Partial Compilers and Dialectica Spaces

Valeria de Paiva

Rearden Commerce

September, 2011

The Compiler Forest Story

This talk is based on: “The Compiler Forest", a preprint by Mihai Budiu, Joel Galenson and Gordon Plotkin, available from `http://homepages.inf.ed.ac.uk/gdp/publications`.

This starts from LINQ a database programming language (wikipedia: LINQ is a Microsoft .NET Framework component that adds native data querying capabilities to .NET languages, released in 2007).

the DryadLINQ compiler and its underlying distributed runtime system Dryad

DryadLINQ translates programs written in the LINQ database programming language (Language INtegrated Query) into large-scale distributed computations that run on shared-nothing computer clusters.

Basic Functionality of DryadLINQ

Compiling programs for execution on a computer cluster.
(1) translating a cluster-level computation into a set of interacting machine-level computations,
(2) translating each machine-level computation into a set of CPU core-level computations, and
(3) implementing each core-level computation in terms of machine code.

The optimization strategies, program transformation rules, and runtime operations invoked by the compiler at the cluster, machine, and core levels are very different.

Goal: Decompose the DryadLINQ compiler into a hierarchy of compilers that cooperate to implement a single computation.

Structure of Paper

Introduction

The theoretical foundations established have immediate applications in practice. Building compilers using forests of partial compilers produces modular system architectures that are easily tested and extended.

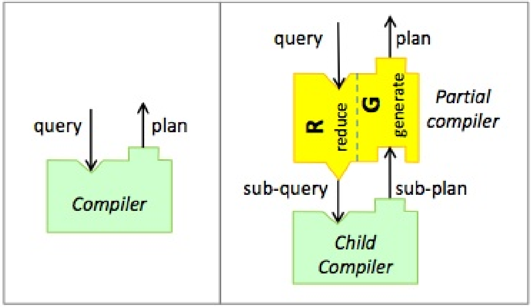
Revisit the original problem of compiling LINQ for computer clusters section 5. A reduced and stylized version of LINQ called μLINQ.

This language is rich enough to express many interesting computations, including the popular MapReduce framework.

Use formal machinery to argue about the correctness of the compilers. Many prior practical systems have been organized by using cooperating compilers. This formalism provides much-needed tools to allow reasoning about such systems rigorously.

Partial Compilers

Terminology: A compiler takes *queries* as inputs and produces *plans* as outputs. A *partial compiler* reduces a query to a sub-query to be handled by a *child compiler*. Given a plan for the sub-query, the partial compiler then generates a plan for the entire query.



Partial Compilers

The motivating example:
the DryadLINQ cluster-level partial compiler generates a plan to distribute the input data among many machines and then instructs each machine to perform a computation on its local data.

To generate a plan for each machine the cluster-level compiler creates a machine-level sub-query, which is handed to a machine-level compiler.
The machine-level compiler generates a machine-level plan.
The global, cluster-level plan contains code to:
(1) move data between machines and
(2) to invoke machine-level plans on the local data.

Partial Compilers in functions...

Compilers C transform queries into plans so they are typed as:
$C\colon query \to plan$.
To execute plans on specified input data, assume available an operation:
$Run\colon plan \times data \to data$
Model plans as computations: $plan = data \to data$
then Run is just function application $Run(p, d) = p(d)$
The formalism models partial compilers as two separate pieces, interdependent, that perform query reduction and plan generation.

Operationally, assume a simply typed lambda-calculus with products, function spaces and labelled sums, as well as lists and base types. Assume there are no side effects, types denote sets, and functions are usual set-theoretic ones. (If we discuss side-effects, then assume a call-by-value monadic semantics along the lines of Moggi's computational lambda calculus.)

Partial Compilers in functions...

Picture was one parent, one child compiler, but can generalize.
A parent partial compiler invoking the services of n child compilers is modeled using a polymorphic composition operation. Notation: $PC^n\langle\langle C_1, \ldots, C_n\rangle\rangle$

The partial compiler PC^n applied to a query Q returns a set of n queries Q_i, one for each C_i, and also a plan generation function G, which is applied to the plans generated by invoking each C_i on the corresponding Q_i.The composition of the partial compiler with its child compilers has type $query \rightarrow plan$:
$PC^n\langle\langle C_1, ..., C_n\rangle\rangle =_{def} \lambda Q$. let $((Q_1, ..., Q_n), G)$ be $PC^n(Q)$ in
let $P_1, ..., P_n$ be $C_1(Q_1), ..., C_n(Q_n)$ in $G(P_1, ..., P_n)$

Examples of Partial Compilers

1. Sequential partial compiler
2. Specialized partial compilers GPU+Core
3. Multiple query and plan types PC of type
$query \to (query' \times (plan' \to plan))$
4. Suppose we are dealing with computations on two different kinds of data: $data$ and $data'$.
So $plan = data \to data$ and $plan' = data' \to data'$.
We would like to compose a partial compiler
$PC\colon query \to (query \times (plan \to plan))$ with a compiler
$C\colon query' \to plan'$. If we have have suitable data conversion functions:
$c : data \to data'$ $c' : data' \to data$ and a suitable query translation
function: $Trans : query \to query'$
Then we can define the intermediate 'conversion' partial compiler
$PC_{convert} : query \to (query' \times (plan' \to plan))$

Compilers as first-class objects

Discuss a set of powerful polymorphic operators, used to manipulate partial compilers

“Structured Programming" manipulating compilers as values

A theory of correctness for composite compilers emerges.
Operations on compilers: tensor, star, conditionals, cases, functor, iteration.

Tensoring Compilers

Given $C_i\colon query_i \to plan_i$ (for $i = 1, 2$) define their tensor
$C_1 \otimes C_2\colon (query_1 \otimes query_2) \to (plan_1 \otimes plan_2)$ by:
$C_1 \otimes C_2(Q_1, Q_2) = (C_1(Q_1), C_2(Q_2))$.
Then, given
$PC^2 : query \to ((query_1 \otimes query_2) \otimes ((plan_1 \otimes plan_2) \to plan))$ say
$PC^2 << C_,C_2 >>$ is an abbreviation for $PC^2 << C_1 \otimes C_2 >>$.

(Other) Operations on Compilers

Given a compiler $C\colon query \to plan$ one can define $C^\star : query^\star \to plan^\star$, the star of C, by applying C pointwise to all elements in a list l of queries: $C(l) = map(C; l)$.
One compiler might be better suited to handle a given query than another, according to some criterion $pred\colon query \to bool$.
We can define a natural *conditional* operation to choose between two compilers
$COND : (query \to bool) \times (query \to plan)^2 \to (query \to plan)$ by:
$COND(p, (C1, C2)) = \lambda Q.$ if $p(Q)$ then $C_1(Q)$ else $C_2(Q)$

Can also define CASES, ITERATION and FUNCTORS.

Traditional compilers usually include a sequence of optimizing passes. An optimizing pass is given by an optimizing function $Opt\colon query \to query$ that transforms the query (program) from a higher-level representation to a lower-level one. We model on optimization pass as the partial compiler $PC = (Opt, id_{plan})$.

Correctness

“The main practical benefit we expect to obtain from the use of partial compilers in building systems is decreased design complexity."

One expects that a monolithic compiler applying a sequence of complex optimization passes to an internal representation should be less robust than a collection of partial compilers solving the same problem. Because the partial compilers communicate through well-defined interfaces and maintain independent representations of their sub-problems.

This makes sense if the correctness of compilers and partial compilers can be treated *modularly*.

Correctness

Given a plan P and a query Q there is generally a natural correctness relation, stating that the plan is correct for the query, written as: $P \models Q$.

Given a compiler $C\colon query \to plan$ and a correctness relation $\models$ on $plan$ and $query$, we say that the compiler C is correct w.r.t. to $\models$, if given a query Q, it returns a plan P such that $P \models Q$.

Given a partial compiler $PC\colon query \to (query' \times (plan' \to plan))$, and correctness relations $\models$ on $plan$ and $query$ and $\models'$ on $plan'$ and $query'$, we say PC is correct w.r.t. $\models$ and $\models'$ if, given a query Q, PC returns a query Q' and a function $G : plan' \to plan$ such that if G is given a plan P', where $P' \models Q'$, then G generates a plan P such that $P \models Q$.

Correctness

Correctness is preserved by all the operations described.
Correct compilers can be assembled from partial compilers or compilers that are only correct for some queries.
Importantly, this restricted correctness can also be treated modularly.
We define a natural *Hoare-type* logic to reason about correctness.

"Patching" bugs in an existing compiler, without access to its implementation. Assume we have a predicate $bug\colon query \rightarrow bool$ describing the queries for which a complex optimizing compiler C_{OPT} generates an incorrect plan. Assume we have a very simple (non-optimizing) compiler C_{SIMPLE} that always generates correct plans. Then the compiler IF bug THEN C_{SIMPLE} ELSE C_{OPT} 'corrects' the bugs of C_{OPT} .

μLINQ?...

LINQ (Language-INtegrated Queries, 2008) is a set of extensions to traditional .Net languages like C and Visual Basic. Essentially a functional, strongly-typed hybrid language, inspired by database language and comprehension calculi.

The main datatype in LINQ computations are lists of values, called (data) collections. The core LINQ operators are named after SQL: Select (a.k.a. map), Where (a.k.a. Þlter), SelectMany, OrderBy, Aggregate (a.k.a. fold), GroupBy (the output of which is a collection of collections), and Join.

LINQ queries are evaluated by LINQ *providers*. A provider is given by a term $P\colon query \times data \to data$ for some 'query' and 'data'. Given a compiler $C : query \to plan$, and a $Run : plan \times data \to data$, we obtain a provider $\lambda Q : query, d : data.Run(C(Q), data)$

μLINQ?...

μLINQ, a reduced version of LINQ with only three operators: SelectMany, Aggregate and GroupBy.
Types of are basic types and $T^{\star}$ the type of collections (finite lists) of elements of type T. A query of type $S^{\star} \to T^{\star}$ denotes a function from S collections to T collections. μLINQ queries consist of sequences of operator applications.

MapReduce can be expressed using μLINQ queries:
MapReduce(map, reduceKey, reduce) is the same as
$SelectMany < S, T > (map)$;
$GroupBy < T, K > (reduceKey)$;
$Select < T^{\star}, U > (reduce)$
$map : S \to T$, $reduceKey : T \to K$, and $reduce : T^{\star} \to U$
where we assume suitable function expressions are available.

Compiling μLINQ?...

Single core:
Basic building block for constructing μLINQ compilers. Start from three simple compilers, each of which can only generate a plan for a query consisting of just one of the operators:
$CSelectMany : FExp \rightarrow MPlan$
$CAggregate : FExp \rightarrow MPlan$ and
$CGroupBy : FExp \rightarrow MPlan$
Use the CASES operation to combine these three elementary compilers into a compiler that can handle simple one-operator queries. use the generalized sequential partial compiler and the star operation to construct a compiler C μLINQ : MQuery MPlan for arbitrary μLINQ queries.
If the above three simple compilers are correct, then, the combination is also correct.

Compiling μLINQ?...

multicore:
The assumption is that each core can execute a plan independently. The most obvious way to take advantage of the available parallelism is to decompose the work to perform by splitting the input data into disjoint parts, performing the work in parallel on each of the parts using a separate core,and then merging the results.
Compilation for distributed execution
The same strategy employed by the multi-core compiler for parallelizing μLINQ query evaluations across cores can be used to parallelize further, across multiple machines, in a manner similar to the DryadLINQ compiler.

Apparently it ALL works, but....

Where is my category theory?

Original Dialectica Categories (de Paiva, 1987) uses a category C, with finite limits, interpreting some type theory and builds the category $Dial(C)$ with objects $A = (U, X, \alpha)$, where U, X are objects of C and α is a sub-object of $U \times X$, that is a monic in $Sub(U \times X)$.
BUT we can do it all simply over Sets! Then Dialectica category $Dial$ has as objects triples, ie two sets and a relation between them. A map from $A = (U, X, \alpha)$ to $B = (V, Y, \beta)$ is a pair of maps (f, F), $f\colon U \to V$, $F\colon U \times Y \to X$ such that $\alpha(u, F(u, y)) \leq \beta(f(u), y)$
What if the *query reduction* and the *plan construction* bits of the partial compiler were our sets U and X? We were told that the partial compiler PC has type : $query \to (query' \times (plan' \to plan))$ But since $A \to (B \times C) \cong (A \to B) \times (A \to C)$, PC is equivalent to $(query \to query') \times (query \to (plan' \to plan))$ which is $\cong (query \times plan' \to plan)$. Hence a partial compiler PC is exactly a Dialectica map between $A = (query, plan)$ and $B = (query', plan')$.
Exactly?...

Where is my category theory?

I lied. Almost exactly. The relationship between the sets ‘query’ and ‘plan’ didn’t play much of a more in the discussion of partial compilers, while it is essential for the logic connection.
The predicate α is not symmetric: you can think of (U, X, α) as $\exists u.\forall x.\alpha(u, x)$, a proposition in the image of the Dialectica interpretation. and the predicate plays a huge part on Dialectica categories, and its generalizations
But the functionals f and F correspond to the Dialectica Interpretation of implication and that is the kernel of the interpretation.
Back to Logic...

Functional Interpretations and Gödel's Dialectica

Starting with Gödel's Dialectica interpretation(1958) a series of "translation functions" between theories

Avigad and Feferman on the Handbook of Proof Theory:

> *This approach usually follows Gödel's original example: first, one reduces a classical theory C to a variant I based on intuitionistic logic; then one reduces the theory I to a quantifier-free functional theory F.*

Examples of functional interpretations:

Kleene's realizability
Kreisel's modified realizability
Kreisel's No-CounterExample interpretation
Dialectica interpretation
Diller-Nahm interpretation, etc...

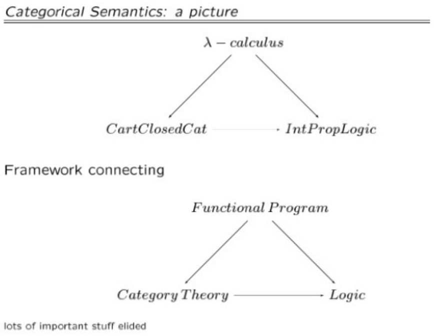
Gödel's Dialectica Interpretation

For each formula A of HA we associate a formula of the form $A^D = \exists u \forall x A_D(u,x)$ (where A_D is a quantifier-free formula of Gödel's system T) inductively as follows: when A_{at} is an atomic formula, then its interpretation is itself.
Assume we have already defined $A^D = \exists u \forall x. A_D(u,x)$ and $B^D = \exists v \forall y. B_D(v,y)$.
We then define:

$$(A \wedge B)^D = \exists u, v \forall x, y.(A_D \wedge B_D)$$
$$(A \rightarrow B)^D = \exists f \colon U \rightarrow V, F \colon U \times X \rightarrow Y, \forall u, y.$$
$$(\ \mathsf{A}_D(u, F(u,y)) \rightarrow B_D(fu; y))$$
$$(\forall z A)^D(z) = \exists f \colon Z \rightarrow U \forall z, x. A_D(z, f(z), x)$$
$$(\exists z A)^D(z) = \exists z, u \forall x. A_D(z,u,x)$$

The intuition here is that if u realizes $\exists u \forall x. A_D(u,x)$ then $f(u)$ realizes $\exists v \forall y. B_D(v,y)$ and at the same time, if y is a counterexample to $\exists v \forall y. B_D(v,y)$, then $F(u,y)$ is a counterexample to $\forall x. A_D(u,x)$.

Where is my category theory?



Categorical Semantics: a picture

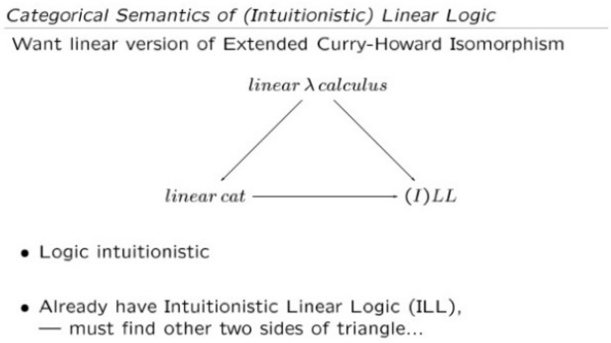
$$\begin{tikzcd}
 & \lambda - calculus \arrow[dl] \arrow[dr] & \\
CartClosedCat \arrow[rr] & & IntPropLogic
\end{tikzcd}$$

Framework connecting

$$\begin{tikzcd}
 & Functional\ Program \arrow[dl] \arrow[dr] & \\
Category\ Theory \arrow[rr] & & Logic
\end{tikzcd}$$

lots of important stuff elided

Where is my category theory?



Categorical Semantics of (Intuitionistic) Linear Logic

Want linear version of Extended Curry-Howard Isomorphism

$$\begin{tikzcd} & linear\,\lambda\,calculus \arrow[dl, dash] \arrow[dr, dash] & \\ linear\,cat \arrow[rr] & & (I)LL \end{tikzcd}$$

- Logic intuitionistic
- Already have Intuitionistic Linear Logic (ILL),
  — must find other two sides of triangle...

More Original Dialectica Categories

My thesis has four chapters, four main definitions and four main theorems. The first two chapters are about the “original" dialectica categories.

Theorem (V de Paiva, 1987)

If C is a ccc with stable, disjoint coproducts, then Dial(C) has products, tensor products, units and a linear function space $(1, \times, \otimes, I, \rightarrow)$ *and Dial(C) is symmetric monoidal closed.*

This means that $Dial(C)$ models **Intuitionistic Linear Logic** (ILL) without modalities. How to get modalities? Need to define a special comonad and lots of work to prove theorem 2...

Original Dialectica Categories

$!A$ must satisfy $!A \to !A \otimes !A$, $!A \otimes B \to !A$, $!A \to A$ and $!A \to !!A$, together with several equations relating them.
The point is to define a comonad such that its coalgebras are commutative comonoids and the coalgebra and the comonoid structure interact nicely.

Theorem (V de Paiva, 1987)

Given C a cartesian closed category with free monoids (satisfying certain conditions), we can define a comonad T on $Dial(C)$ such that its Kleisli category $Dial(C)^T$ is cartesian closed.

Define T by saying $A = (U, X, \alpha)$ goes to (U, X^*, α^*) where X^* is the free commutative monoid on X and α^* is the multiset version of α.
Loads of calculations prove that the linear logic modalitiy ! is well-defined and we obtain a full model of ILL and IL, a posteriori of CLL.
Construction generalized in many ways, cf. dePaiva, TAC, 2006.

What is the point of (these) Dialectica categories?

First, the construction ends up as a model of Linear Logic, instead of constructive logic. This allows us to see where the assumptions in Godel's argument are used (Dialectica still a bit mysterious...)

It justifies linear logic in terms of a more traditional logic tool and conversely explains the more traditional work in terms of a 'modern' (linear, resource conscious) decomposition of concerns.

Theorems(87/89): Dialectica categories provide models of linear logic as well as an internal representation of the dialectica interpretation. Modeling the exponential ! is hard, first model to do it. Still (one of) the best ones.

Dialectica categories: 20 years later...

It is pretty: produces models of Intuitionistic and clasical linear logic and special connectives that allow us to get back to usual logic.
Extended it in a number of directions (a robust proof can be pushed in many ways): used in CS as a model of Petri nets (more than 2 phds),
it has a non-commutative version for Lambek calculus (linguistics),
it has been used as a model of state (with Correa and Hausler)
and even of cardinalities of the continuum (Blass, Votjas(independently), Morgan&daSilva).
Also used for generic models (with Schalk04) of Linear Logic, for Dialectica interpretations of Linear Logic and Games (Shirahata and Oliveira) and fibrational versions (B. Biering and P. Hofstra).

Conclusions

Working in interdisciplinary areas is hard, but rewarding
The frontier between logic, computing, linguistics and categories is a fun place to be
Mathematics teaches you a way of thinking, more than specific theorems
Fall in love with your ideas and enjoy talking to many about them...

(ok these conclusions were to promote mathematics for young women at Sonoma State, so maybe you don't need them...)

Thanks Vlad for keeping BACAT alive, Mike for organizing today, Mihai, Joel and Gordon for a great paper and all present for a fun discussion!

References

Categorical Semantics of Linear Logic for All, Manuscript.
Dialectica and Chu constructions: Cousins?Theory and Applications of Categories, Vol. 17, 2006, No. 7, pp 127-152.
A Dialectica Model of State. (with Correa and Haeusler). In CATS'96, Melbourne, Australia, Jan 1996.
Full Intuitionistic Linear Logic (extended abstract). (with Martin Hyland). Annals of Pure and Applied Logic, 64(3), pp.273-291, 1993.
Thesis TR: The Dialectica Categories

http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-213.html

A Dialectica-like Model of Linear Logic.In Proceedings of CTCS, Manchester, UK, September 1989. LNCS 389
The Dialectica Categories. In Proc of Categories in Computer Science and Logic, Boulder, CO, 1987. Contemporary Mathematics, vol 92, American Mathematical Society, 1989

all available from `http://www.cs.bham.ac.uk/~vdp/publications/papers.html`