The Compiler Forest

omitted for submission

Abstract

Compilers targeting complex execution environments, such as computer clusters composed of machines with multi-core CPUs and GPUs, are difficult to write. To address this problem, we introduce *partial compilers*, which can pass subtasks to child compilers and combine the various plans they create, as a generalization of traditional compilers. We define a set of high-level polymorphic operations that manipulate partial compilers as first-class values. These mechanisms provide a software architecture for modular compiler construction, which allows the building of a forest of compilers. We explore the mathematical structure of partial compilers and its ties to theorem proving and category theory. Motivated by the problem of distributed computation, we demonstrate the software engineering advantages of our approach by building some complex compilers. We describe the complete implementation of a large-scale query-processing system written as a modular combination of many simple partial compilers.

1 Introduction

Today's computers are routinely composed of multiple computational units: multi-core processors, hyperthreaded processors, graphics processors, and multi-processors; we will use the term "execution engine" for these computational resources. Writing software that effectively exploits a complex hierarchy of execution engines is a daunting task.

The work presented in this paper was motivated by the DryadLINQ compiler [33] and its underlying distributed runtime system Dryad [16]. DryadLINQ translates programs written in the LINQ programming language (Language INtegrated Query) [20] into large-scale distributed computations that run on shared-nothing computer clusters, using multiple cores for each machine.

The basic functionality provided by DryadLINQ is to compile programs for execution on a computer cluster. A cluster can be seen as a big execution engine composed of many computers that do the actual work. At a finer grain each computer may have multiple CPU cores. Logically, the core DryadLINQ compilation process is correspondingly structured as a three-stage process: (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 (via .Net). In the case of DryadLINQ, the source language for all these layers is essentially the same, viz LINQ, but the optimization strategies, program transformation rules, and runtime operations invoked by the compiler at the cluster, machine, and core levels are very different. However, the DryadLINQ compiler has a monolithic implementation, including several other modules besides the three mentioned above. Maintaining and updating the DryadLINQ codebase is quite onerous; its monolithic nature makes it brittle to changes and hard to understand. Our goal is to refactor the compiler into a *hierarchy* of completely independent compilers that cooperate to implement a single translation.

Our main contribution is a novel software architecture constructed using a standard type-theoretical interface for building cooperating compilers. We present in Section 2 the notion of a *partial compiler*; this is a compiler that needs "help" from other compilers to produce a complete result. The resulting composite compilers form our *compiler forests*. Formally, one uses *polymorphic composition operations* on compilers and partial compilers.

Mathematically, partial compilers and their composition, as well as our other polymorphic operations, can be placed on a solid foundation using tools from category theory, particularly categorical logic, described in Section 6. The categorical formalism suggests a set of powerful polymorphic operators that can be used to manipulate partial compilers as first-class objects, particularly the composition and tensor operations given in Section 2, as well as others, given in Section 3. These operations can be seen as a form of "structured programming" manipulating compilers as values.

Based on this small set of well-defined operations, we obtain a theory of correctness of composite compilers. This is described in Section 4; it is a Hoare-type logic for reasoning about partially-correct partial compilers and their compositions.

The theoretical foundations we establish have immediate applications in practice. We have used our approach successfully for writing compilers that target complex execution engines (such as computer clusters). A compiler built as a tree of partial compilers has a modular architecture (to some degree induced by the structure of the execution engine) and is easily tested and extended. To demonstrate this, we revisit the original problem of compiling LINQ for computer clusters in Section 5. In order to expose the fundamental ideas without undue detail, we consider only a reduced and stylized version of LINQ called μLINQ. This language is rich enough to express many interesting computations, including the popular MapReduce [8] computation model. In the text we build a fully functional compiler for μLINQ that executes programs on a computer cluster with multi-core machines. We use our formal machinery to argue about the correctness of the μLINQ compiler.

We have validated this architecture by two preliminary compiler implementations: a (simplified) implementation of DryadLINQ, and a large-scale matrix computations compiler, both described briefly in Section 5.4 (a detailed description is reserved for a separate publication). We conjecture that partial compilers will prove to be a useful concept even for structuring traditional compilers, since the formalism of partial compilers does not depend on the existence of a complex composite execution engine.

[Copyright notice will appear here once 'preprint' option is removed.]

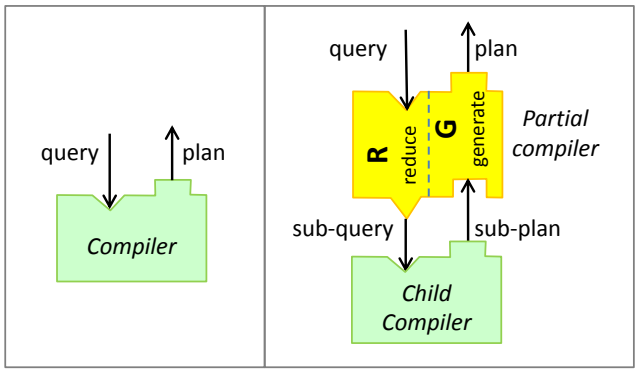


Figure 1. *Left*: A compiler translates queries to plans. *Right*: A partial compiler is unable to completely compile the input query; it reduces the query to a sub-query to be handled by a child compiler. Given a plan for the sub-query, the partial compiler then generates a plan for the entire query.

Our type-theoretic approach was originally inspired by Milner's tactics, which are used in theorem proving and proof assistants (Section 7.1). We were then influenced by the categorical ideas that inspired the precise typing of our partial compilers, as well, as noted above, as the operations we use over them. It is fascinating to note that, in their turn, the categorical ideas derive from Gödel's 1958 Dialectica interpretation of intuitionistic logic [1]. Since that interpretation can be viewed as a form of intuitionistic logical consequence that may not be ultimately surprising. Many prior practical systems have been organized by using cooperating compilers (Section 7.2). Our formalism provides us with much-needed tools to allow rigorous reasoning about such systems.

Our theory and practice evolved concurrently: the practice defined and constrained the problems to be solved while the theory enabled us to refactor our implementations to correspond to the emerging formalism more directly and to generalize the scope of our abstractions. This iterative process is not yet complete. This paper is only a first step: much theoretical and practical work remains to be done (Section 8).

2 Compilers and partial compilers

Some of our terminology is inherited from LINQ, which in turn inherited terminology from databases. We call the programs fed as inputs to a compiler *"queries"* and the outputs generated by the compiler *"plans"* (as shown on the left of Figure 1). We strongly emphasize, however, that our framework is not at all tied to query languages. All computations involved act on data. We assume that plans can be *executed* on specified input data.

Partial compilers are a generalization of compilers. The main intuition is shown on the right of Figure 1. On receipt of a query, a partial compiler needs the help of some other *child* compilers to handle it. For example, the DryadLINQ cluster-level partial compiler (top) generates a plan to distribute the input data among many machines and then instructs each machine to perform a computation on its local data. In order 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 (bottom). 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.

2.1 Definition

We now formalize all these intuitions with a typed lambda calculus. We do not spell the calculus out in detail, but we make use of product and function types, labeled sum types, and list types, as well as base types. Operationally, we generally assume there are no side effects, that types denote sets, and that functions are the usual set-theoretic ones. If we discuss side-effects then we assume a call-by-value monadic semantics along the lines of that of Moggi's computational lambda calculus [2, 23].

Compilers C transform queries into plans so they are typed as:

$$C : \mathrm{query} \longrightarrow \mathrm{plan}$$

as pictured on the left of Figure 1.

All computations involved act on data, which we take to be values of a given type "data". In order to execute plans on specified input data we assume available a suitable "run" operation:

$$\mathrm{run} : \mathrm{plan} \times \mathrm{data} \longrightarrow \mathrm{data}$$

In order to avoid formalizing details of the runtime used for executing plans, we often model plans as computations:

$$\mathrm{plan} = \mathrm{data} \to \mathrm{data}$$

and then run is just function application: $\mathrm{run}(p, d) = p(d)$. We do not specify the relationship between query and plan; in particular, the plan type of some compiler may be the same or a subset of the query type of some other compiler.

Our formalism models partial compilers as two separate pieces that perform, respectively, *query reduction* and *plan generation* (see Figure 1, right). We begin by considering *unary partial compilers*, which need just one child compiler. Given an input query, such a partial compiler returns a query: it "reduces" the original query to a "simpler" (sub-)query, to be fed to its child. This is done by a *query reduction function*

$$\mathbf{R} : \mathrm{query} \longrightarrow \mathrm{query}'$$

Note that the sub-query may be written in a different language than the original query. This extra generality is natural, since the sub-query may be targeted to a different engine than the query.

The partial compiler also contains a *plan generation* function

$$\mathbf{G} : \mathrm{query} \times \mathrm{plan}' \longrightarrow \mathrm{plan}$$

that, given the original query together with a (sub-)plan for the sub-query, generates a plan for the original query. Much as before, the sub-plan may be of a different type than the original plan. A partial compiler PC is therefore typed as:

$$PC : (\mathrm{query} \to \mathrm{query}') \times (\mathrm{query} \times \mathrm{plan}' \to \mathrm{plan})$$

To make formulas more readable we employ syntactic sugar for both types and programs. We write

$$(\mathrm{query}, \mathrm{plan}) \multimap (\mathrm{query}', \mathrm{plan}')$$

for the above partial compiler type, reading the type as "going from query to query′ and then back from plan′ to plan"; and we write

$$\begin{array}{l} \mathit{Compiler}\ \ Q : \mathrm{query}. \\ \ \mathit{Reduction}\ \ \mathbf{R}(Q), \\ \ \mathit{Generation}\ \ P' : \mathrm{plan}'. \mathbf{G}(Q, P') \end{array}$$

for the partial compiler defined by $(\mathbf{R}, \mathbf{G})$. Note that Q is bound in both the reduction and generation clauses.

Figure 1 (right) shows a simple compiler tree, consisting of a parent partial compiler invoking the services of a child compiler. In our formalism this is modeled using a polymorphic *composition* operation, which returns a compiler given a partial compiler and a compiler. Let $PC = (\mathbf{R}, \mathbf{G})$ be the (parent) partial compiler and

let $C : \text{query}' \to \text{plan}'$ be the (child) compiler. We use angle brackets to denote their composition:

$$PC\langle\langle C\rangle\rangle =_{\text{def}} \lambda Q : \text{query}.\, \mathbf{G}(Q, C(\mathbf{R}(Q))) : \text{query} \to \text{plan}$$

We also define the composition

$$PC\langle\langle PC'\rangle\rangle : (\text{query}, \text{plan}) \multimap (\text{query}'', \text{plan}'')$$

of the partial compiler $PC = (\mathbf{R}, \mathbf{G})$ with a partial compiler

$$PC' = (\mathbf{R}', \mathbf{G}') : (\text{query}', \text{plan}') \multimap (\text{query}'', \text{plan}'')$$

to be:

$$\begin{array}{l} \mathit{Compiler}\ \ Q : \text{query}. \\ \quad \mathit{Reduction}\ \ \mathbf{R}'(\mathbf{R}(Q)), \\ \quad \mathit{Generation}\ \ P'' : \text{plan}''.\, \mathbf{G}(Q, \mathbf{G}'(\mathbf{R}(Q), P'')) \end{array}$$

Partial compiler composition is *associative*:

$$PC\langle\langle PC'\langle\langle PC''\rangle\rangle\rangle\rangle = PC\langle\langle PC'\rangle\rangle\langle\langle PC''\rangle\rangle$$

and the two kinds of composition are compatible, as shown by the following *action* equation:

$$PC\langle\langle PC'\langle\langle C\rangle\rangle\rangle\rangle = PC\langle\langle PC'\rangle\rangle\langle\langle C\rangle\rangle$$

The partial compiler $\text{Id} =_{\text{def}} (\lambda Q.Q, \lambda Q, P.P)$ passes a given query to its child and then passes back the plan generated by its child unchanged. It is the identity element for composition, by which we mean that the following *identity* equations hold:

$$\text{Id}\langle\langle PC\rangle\rangle = PC = PC\langle\langle \text{Id}\rangle\rangle$$
$$\text{Id}\langle\langle C\rangle\rangle = C$$

Unary partial compilers can be generalized to n-ary ones, $PC^n = (\mathbf{R}^n, \mathbf{G}^n)$, where:

$$\mathbf{R}^n : \text{query} \to (\text{query}'_1 \times \ldots \times \text{query}'_n)$$
$$\mathbf{G}^n : \text{query} \times (\text{plan}'_1 \times \ldots \times \text{plan}'_n) \to \text{plan}$$

They can be reduced to unary partial compilers if we take query' to be $\text{query}'_1 \times \ldots \times \text{query}'_n$ and plan' to be $\text{plan}'_1 \times \ldots \times \text{plan}'_n$. Compilers can be thought of as 0-ary partial compilers. The ability to write n-ary partial compilers that can communicate with several children, which may be addressing different execution engines, is crucial to our approach.

To define composition on n-ary partial compilers we use a pairing operations—called *tensor*—on compilers, and partial compilers. Given two compilers $C_i : \text{query}_i \to \text{plan}_i$ (for $i = 1, 2$) their tensor

$$C_1 \otimes C_2 : (\text{query}_1 \times \text{query}_2) \to (\text{plan}_1 \times \text{plan}_2)$$

is defined by:

$$C_1 \otimes C_2(Q_1, Q_2) = (C_1(Q_1), C_2(Q_2))$$

Then, given an n-ary partial compiler PC^n as above and n compilers $C_i : \text{query}'_i \to \text{plan}'_i$ (for $i = 1, n$) the composition $PC^n\langle\langle C_1, \ldots, C_n\rangle\rangle$ is an abbreviation for the unary composition $PC^n\langle\langle C_1 \otimes \ldots \otimes C_n\rangle\rangle$. The n-fold tensor is the iterated binary one, associated to the left, and it is the trivial compiler for $n = 0$.

One proceeds analogously for the composition of an n-ary partial compiler with n unary partial compilers, via an analogous tensor of partial compilers. One then obtains suitable generalizations of the above associativity, action, and unit equations.

We could have defined the general n-ary compositions directly, but the pairing operations and unary composition relate well to the categorical context: see Section 6.

2.2 An example: the sequential partial compiler

Here we consider compiling functional programs written as sequences of function compositions. This is not entirely contrived:

Operation	Symbol	Compilers	Partial Compilers	Section
Composition	$\langle\langle\rangle\rangle$	Yes	Yes	2.1
Tensor	$\otimes$	Yes	Yes	2.1
Star	$*$	Yes	No	3.1
Conditional	COND	Yes	Yes	3.2
Cases	*CASES*	Yes	Yes	3.3
Functor	PC_{Func}	No	Yes	3.4

Table 1. Generic compiler operations described in this paper.

as we further discuss in Section 5, LINQ itself is a functional language and LINQ programs rely heavily on function composition.

We consider queries Q that are obtained from the composition of sub-queries $\text{prefix}(Q)$ and $\text{suffix}(Q)$, where:

$$\text{prefix}, \text{suffix} : \text{query} \longrightarrow \text{query}$$

The binary partial compiler

$$PC^2_{\text{SEQ}} : (\text{query}, \text{plan}) \multimap (\text{query} \times \text{query}, \text{plan} \times \text{plan})$$

generates sub-queries for the query's prefix and suffix. The plans for these two queries are composed, with the suffix plan invoking the prefix plan:

$$\begin{array}{l} \mathit{Compiler}\ \ Q : \text{query}. \\ \quad \mathit{Reduction}\ \ (\text{prefix}(Q), \text{suffix}(Q)), \\ \quad \mathit{Generation}\ \ P_{\text{prefix}}, P_{\text{suffix}} : \text{plan}. \\ \quad\ \ \lambda d : \text{data}.\, \text{run}(P_{\text{suffix}}, \text{run}(P_{\text{prefix}}, d)) \end{array}$$

3 Compilers as first-class objects

While composition and tensor are the main operations on compilers and partial compilers, we now discuss four more operations (shown in Table 1). We can also define an iteration operation that repeatedly invokes a child partial compiler, but we omit its definition due to space constraints, as it is not used in any of our examples.

3.1 Star

So far we have considered partial compilers whose arity is constant. We generalize, defining partial compilers that operate with lists of queries and plans. Given a compiler $C : \text{query} \to \text{plan}$ one can define $C^* : \text{query}^* \to \text{plan}^*$, the *star* of C, by applying C pointwise to all elements in a list l of queries:

$$C^*(l) = \text{map}(C, l)$$

Consider the partial compiler

$$PC_{\text{SEQ}} : (\text{query}, \text{plan}) \multimap (\text{query}^*, \text{plan}^*)$$

that generalizes the sequential compiler PC^2_{SEQ} from Section 2.2 by decomposing a query Q into a list $[Q_1, \ldots, Q_n]$ of its components. Given a compiler $C : \text{query}' \to \text{plan}$ for simple queries, the composition $PC_{\text{SEQ}}\,\langle\langle C^*\rangle\rangle$ is a compiler for queries that are an arbitrary composition of sub-queries. A practical example involving the star operation is given in Section 5.3.1.

3.2 Conditionals

The partial compiler operations we have constructed so far are all independent of the queries involved; by allowing dependence we obtain a richer class of compiler composition operations. For example, it may be that one compiler is better suited to handle a given query than another, according to some criterion:

$$\text{pred} : \text{query} \to \text{bool}$$

We can define a natural conditional operation to choose between two compilers

$$\mathrm{COND} : (\mathrm{query} \to \mathrm{bool}) \times (\mathrm{query} \to \mathrm{plan})^2 \to (\mathrm{query} \to \mathrm{plan})$$

by:

$$\mathrm{COND}(p, (C_1, C_2)) = \lambda Q.\ \mathit{if}\ p(Q)\ \mathit{then}\ C_1(Q)\ \mathit{else}\ C_2(Q)$$

We may use the mnemonic

$$\mathit{IF}\ \mathrm{pred}\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2$$

for $\mathrm{COND}(\mathrm{pred}, (C_1, C_2))$. There is an evident analogous conditional operation on partial compilers.

We can use the conditional to "patch" bugs in an existing compiler without having access to its implementation. Assume we have a predicate $\mathrm{bug} : \mathrm{query} \to \mathrm{bool}$ that describes (a superset of) the queries for which a specific complex optimizing compiler C_{OPT} generates an incorrect plan. Let us also assume that we have a very simple (non-optimizing) compiler C_{SIMPLE} that always generates correct plans. Then the compiler $\mathit{IF}\ \mathrm{bug}\ \mathit{THEN}\ C_{\mathrm{SIMPLE}}\ \mathit{ELSE}\ C_{\mathrm{OPT}}$ masks the bugs in C_{OPT}.

3.3 Cases

Similar to the * operation, but replacing list types by sum types, we can define a "cases" operation, a useful generalization of conditional composition. For convenience in our description we use labeled sum types (see, e.g., [27]). Given n individual compilers $C_{\mathrm{i}} : \mathrm{query}_i \to \mathrm{plan}$ (for $i = 1, n$) together with a function $W : \mathrm{query} \to l_1 : \mathrm{query}_1 + \ldots + l_n : \mathrm{query}_n$, one can define

$$\mathit{CASES}\ W\ \mathit{OF}\ l_1 : C_1, \ldots, l_n : C_{\mathrm{n}}$$

to be the compiler $C : \mathrm{query} \to \mathrm{plan}$ where:

$$C(Q) = \mathit{cases}\ W(Q)\ \mathit{of}\ l_1 : C_1(Q), \ldots, l_n : C_{\mathrm{n}}(Q)$$

We give a practical example using CASES in Section 5.3.1.

There is an evident analogous cases operation on partial compilers. Given

$$PC_{\mathrm{i}} = (\mathbf{R}_i, \mathbf{G}_{\mathrm{i}}) : (\mathrm{query}_i, \mathrm{plan}) \multimap (\mathrm{query}', \mathrm{plan}')$$

one defines

$$\mathit{CASES}\ W\ \mathit{OF}\ l_1 : PC_1, \ldots, l_n : PC_{\mathrm{n}}$$

to be the partial compiler

$$PC = (\mathbf{R}, \mathbf{G}) : (\mathrm{query}, \mathrm{plan}) \multimap (\mathrm{query}', \mathrm{plan}')$$

where:

$$\mathbf{R}(Q) = \mathit{cases}\ W(Q)\ \mathit{of}\ l_1 : \mathbf{R}_1(Q), \ldots, l_n : \mathbf{R}_n(Q)$$
$$\mathbf{G}(Q, P') = \mathit{cases}\ W(Q)\ \mathit{of}\ l_1 : \mathbf{G}_1(Q, P'), \ldots, l_n : \mathbf{G}_{\mathrm{n}}(Q, P')$$

3.4 Functor

Given functions $f : \mathrm{query} \to \mathrm{query}'$ and $g : \mathrm{plan}' \to \mathrm{plan}$, there is a partial compiler

$$PC_{\mathrm{Func}}(f, g) : (\mathrm{query}, \mathrm{plan}) \multimap (\mathrm{query}', \mathrm{plan}')$$

defined as:

$$\begin{array}{l} \mathit{Compiler}\ \ Q : \mathrm{query}. \\ \quad \mathit{Reduction}\ \ f(Q), \\ \quad \mathit{Generation}\ \ P : \mathrm{plan}.\, g(P) \end{array}$$

This operation is *functorial*, meaning that this equation holds:

$$PC_{\mathrm{Func}}(f, g)\langle\!\langle PC_{\mathrm{Func}}(f', g')\rangle\!\rangle = PC_{\mathrm{Func}}(f' \circ f, g \circ g')$$

We describe two useful applications of the functor in which g is the identity $\mathrm{id}_{\mathrm{plan}}$ on plan.

Traditional compilers usually include a sequence of optimizing passes. An optimizing pass is given by an optimizing transformation function $\mathrm{Opt} : \mathrm{query} \to \mathrm{query}$ that completely transforms the query (program) from a higher-level representation to a lower-level representation. We can model an optimization pass as the partial compiler $PC_{\mathrm{Func}}(\mathrm{Opt}, \mathrm{Id}_{\mathrm{plan}})$.

Staged compilers are frequently built from a sequence of transformations between (progressively lower-level) intermediate representations:

$$\mathrm{query}_1 \xrightarrow{\mathrm{Trans}_1} \ldots \xrightarrow{\mathrm{Trans}_{n-1}} \mathrm{query}_n$$

We can model this structure composing partial compilers $PC_{\mathrm{Func}}(\mathrm{Trans}_i, \mathrm{Id}_{\mathrm{plan}})$, obtaining the partial compiler

$$PC_{\mathrm{Func}}(\mathrm{Trans}_1, \mathrm{Id}_{\mathrm{plan}})\langle\!\langle \ldots \langle\!\langle PC_{\mathrm{Func}}(\mathrm{Trans}_{n-1}, \mathrm{Id}_{\mathrm{plan}})\rangle\!\rangle \ldots\rangle\!\rangle$$

of type $(\mathrm{query}_1, \mathrm{plan}) \multimap (\mathrm{query}_n, \mathrm{plan})$.

4 Correctness

The main practical benefit we expect to obtain from the use of partial compilers in building systems is decreased design complexity by reducing the number of possible interactions between the components involved. The partial compilers communicate through well-defined interfaces and maintain independent representations of their sub-problems.

We bolster this argument by showing that the correctness of compilers and partial compilers can be treated modularly. Correctness is preserved by all our operations (Section 4.1). However, in practice, correct compilers can be assembled from partial compilers or compilers that are only correct for some queries. Importantly, such restricted correctness can also be treated modularly, and we define a natural Hoare-type logic to reason about correctness in this case (Section 4.2). We regard the results presented here as a useful complement to the existing rich body of work on compiler correctness, since we prove our compilers correct under the assumptions that the component parts are correct.

4.1 Compositional correctness

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

$$P \models Q$$

For example, $P \models Q$ may hold if the action of the plan on the input data is in accord with the semantics of the query. We give examples of such correctness relations for the μLINQ language in Section 5.3.1. We define what it means for a compiler, or a partial compiler, to be correct, with respect to such correctness relations, and ensure that our various ways of combining compilers and partial compilers preserve correctness. We do all this informally, although it is straightforward to spell out the relevant mathematics in terms of the semantics of the lambda calculus without recursion or other effects. We do not consider correctness in the presence of effects. (This is not an unreasonable assumption since practical real compilers are essentially pure functions.)

Suppose we are given a compiler $C : \mathrm{query} \to \mathrm{plan}$ and a correctness relation $\models$ on plan and query. Then C is *correct w.r.t. to* $\models$ if, given a query Q, it returns a plan P such that $P \models Q$. Suppose next that we are given a partial compiler

$$PC = (\mathbf{R}, \mathbf{G}) : (\mathrm{query}, \mathrm{plan}) \multimap (\mathrm{query}', \mathrm{plan}')$$

and correctness relations $\models \subseteq \mathrm{plan} \times \mathrm{query}$ and $\models' \subseteq \mathrm{plan}' \times \mathrm{query}'$. Then PC is *correct w.r.t.* $\models$ *and* $\models'$ if, given a query Q and a plan P', $P' \models \mathbf{R}(Q)$ implies $\mathbf{G}(Q, P') \models Q'$.

Given relations $\models_i \subseteq \mathrm{plan}_i \times \mathrm{query}_i$, for $i = 1, 2$, define a relation $\models_1 \otimes \models_2$ between $\mathrm{plan}_1 \times \mathrm{plan}_2$ and $\mathrm{query}_1 \times \mathrm{query}_2$ by: $(P_1, P_2) \models_1 \otimes \models_2 (Q_1, Q_2)$ iff $P_i \models_i Q_i$, for $i = 1, 2$. Then, as an example, we would expect the sequential partial compiler PC^2_{SEQ} (Section 2.2) to be correct w.r.t. $\models$ and $\models \otimes \models$ for the

suitable $\models$, as given correct plans for the prefix and suffix of a query, their composition should be correct for the query. A formal proof can be written in terms of the correctness notion at hand and the semantics of queries.

Suppose $PC = (\mathbf{R}, \mathbf{G})$, as above, is a correct partial compiler, w.r.t. $\models$ and $\models'$, and $C : \text{query}' \to \text{plan}'$ is a correct compiler, w.r.t. $\models'$. Then their composition $PC\langle\langle C\rangle\rangle$ is correct w.r.t. $\models$. For given a query Q, $PC\langle\langle C\rangle\rangle$ generates the plan $\mathbf{G}(Q, C(\mathbf{R}(Q)))$. Since C is correct, w.r.t. $\models'$, we know that $C(\mathbf{R}(Q)) \models \mathbf{R}(Q)$. So, since PC is correct, w.r.t. $\models$ and $\models'$, we know that $\mathbf{G}(Q, C(\mathbf{R}(Q))) \models Q$. But this is what we need to show to prove that $PC\langle\langle C\rangle\rangle$ is correct, w.r.t. $\models$.

We can make similar arguments about the other operations on compilers and partial compilers defined in Sections 2 and 3 above. For example, the tensor operations preserve correctness. That is, if C_i is correct w.r.t. $\models_i$, for $i = 1, 2$, then $C_1 \otimes C_2$ is correct w.r.t. $\models_1 \otimes \models_2$; the corresponding assertion holds for partial compilers. Combining all this with composition we see that n-ary composition also preserves correctness.

Again, if $C : \text{query} \to \text{plan}$ is correct w.r.t. $\models$ then C^* is correct w.r.t. $\models^*$, where $[P_1, \ldots, P_m] \models^* [Q_1, \ldots, Q_n]$ iff $m = n$ and $P_i \models Q_i$, for $i = 1, m$. The compiler in Section 3.1 provides an example. Leaving correctness relations implicit, the generalized sequential partial compiler PC_{SEQ} is surely correct. So if C correct, so is the compiler $PC_{\text{SEQ}}\ \langle\langle C^*\rangle\rangle$.

Conditional composition also preserves correctness. In fact, an even stronger property is true: if C_1 is correct w.r.t. $\models$ on queries for which pred holds and if C_2 is correct w.r.t. $\models$ on queries for which pred does not hold, then IF pred $THEN\ C_1\ ELSE\ C_2$ is correct w.r.t. $\models$. An interesting use was given in Section 3.2.

4.2 A Hoare logic for compiler composition

We now sketch a form of "Hoare logic" that can be used to prove compilers correct, even if their components are only correct for some queries. We work informally, presenting suitable correctness notions and some of their properties. It would be straightforward to give a formal logic and a set-theoretic interpretation.

For every compiler $C : \text{query} \to \text{plan}$, correctness relation $\models$ on plan and query, and predicate $\varphi(Q)$ on queries $Q \in \text{query}$, we define a "Hoare pair" by:

$$\{\varphi(Q)\}_{\models} C \quad \equiv \quad \forall Q.\, \varphi(Q) \Rightarrow C(Q) \models Q$$

read as "the compiler C correctly translates the queries where predicate φ is true." Next, for every partial compiler

$$PC = (\mathbf{R}, \mathbf{G}) : (\text{query}, \text{plan}) \multimap (\text{query}', \text{plan}')$$

correctness relations $\models$ (on plan and query) and $\models'$ (on plan$'$ and query$'$), and predicates $\varphi(Q)$ and $\varphi'(Q')$, on queries $Q \in \text{query}$ and $Q' \in \text{query}'$, respectively, we define a "Hoare triple" by:

$$\{\varphi(Q)\}PC\{\varphi'(Q')\}_{\models,\models'} \equiv \forall Q.\, \varphi(Q) \Rightarrow \varphi'(\mathbf{R}(Q)) \wedge (\forall P'.\, P' \models' \mathbf{R}(Q) \Rightarrow \mathbf{G}(Q, P') \models Q)$$

As before, the predicate $\varphi(Q)$ specifies the queries on which PC operates correctly; the predicate $\varphi'(Q')$ gives a condition satisfied by the sub-query produced by PC. When the correctness relation indices are clear from the context, we may omit them from the Hoare assertions.

There are rules for unary composition of a partial compiler with a compiler or with another partial compiler:

$$\frac{\{\varphi(Q)\}PC\{\varphi'(Q')\} \quad \{\varphi'(Q')\}C}{\{\varphi(Q)\}PC\langle\langle C\rangle\rangle}$$

$$\frac{\{\varphi(Q)\}PC_1\{\varphi'(Q')\} \quad \{\varphi'(Q')\}PC_2\{\varphi''(Q'')\}}{\{\varphi(Q)\}PC_1\langle\langle PC_2\rangle\rangle\{\varphi''(Q'')\}}$$

There is a rule for the tensor of two compilers:

$$\frac{\{\varphi_1(Q_1)\}C_1 \quad \{\varphi_2(Q_2)\}C_2}{\{\varphi_1(\text{fst}(Q)) \ \wedge \ \varphi_2(\text{snd}(Q))\}C_1 \otimes C_2}$$

There is a rule for the star of a compiler:

$$\frac{\{\varphi(Q)\}_{\models} C}{\{\varphi^*(Q^*)\}_{\models^*} C^*}$$

where $\varphi^*([Q_1, \ldots, Q_n])$ holds iff $\varphi(Q_i)$ does, for $i = 1, n$.
There is a rule for the conditional composition of two compilers:

$$\frac{\{\text{pred}(Q) = 1 \ \wedge \ \varphi(Q)\}C_1 \quad \{\text{pred}(Q) = 0 \ \wedge \ \varphi(Q)\}C_2}{\{\varphi(Q)\}IF\ \text{pred}\ THEN\ C_1\ ELSE\ C_2}$$

There are similar rules, which we omit, for the tensor of two partial compilers, for the conditional composition of two partial compilers, and for the other operations discussed in Sections 2 and 3 above; there are also two evident predicate implication rules, one for partial compilers and one for compilers.

As an example, the conditional composition rule would apply to the bug-patching compiler of Section 4.1, taking pred $=$ bug, $C_1 = C_{\text{SIMPLE}}$, $C_2 = C_{\text{OPT}}$, and $\varphi(P) = \top$.

In the next section we construct a cluster compiler and use the Hoare logic rules to reason about its correctness.

5 Application to query processing

In this section we return to the original problem motivating our work: compiling LINQ. We introduce essential aspects of LINQ and give a much simplified version, called μLINQ, that is small enough to be tractable in a paper but rich enough to express interesting computations. We develop a hierarchy of partial compilers that, composed together, provide increasingly more powerful compilers for μLINQ.

5.1 LINQ

LINQ (Language-INtegrated Queries) was introduced in 2008 as a set of extensions to traditional .Net languages such as C# and F#. It is essentially a functional, strongly-typed language, inspired by the database language SQL (or the relational algebra) and comprehension calculi [4]. Like LISP, the main datatype manipulated by LINQ computations is that of lists of values; these are thought of as *(data) collections*.

LINQ operators transform collections into other collections. Queries are (syntactic) compositions of LINQ operators. For example, the query `C.Select(e => f(e))` uses the `Select` operator (often called *map*) to apply the function `f` to every element of a collection `C`. The result is a collection of the same size as the input collection. The notation `e => f(e)` is syntactically equivalent to the lambda expression λ`e.f(e)`. The elements `e` of `C` can have any .Net type, and the function `f` can be any .Net computation that returns a value, e.g., an expression, method, or a delegate.

The core LINQ operators are named after SQL. They are: `Select` (a.k.a. map), `Where` (a.k.a. filter), `SelectMany`, `OrderBy`, `Aggregate` (a.k.a. fold), `GroupBy` (the output of which is a collection of collections), and `Join`. All these operators are second-order, as their arguments are functions.

5.2 μLINQ

The LINQ language is quite complex, featuring more than 150 different operators if one counts the overloaded versions. We now define a reduced version of LINQ with only three operators, called *μLINQ*; this serves to model all the features of LINQ needed to illustrate the uses of partial compilers.

5.2.1 μLINQ syntax

The basic data *types* are denoted by the symbols S, T, U, and K; they are given by the grammar: $\mathtt{T} ::= \mathtt{B} \mid \mathtt{T}^*$ where B ranges over a given set of *primitive* types, such as Int, the type of integers. The type $\mathtt{T}^*$ stands for the type of collections (taken to be finite lists) of elements of type T. The corresponding .NET type is $\mathtt{IEnumerable}\langle\mathtt{T}\rangle$.

μLINQ queries consist of sequences of operator applications; they are not complete programs as the syntax does not specify the input data collections. They are specified by the grammar

$$\begin{array}{lcl} \mathtt{Query} & ::= & \mathtt{OpAp}_1;\ldots;\mathtt{OpAp}_n \qquad (n \geq 0) \\ \mathtt{OpAp} & ::= & \mathtt{SelectMany<S,T>(FExp)} \mid \\ & & \mathtt{Aggregate<T>(FExp,Exp)} \mid \\ & & \mathtt{GroupBy<T,K>(FExp)} \end{array}$$

Here Exp ranges over a given set of *expressions*, each of a given type T, and FExp ranges over a given set of *function expressions*, each of a given type $\mathtt{S}_1 \times \ldots \times \mathtt{S}_n \to \mathtt{T}$. We leave unspecified details of the given primitive types, expressions, and function expressions.

Only well-formed operator applications and queries are of interest. The following rules specify these and their associated types:

$$\frac{\mathtt{OpAp}_i : \mathtt{T}_i \to \mathtt{T}_{i+1} \qquad (i = 1, n)}{\mathtt{OpAp}_1;\ldots;\mathtt{OpAp}_n : \mathtt{T}_1 \to \mathtt{T}_{n+1}}$$

$\mathtt{SelectMany<S,T>(FExp)} : \mathtt{S}^* \to \mathtt{T}^*$ (if FExp has type $\mathtt{S} \to \mathtt{T}^*$)
$\mathtt{Aggregate<T>(FExp,Exp)} : \mathtt{T}^* \to \mathtt{T}^*$
(if FExp has type $\mathtt{T} \times \mathtt{T} \to \mathtt{T}$, and Exp has type T)
$\mathtt{GroupBy<T,K>(FExp)} : \mathtt{T}^* \to \mathtt{T}^{**}$ (if FExp has type $\mathtt{T} \to \mathtt{K}$)

5.2.2 μLINQ semantics

We now give an informal explanation of the semantics of μLINQ. We use $\cdot$ for list concatenation. A query of type $\mathtt{S}^* \to \mathtt{T}^*$ denotes a function from S collections to T collections. SelectMany<S,T>(FExp) applied to a collection returns the result of applying FExp to all its elements and concatenating the results. So, for example, SelectMany<nat,nat>(n => [n,n+1]) applied to $I =_{\text{def}} [1,2,3,4,5]$ results in $[1,2,2,3,3,4,4,5,5,6]$.

Aggregate<T>(FExp,Exp) denotes a singleton list containing the result of a fold operation [14] performed using FExp and Exp. So, for example, Aggregate<nat,nat>((m,n) => m+n,6) applied to I results in $[1+(2+(3+(4+(5+6))))] = [21]$. Some compilers require that FExp forms a monoid with Exp as a unit element (i.e., an associative function with a unit).

GroupBy<T,K>(FExp) groups all the elements of a collection into a collection of sub-collections, each sub-collection consisting of all the elements in the original collection sharing a common key, as found using FExp. The sub-collections occur in the order of the occurrences of their keys, via FExp, in the original collection, and the elements in the sub-collections occur in their order in the original collection. So, for example, GroupBy(n => n mod 2) applied to I results in [[1,3,5],[2,4]].

Finally, composite queries are constructed with semicolons and represent the composition, from left to right, of the functions denoted by their constituent operator applications.

The formal definition of μLINQ can be completed by giving a complete denotational semantics. We show the semantics for a language fragment; it is easy (but slightly tedious) to spell it out for a full language. We assign a set $[\![\mathtt{T}]\!]$ to every μLINQ type T, assuming every primitive type already has an assigned such set. Next, to any well-typed operator application $\mathtt{OpApp} : \mathtt{S} \to \mathtt{T}$ we assign a function $[\![\mathtt{OpApp}]\!] : [\![\mathtt{S}]\!] \to [\![\mathtt{T}]\!]$, given a denotation $[\![\mathtt{Exp}]\!] \in [\![\mathtt{T}]\!]$ for each expression $\mathtt{Exp} : \mathtt{T}$. For example:

$$[\![\mathtt{Aggregate<T>(FExp,Exp)}]\!](d) = [\mathrm{fold}([\![\mathtt{FExp}]\!], [\![\mathtt{Exp}]\!], d)]$$

Finally, to any well-typed query $\mathtt{Query} : \mathtt{S} \to \mathtt{T}$ we assign a function $[\![\mathtt{Query}]\!] : [\![\mathtt{S}]\!] \to [\![\mathtt{T}]\!]$ by:

$$[\![\mathtt{OpAp}_1;\ldots;\mathtt{OpAp}_n]\!] = [\![\mathtt{OpAp}_n]\!] \circ \ldots \circ [\![\mathtt{OpAp}_1]\!] \qquad (n \geq 0)$$

5.2.3 μLINQ, LINQ, and MapReduce

The LINQ operators Select and Where can be defined using SelectMany. For example, Select<S,T>(x => f(x)) is the same as SelectMany<S,T>(x => [f(x)]).

The popular MapReduce [8] programming model can be succinctly expressed in μLINQ:

```
MapReduce(map, key, red)
```

is the same as

```
SelectMany(map);GroupBy(key);Select(red)
```

5.3 Compiling μLINQ

5.3.1 A single-core compiler

We start by defining the types for queries and plans. Let us assume we are given a type FExp corresponding to the set of function expressions. Then we can define types OpAp and MQuery, corresponding to the sets of μLINQ operator applications and queries:

$$\begin{array}{lcl} \mathrm{OpAp} & = & \mathit{SelectMany} : \mathrm{FExp} + \\ & & \mathit{Aggregate} : \mathrm{FExp} + \\ & & \mathit{GroupBy} : \mathrm{FExp} \\ \mathrm{MQuery} & = & \mathrm{OpAp}^* \end{array}$$

For μLINQ plans we take

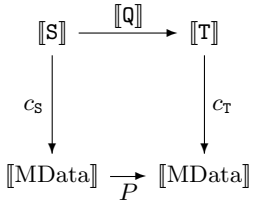
$$\mathrm{MPlan} = \mathrm{MData} \longrightarrow \mathrm{MData}$$

for a suitable given type "MData" of μLINQ data. We assume MData denotes lists of items, where items are either elements of (the semantics of) a basic μLINQ type B, or lists of such items.

There are natural functions:

$$c_{\mathtt{T}} : [\![\mathtt{T}]\!] \to [\![\mathrm{MData}]\!]$$

where we refer to the semantics of the lambda calculus on the right. Using this, one can define a natural correctness relation $P \models Q$ between plans (elements of $[\![\mathrm{MPlan}]\!]$) and queries (μLINQ queries, but identified with elements of $[\![\mathrm{MQuery}]\!]$). The correctness relation for μLINQ queries is given by the following commutative diagram, where Q is a μLINQ query of type $\mathtt{S} \to \mathtt{T}$:



and there is an analogous correctness relation for μLINQ operator applications.

One often computes on collections that are bags, i.e., (unordered) multisets (e.g., in the MapReduce model). In the underlying implementation lists may be used to represent bags. In this case a different "bag correctness" relation should be used that requires that $[\![\mathtt{Q}]\!] \circ c_{\mathtt{T}}$ and $c_{\mathtt{S}} \circ P$ give the same results up to permutation. Such correctness is only expected if all aggregations Aggregate(FExp,Exp) are *commutatively monoidal* (such an aggregation is *monoidal* if FExp is associative with unit Exp; commutativity further requires FExp to be commutative). Under that assumption $[\![\mathtt{Q}]\!]$ preserves permutation equivalence; as $c_{\mathtt{S}}$ and $c_{\mathtt{T}}$ evidently also preserve permutation equivalence, we see that bag correct Ps preserve permutation equivalence on the range of $c_{\mathtt{S}}$.

As a basic building block for constructing μLINQ compilers, we start from three very simple compilers, each of which can only generate a plan for a query consisting of just one of the operators:

$$C_{\text{SelectMany}} : \text{FExp} \longrightarrow \text{MPlan} \quad C_{\text{Aggregate}} : \text{FExp} \longrightarrow \text{MPlan}$$

$$C_{\text{GroupBy}} : \text{FExp} \longrightarrow \text{MPlan}$$

The denotational semantics of the μLINQ operators from Section 5.2.2 is a blueprint for a possible implementation of these compilers.

We use the *CASES* operation from Section 3.3 to combine these three elementary compilers into a compiler that can handle simple one-operator queries:

$$\begin{aligned} C_{\text{OO}} \quad = \quad & \mathit{CASES}\ (\lambda Q : \text{OpAp}.\, Q)\ \mathit{OF} \\ & \mathit{SelectMany} : C_{\text{SelectMany}}, \\ & \mathit{Aggregate} : C_{\text{Aggregate}}, \\ & \mathit{GroupBy} : C_{\text{GroupBy}} \end{aligned}$$

Finally, we use the generalized sequential partial compiler PC_{SEQ} and the star operation, both introduced in Section 3.1, to construct a compiler $C_{\mu\text{LINQ}} : \text{MQuery} \to \text{MPlan}$ for arbitrary μLINQ queries, by:

$$C_{\mu\text{LINQ}} = PC_{\text{SEQ}} \langle\!\langle C^*_{\text{OO}} \rangle\!\rangle$$

If the above three simple compilers are correct, then, by the discussion in Section 4.1, $C_{\mu\text{LINQ}}$ is also correct.

5.3.2 A multi-core compiler

In this example we construct a special partial compiler PC_{MC} to allow our single-core compiler to target a multi-core machine. We assume 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 part using a separate core, and then merging the results.

A partial compiler for operator applications for a multi-core machine with two identical cores c_1 and c_2

$$PC_{\text{MC}} : (\text{OpAp}, \text{MPlan}) \multimap (\text{OpAp}, \text{MPlan})$$

can be defined as:

$$\begin{aligned} & \mathit{Compiler}\ \ Q : \text{OpAp}. \\ & \ \mathit{Reduction}\ \ Q, \\ & \ \mathit{Generation}\ \ \mathit{Generation}\ \ P : \text{MPlan}. \\ & \ \lambda d : \text{MData}.\ \mathit{let}\ d'\ \mathit{be}\ \text{part}_1(Q, d)\ \mathit{in} \\ & \quad \text{collate}(Q, \text{run}_{c_1}(P, d'), \text{run}_{c_2}(P, d \backslash d')) \end{aligned}$$

The construct

$$\text{run}_e(P, d)$$

is to be thought of as a remote procedure call that runs P on computational engine $e : \text{Engine}$, computing on the data d. From the point of view of the lambda calculus we regard $\text{run}_e(P, d)$ as equivalent to $P(d)$, the type Engine as the unit type, and the semantics of remote procedure calls as an implementation matter that we do not formalize here.

For any list d, $\text{part}_1(d)$ and $d \backslash \text{part}_1(d)$ constitute a division of d into two parts, in a query-dependent manner (we assume that $d \backslash d'$ is chosen so that $d = d' \cdot (d \backslash d')$, if that is possible). The function collate assembles the results of the computations together, also in a query-dependent manner. There are many possible ways to define part_1 and collate; one reasonable specification is shown in Table 2 and described below. In the table $\text{prefix}(d)$ gives a prefix of d (so that $d = \text{prefix}(d) \cdot (d \backslash \text{prefix}(d))$); $\text{head}(d)$ is the first element of d, assuming d is non-empty, and $[\![\texttt{Exp}]\!]$ otherwise; and $\text{setr}(d)$ consists of d with all repetitions of an element on its right deleted.

The `SelectMany` operator is homomorphic with respect to collection concatenation, so it can be computed by partitioning the input collection d into an arbitrary prefix and suffix, applying the same operator recursively, and concatenating the results.

Similarly, if monoidal, `Aggregate(FExp,Exp)` is homomorphic w.r.t. the aggregation function `FExp`, so it can be applied to an arbitrary partition of d, combining the two results using `FExp`.

Finally, `GroupBy` partitions the input collection d so that values with the same key end up in the same partition. (It does so by splitting the codomain of the key function `FExp` into two arbitrary disjoint sets.) The results of recursively applying `GroupBy` on these partitions can be concatenated as the groups will also be disjoint.

The partial compiler PC_{MC} is correct when restricted to the predicate $\text{Mon}(Q)$ that asserts that if Q is an aggregation then it is monoidal. The correctness assertion for PC_{MC} is therefore:

$$\{\text{Mon}(Q)\} PC_{\text{MC}} \{\text{Mon}(Q)\}$$

The proof of this formula is based on the semantics of μLINQ's three operators.

The complete multi-core μLINQ compiler is given by

$$PC_{\text{SEQ}} \langle\!\langle PC_{\text{MC}} \langle\!\langle C_{\text{OO}} \rangle\!\rangle^* \rangle\!\rangle$$

This could be generalized to a machine with n cores by suitably modifying part_1 and collate.

Regarding correctness, since C_{OO} is correct, by the weakening, composition, and star rules we have: $\{\text{Mon}^*(Q^*)\} PC_{\text{MC}} \langle\!\langle C_{\text{OO}} \rangle\!\rangle^*$. We also have $\{\text{Mon}^*(Q^*)\} PC_{\text{SEQ}} \{\text{Mon}^*(Q^*)\}$ as PC_{SEQ} is correct and its query reduction function is the identity. It follows by the composition rule that the multi-core compiler is correct when restricted to $\text{Mon}^*(Q^*)$; note that this predicate holds of a query if and only if it contains only monoidal aggregations.

Note that we have achieved a non-trivial result: we have built a real μLINQ compiler targeting multi-cores by writing just a few lines of code, combining several simple compilers. This implementation is certainly not optimal as it repartitions the data around each operation, but we can transform it into a smarter compiler by using the same techniques (as we have done in our actual implementation). The functionality it provides is essentially that of PLINQ [9], the parallel LINQ implementation.

5.3.3 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. We add one additional twist by modeling resource allocation and scheduling in the plan language. Consider an example of a cluster of machines, and suppose we are dealing with a large collection, stored on a distributed filesystem, by splitting the collection into many partitions resident on different cluster machines (each machine may have multiple partitions). The goal of the generated plan is to process the partitioned collections in an efficient way, ideally having each piece of data be processed by the machine where it is stored. We assume in this example that the cluster is composed of just two machines, m_1 and m_2.

We define the unary partial operator application compiler PC_{CLUSTER} to be:

$$\begin{aligned} & \mathit{Compiler}\ \ Q : \text{OpAp}. \\ & \ \mathit{Reduction}\ \ Q, \\ & \ \mathit{Generation}\ \ P : \text{MPlan}. \\ & \ \lambda d : \text{MData}.\ \text{machines}\ m_1, m_2. \\ & \quad \mathit{let}\ d'\ \mathit{be}\ \text{MPart}_1(d, m_1, m_2)\ \mathit{in} \\ & \qquad \text{collate}(\text{run}_{m_1}(P, d'), \text{run}_{m_2}(P, d \backslash d')) \end{aligned}$$

"machines $m_1, m_2.\ P$" is an implementation-level *scheduling* construct to obtain two different machines and then run the plan P, which will involve the two machines. The construct

$$\text{MPart}_1(d, m_1, m_2)$$

Q	$\text{collate}(Q, l, r)$	$\text{part}_1(Q, d)$
`SelectMany(FExp)`	$l \cdot r$	$\text{prefix}(d)$
`Aggregate(FExp,Exp)`	$[[\![\texttt{FExp}]\!](\text{head}(l), \text{head}(r))]$	$\text{prefix}(d)$
`GroupBy(FExp)`	$l \cdot r$	$[x \in d \mid [\![\texttt{FExp}]\!](x) \in \text{prefix}(\text{setr}(\text{map}([\![\texttt{FExp}]\!], d)))]$

Table 2. Compiling a query Q for a dual-core computer.

returns the first part of a partition of the data d into two (dependent on m_1 and m_2). We assume that, in the implementation, each data item is tagged with the machine it resides on; the data-division construct could then, for example, put a data item into the first part of the partition if m_1 is closer to its machine than m_2. In terms of the underlying lambda calculus, we regard $\text{MPart}_1(d, m_1, m_2)$ as equivalent to $\text{MPart}_1(d)$, which nondeterministically returns the set of the first parts of all possible partitions of d into two[1].

The cluster-level operator application compiler can then be obtained by, for example, composing the cluster partial compiler with the multi-core compiler described previously

$$PC_{\text{CLUSTER}}\langle\!\langle PC_{\text{MC}}\langle\!\langle C_{\text{OO}}\rangle\!\rangle\rangle\!\rangle$$

and then the complete compiler is:

$$PC_{\text{SEQ}}\langle\!\langle PC_{\text{CLUSTER}}\langle\!\langle PC_{\text{MC}}\langle\!\langle C_{\text{OO}}\rangle\!\rangle\rangle\!\rangle^*\rangle\!\rangle$$

The cluster-level compiler is structurally similar to the multi-core compiler. However, in that case the collections themselves are already partitioned and the compiler uses the collection structure to allocate the computation's runtime resources. To establish correctness, we should consider the bag correctness relation, defined above, and also restrict the input language to queries such that all the binary functions occurring in `Aggregate` operations are associative and commutative, with unit the corresponding constant[2].

This compiler is in some respects more powerful than MapReduce, because (1) it can handle more complex queries, including chains of MapReduce computations and (2) it parallelizes the computation both across cores and across machines[3].

5.4 Implementation

We have implemented two useful real compiler forests: one for handling LINQ, and one for handling matrix computations. There is no obvious metric for comparing two implementations of a compiler, one monolithic and one modular (especially since they do not contain the same optimizations), so we do not provide any measurements. We have reused multiple partial compilers. The correctness proof presented above for μLINQ can be extended to reason informally about a larger subset of LINQ and its implementation.

5.4.1 Compiling LINQ

We have implemented a large number of compilers and partial compilers as building blocks for compiling the LINQ language, including partial compilers for multi-core execution (PC_{MC}), GPU execution (by wrapping the Microsoft Research Accelerator compiler [31]), sequential composition (PC_{SEQ}), cluster-level execution (PC_{CLUSTER}) (using Dryad as a runtime for the cluster), and complex compositions of all of these. Some of the compilers are complete and can handle essentially all of LINQ while others fail to compile some queries by generating a plan wrapping an error. For example, the GPU compiler cannot handle operations on .Net collections with complex datatypes. In our prototype the leaves of the compiler forest are frequently the LINQ-to-Objects implementation that is part of the default .Net distribution. Our implementation is only preliminary, but it performs well and has served to validate the architectural design. We are actively working to apply these ideas on a larger scale by building an industrial-strength compiler hierarchy that peforms more sophisticated optimizations at multiple levels (cluster/machine/core/GPU).

5.4.2 Compiling Matrix Algebra

We have also defined a functional language for computing on matrices. The language contains operators such as addition, multiplication, transposition, solving linear systems, Cholesky factorization, and LU decomposition. The matrices are modelled as two-dimensional collections of tiles, where the tiles are smaller matrices. This design is useful for dense matrices, but by making tiles very small it can also accommodate sparse matrices. Our compiler handles very large matrices, which can be partitioned across multiple machines. The top-level partial compiler translates matrix operations into operations on collections of tiles. The collection operations are translated by a second-level partial compiler into LINQ computations and passed to our LINQ compilers. The leaf compilers handle tile operations. The result is a compiler that can parallelize large matrix computations and generate linear algebra code for execution on a large cluster, also using multiple-cores. The matrix compiler takes advantage of the modularity of our LINQ compiler forest, by reusing multiple pieces as black-boxes, while also defining a set of new matrix-related partial compilers and compilers. The matrix compiler can be roughly described as:

$$PC_{\text{SEQ}}\langle\!\langle PC_{\text{Matrix}}\langle\!\langle C_{\text{Tile}}, C_{\text{LINQ}}\rangle\!\rangle^*\rangle\!\rangle$$

where C_{LINQ} is the distributed LINQ compiler from Section 5.4.1.

6 Categorical remarks

Remarkably, the idea of partial compilers originates in the categorical literature under the rubric of Dialectica categories [6, 15]. As the name suggests, these were invented to give a categorical account of Gödel's Dialectica interpretation of intuitionistic logic in type theory [1]. For simplicity, we explain the idea in terms of sets and functions, although it can be put in much more general terms. The Dialectica category has as objects pairs of sets (P, S), where, following Blass [3], we may think of P as a set of problems and S as a set of solutions. A morphism, $(f, g) : (P, S) \longrightarrow (P', S')$ consists of a *reduction* function $f : P \longrightarrow P'$ and a *solution* function $g : P \times S' \longrightarrow S$; we may picture it as in Figure 2. It is exactly a partial compiler $h : (P, S) \multimap (P', S')$, if we overlook the difference between the syntax for partial compilers, i.e., lambda expressions, and their semantics, i.e., functions.

The composition $(f, g) = (f_2, g_2) \circ (f_1, g_1)$ of two morphisms $(P_1, S_1) \xrightarrow{(f_1, g_1)} (P_2, S_2) \xrightarrow{(f_2, g_2)} (P_3, S_3)$ is given by:

$$f(p_1) = f_2(f_1(p_1)) \quad \text{and} \quad g(p_1)(s_3) = g_1(p_1, g_2(f_1(p_1), s_3))$$

[1] We are then no longer working with the straightforward set-theoretic semantics and must use the finite subsets monad (see Section 6).

[2] A complete formal treatment would also require us to adapt the definitions of correctness from Section 4 to handle non-determinism. The above rules for the Hoare logic remain valid.

[3] With just a little more work one can also add the only important missing optimization performed by MapReduce, which is to perform early aggregation in the map stage.

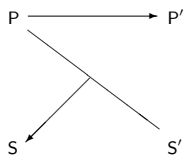


Figure 2. A morphism from (P, S) to (P', S').

This corresponds to the composition of the corresponding partial compilers. If h_i is (f_i, g_i) (for $i = 1, 2$) then $h_1 \langle\langle h_2 \rangle\rangle$ is $(f_2, g_2) \circ (f_1, g_1)$: note the reversal of order. The identity morphism on (P, S) is (id_S, π_2), which is just the identity partial compiler.

So we have a categorical explanation of unary partial compilers and their composition. Turning to binary partial compilers, the category has a tensor product. This is given on objects by:

$$(P_1, S_1) \otimes (P_2, S_2) = (P_1 \times P_2, S_1 \times S_2)$$
$$(f_1, g_1) \otimes (f_2, g_2) : (P_1 \times P_2, S_1 \times S_2) \longrightarrow (P_1' \times P_2', S_1' \times S_2')$$

the tensor product of two morphisms $(f_i, g_i) : (P_i, S_i) \to (P_i', S_i')$ $(i = 1, 2)$ is (f, g), where $f(p_1, p_2) = (f_1(p_1), f_2(p_2))$ and where $g(p_1, p_2)(s_1', s_2') = (g_1(p_1, s_1'), g_2(p_2, s_2'))$. This corresponds to the tensor of the corresponding partial compilers: if h_i is (f_i, g_i) then $h_1 \otimes h_2$ is $(f_1, g_1) \otimes (f_2, g_2)$.

All the operations can be seen as arising from categorical considerations; we briefly discuss some of them. The cases operation arises in a standard way from the fact that categorical sums exist when the sets of solutions are all the same [6], in particular $(P_1, S) + (P_2, S) = (P_1 + P_2, S)$ (however they do not exist in general). Finally the functorial operation corresponds to the evident functor from the the Girard category to the above Dialectica category. (The Girard category is a simpler version of the Dialectica category: see [7, Prop. 52] for the relationship between them.)

Dialectica categories also provide an account of correctness. One changes the category a little, using as objects triples $(P, S, \models)$ with a "solution" relation $\models \subseteq S \times P$. A morphism is then a pair $(f, g) : (P, S, \models) \longrightarrow (P', S', \models')$ that is a morphism from (P, S) to (P', S') as before, such that, additionally, for any $p \in P$ and $s' \in S'$, if $s' \models' f(p)$ then $g(p)(s') \models p$. (This is the converse of de Paiva [6], and instead follows Blass [3].)

The tensor product $(P_1, S_1, \models_1) \otimes (P_2, S_2, \models_2)$ of two objects is $(P_1 \times P_2, S_1 \times S_2, \models)$ where $(s_1, s_2) \models (p_1, p_2)$ if and only if $s_1 \models_1 p_1$ and $s_2 \models_2 p_2$; the composition and tensor product of morphisms are defined as before, verifying, as one goes along, that the extra conditions involving the relations are satisfied. In terms of partial compilers, correct partial compilers now correspond to morphisms in the Dialectica category with relations, and the fact that composition preserves correctness corresponds to the fact that the composition and tensor product of morphisms are defined in the same way, with and without relations.

The connection with categorical ideas is not only pleasant but also useful: as we have seen our operations on partial compilers were inspired by the corresponding categorical constructs. The literature on the Dialectica categories contains further functorial constructions that may also prove useful—for example, the sequential construction of Blass [3] is intriguing. Finally we note that the internal language of Dialectica categories may help formulate a useful language for the expression of partial compilers.

It is interesting to consider adding effects to the Dialectica categories. After Moggi's work [2, 23], one would naturally do so via a suitable monad, T, say. Martin Hyland suggested[4] using the underlying Kleisli category, taking morphisms $(f, g) : (P, S) \to (P', S')$ to be pairs $f : P \to T(P')$, $g : P \times S' \to T(S)$, with the evident composition and identities (we consider the variant with no correctness relation). One still has sums as above when the second component of the objects are all the same. One can also still define a tensor operation, but one obtains a premonoidal structure [29] rather than a functor; this corresponds to the fact that side-effects may cause the order in which child compilers are called to matter.

[4] Personal communication

7 Related work

7.1 Tactics and problem reductions

As we have said, partial compilers arose by analogy with Milner's tactics [12, 22]. Milner cared about sequents and theorems, whereas we care about queries and plans; our partial compilers can be n-ary, his tactics produce lists. So rather than partial compilers:

$$(\mathrm{query}, \mathrm{plan}) \multimap (\mathrm{query}', \mathrm{plan}')$$

equivalently

$$(\mathrm{query} \to \mathrm{query}') \times (\mathrm{query} \times \mathrm{plan}' \to \mathrm{plan})$$

he has tactics of the form:

$$\mathrm{sequent} \to (\mathrm{sequent}^* \times (\mathrm{theorem}^* \to \mathrm{theorem}))$$

Without side-effects, these correspond to partial compilers of type:

$$(\mathrm{sequent}, \mathrm{theorem}) \multimap (\mathrm{sequent}^*, \mathrm{theorem}^*)$$

Our methods of combining partial compilers correspond, more or less, to his tacticals. For example, we both use a composition operation, though his is adapted to lists, and the composition of two tactics may fail. His makes use of an OR tactical, which tries a tactic and if that fails (by raising a failure exception) tries an alternate; we replaced that by our conditional partial compiler.

We have avoided exceptions by making careful use of lambda calculus types. This enables us to use a side-effect free lambda calculus; it also strengthens the connection with category theory, discussed in Section 6. Correctness is also considered by Milner and Bird [22], where it is called "validity" instead.

7.2 Practical related work

Similar problems arise in federated and heterogeneous distributed databases. In the former, a query must be converted into queries against the component databases [30], and most of the work concentrates on optimizations in such a setting. Wrappers are used to translate queries between different component databases [18]. Our partial compilers serve a similar role but are more general as they can have multiple children while wrappers operate on a single database. A system such as Dremel [21] that uses trees to execute queries would benefit from our approach.

Others have considered compilers (or translators) at an abstract level, for example the work of Earley and Sturgis [10]. However, their emphasis is different: they vary three languages, source, target, and "meta" (the one in which the compiler is written), whereas we fix the metalanguage but bring in partiality.

The Haskell community has investigated separating an algorithm from its parallel behavior by introducing compositional strategies that control the parallel evaluation of functions [32]. However, these strategies are only used to specify parallelism and are composed mainly to operate on complicated data types or to control fine-grained parallelism rather than to combine different functionalities (e.g., combining compilers for CPUs and GPUs). To use an existing strategy on a newly-defined datatype, one must write a new strategy. Strategies can also be used to express parallel paradigms such as producer/consumer and pipeline; an implementation of our compiler forests could support these as well.

Lerner et. al [19] compose analyses by integrating them with their associated transformation to use graph transformations to

allow multiple analyses to communicate. Chang et al. [5] propose cooperating decompilers, where individual abstract interpretations communicate to share information. Our approach also supports these applications using the iteration operation mentioned above.

8 Discussion and conclusions

We made several simplifying assumptions in order to concentrate on the main points: partial compilers and their compositions. For example, in μLINQ we did not add a join operator, and the nature of expressions was left unspecified; in particular we did not consider (function) expressions containing nested queries. The join operator would have led to the use of tree-shaped queries rather than lists, and nested queries would have led to the use of DAGs: indeed DryadLINQ plans are DAGs. We note that a natural treatment of DAGs for functional programming seems to be missing from the literature. There is a natural version of the star operator of Section 3.1 for trees, and there should also be one for DAGs.

We also identified the runtime (plan) language (or languages) with the compiler language, thereby avoiding the need to employ two-level languages [11, 24, 26]. In reality, different computational engines can support different languages. It would be interesting to pursue formal treatments of the resource manager, which allocates engines, and the remote procedure calls, used by plans to invoke computations on different engines. Existing work on distributed computation in a functional setting (e.g., [17, 25]) may be helpful. A potential downside of modularity is that it may prevent partial compilers from sharing results of their analyses; this problem can be solved by using partial compilers whose query language is the same as the intermediate language of their parents.

We have generally assumed that the semantics of the compiler lambda calculus could be taken to be simply that of sets and functions. However, for realism, one should at least allow recursion, and so nontermination, and also, perhaps, other effects such as nondeterminism, given the discussion in Section 5.3.3. The plan languages may perhaps also profitably employ effects: probabilistic choice comes to mind as one natural possibility. The presence of effects raises the question of how to handle correctness in the presence of effects or how to adapt the Hoare logic (see, e.g., [13, 28]).

The DryadLINQ compiler extends the LINQ language with operations to partition data, similar to our partitioning operations. However, these extensions are invisible to the machine-level compilers. This suggests that partial compilers can be used to extend a legacy language implementation while treating it as a black box.

Nothing in our formalism is deeply dependent on plans being executable programs. Our software architecture may therefore be much more broadly applicable to other settings involving divide-and-conquer strategies.

We have had great success building useful practical compilers from partial compilers. Let us note that adopting a partial compiler architecture does not preclude one from building monolithic compilers; it merely offers an additional tool for *enforcing* modularity in the implementation. Our evidence so far is that the modularity has very useful outcomes, such as easing the reasoning about correctness and promoting module reuse.

References

[1] J. Avigad and S. Feferman. *Gödel's Functional ("Dialectica") Interpretation*, chapter 6, The Handbook of Proof Theory, pages 337–405. North Holland, 1998.

[2] N. Benton, J. Hughes, and E. Moggi. Monads and effects. In *APPSEM*, pages 42–122, 2000.

[3] A. Blass. Questions and answers – a category arising in linear logic, complexity theory, and set theory. In *Advances in Linear Logic, London Math. Soc. Lecture Notes 222*, pages 61–81, 1995.

[4] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[5] B.-Y. Evan Chang, M. Harren, and G. C. Necula. Analysis of low-level code using cooperating decompilers. In *SAS*, pages 318–335, 2006.

[6] V. de Paiva. The Dialectica categories. In *Proc. Cat. in Comp. Sci. and Logic, 1987. Cont. Math. , vol 92*, pages 47–62. AMS, 1989.

[7] V. de Paiva. *The Dialectica Categories*. PhD thesis, University of Cambridge, 1991.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.

[9] J. Duffy. *Concurrent Programming on Windows*. Addison Wesley, 2008.

[10] J. Earley and H. Sturgis. A formalism for translator interactions. *Commun. ACM*, 13:607–617, 1970.

[11] M. J. Gabbay and D. P. Mulligan. Two-level lambda-calculus. *Electr. Notes Theor. Comput. Sci.*, 246:107–129, 2009.

[12] M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF*. Springer-Verlag, 1979.

[13] J. Goubault-Larrecq, S. Lasota, and D. Nowak. Logical relations for monadic types. *MSCS*, 18(6):1169–1217, 2008.

[14] G. Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9(4):355–372, 1999.

[15] J. M. E. Hyland. Proof theory in the abstract. *Ann. Pure Appl. Logic*, 114(1-3):43–78, 2002.

[16] M. Isard et al. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[17] L. Jia and D. Walker. Modal proofs as distributed programs (extended abstract). In *ESOP*, pages 219–233, 2004.

[18] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32:422–469, 2000.

[19] S. Lerner et al. Composing dataflow analyses and transformations. In *Proc. 29th. POPL*, pages 270–282. ACM, 2002.

[20] E. Meijer et al. LINQ: reconciling object, relations and XML in the .NET framework. In *Proc. SIGMOD '06*, pages 706–706. ACM, 2006.

[21] S. Melnik et al. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3:330–339, 2010.

[22] R. Milner and R.S. Bird. The use of machines to assist in rigorous proof. *Phil. Trans. R. Soc. Lond. A*, 312(1522):411–422, 1984.

[23] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[24] E. Moggi and S. Fagorzi. A monadic multi-stage metalanguage. In *FoSSaCS*, pages 358–374, 2003.

[25] T. Murphy VII. *Modal types for mobile code*. PhD thesis, CMU, 2008.

[26] F. Nielson and H. Riis Nielson. *Two-level functional languages*. Cambridge University Press, 1992.

[27] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[28] G. D. Plotkin and M. Pretnar. A logic for algebraic effects. In *LICS*, pages 118–129. IEEE Computer Society, 2008.

[29] J. Power and E. Robinson. Premonoidal categories and notions of computation. *MSCS*, 7(5):453–468, 1997.

[30] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22:183–236, 1990.

[31] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *Proc. ASPLOS-XII*, pages 325–335. ACM, 2006.

[32] P. W. Trinder et al. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.

[33] Y. Yu et al. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.