Functions Compute, Relations Co-ordinate

Manuel M. T. Chakravarty

Technische Universität Berlin / GMD FIRST

1. Declarative parallel programming (GOFFIN)
2. Semantic framework (logic and operational semantics)
3. Multi-threaded execution model (to hide latencies)

DECLARATIVE PARALLEL PROGRAMMING

abstraction $\Rightarrow$ portability

clean semantics $\Rightarrow$
- automatic parallelization (exploits freedom in the evaluation order)
- optimizing transformations (alleviate inefficiencies due to abstraction)

automatic parallelization

$\Downarrow$

poor performance

$\Uparrow$

irregular access to data structures

(across PE boundaries)

explicit, but declarative parallelism

— multi-threading to tolerate latencies

What is declarative parallelism?

declarative	–	abstract semantics corresponds to mathematical logic
parallelism	–	operational semantics defines parallel behaviour

computation	step-at-a-time activities	functional specification
co-ordination	creates & composes activities and organises communication	relational description
	(Carriero & Gelernter 1992)	(Chakravarty, Köhler, Guo & Lock 1995)

Goffin (Chakravarty, Köhler, Guo & Lock 1997):

- ➜ computational sublanguage: Haskell
- ➜ co-ordinating sublanguage:
  - a new type statically identifies co-ordinative agents
  - parallel & sequential composition, synchronization & communication, committed-choice indeterminism

Properties of the language:

- ➜ type system preserves the semantics of Haskell
- ➜ supports the definition of higher-order co-ordination forms (skeletons)

Contributions of the reported work:

- ➜ abstract and operational semantics for GOFFIN-like languages
- ➜ implementation technology for GOFFIN-like languages

$$
\begin{aligned}
dc \quad :: \quad & (\alpha \to Bool) \\
\to \quad & (\alpha \to \beta) \\
\to \quad & (\alpha \to (\alpha, \alpha)) \\
\to \quad & (\beta \to \beta \to \beta) \\
\to \quad & (\alpha \to \beta \to O)
\end{aligned}
$$

$$
\begin{array}{l}
dc \;\; istriv\ solve\ split\ merge\ p\ s \\
\quad \begin{array}{lcl}
\mid istriv\ p & \Rightarrow & \{s \leftarrow solve\ p\} \\
\mid otherwise & \Rightarrow & \{p1, p2, s1, s2 \\
 & & \quad \textbf{in} \\
 & & \qquad (p1, p2) \leftarrow split\ p, \\
 & & \qquad rdc\ p1\ \ s1, \\
 & & \qquad rdc\ p2\ \ s2, \\
 & & \qquad s \leftarrow merge\ s1\ \ s2\} \\
 & & \quad \textbf{where} \\
 & & \qquad rdc = dc\ istriv\ solve\ split\ merge
\end{array}
\end{array}
$$

$$
\begin{array}{lcl}
msort & :: & Ord\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha] \rightarrow O \\
msort & = & dc\ ((<= 1)\ .\ length)\ id\ msplit\ mmerge
\end{array}
$$

$$
\begin{array}{llcl}
msplit & & :: & [\alpha] \rightarrow ([\alpha], [\alpha]) \\
msplit\ l & & = & splitAt\ (length\ l\ \text{`}div\text{`}\ 2)\ l \\
 & & & \\
mmerge & & :: & Ord\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha] \rightarrow [\alpha] \\
mmerge\ []\ \quad ys & & = & ys \\
mmerge\ xs\ \quad [] & & = & xs \\
mmerge\ (x : xs)\ (y : ys) & \mid x < y & = & x : mmerge\ xs\ (y : ys) \\
 & \mid otherwise & = & y : mmerge\ (x : xs)\ ys
\end{array}
$$

$$
\begin{array}{lcl}
msort & :: & Ord\ a \Rightarrow [\alpha] \rightarrow [\alpha] \rightarrow O \\
msort & = & dc\ ((<= threshold)\ .\ length)\ seqsort\ msplit\ mmerge
\end{array}
$$

Semantic Framework

$\mathcal{D}$—a kernel language for Goffin

typed lambda calculus with some syntactic sugar

$\lambda v.E$	abstraction	$E_1\ E_2$	application
$a\ \overline{E}$	constructor term	case E of $\overline{X}$	case selection
let $\overline{B}$ in E	local binding	$E{\downarrow}$	dereferencing

co-ordinating agents of type o

0	failure	1	success
$A_1 \parallel A_2$	parallel comp.	$A_1 \Rightarrow A_2$	sequential comp.
$\nu x.E$	unbound var.	$E_1 \doteq E_2$	eq. constraint

$\overline{v}_1 : \overline{G}_1 \to E_1 \mathbin{[\!]} \cdots \mathbin{[\!]} \overline{v}_n : \overline{G}_n \to E_n$ choice expression

Examples: $\nu x, y.(x \doteq f\ a) \parallel (y \doteq g\ b) \parallel p\ x\ y$ and $\lambda p_1, p_2.(p_1 \parallel p_2)$

Classical logic cannot properly model

- ➜ committed-choice indeterminism

$$G_1 \to E_1 \mathbin{[\!]} G_2 \to E_2 \qquad \text{where } G_1 \text{ and } G_2 \text{ overlap}$$

- ➜ rewriting (no concept of consumption)

An alternative: Girard's *linear logic*

- ➜ weakening and contraction not as structural rules
- ➜ additive & multiplicative version of the connectives
- ➜ explicit control over resources (modalities)

Abstract semantics for $\mathcal{D}$ on the basis of linear logic:

- ➜ each $\mathcal{D}$ expression E corresponds to a formula $\mathcal{E}[\![E]\!]$
- ➜ the value of E is v iff $\mathbf{D} \vdash \forall r.\mathcal{E}[\![E]\!]\, r \multimapinv (r\, v \parr \cdots)$
  ($\multimapinv$ denotes linear implementation)

Derivation of an operational semantics by an operational interpretation of linear logic proofs:

"reduction as deduction" (Guo 1994)

① Linear implication induces a transition relation $\rightrightarrows$:

$$E_1 \rightrightarrows E_2 \quad \text{iff} \quad \mathbf{D} \vdash \forall r.\mathcal{E}[\![E_1]\!]\,r \multimapinv \mathcal{E}[\![E_2]\!]\,r \quad \text{(for all } E_1, E_2 \in \mathcal{D})$$

② Derive transition *rules* from the transition relation $\rightrightarrows$

③ Define the parallel operational semantics by the transition rules

➜ soundness theorem & completeness conjectured

A MULTI-THREADED EXECUTION MODEL

Spineless Tagless G-machine: compiled graph-reduction (Peyton Jones 1992)

- ➜ distributed heap implies frequent remote access: latency

$$\boxed{(a+b)*(c+d)} \rightsquigarrow \boxed{\langle\text{other task}\rangle}$$

- ➜ task switches are expensive (cache, registers, ...)

Multi-threading exploits freedom in the evaluation order

- ➜ observe: $c+d$ and $a+b$ share context
- ➜ hide latency by evaluating nearby expressions
- ➜ an abstract notion of threads:

$$\texttt{letpar}\ \underbrace{v_1 = e_1}_{\text{thread \#1}}, \ldots, \underbrace{v_n = e_n}_{\text{thread \#}n}\ \texttt{in}\ e$$

Thread partitioning as source-to-source transformation:

$$
\begin{array}{l}
\texttt{letpar}\ av = a\ \texttt{[]} \\
\texttt{in letpar}\ bv = b\ \texttt{[]} \\
\texttt{in letpar}\ ab = av + bv \\
\texttt{in letpar}\ cv = c\ \texttt{[]} \\
\texttt{in letpar}\ dv = d\ \texttt{[]} \\
\texttt{in letpar}\ cd = cv + dv \\
\texttt{in}\ ab * cd
\end{array}
\quad \overset{\text{(optimisation)}}{\Longrightarrow} \quad
\begin{array}{l}
\texttt{letpar} \\
\quad ab = \boxed{\begin{array}{l}\texttt{letpar} \\ \quad av = \boxed{a\ \texttt{[]}} \\ \quad bv = \boxed{b\ \texttt{[]}} \\ \texttt{in}\ av + bv\end{array}} \\
\quad cd = \boxed{\begin{array}{l}\texttt{letpar} \\ \quad cv = \boxed{c\ \texttt{[]}} \\ \quad dv = \boxed{d\ \texttt{[]}} \\ \texttt{in}\ cv + dv\end{array}} \\
\texttt{in}\ ab * cd
\end{array}
$$

Extension of the Spineless Tagless G-machine:

- ➜ co-ordinating agents
- ➜ multi-threading based on the new `letpar` construct
- ➜ formalized in Haskell (executable specification)

Prototypical implementation of the new abstract machine:

- ➜ emulator
- ➜ determines the *depth* and *work* complexity

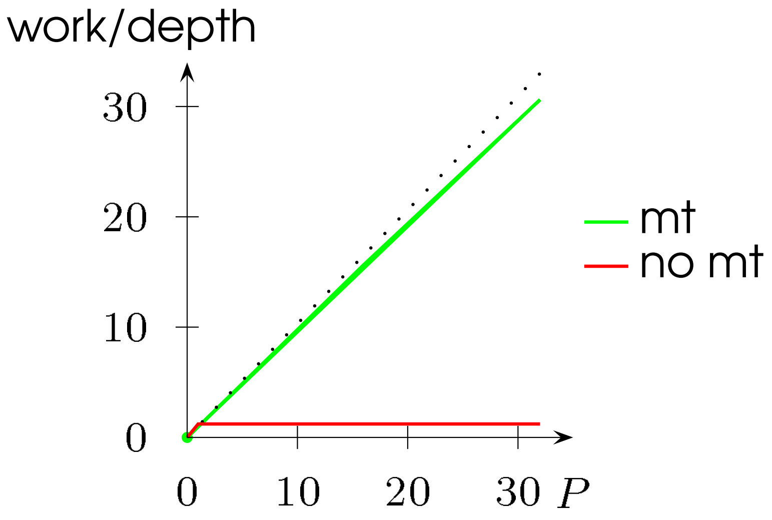
depth	–	critical path
work	–	sequential complexity
work/depth	–	average parallelism

An example ($farm$ is a parallel variant of map):

$$
\begin{aligned}
farm &:: (\alpha \to \beta) \to [\alpha] \to [\beta] \to O\\
farm\ f\ []\ r &= \{r \leftarrow []\}\\
farm\ f\ (x : xs)\ r &= \{ys\ \mathbf{in}\ r \leftarrow (f\ x) : ys, farm\ f\ xs\ ys\}
\end{aligned}
$$

$$
\begin{aligned}
sum &:: [Int] \to Int\\
sum\ [] &= 0\\
sum\ (x : xs) &= x + sum\ xs
\end{aligned}
$$

$$\{result, l'\ \mathbf{in}\ farm\ f\ l\ l', result \leftarrow sum\ l'\}$$



SUMMARY

$\mathcal{D}$ as a kernel language for GOFFIN:

- ➜ abstract semantics by an encoding into linear logic
- ➜ derivation of a parallel operational semantics
- ➜ soundness & completeness

Extension of the Spineless Tagless G-machine:

- ➜ multi-threading by an abstract notion of threads
- ➜ first experimental results support the expected behaviour

Future work:

- ➜ algebraic semantics
- ➜ alternative logics
- ➜ completion of a compiler
- ➜ extension towards distributed computing