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

THE FRAMEWORK PROGRAMME FOR RESEARCH AND INNOVATION
HORIZON 2020

University of St Andrews

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

IBM

evopro it's possible!

Universidad Carlos III de Madrid

PRQA Programming Research

scch software competence center hagenberg

UNIVERSITÀ DI PISA

UNIVERSITÀ DEGLI STUDI DI TORINO
ALMA UNIVERSITAS TAURINENSIS

REPHRASE

Scottish Enterprise

University of 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

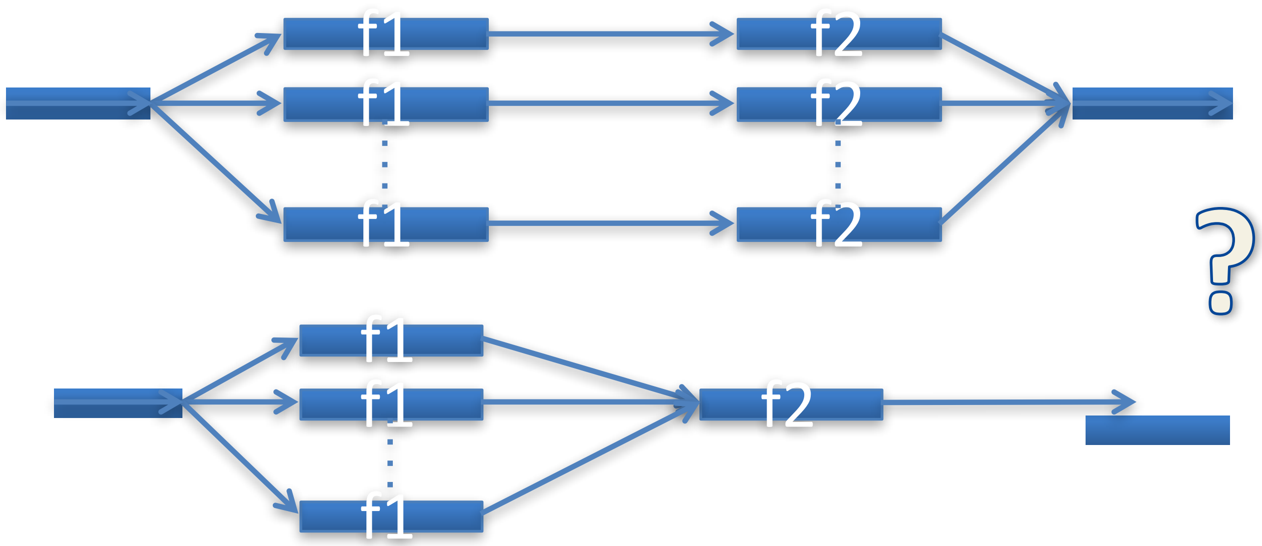
Coordinated by Kevin Hammond, St Andrews

ParaFormance™

The Problem

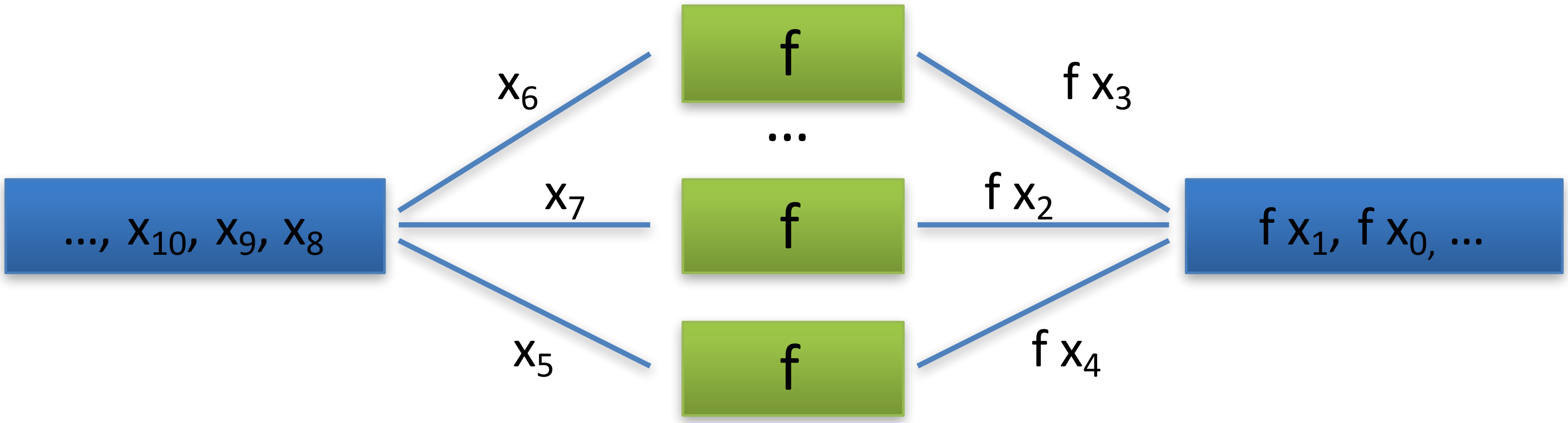
- We need to choose the best parallel abstractions
  - *Algorithmic skeletons* [Cole 1989] implement patterns

- We need a formal way to reason about parallel structure
  - Correctness of transformations
  - Reasoning about performance



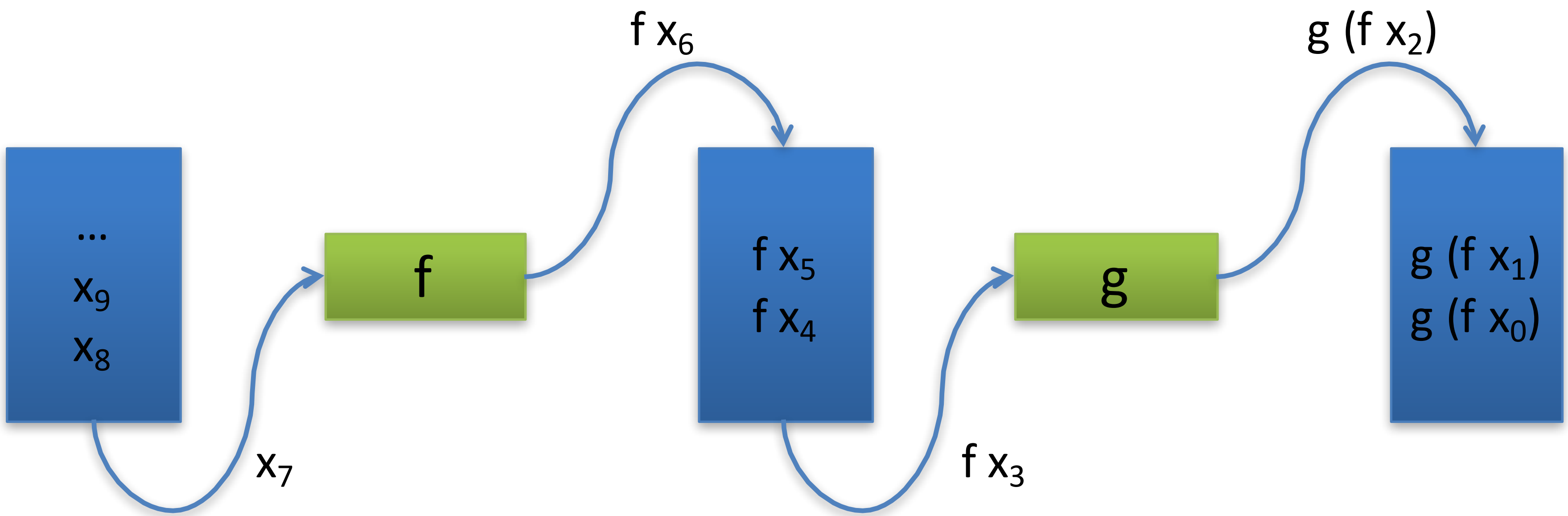
Example Skeleton: Parallel Task Farms

- **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:

$$\text{imgMerge}_1 = \text{farm}\ n\ (\text{fun}\ (\text{merge} \circ \text{mark}))$$
$$\text{imgMerge}_2 = \text{farm}\ n\ (\text{fun mark}) \parallel \text{farm}\ m\ (\text{fun merge})$$
$$\text{imgMerge}_3 = \text{farm}\ n\ (\text{fun merge}) \parallel \text{fun mark}$$
$$\ldots$$

Choosing an Implementation

University of St Andrews

Decorate the function type with IM(n,m)

$$\mathsf{imgMerge} \;:\; \mathsf{List}(\mathsf{Img} \times \mathsf{Img}) \xmapsto{\mathrm{IM}\ (n,m)} \mathsf{List\ Img}$$
$$\mathsf{imgMerge} \;=\; \mathsf{map}\ (\mathsf{merge} \circ \mathsf{mark})$$

where

$$\textsc{im}(n, m) \;=\; \textsc{farm}\ n\ (\textsc{fun a}) \parallel \textsc{farm}\ m\ (\textsc{fun a})$$

Now the type system automatically selects

$$\mathsf{imgMerge}_2 \;=\; \mathsf{farm}\ n\ (\mathsf{fun\ mark}) \parallel \mathsf{farm}\ m\ (\mathsf{fun\ merge})$$

We can *guarantee* that this is *functionally equivalent* to 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) = min cost (_ || FARM m _)

replaces _ with the least cost structures.

We can choose the provably least cost skeleton

Basic Semantics

Syntax of Skeletons

University of St Andrews

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

fun_T	lifts an atomic function to a collection type T
dc	represents divide and conquer over collection T
fb	introduces feedback

Skeleton Denotational Semantics

University of St Andrews

Base semantics, S (ρ is a global environment of function defns)

$$
\begin{array}{lcl}
\mathcal{S}[\![p \;:\; T\,A \rightarrow T\,B]\!] & : & [\![A \rightarrow B]\!] \\
\mathcal{S}[\![\mathsf{fun}\ f]\!] & = & \hat{\rho}(f) \\
\mathcal{S}[\![p_1 \ \|\ p_2]\!] & = & \mathcal{S}[\![p_2]\!] \circ \mathcal{S}[\![p_1]\!] \\
\mathcal{S}[\![\mathsf{farm}\ n\ p]\!] & = & \mathcal{S}[\![p]\!] \\
\mathcal{S}[\![\mathsf{fb}\ p]\!] & = & iter\ \mathcal{S}[\![p]\!] \\
\mathcal{S}[\![\mathsf{dc}_{n,T,F}\ f\ g]\!] & = & cata_F\ (\hat{\rho}(f)) \circ ana_F\ (\hat{\rho}(g))
\end{array}
$$

Lifted to a streaming form, P over collection type T

$$
\begin{array}{lcl}
[\![p \;:\; T\,A \rightarrow T\,B]\!] & : & [\![T\,A \rightarrow T\,B]\!] \\
[\![p]\!] & = & map_T\ \mathcal{S}[\![p]\!]
\end{array}
$$

Morphisms for Divide-and-Conquer

$$\mathcal{S}[\![\mathsf{dc}_{n,T,F}\ f\ g]\!] \quad = \quad cata_F\ (\hat{\rho}(f)) \circ ana_F\ (\hat{\rho}(g))$$

Catamorphism (fold)

$$\begin{array}{lcl} cata_F & : & (F\ A \rightarrow A) \rightarrow \mu F \rightarrow A \\ cata_F\ f & = & f \circ F\ (cata_F\ f) \circ out_F \end{array}$$

Anamorphism (unfold)

$$\begin{array}{lcl} ana_F & : & (A \rightarrow F\ A) \rightarrow A \rightarrow \mu F \\ ana_F\ g & = & in_F \circ F\ (ana_F\ g) \circ g \end{array}$$

Morphisms for Streams

University of St Andrews

$$[\![p]\!] \quad = \quad map_T\, S[\![p]\!]$$

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

$$\begin{aligned} map_T\, f &= cata_{G\,A}(in_{G\,B} \circ G\, f\, id) \\ &= ana_{G\,B}(G\, f\, id \circ out_{G\,A}) \end{aligned}$$

Iteration

$$\mathcal{S}[\![\mathsf{fb}\ p]\!] \quad = \quad iter\ \mathcal{S}[\![p]\!]$$

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

$$\begin{array}{lcl} iter & : & (A \to A + B) \to A \to B \\ iter\ f & = & \mathsf{Y}\ (\lambda\ g.(g \nabla id) \circ f) \end{array}$$

Generalising Recursion Patterns

Hylomorphisms

Hylomorphisms are general recursion patterns

$$
\begin{array}{lcl}
hylo_F & : & (F\ B \to B) \to (A \to F\ A) \to A \to B \\
hylo_F\ f\ g & = & f \circ F\ (hylo_F\ f\ g) \circ g
\end{array}
$$

For $hylo_F$ f g

	μF	recursive call tree
	g	how inputs are split
	f	how results are combined

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

$$
\begin{array}{lcl}
T\ A & = & \mu(F\ A) \\
map_T\ f & = & hylo_{F\ A}\ (in_{F\ B} \circ (F\ f\ id))\ out_{F\ A}, \\
 & & \text{where } A = dom(f) \text{ and } B = codom(f) \\
cata_F\ f & = & hylo_F\ f\ out_F \\
ana_F\ f & = & hylo_F\ in_F\ f
\end{array}
$$

Example: Quicksort

University of St Andrews

$$
\begin{array}{lcl}
\mathsf{split} & : & \mathsf{List}\ A \to T\ A\ (\mathsf{List}\ A) \\
\mathsf{split\ nil} & = & inj_1\ () \\
\mathsf{split}\ (\mathsf{cons}\ x\ l) & = & inj_2\ (x,\ \mathsf{leq}\ x\ l,\ \mathsf{gt}\ x\ l)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{join} & : & T\ A\ (\mathsf{List}\ A) \to \mathsf{List}\ A \\
\mathsf{join}\ (inj_1\ ()) & = & \mathsf{nil} \\
\mathsf{join}\ (inj_2\ (x, l, r)) & = & l + \!\! + \ \mathsf{cons}\ x\ r
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{qsort} & : & \mathsf{List}\ A \to \mathsf{List}\ A \\
\mathsf{qsort} & = & cata_{T\ A}\ \mathsf{join} \circ ana_{T\ A}\ \mathsf{split}
\end{array}
$$

or

$$\mathsf{qsort} = hylo_{T\ A}\ \mathsf{join\ split}$$

All the World's a Hylomorphism!

University of St Andrews

$$
\begin{array}{lcl}
e \in E & ::= & s \;\mid\; \mathsf{par}_T\; p \\
s \in S & ::= & f \;\mid\; e_1 \circ e_2 \;\mid\; \mathsf{hylo}_F\; e_1\; e_2 \\
p \in P & ::= & \mathsf{fun}\; s \mid p_1 \,||\, p_2 \mid \mathsf{dc}_{n,F}\; s_1\; s_2 \mid \mathsf{farm}\; n\; p \mid \mathsf{fb}\; p
\end{array}
$$

$$
\begin{array}{lcl}
[\![\mathsf{par}_T\; p]\!] & = & map_T\; \mathcal{S}[\![p]\!] \\
\ldots & & \\
\mathcal{S}[\![\mathsf{fun}\; e]\!] & = & [\![e]\!] \\
\mathcal{S}[\![\mathsf{dc}_{n,F}\; e_1\; e_2]\!] & = & hylo_F\; [\![e_2]\!]\; [\![e_1]\!] \\
\ldots & &
\end{array}
$$

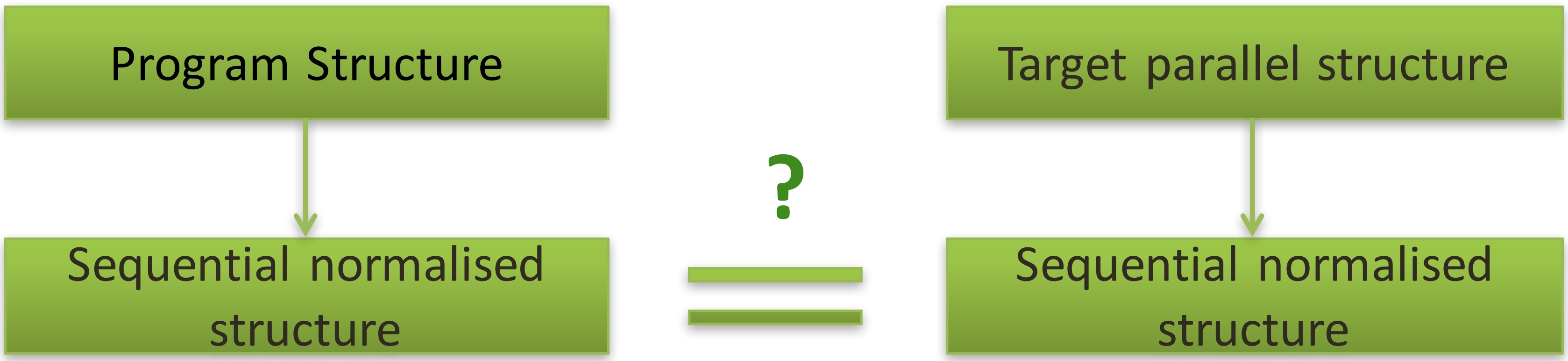
$$
iter\, f \;=\; hylo_{(+B)}\; (id \triangledown id)\; f
$$

University of St Andrews

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

$$e : A \xrightarrow{\sigma} B,$$

$$
\begin{array}{llllll}
\sigma \in \Sigma & ::= & \sigma_{\mathsf{s}} \mid \mathrm{PAR}_F\ \sigma_{\mathsf{p}} & & & \\
\sigma_{\mathsf{s}} \in \Sigma_{\mathsf{s}} & ::= & \mathrm{A} \mid \sigma \circ \sigma \mid \mathrm{HYLO}_F\ \sigma\ \sigma & & & \\
\sigma_{\mathsf{p}} \in \Sigma_{\mathsf{p}} & ::= & \mathrm{FUN}\ \sigma_{\mathsf{s}} & \mid & \mathrm{DC}_{n,F}\ \sigma_{\mathsf{s}}\ \sigma_{\mathsf{s}} & \\
 & \mid & \sigma_{\mathsf{p}} \parallel \sigma_{\mathsf{p}} & \mid & \mathrm{FARM}_n\ \sigma_{\mathsf{p}} & \mid \quad \mathrm{FB}\ \sigma_{\mathsf{p}}
\end{array}
$$

Structure-Annotated Type Rules

$$\frac{\rho(f) = A \to B}{\vdash f \;:\; A \xrightarrow{\mathrm{A}} B}$$

$$\frac{\vdash e_1 \;:\; B \xrightarrow{\sigma_1} C \qquad \vdash e_2 \;:\; A \xrightarrow{\sigma_2} B}{\vdash e_1 \circ e_2 \;:\; A \xrightarrow{\sigma_1 \circ \sigma_2} C}$$

$$\frac{\vdash e_1 \;:\; F\,B \xrightarrow{\sigma_1} B \qquad \vdash e_2 \;:\; A \xrightarrow{\sigma_2} F\,A \qquad G = \mathsf{base}\; F}{\vdash \mathsf{hylo}_F\; e_1\; e_2 \;:\; A \xrightarrow{\mathrm{HYLO}_G\; \sigma_1\; \sigma_2} B}$$

$$\frac{\vdash p \;:\; T\,A \xrightarrow{\sigma} T\,B \qquad F = \mathsf{base}\; T}{\vdash \mathsf{par}_T\; p \;:\; T\,A \xrightarrow{\mathrm{PAR}_F\; \sigma} T\,B}$$

$$\frac{\vdash s \;:\; A \xrightarrow{\sigma} B}{\vdash \mathsf{fun}\; s \;:\; T\,A \xrightarrow{\mathrm{FUN}\; \sigma} T\,B}$$

$$\frac{\vdash s_1 \;:\; F\,B \xrightarrow{\sigma_1} B \qquad \vdash s_2 \;:\; A \xrightarrow{\sigma_2} F\,A \qquad G = \mathsf{base}\; F}{\vdash \mathsf{dc}_{n,F}\; s_1\; s_2 \;:\; T\,A \xrightarrow{\mathrm{DC}_{n,G}\; \sigma_1\; \sigma_2} T\,B}$$

$$\frac{n \;:\; \mathbb{N} \qquad \vdash p \;:\; T\,A \xrightarrow{\sigma} T\,B}{\vdash \mathsf{farm}\; n\; p \;:\; T\,A \xrightarrow{\mathrm{FARM}_n\; \sigma} T\,B}$$

$$\frac{\vdash p_1 \;:\; T\,A \xrightarrow{\sigma_1} T\,B \qquad \vdash p_2 \;:\; T\,B \xrightarrow{\sigma_2} T\,C}{\vdash p_1 \parallel p_2 \;:\; T\,A \xrightarrow{\sigma_1 \parallel \sigma_2} T\,C}$$

$$\frac{\vdash p \;:\; T\,A \xrightarrow{\sigma} T\,(A + B)}{\vdash \mathsf{fb}\; p \;:\; T\,A \xrightarrow{\mathrm{FB}\; \sigma} T\,B}$$

Convertibility

$$\frac{\vdash e : A \xrightarrow{\sigma_1} B \qquad \sigma_1 \equiv \sigma_2}{\vdash e : A \xmapsto{\sigma_2} B}$$

$$\frac{\sigma_1 \equiv_{\mathsf{s}} \sigma_2}{\sigma_1 \equiv \sigma_2} \qquad \frac{\sigma_1 \equiv_{\mathsf{p}} \sigma_2}{\mathrm{PAR}_F\ \sigma_1 \equiv \mathrm{PAR}_F\ \sigma_2}$$

$$\mathrm{PAR}_F\ (\mathrm{FUN}\ \sigma) \equiv \mathrm{MAP}_F\ \sigma \qquad (\mathrm{PAR\text{-}EQUIV})$$

$$\begin{array}{rcll} \mathrm{FUN}\ \sigma_1 \parallel \mathrm{FUN}\ \sigma_2 & \equiv_{\mathsf{p}} & \mathrm{FUN}\ (\sigma_2 \circ \sigma_1) & (\mathrm{PIPE\text{-}EQUIV}) \\ \mathrm{DC}_{n,F}\ \sigma_1\ \sigma_2 & \equiv_{\mathsf{p}} & \mathrm{FUN}\ (\mathrm{HYLO}_F\ \sigma_1\ \sigma_2) & (\mathrm{DC\text{-}EQUIV}) \\ \mathrm{FARM}_n\ \sigma & \equiv_{\mathsf{p}} & \sigma & (\mathrm{FARM\text{-}EQUIV}) \\ \mathrm{FB}(\mathrm{FUN}\ \sigma) & \equiv_{\mathsf{p}} & \mathrm{FUN}\ (\mathrm{ITER}\ \sigma) & (\mathrm{FB\text{-}EQUIV}) \end{array}$$

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

$$
\begin{array}{lll}
\mathrm{FARM}_n\ \sigma_{\mathsf{p}} & \leadsto_{\mathsf{p}} & \sigma_{\mathsf{p}} \\
\mathrm{FUN}\ \sigma_1 \parallel \mathrm{FUN}\ \sigma_2 & \leadsto_{\mathsf{p}} & \mathrm{FUN}\ (\sigma_1 \circ \sigma_2) \\
\mathrm{DC}_{n,F}\ \sigma_1\ \sigma_2 & \leadsto_{\mathsf{p}} & \mathrm{FUN}\ (\mathrm{HYLO}_F\ \sigma_1\ \sigma_2) \\
\mathrm{FB}\ (\mathrm{FUN}\ \sigma_1) & \leadsto_{\mathsf{p}} & \mathrm{FUN}\ (\mathrm{ITER}\ \sigma_1) \\
\\
\mathrm{PAR}_T\ (\mathrm{FUN}\ \sigma_{\mathsf{s}}) & \leadsto_{\mathsf{p}} & \mathrm{MAP}_T\ \sigma_{\mathsf{s}}
\end{array}
$$

Repeated to produce a confluent rewriting system

$$
\begin{array}{lll}
\mathsf{erase} & : & \Sigma \rightarrow \overline{\Sigma}_{\mathsf{s}} \\
\mathsf{erase}\ \sigma & = & \sigma',\ \text{s.t. } \sigma \leadsto_{\mathsf{p}}^{*} \sigma' \ \wedge\ \nexists \sigma'' \text{ s.t. } \sigma'' \leadsto_{\mathsf{p}} \sigma''
\end{array}
$$

Normalisation

The rewrite rules are derived from basic laws

$$\begin{array}{lclcll}
\text{HYLO}_F\ \sigma_1\ \sigma_2 & \rightsquigarrow_s & \text{CATA}_F\ \sigma_1 \circ \text{ANA}_F\ \sigma_2 & \Leftarrow & \sigma_1 \neq \text{IN} \wedge \sigma_2 \neq \text{OUT} & (\text{HYLO-SPLIT}) \\
\text{CATA}_F\ (\sigma_1 \circ F\ \sigma_2) & \rightsquigarrow_s & \text{CATA}_F\ \sigma_1 \circ \text{MAP}_F\ \sigma_2 & \Leftarrow & \sigma_1 \neq \text{IN} & (\text{CATA-SPLIT}) \\
\text{ANA}_F\ (F\ \sigma_1 \circ \sigma_2) & \rightsquigarrow_s & \text{MAP}_F\ \sigma_1 \circ \text{ANA}_F\ \sigma_2 & \Leftarrow & \sigma_2 \neq \text{OUT} & (\text{ANA-SPLIT})
\end{array}$$

...

Used to define a normalisation procedure

$$\begin{array}{lcl}
\text{norm}_s\ \sigma & = & \sigma',\ \text{s.t.}\ \sigma \rightsquigarrow_s^* \sigma' \wedge \nexists \sigma''\ \text{s.t.}\ \sigma' \rightsquigarrow_s \sigma'' \\
\text{norm} & = & \text{norm}_s \circ \text{erase}
\end{array}$$

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

$$\mathsf{qsorts} \;:\; \mathsf{List}(\mathsf{List}\ A) \rightarrow \mathsf{List}(\mathsf{List}\ A)$$
$$\mathsf{qsorts} \;=\; \mathsf{map}_{\mathsf{List}}\ (\mathsf{hylo}_{F\ A}\ \mathsf{merge\ div})$$

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

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

This is easily done using a simple parallelism erasure

Inferrring More Complex Parallel Structure

Now consider a more complex structure

$$\text{MAP}_L\ (\text{HYLO}_F\ \text{A}\ \text{A}) \cong \text{PAR}_L\ (\text{FARM}_n\ \_ \parallel \_)$$

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

$$\text{MAP}_L\ (\text{HYLO}_F\ \text{A}\ \text{A}) \rightsquigarrow^* \text{MAP}_L\ (\text{CATA}_F\ \text{A}) \circ \text{MAP}_L\ (\text{ANA}_F\ \text{A})$$

Parallelism erasure on the RHS gives

$$\text{PAR}_L\ (\text{FARM}\ n\ m_2 \parallel m_1) \rightsquigarrow^* \text{MAP}_L\ (m_1' \circ m_2')$$
$$\delta = \{m_1 \sim \text{FUN}\ m_1', m_2 \sim \text{FUN}\ m_2'\}$$

Normalisation of the RHS gives

$$\text{MAP}_L\ (m_1' \circ m_2') \rightsquigarrow^* \text{MAP}_L\ m_1' \circ \text{MAP}_L\ m_2'$$

Inferrring More Complex Parallel Structure (2)

University of St Andrews

We need to unify the normalised forms

$$\text{MAP}_L\ (\text{CATA}_F\ \text{A})\ \circ\ \text{MAP}_L\ (\text{ANA}_F\ \text{A})$$

and

$$\begin{array}{c}\text{MAP}_L\ (\text{CATA}_F\ \text{A})\ \circ\ \text{MAP}_L\ (\text{ANA}_F\ \text{A}) \sim \text{MAP}_L\ m_1' \circ \text{MAP}_L\ m_2' \\ \Rightarrow \Delta_1 = \{m_1' \sim \text{CATA}_F\ \text{A}, m_2' \sim \text{ANA}_F\ \text{A}\}\end{array}$$

$$\Delta = \{\delta\} \otimes \Delta_1$$

Inferrring More Complex Parallel Structure (3)

Substituting back gives us the desired parallel form

$$\text{PAR}_L\ (\text{FARM}_n\ (\text{FUN}\ (\text{ANA}_F\ \text{A}))\ \|\ \text{FUN}\ (\text{CATA}_F\ \text{A}))$$

We can then use equivalence to give the actual program

$$\mathsf{map}_{\mathsf{List}}(\mathsf{hylo}_{F\ A}\mathsf{merge\ div}) \rightsquigarrow^{*}$$
$$\mathsf{par}_{\mathsf{List}}\ (\mathsf{farm}\ n\ (\mathsf{fun}\ (\mathsf{ana}_{F\ A}\ \mathsf{div}))\ \|\ \mathsf{fun}\ (\mathsf{cata}_{F\ A}\mathsf{merge}))$$

Costs

University of St Andrews

For 1000 lists of 30,000,000 elements

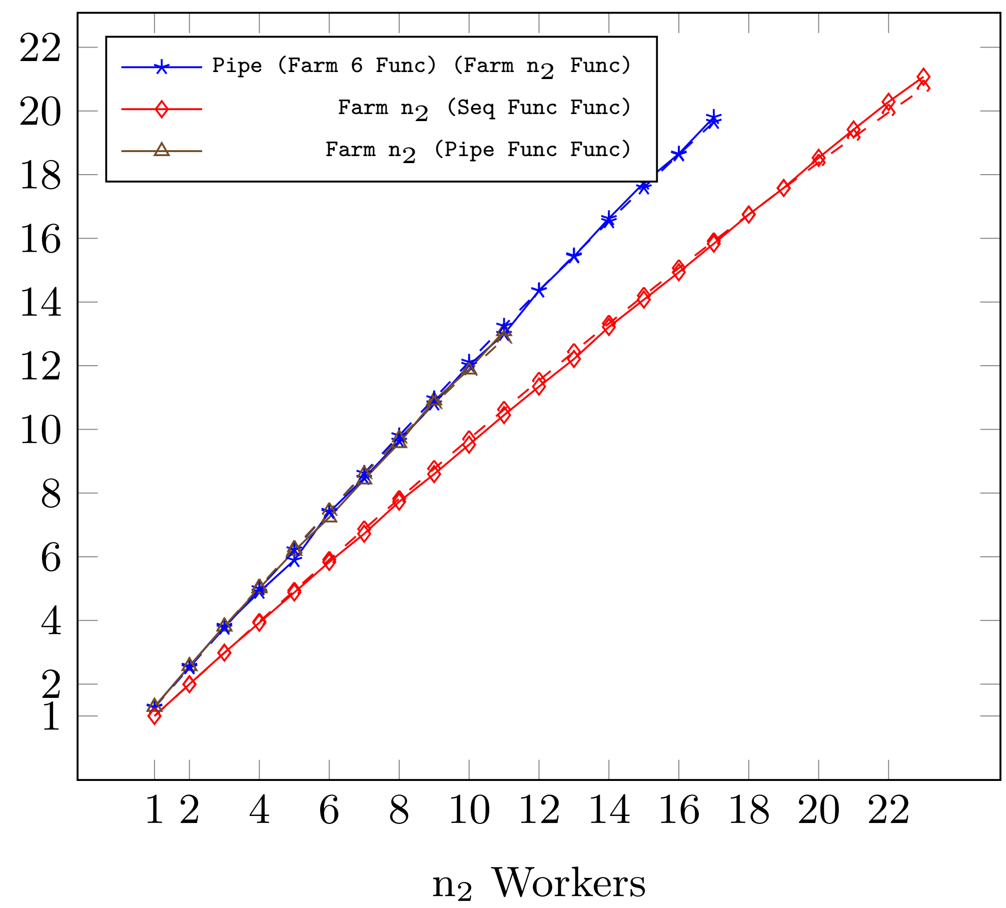
$$\text{qsorts} \;:\; \text{List}(\text{List } A) \xrightarrow{\text{min cost}} \text{List}(\text{List } A)$$

$$\begin{aligned}
&\text{cost } (\text{PAR}_L\ (\text{DC}_{n,F}\ \text{A}_{c_1}\ \text{A}_{c_2}))\ sz \\
&= \max\{ \max_{1 \le i \le n} \{c_2\ (|\text{A}_{c_2}|^i sz) \\
&\quad, \text{cost } (\text{HYLO}_F\ \text{A}_{c_1}\ \text{A}_{c_2})\ (|\text{A}_{c_2}|^n sz) \\
&\quad, \max_{1 \le i \le n} \{c_1\ (|\text{A}_{c_1}|^i |\text{A}_{c_2}|^n sz)\}\} + \kappa_3(n) = 42602.72ms
\end{aligned}$$

$$\begin{aligned}
&\text{cost } (\text{PAR}_L\ (\text{FARM}_n\ (\text{FUN}\ (\text{ANA}_L\ \text{A}_{c_2}))\ \|\ (\text{FUN}\ (\text{CATA}_L\ \text{A}_{c_1}))))\ sz \\
&= 27846.13ms
\end{aligned}$$

$$\begin{aligned}
&\text{cost } (\text{PAR}_L\ (\text{FARM}_n\ (\text{FUN}\ (\text{HYLO}_F\ \text{A}_{c_1}\ \text{A}_{c_2}))))\ sz \\
&= 32179.77ms
\end{aligned}$$

Predicted v. Actual Speedup



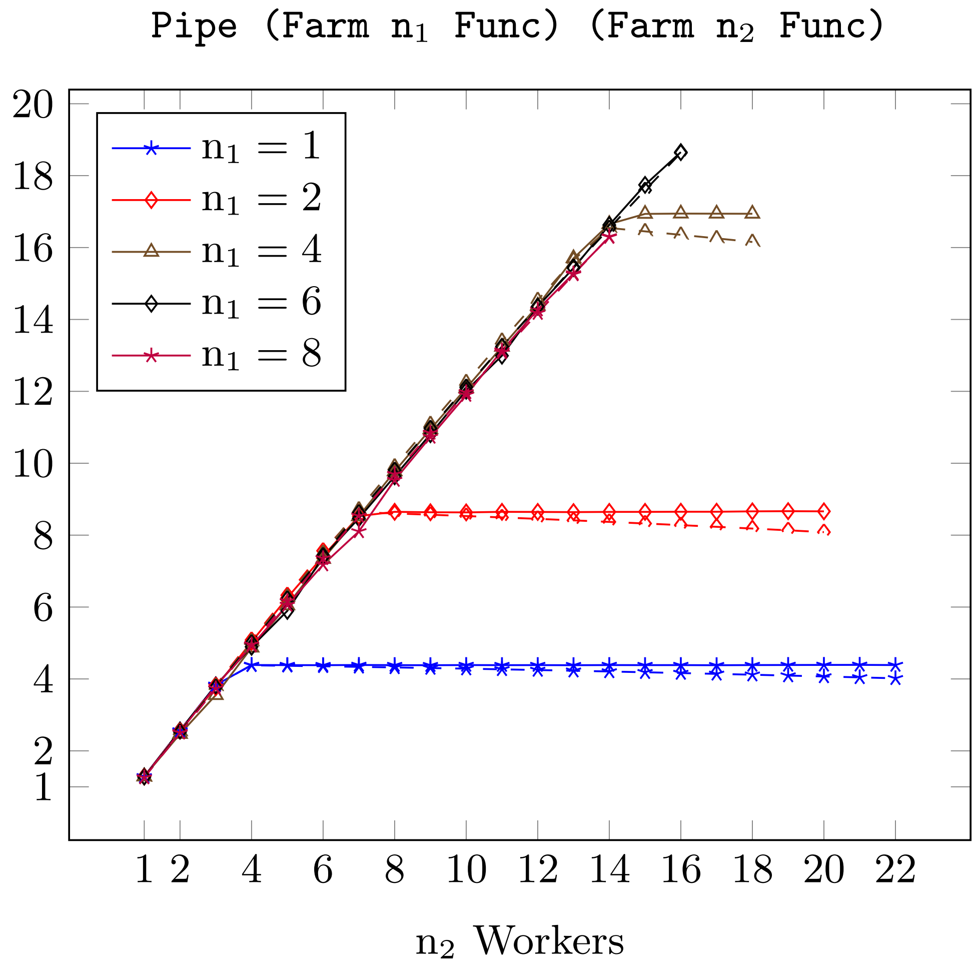


Image Convolution of 500 images on *titanic,* a 2.4GHz 24-core, AMD Opteron 6176 architecture, running Centos Linux 2.6.18-274.e15. Dashed lines are predictions.

Conclusion

Conclusions

University of St Andrews

- 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

University of St Andrews

- 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

Paper Available on Request

University of St Andrews

To appear in ICFP 2016

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

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

University of St Andrews

THANK YOU!

http://rephrase-ict.eu

http://paraphrase-ict.eu

@rephrase_eu