FUN } A \right)
\]

Now the type system automatically selects

\[
\text{imgMerge}_2 = \text{farm } n \left( \text{fun mark} \right) \ || \ \text{farm } m \left( \text{fun merge} \right)
\]

We can guarantee that this is functionally equivalent to \text{imgMerge}
Inferring parallel structures

We can leave holes in the types, e.g.

\[ IM(n,m) = _ || FARM m _ \]

replaces \_ with the simplest possible structures

\[ IM(n,m) = \text{min cost} (_ || FARM m _) \]

replaces \_ with the least cost structures.

We can choose the provably least cost skeleton
Basic Semantics
Syntax of Skeletons

\[ p \in P ::= \text{fun}_T f \mid p_1 \parallel p_2 \mid \text{dc}_{n,T,F} f g \mid \text{farm } n \ p \ | \ \text{fb} \ p \]

- `fun_T` lifts an atomic function to a collection type \( T \)
- `dc` represents divide and conquer over collection \( T \)
- `fb` introduces feedback
Base semantics, $S$ ($\rho$ is a global environment of function defns)

$$S[p : T A \rightarrow T B] : [A \rightarrow B]$$

$$S[\text{fun } f] = \hat{\rho}(f)$$

$$S[p_1 \| p_2] = S[p_2] \circ S[p_1]$$

$$S[\text{farm } n \ p] = S[p]$$

$$S[\text{fb } p] = \text{iter } S[p]$$

$$S[\text{dc}_{n,T,F} \ f \ g] = \text{cata}_F (\hat{\rho}(f)) \circ \text{ana}_F (\hat{\rho}(g))$$

Lifted to a streaming form, $P$ over collection type $T$

$$[p : T A \rightarrow T B] : [T A \rightarrow T B]$$

$$[p] = \text{map}_T S[p]$$
Morphisms for Divide-and-Conquer

\[ S[dc_{n,T,F} f g] = \text{cata}_F (\hat{\rho}(f)) \circ \text{ana}_F (\hat{\rho}(g)) \]

Catamorphism (fold)

\[ \text{cata}_F f : (F A \to A) \to \mu F \to A \]
\[ \text{cata}_F f = f \circ F (\text{cata}_F f) \circ \text{out}_F \]

Anamorphism (unfold)

\[ \text{ana}_F g : (A \to F A) \to A \to \mu F \]
\[ \text{ana}_F g = \text{in}_F \circ F (\text{ana}_F g) \circ g \]
Morphisms for Streams

Given a bifunctor, $G$, maps over collection $T$ are

$$\left[p\right] = \text{map}_T S[p]$$

$$\text{map}_T f = \text{cata}_{GA}(\text{in}_{GB} \circ Gf \circ \text{id})$$

$$= \text{ana}_{GB}(Gf \circ \text{id} \circ \text{out}_{GA})$$
Iteration

Easy to define using the fix-point combinator, \( Y f = f (Y f) \)

\[
\text{iter} : (A \rightarrow A + B) \rightarrow A \rightarrow B
\]

\[
\text{iter } f = Y (\lambda g. (g \triangleright \text{id}) \circ f)
\]
Generalising
Recursion Patterns
**Hylomorphisms** are general recursion patterns

\[
\text{hylo}_F \quad : \quad (F \; B \to B) \to (A \to F \; A) \to A \to B
\]

\[
\text{hylo}_F \; f \; g \quad = \quad f \circ F \; (\text{hylo}_F \; f \; g) \circ g
\]

For \( \text{hylo}_F \; f \; g \)
- \( \mu F \) recursive call tree
- \( g \) how inputs are split
- \( f \) how results are combined

**map, cata** and **ana** are just special cases of hylomorphisms

\[
T \; A \quad = \quad \mu (F \; A)
\]

\[
\text{map}_T \; f \quad = \quad \text{hylo}_{F \; A} \; (\text{in}_F \; B \circ (F \; f \; \text{id})) \; \text{out}_{F \; A},
\] where \( A = \text{dom}(f) \) and \( B = \text{codom}(f) \)

\[
\text{cata}_F \; f \quad = \quad \text{hylo}_F \; f \; \text{out}_F
\]

\[
\text{ana}_F \; f \quad = \quad \text{hylo}_F \; f \; \text{in}_F \; f
\]
Example: Quicksort

\[
\begin{align*}
\text{split} & : \text{List } A \rightarrow T \text{ } A (\text{List } A) \\
\text{split } \text{nil} & = \text{inj}_1 () \\
\text{split } \text{(cons } x \text{ } l) & = \text{inj}_2 (x, \text{leq } x \text{ } l, \text{gt } x \text{ } l) \\
\text{join} & : T \text{ } A (\text{List } A) \rightarrow \text{List } A \\
\text{join } \text{(inj}_1 ()) & = \text{nil} \\
\text{join } \text{(inj}_2 (x, l, r)) & = l ++ \text{cons } x \text{ } r \\
\text{qsort} & : \text{List } A \rightarrow \text{List } A \\
\text{qsort} & = \text{cata}_{T \text{ } A} \text{join } \circ \text{ana}_{T \text{ } A} \text{split} \\
\text{or} & \\
\text{qsort} & = \text{hylo}_{T \text{ } A} \text{join } \text{split}
\end{align*}
\]
All the World’s a Hylomorphism!

\[
e \in E := s \mid \text{par}_T p
\]
\[
s \in S := f \mid e_1 \circ e_2 \mid \text{hylo}_F e_1 e_2
\]
\[
p \in P := \text{fun } s \mid p_1 \parallel p_2 \mid \text{dc}_{n,F} s_1 s_2 \mid \text{farm } n p \mid \text{fb } p
\]

\[
[\text{par}_T p] = \text{map}_T S[p]
\]
\[
\ldots
\]
\[
S[\text{fun } e] = [e]
\]
\[
S[\text{dc}_{n,F} e_1 e_2] = \text{hylo}_F [e_2] [e_1]
\]
\[
\ldots
\]

\[
\text{iter } f = \text{hylo}_{(+B)} (id \nabla id) f
\]
Structure in Types
Introducing Parallel Patterns

- The type system uses a structure-equivalence relation that describes when two programs are extensionally equivalent.
- The type-checking algorithm needs to decide these structure-equivalences.

The type-checking algorithm also needs to unify structures, modulo this structure-equivalence relation.
Syntax of Structured Types

\[ e : A \xrightarrow{\sigma} B \]

\[
\begin{align*}
\sigma & \in \Sigma \\
\sigma_s & \in \Sigma_s \\
\sigma_p & \in \Sigma_p \\
\sigma_s & ::= A \mid \sigma \circ \sigma \mid HYLO_F \sigma \sigma \\
\sigma_p & ::= FUN \sigma_s \mid DC_{n,F} \sigma_s \sigma_s \\
& \mid \sigma_p \| \sigma_p \mid FARM_n \sigma_p \mid FB \sigma_p
\end{align*}
\]
Structure-Annotated Type Rules

\[ \rho(f) = A \rightarrow B \]
\[ \vdash e_1 : B \xrightarrow{\sigma_1} C \]
\[ \vdash e_2 : A \xrightarrow{\sigma_2} B \]
\[ \vdash e_1 \circ e_2 : A \xrightarrow{\sigma_1 \circ \sigma_2} C \]
\[ \vdash e_1 : F B \xrightarrow{\sigma_1} B \]
\[ \vdash e_2 : A \xrightarrow{\sigma_2} FA \]
\[ G = \text{base } F \]
\[ \vdash \text{hylo}_F e_1 e_2 : A \xrightarrow{\text{HYLO}_G \sigma_1 \sigma_2} B \]
\[ \vdash p : T A \xrightarrow{\sigma} T B \]
\[ F = \text{base } T \]
\[ \vdash \text{par}_T p : T A \xrightarrow{\text{PAR}_F \sigma} T B \]

\[ \vdash s : A \xrightarrow{\sigma} B \]
\[ \vdash \text{fun } s : T A \xrightarrow{\text{FUN } \sigma} T B \]
\[ \vdash s_1 : F B \xrightarrow{\sigma_1} B \]
\[ \vdash s_2 : A \xrightarrow{\sigma_2} FA \]
\[ G = \text{base } F \]
\[ \vdash d_{Cn,F} s_1 s_2 : T A \xrightarrow{d_{Cn,G} \sigma_1 \sigma_2} T B \]
\[ n : N \]
\[ \vdash p : T A \xrightarrow{\sigma} T B \]
\[ \vdash p_1 : T A \xrightarrow{\sigma_1} T B \]
\[ \vdash p_2 : T B \xrightarrow{\sigma_2} T C \]
\[ \vdash p_1 \parallel p_2 : T A \xrightarrow{\sigma_1 \parallel \sigma_2} T C \]
\[ \vdash p : T A \xrightarrow{\sigma} T (A + B) \]
\[ \vdash \text{fb } p : T A \xrightarrow{\text{FB } \sigma} T B \]
Convertibility

Plus some other rules derived from the hylomorphism laws. We use this to produce a confluent rewriting system.
Parallelism Erasure

Rewrite rules derived from convertibility

\[
\begin{align*}
FARM_n \sigma_p & \rightsquigarrow_p \sigma_p \\
\text{FUN } \sigma_1 \parallel \text{ FUN } \sigma_2 & \rightsquigarrow_p \text{ FUN } (\sigma_1 \circ \sigma_2) \\
\text{DC}_{n,F} \sigma_1 \sigma_2 & \rightsquigarrow_p \text{ FUN } (\text{HYLO}_F \sigma_1 \sigma_2) \\
\text{FB (FUN } \sigma_1) & \rightsquigarrow_p \text{ FUN } (\text{ITER } \sigma_1) \\
\text{PAR}_T \text{ (FUN } \sigma_s) & \rightsquigarrow_p \text{ MAP}_T \sigma_s
\end{align*}
\]

Repeated to produce a confluent rewriting system

\[
\begin{align*}
\text{erase } & : \ \Sigma \rightarrow \overline{\Sigma}_s \\
\text{erase } \sigma & = \sigma', \text{ s.t. } \sigma \rightsquigarrow^*_p \sigma' \wedge \nexists \sigma'' \text{ s.t. } \sigma'' \rightsquigarrow_p \sigma''
\end{align*}
\]
Normalisation

The rewrite rules are derived from basic laws

\[
\begin{align*}
\text{HYLO}_F \sigma_1 \sigma_2 & \sim_s \quad \text{CATAF} \sigma_1 \circ \text{ANAF} \sigma_2 \iff \sigma_1 \neq \text{IN} \land \sigma_2 \neq \text{OUT} \quad \text{(HYLO-SPLIT)} \\
\text{CATAF} (\sigma_1 \circ F \sigma_2) & \sim_s \quad \text{CATAF} \sigma_1 \circ \text{MAPF} \sigma_2 \iff \sigma_1 \neq \text{IN} \quad \text{(CATA-SPLIT)} \\
\text{ANAF} (F \sigma_1 \circ \sigma_2) & \sim_s \quad \text{MAPF} \sigma_1 \circ \text{ANAF} \sigma_2 \iff \sigma_2 \neq \text{OUT} \quad \text{(ANA-SPLIT)}
\end{align*}
\]

Used to define a normalisation procedure

\[
\begin{align*}
\text{norm}_s \sigma & = \quad \sigma', \text{ s.t. } \sigma \sim_s^* \sigma' \land \exists \sigma'' \text{ s.t. } \sigma' \sim_s \sigma'' \\
\text{norm} & = \quad \text{norm}_s \circ \text{erase}
\end{align*}
\]

We can now prove equivalence of two parallel terms by:

i) erasing parallelism using erase,

ii) normalising using norm, and

iii) testing for equivalence
Example
QuickSort Revisited

Start with a sequential version

$$qsorts : \text{List}(\text{List } A) \rightarrow \text{List}(\text{List } A)$$
$$qsorts = \text{map}_{\text{List}}(\text{hylo}_F A \text{ merge div})$$

To create a parallel divide-and-conquer version, we need to decide

$$\text{MAP}_L(\text{HYLO}_F A A) \simeq \text{PAR}_L(\text{DC}_{n,F} A A)$$

This is easily done using a simple parallelism erasure
Inferring More Complex Parallel Structure

Now consider a more complex structure

\[ \text{MAP}_L \left( \text{HYLO}_F \ A \ A \right) \cong \text{PAR}_L \left( \text{FARM}_n \ | \ | \ - \right) \]

Normalisation of the LHS (using HYLO-SPLIT etc) gives

\[ \text{MAP}_L \left( \text{HYLO}_F \ A \ A \right) \leadsto^* \text{MAP}_L \left( \text{CATA}_F \ A \right) \circ \text{MAP}_L \left( \text{ANA}_F \ A \right) \]

Parallelism erasure on the RHS gives

\[ \text{PAR}_L \left( \text{FARM} \ n \ m_2 \ | \ | \ m_1 \right) \leadsto^* \text{MAP}_L \left( m'_1 \circ m'_2 \right) \]
\[ \delta = \{ m_1 \sim \text{FUN} \ m'_1, m_2 \sim \text{FUN} \ m'_2 \} \]

Normalisation of the RHS gives

\[ \text{MAP}_L \left( m'_1 \circ m'_2 \right) \leadsto^* \text{MAP}_L \ m'_1 \circ \text{MAP}_L \ m'_2 \]
Inferring More Complex Parallel Structure (2)

We need to unify the normalised forms

\[ \text{MAP}_L (\text{CAT}_F \ A) \circ \text{MAP}_L (\text{ANA}_F \ A) \]

and

\[ \text{MAP}_L (\text{CAT}_F \ A) \circ \text{MAP}_L (\text{ANA}_F \ A) \sim \text{MAP}_L m'_1 \circ \text{MAP}_L m'_2 \\
\Rightarrow \Delta_1 = \{ m'_1 \sim \text{CAT}_F \ A, m'_2 \sim \text{ANA}_F \ A \} \]

\[ \Delta = \{ \delta \} \otimes \Delta_1 \]
Inferrring More Complex Parallel Structure (3)

Substituting back gives us the desired parallel form

\[ \text{PAR}_{L} (\text{FARM}_n (\text{FUN} (\text{ANA}_F A))) \parallel \text{FUN} (\text{CATA}_F A)) \]

We can then use equivalence to give the actual program

\[ \text{map}_{\text{List}} (\text{hylo}_{F}^{A} \text{merge} \text{ div}) \sim^{*} \]
\[ \text{par}_{\text{List}} (\text{farm} \ n (\text{fun} (\text{ana}_{F} A \text{ div})) \parallel \text{fun} (\text{cata}_{F} A \text{merge})) \]
Costs

For 1000 lists of 30,000,000 elements

\[
\text{qsorts} : \text{List}(\text{List } A) \xrightarrow{\text{min cost}} \text{List}(\text{List } A)
\]

\[
\begin{align*}
\text{cost} & \left( \text{PAR}_L \left( \text{DC}_{n,F} A_{c_1} A_{c_2} \right) \right) \text{ sz} \\
& = \max \left\{ \max \left\{ c_2 \left( |A_{c_2}|^i \text{ sz} \right) \\
& \quad , \text{cost} \left( \text{HYLO}_F A_{c_1} A_{c_2} \right) \left( |A_{c_2}|^n \text{ sz} \right) \\
& \quad , \max \left\{ c_1 \left( |A_{c_1}|^i |A_{c_2}|^n \text{ sz} \right) \right\} \right\} + \kappa_3(n) = 42602.72ms \\
\end{align*}
\]

\[
\begin{align*}
\text{cost} & \left( \text{PAR}_L \left( \text{FARM}_n \left( \text{FUN} \left( \text{ANAL}_L A_{c_2} \right) \right) \parallel \left( \text{FUN} \left( \text{CATAL}_L A_{c_1} \right) \right) \right) \right) \text{ sz} \\
& = 27846.13ms \\
\end{align*}
\]

\[
\begin{align*}
\text{cost} & \left( \text{PAR}_L \left( \text{FARM}_n \left( \text{FUN} \left( \text{HYLO}_F A_{c_1} A_{c_2} \right) \right) \right) \right) \text{ sz} \\
& = 32179.77ms \\
\end{align*}
\]
Predicted v. Actual Speedup

Image Convolution of 500 images on *titanic*, a 2.4GHz 24-core, AMD Opteron 6176 architecture, running CentOS Linux 2.6.18-274.e15. Dashed lines are predictions.
Conclusion
Conclusions

• First-ever treatment of parallelism that reflects parallel structure in types

• Several advantages to exposing parallel structure in types
  • clear separation between the structure and the functionality
  • documentation of how a program was parallelized
  • easy to change the parallel structure of a program without modifying the functional behaviour

• Reasoning about costs of different parallel structure is very powerful
  ▪ Automatically find suitable parallel structures
  ▪ Compile-time information about the run-time behaviour
  ▪ Automatically rewrite programs to minimize costs
Future Work

- Other patterns, e.g. stencil and bulk synchronous parallelism
- More detailed cost models (see e.g. Hammond et al, 2016)
- Dynamic Analysis is also possible
- Allow (certain kinds of) side effects in the workers
- Implement back-ends. Run our structured programs!
- Larger case studies
Farms, Pipes, Streams and Reforestation: Reasoning about Structured Parallel Processes using Types and Hylomorphisms

David Castro, Kevin Hammond, Susmit Sarkar
School of Computer Science, University of St Andrews, St Andrews, Scotland
{dc84, kh8, ss265}@st-andrews.ac.uk

Abstract
The increasing importance of parallelism has motivated the creation of better abstractions for writing parallel software, including structured parallelism using nested algorithmic skeletons. Such approaches provide high-level abstractions that avoid common problems, such as race conditions, and often allow strong cost models to be defined. However, choosing a combination of algorithmic skeletons that yields good parallel speedups for a program on some platform is critical, and the decision is often made using heuristics. This paper presents a new approach, a type-based mechanism that enables us to reason about the safe introduction of parallelism, while also providing a good abstraction to reason about cost. This mechanism exploits strong program structure in the form of structured parallel processes [3], combined with properties of hylomorphisms [22].

1.1 Motivating Example
We introduce our approach using a simple example, image merge, which merges pairs of images taken from an input stream. We start
THANK YOU!

http://rephrase-ict.eu
http://paraphrase-ict.eu

@rephrase_eu