Automatically Deriving Cost Models for Structured Parallel Programs using Types and Hylomorphisms

David Castro, Kevin Hammond and Susmit Sarkar

University of St Andrews, UK

T: @rephrase_eu, @khstandrews
E: kevin@kevinhammond.net
W: http://www.rephrase-ict.eu
http://www.paraformance.com
Motivation

• Parallel patterns are great
  • **BUT we need to choose the best implementation**
  • *For a specific (heterogeneous) parallel architecture*

• We need a way to reason about parallel **structure**
  ✓ Correctness of transformations (done! ICFP2016)
  ☐ Reasoning about performance (in progress)
Example Skeleton: Parallel Task Farms

- Task Farms use a fixed number of workers (farm n f)
  - 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 \((f \ |\ |\ g)\)
  - over the elements of an input stream
  - \(f\) and \(g\) are run in parallel
Example: Parallel Image Merge

*Image merge* \( (\text{im}) \) composes *mark* and *merge*

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

There are many alternative parallel implementations
- **even just** using farms \( (\text{farm}) \) and pipelines \( (\|\|) \)

\[
\text{im}_1 = \text{farm n (fun (merge} \circ \text{mark))}
\]
\[
\text{im}_2 = \text{farm n (fun mark)} \| \text{farm m (fun merge)}
\]
\[
\text{im}_3 = \text{farm n (fun mark)} \| \text{fun merge}
\]

...
So, why types?

- According to the types community:
  - **Soundness**: “Well-typed programs cannot go wrong”
  - **Documentation**: “Type signatures provide valuable docs.”

- The benefits we are really interested in:
  - **Soundness**: 
    - “Well-typed programs can be parallelised as described by the types”
  - **Documentation**: 
    - “Type-level parallel structures clearly separate structure & functionality”
  - **Reusing well-understood techniques.**
    - E.g. algorithms for type unification and inference.
Selecting an Implementation

Decorate the function type with \( IM(n,m) \)

\[
\text{im} : IM(n,m) \sim \text{List (Img, Img)} \rightarrow \text{List Img} \\
\text{im} = \text{map (merge } \circ \text{ mark)}
\]

\( IM(n,m) = \text{FARM n (FUN A) } \parallel \text{FARM m (FUN A)} \)

The type system now automatically selects

\[
\text{im}_2 = \text{farm n (fun mark) } \parallel \text{farm m (fun merge)}
\]

We can \textit{guarantee} that this is functionally equivalent to \( \text{im} \)
Introducing/Transforming Parallel Patterns

Original program structure

Target parallel structure

Sequential normalised structure

Sequential normalised structure
How do we decide semantic equivalences?

- We can use the laws and properties of **hylomorphisms**!

- Hylomorphisms are a generalisation of a divide and conquer.

\[
\text{hylo}_F \ g \ h = f
\]

where \( f = g \circ F \ f \circ h \)

- “h” splits the input into a structure “F”, then recursive calls are mapped in structure “F”, the results are combined by “g”.

- Algorithmic skeletons can be described as instances of hylomorphisms
Hylomorphism Example

type \( T \text{ A} = \text{Empty} \mid (\text{A}, \text{List A}, \text{List A}) \)

\text{quicksort} : \text{List A} \rightarrow \text{List A}
\text{quicksort} = \text{hylo}_T \text{merge split}

\text{merge} : T A \rightarrow \text{List A}
\text{merge} = \ldots

\text{split} : \text{List A} \rightarrow T A
\text{split} = \ldots
Introducing Parallelism

We start with a streaming sequential version

\[
\text{quicksort} : \text{List} \ (\text{List} \ A) \rightarrow \text{List} \ (\text{List} \ A) \\
quicksort = \text{map}_{\text{List}} \ (\text{hylo}_T \ \text{merge} \ \text{split})
\]

To create a task farm and pipeline version, just change the type!

\[
\text{quicksort} : \text{PAR}_L \ (\text{FARM} \ n \ _ \ || \ _)\sim \\
\text{List} \ (\text{List} \ A) \rightarrow \text{List} \ (\text{List} \ A)
\]

To create a parallel divide-and-conquer version, change the type again!

\[
\text{quicksort} : \text{PAR}_L \ (\text{DC}_{n,T} \ A \ A) \sim \\
\text{List} \ (\text{List} \ A) \rightarrow \text{List} \ (\text{List} \ A)
\]
Base Semantics

- Defined using well-known recursion schemes:
  - map (replication, $\text{map}_F$)
  - fold (“catamorphism”, $\text{cata}_F$)
  - unfold (“anamorphism”, $\text{ana}_F$)
- plus sequential composition, $\circ$

\[
\begin{align*}
S[p : T A \to T B] &: [A \to B] \\
S[\text{fun}_T f] &= \text{env}(f) \\
S[p_1 \parallel p_2] &= S[p_2] \circ S[p_1] \\
S[\text{farm} n p] &= S[p] \\
S[\text{dc}_{n,T,F} f g] &= \text{cata}_F (\text{env}(f)) \circ \text{ana}_F (\text{env}(g))
\end{align*}
\]

\[
[p : T A \to T B] = \text{map}_T S[p]
\]
Deciding Semantic Equivalences

- Equivalence of parallel programs is reduced to equivalence of recursion schemes.
Recursion Schemes are Hylomorphisms!

\[ T \ A = \mu(F \ A) \]
\[ map_T f = \text{hylo}_{F \ A} (in_{F \ B} \circ (F \ f \ id)) \ out_{F \ A}, \]
where \( A = \text{dom}(f) \) and \( B = \text{codom}(f) \)
\[ \text{cata}_F f = \text{hylo}_F f \ out_F \]
\[ \text{ana}_F f = \text{hylo}_F \ in_F f \]
Inferring Parallel Structures

We can leave holes in the types

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

**Type unification** replaces \_ with any suitable parallel structure

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

Type unification replaces \_ with the parallel structure that *minimises* the cost model!
Example: Cost Model for Task Farms

\[ q_{farm}(n, \mathcal{P})(Q_0, Q_1) = \underbrace{\mathcal{P}(Q_0, Q_1) \parallel \ldots \parallel \mathcal{P}(Q_0, Q_1)}_{n \text{ times}} \]

This cost depends on the number of contending threads.

If \( \mathcal{P} \) takes time \( \mathcal{T} \), then the cost of each \( \mathcal{P}(Q_0, Q_1) \) is

\[ \mathcal{T} + \mathcal{T}_{\text{dequeue}}(Q_0) + \mathcal{T}_{\text{enqueue}}(Q_1). \]

If \( \mathcal{P} \) produces \( p \) number of outputs, then the task farm produces \( n \times p \) number of outputs, so the resulting cost needs to be divided by \( n \times p / p \), or \( n \):

\[ \frac{\mathcal{T} + \mathcal{T}_{\text{dequeue}}(Q_0) + \mathcal{T}_{\text{enqueue}}(Q_1)}{n}. \]
Predicting Parallel Execution Costs

Matrix multiplication, NxN matrices
24-core AMD Opteron
Predicting Parallel Execution Costs

\[
(FARM_{n_1}(\text{FUN} \sigma_1)) \parallel (FARM_{n_2}(\text{FUN} \sigma_2))
\]

Image Convolution
24-core AMD Opteron
Alternative Parallel Structures

Image Convolution
24-core AMD Opteron
Alternative Parallel Structures

Image Convolution
64-core Intel Xeon
Conclusions

- Deriving costs of parallel structures from an operational semantics is very powerful:
  - **Automatically** derive a **cost equation** from an “implementation”
  - Compile-time information about run-time behaviour based on a simple and easy to understand model.
  - When combined with our previous work (ICFP 2016), we can **automatically** rewrite programs to **minimize costs**

- Our cost model accurately predicts lower bounds on speedups

- We can choose between alternative parallel implementations
  - different patterns
  - CPU/GPU, manycore/multicore
Future Work

- Other patterns, e.g. stencil and bulk synchronous parallelism

- More general recursion patterns:
  - e.g. adjoint folds or conjugate hylomorphisms (Hinze)

- Apply to real languages (e.g. Haskell, Erlang)
  - Build a full implementation
RePhrase Project: Refactoring Parallel Heterogeneous Software – a Software Engineering Approach (ICT-644235), 2015-2018, €3.5M budget

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

Coordinated by @khstandrews
Bringing Functional Ideas to the Masses!

ParaFormance

Democratising Parallel Software

Zloties 2.5M
THANK YOU!

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

@rephrase_eu
Type System

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

Figure 5: Structure-Annnotated Type System for E.

\[ \vdash s : A \sigma \rightarrow B \]
\[ \vdash \text{fun } s : T A \text{FUN } \sigma \rightarrow T B \]
\[ \vdash s_1 : F B \sigma_1 \rightarrow B \]
\[ \vdash s_2 : A \sigma_2 \rightarrow FA G = \text{base } F \]
\[ \vdash \text{dc}_{n,F} s_1 s_2 : T A \text{DC}_{n,G} \sigma_1 \sigma_2 \rightarrow T B \]
\[ \vdash n : N \]
\[ \vdash p : T A \sigma \rightarrow T B \]
\[ \vdash \text{farm } n p : T A \text{FARM}_n \sigma \rightarrow T B \]
\[ \vdash p_1 : T A \sigma_1 \rightarrow T B \]
\[ \vdash p_2 : T B \sigma_2 \rightarrow T C \]
\[ \vdash p_1 \parallel p_2 : T A \sigma_1 \parallel \sigma_2 \rightarrow T C \]
\[ \vdash p : T A \sigma \rightarrow T (A + B) \]
\[ \vdash \text{fb } p : T A \text{FB } \sigma \rightarrow T B \]

- \( \sigma \sim A \rightarrow B \) is an alternative notation for \( A \sigma \rightarrow B \)

\[ \vdash e : A \sigma_1 \rightarrow B \]
\[ \sigma_1 \sim \sigma_2 \]
\[ \vdash e : A \sigma_2 \rightarrow B \]