The Compiler Forest

Mihai Budiu[1], Joel Galenson[1,2], and Gordon D. Plotkin[1,3]

[1] Microsoft Research, Silicon Valley
[2] University of California, Berkeley
[3] University of Edinburgh

Abstract. We address the problem of writing compilers targeting *complex execution environments*, such as computer clusters composed of machines with multi-core CPUs. To that end we introduce *partial compilers*. These compilers can pass sub-programs to several child (partial) compilers, combining the code generated by their children to generate the final target code. We define a set of high-level polymorphic operations manipulating both compilers and partial compilers as first-class values. These mechanisms provide a software architecture for modular compiler construction. This allows the building of a forest of compilers, providing a structured treatment of multistage compilers.

1 Introduction

Today's computers are routinely composed of multiple computational units: multi-core processors, hyperthreaded processors, graphics processors, and multi-processors; we use the term "execution engine" for these computational resources. The work presented in this paper was motivated by the DryadLINQ compiler [27]. DryadLINQ translates programs written in the LINQ programming language (Language INtegrated Query) [17] into distributed computations that run on shared-nothing computer clusters, using multiple cores on each machine. The core DryadLINQ compilation is 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.

Modifying a compiler stage requires deep understanding of both the compiler architecture and its implementation. We would prefer to be able to experiment easily, replacing some stages without knowing the implementation of others. Our goal is therefore to develop a general modular software architecture enabling compilers for distributed execution environments to be factored into a *hierarchy* of completely independent compilers, or "pieces" of compilers that cooperate via well-defined interfaces; the architecture should allow different pieces to be mixed and matched, with no access to source code or knowledge of internals.

To this end we propose a novel architecture employing a standard type-theoretical interface. In Section 2 we present *partial compilers*, a formalization of a "piece" of a compiler: partial compilers need "help" from one or more *child* compilers to produce a complete result. The resulting composite compilers form *compiler forests*. Formally, one uses *polymorphic composition operations* on compilers and partial compilers. The interface between component compilers is surprisingly simple and succinct. Traditional compiler stages can be recast as partial compilers.

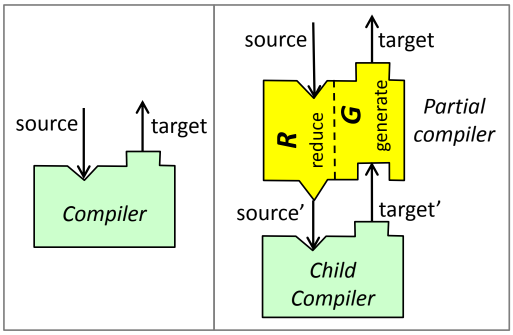


Fig. 1. L: A compiler translates sources to targets. **R**: A partial compiler invokes the service of a child compiler.

We present other natural polymorphic composition operations on compilers and partial compilers in Sections 2 and 3. Taken together, these operations can be seen as a form of "structured programming" manipulating compilers and partial compilers as first-class values. We thereby support dynamic compiler construction and extension, enabling sophisticated users to construct, customize, and extend compilers by mixing predefined and custom-built compiler components.

The theoretical foundations we establish have immediate practical applications. To demonstrate this, we revisit the original problem of compiling LINQ for computer clusters. In order to expose the fundamental ideas without undue detail, we use a stylized version of LINQ, called μLINQ. This language is rich enough to express many interesting computations, including the popular MapReduce [6] large-scale computation model. In Section 4 we build a fully functional compiler for μLINQ that executes programs on a computer cluster with multi-core machines.

Remarkably, partial compliers have their origins in work on categorical logic and on computer-assisted theorem proving, specifically de Paiva and Hyland's Dialectica categories [5,12] and Milner's tactics [10,19], the building blocks of his approach to computer-aided theorem proving. Section 5 treats the mathematical foundations of partial compilers in terms of a slight variant of the Dialectica category incorporating compile-time effects via a suitable monad. The morphisms of this category can be viewed as providing (the semantics of) a typed version of Milner's tactics. The polymorphic operations on partial compilers and compilers that we use to manipulate them as first-class objects were inspired by categorical considerations. For example, the composition and tensor operations of Section 2 correspond to compositions and tensors of morphisms.

We have also validated the partial compiler architecture with two proof-of-concept compiler implementations: a (simplified) reimplementation of DryadLINQ, and a compiler for large-scale matrix expressions. They are described briefly in Section 6. Finally, Sections 7 and 8 discuss related work and conclude.

2 Compilers and Partial Compilers

We call the program fed as input to a compiler a *"source"* (usually denoted by S), and the output generated by the compiler a *"target"* (usually denoted by T). The intuition behind partial compilers is shown on the right of Figure 1. There a partial compiler *reduces* the source program to a source' program, to be handled by a *child* compiler. Given a target' result obtained from the source' program by the child compiler, the partial compiler then *generates* the target for the original source program.

More generally, a partial compiler may use several child compilers. For example, given a source, a cluster-level partial compiler may generate a target to distribute input data among many machines, of various types, instructing each machine to perform a computation on its local data. In order to generate the target code running on each machine, the cluster-level compiler creates machine-level source programs source', which are handed to machine-level child compilers, one for each type of machine; these, in turn, generate the needed machine-level target's. The global, cluster-level target contains code to (1) move data between machines and (2) invoke machine-level target's of appropriate types on cluster machines and their local data.

2.1 Definitions

With these intuitions in mind, we can now give a theory of partial compilers. We take a call-by-value typed lambda calculus as our *compiler* language, and use it to define partial compilers and operations on them. We do not detail the calculus, but we make use of product and function types, labeled sum types (see [22]), list types, and base types. Our theory permits the lambda calculus to be effectful, i.e., we permit *compile-time effects*; it also permits recursion. However neither our examples nor our implementations make use of either of these two possibilities.

Formally, we take the calculus to be a suitable extension of Moggi's computational lambda calculus [20,21,1] to allow for compile-time effects. For its semantics we assume available a Cartesian closed category equipped with a strong *"compile-time"* monad $\mathrm{T}_{\mathrm{comp}}$ and suitable extra structure to accommodate the sum types, etc. As our examples and implementations use neither compile-time effects nor recursion, the reader can assume there that the category is that of sets and functions, so that types denote sets and terms denote elements of them.

Compilers transform sources into targets so they are terms C typed as:

$$C : \mathrm{source} \longrightarrow \mathrm{target}$$

as pictured on the left of Figure 1. We do not specify the relationship between source and target; in particular, the target type of some compiler may be the source type of some other compiler.

Rather than making specific choices of target languages, we use a lambda calculus to define the semantics $[\![T]\!]$ of targets T output by compilers. We assume that the target computations output by compilers act on a type "data" so that this semantics has the form:

$$[\![T]\!] : \mathrm{data} \rightarrow \mathrm{data}$$

As in the case of the compiler language, we do not detail such a *run-time* lambda calculus, but, in particular, it may have *run-time effects*. In general, target languages may differ in both the data their targets handle and the run-time effects they create; however, for simplicity, we keep both fixed in the examples.

Formally, we (again) use the computational lambda calculus, but for the semantics we now use *"run-time"* monads $\mathrm{T}_{\mathrm{run}}$ to account for run-time effects. As it suffices for the examples at hand, we work in the category of sets, but nothing depends on that.

We define (unary) partial compilers to be terms of type:

$$PC : \mathrm{source} \rightarrow (\mathrm{source}' \times (\mathrm{target}' \rightarrow \mathrm{target}))$$

As discussed above, the idea is that, given a source program, a partial compiler "reduces" it to a source' program, to be handled by a child compiler, and also produces a "generation" function that, given a target' obtained from the child, returns the required target. With this type, compile-time effects can occur at two points: when reducing the original source, and when computing the target.

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

$$(\mathrm{source}, \mathrm{target}) \multimap (\mathrm{source}', \mathrm{target}')$$

for the above partial compiler type, reading the type as "going from source to source' and then back from target' to target"; and we write

$$\begin{array}{l} \mathit{Compiler}\ \ S : \mathrm{source}. \\ \mathit{Reduction}\ \ R, \\ \mathit{Generation}\ \ T' : \mathrm{target}'. G \end{array}$$

for the partial compiler

$$\lambda S : \mathrm{source}.\,\mathrm{let}\, S' : \mathrm{source}'\ \mathrm{be}\, R\, \mathrm{in}\, (S', \lambda T' : \mathrm{target}'.\, G)$$

Note that S 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. We model this by a polymorphic *composition* operation, which returns a compiler, given a (parent) partial compiler and a (child) compiler. Let PC be a partial compiler, as above, and $C : \mathrm{source}' \to \mathrm{target}'$ be a compiler. We write their composition using angle brackets:

$$PC\langle\langle C\rangle\rangle : \mathrm{source} \to \mathrm{target}$$

and define it to be:

$$\lambda S : \mathrm{source}.\,\mathrm{let}\, (S', G)\ \mathrm{be}\, PC(S)\, \mathrm{in}\, G(C(S'))$$

If there are no compile-time effects, we can view the operation of the compiler $PC\langle\langle C\rangle\rangle$ on a source S' as going through a sequence of compiler stages or passes:

$$S \xrightarrow{\mathrm{fst} \circ PC} S' \xrightarrow{C} T' \xrightarrow{\mathrm{snd}(PC(S))} T$$

where the last pass $\mathrm{snd}(PC(S))$ is a function of the initial source. In contrast, the operation of the partial compiler PC is a "partial" sequence of passes:

$$S \xrightarrow{\mathrm{fst} \circ PC} S' \xrightarrow{?} T' \xrightarrow{\mathrm{snd}(PC(S))} T$$

The core function of our methodology is to generate useful patterns of such passes in a structured way, including combining partial passes. We define the composition

$$PC\langle\langle PC'\rangle\rangle : (\mathrm{source}, \mathrm{target}) \multimap (\mathrm{source}'', \mathrm{target}'')$$

of a partial compiler PC with a partial compiler

$$PC' : (\mathrm{source}', \mathrm{target}') \multimap (\mathrm{source}'', \mathrm{target}'')$$

to be:

$$\lambda S : \mathrm{source}.\ \mathrm{let}\, (S', G)\ \mathrm{be}\, PC(S)\, \mathrm{in}\, \mathrm{let}\, (S'', G')\ \mathrm{be}\, PC'(S')\, \mathrm{in}\, (S'', G \circ G')$$

In terms of a partial sequence of passes this is:

$$S \xrightarrow{\mathrm{fst} \circ PC} S' \xrightarrow{\mathrm{fst} \circ PC'} S'' \xrightarrow{?} T'' \xrightarrow{\mathrm{snd}(PC'(S'))} T' \xrightarrow{\mathrm{snd}(PC(S))} T$$

Certain *equations* hold in the computational lambda calculus, for all compiler-time effects. 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 compositions are compatible, as shown by the *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 Id $=_{\mathrm{def}} (\lambda S.S, \lambda(S,T).T)$ passes a given source to its child and then passes back the target generated by its child unchanged. It is the identity element for composition, i.e., the following *identity* equations hold:

$$\mathrm{Id}\langle\!\langle PC\rangle\!\rangle = PC = PC\langle\!\langle \mathrm{Id}\rangle\!\rangle \qquad \mathrm{Id}\langle\!\langle C\rangle\!\rangle = C$$

Unary partial compilers can be generalized to n-ary terms PC^n of type

$$\mathrm{source} \longrightarrow ((\mathrm{source}'_1 \times \ldots \times \mathrm{source}'_n) \times (\mathrm{target}'_1 \times \ldots \times \mathrm{target}'_n) \to \mathrm{target})$$

One can reduce such n-ary partial compilers to unary partial compilers by taking source' to be $\mathrm{source}'_1 \times \ldots \times \mathrm{source}'_n$ and target' to be $\mathrm{target}'_1 \times \ldots \times \mathrm{target}'_n$. 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 iterate two pairing operations, which are both called *tensor*, on compilers and partial compilers. For the first, given compilers $C_{\mathrm{i}} : \mathrm{source}_i \to \mathrm{target}_i$ (for $i = 1, 2$), we define their tensor

$$C_1 \otimes C_2 : (\mathrm{source}_1 \times \mathrm{source}_2) \to (\mathrm{target}_1 \times \mathrm{target}_2)$$

to be:

$$C_1 \otimes C_2 = \lambda(S_1, S_2).\, (C_1(S_1), C_2(S_2))$$

Given an n-ary partial compiler PC^n and n compilers $C_{\mathrm{i}} : \mathrm{source}'_i \to \mathrm{target}'_i$ (for $i = 1, \ldots, n$) the n-ary composition $PC^n\langle\!\langle C_1, \ldots, C_{\mathrm{n}}\rangle\!\rangle$ is an abbreviation for the unary composition $PC^n\langle\!\langle C_1 \otimes \ldots \otimes C_{\mathrm{n}}\rangle\!\rangle$. The n-fold tensor is the iterated binary one, associated to the left; it is the trivial compiler for $n = 0$.

Next, we define the binary tensor

$$\begin{aligned} PC_1 \otimes PC_2 : (\mathrm{source}_1 \times \mathrm{source}_2, \mathrm{target}_1 \times \mathrm{target}_2) \multimap \\ (\mathrm{source}'_1 \times \mathrm{source}'_2, \mathrm{target}'_1 \times \mathrm{target}'_2) \end{aligned}$$

of two partial compilers

$$PC_i : (\mathrm{source}_i, \mathrm{target}_i) \multimap (\mathrm{source}'_i, \mathrm{target}'_i)$$

to be:

$$\begin{aligned} \lambda S_1, S_2.\ & \mathrm{let}\,(S'_1, G_1)\ \mathrm{be}\, PC_1(S_1)\ \mathrm{in} \\ & \mathrm{let}\,(S'_2, G_2)\ \mathrm{be}\, PC_2(S_2)\ \mathrm{in} \\ & ((S'_1, S'_2), \lambda T_1, T_2.\, \mathrm{let}\, T'_2, T'_1\ \mathrm{be}\, G_2(T_2), G_1(T_1)\ \mathrm{in}\, (T'_1, T'_2)) \end{aligned}$$

The reason for the "twist" in the order of the G's is explained in Section 5. Intuitively, G_2's effects are "well-bracketed" by G_1's.

Using this tensor, one defines the composition of an n-ary partial compiler with n partial compilers via iterated tensors, analogously to the case of compilers. One then obtains suitable n-ary generalizations of the above unary associativity, action, and unit equations for the two n-ary compositions. These hold when there are no compile-time effects; Section 5 discusses the general case.

2.2 An Example: The Sequential Partial Compiler

We give an example of a binary partial compiler and its composition with two compilers; Section 3.4 makes use of the composition of partial compilers. We consider compiling source programs S obtained from the composition of two simpler sources $\mathrm{prefix}(S)$ and $\mathrm{suffix}(S)$, where:

$$\mathrm{prefix}, \mathrm{suffix} : \mathrm{source} \longrightarrow \mathrm{source}$$

The binary partial compiler

$$PC^2_{\mathrm{SEQ}} : (\mathrm{source}, \mathrm{target}) \multimap (\mathrm{source} \times \mathrm{source}, \mathrm{target} \times \mathrm{target})$$

generates (partial) sources from the source prefix and suffix, and the targets obtained for these two sources are composed:

$$\begin{array}{l} \mathit{Compiler}\ \ S : \mathrm{source}. \\ \ \ \mathit{Reduction}\ \ (\mathrm{prefix}(S), \mathrm{suffix}(S)), \\ \ \ \mathit{Generation}\ \ T_{\mathrm{prefix}} : \mathrm{target}, T_{\mathrm{suffix}} : \mathrm{target}. \mathrm{Comp}(T_{\mathrm{suffix}}, T_{\mathrm{prefix}}) \end{array}$$

where Comp is an assumed available composition operation with semantics:

$$[\![\mathrm{Comp}(T_{\mathrm{suffix}}, T_{\mathrm{prefix}})]\!] = \lambda d : \mathrm{data}.\ [\![T_{\mathrm{suffix}}]\!]([\![T_{\mathrm{prefix}}]\!](d))$$

Suppose we wish to run our computation on a computer with a CPU and a graphics card (GPU). Assume we have compilers C_{GPU}, generating a GPU target', and C_{CPU}, generating a CPU target, and a term $\mathrm{run}_{\mathrm{GPU}} : \mathrm{target}' \to \mathrm{target}$ that, given T', produces a T with the same semantics that loads T' on the GPU and then runs it on the data supplied to T, returning the result to the CPU. The composition of C_{GPU} with $\mathrm{run}_{\mathrm{GPU}}$ then defines a compiler $C_{\mathrm{G}} : \mathrm{source} \to \mathrm{target}$ such that, for all source's S and data d:

$$[\![C_{\mathrm{G}}(S)]\!](d) = [\![\mathrm{run}_{\mathrm{GPU}}(C_{\mathrm{GPU}}(S))]\!](d) = [\![C_{\mathrm{GPU}}(S)]\!](d)$$

Given a source program, we can then run its prefix on the GPU and its suffix on the CPU, using the binary composition $PC^2_{\mathrm{SEQ}}\langle\!\langle C_{\mathrm{G}}, C_{\mathrm{CPU}}\rangle\!\rangle$ of the binary partial compiler PC^2_{SEQ} with the two compilers C_{G} and C_{CPU}.

3 Compilers and Partial Compilers as First-Class Objects

While composition and tensor are the main operations on compilers and partial compilers, we now discuss five more, shown in Table 1.

Table 1. Generic compiler operations described in this paper

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
Iteration	DO	No	Yes	3.5

3.1 Star

So far we have considered partial compilers whose arity is constant. We generalize, defining partial compilers that operate with lists of sources and targets. For any compiler C : source $\rightarrow$ target, we define C^* : source* $\rightarrow$ target*, the *star* of C, to be the pointwise application of C to all elements of a given list l of sources:

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

Consider the partial compiler PC_{SEQ} : $(\mathrm{source}, \mathrm{target}) \multimap (\mathrm{source}^*, \mathrm{target}^*)$ that generalizes the sequential compiler PC^2_{SEQ} from Section 2.2 by decomposing a source S that is function composition into a list $[S_1, \ldots, S_n]$ of its components. Given a compiler C : source $\rightarrow$ target for simple sources, the composition PC_{SEQ} $\langle\langle C^* \rangle\rangle$ is a compiler for queries that are an arbitrary composition of simple sources. A practical example involving the star operation is given in Section 4.2.

3.2 Conditionals

The partial compiler operations we have constructed so far are all independent of the sources 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 source than another, according to some criterion:

$$Pred : \mathrm{source} \rightarrow \mathrm{bool}$$

We define a natural conditional operation to choose between two compilers

$$\mathrm{COND} : (\mathrm{source} \rightarrow \mathrm{bool}) \times (\mathrm{source} \rightarrow \mathrm{target})^2 \rightarrow (\mathrm{source} \rightarrow \mathrm{target})$$

by:

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

We may write $\mathit{IF}\ \mathit{Pred}\ \mathit{THEN}\ C_1\ \mathit{ELSE}\ C_2$ instead of $\mathrm{COND}(\mathit{Pred}, (C_1, C_2))$. There is an evident analogous conditional operation on partial compilers.

We can use the conditional to "patch" bugs in a compiler without access to its implementation. Assume we have a predicate bug : source $\rightarrow$ bool that describes (a superset of) the sources for which a specific complex optimizing compiler C_{OPT} generates an incorrect target. Let us also assume that we have a simple (non-optimizing) compiler C_{SIMPLE} that always generates correct targets. Then the compiler

$$\mathit{IF}\ \text{bug}\ \mathit{THEN}\ C_{\text{SIMPLE}}\ \mathit{ELSE}\ C_{\text{OPT}}$$

"hides" the bugs in C_{OPT}.

3.3 Cases

Similar to the $*$ operation, but replacing list types by labeled sum types, we can define a "cases" operation, a useful generalization of conditional composition. Given n individual compilers C_{i} : source$_i$ $\rightarrow$ target (for $i = 1, \ldots, n$) together with a function W : source $\rightarrow l_1$: source$_1$ $+ \ldots + l_n$: source$_n$, we define

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

to be the compiler C : source $\rightarrow$ target where:

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

We give a practical example using CASES in Section 4.2.

There is an evident analogous cases operation on partial compilers. Given two partial compilers PC_{i} : (source$_i$, target) $\multimap$ (source$'$, target$'$), we define

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

to be the partial compiler PC : (source, target) $\multimap$ (source$'$, target$'$) given by:

$$\lambda S.\ \mathit{cases}\, W(S)\ \ \mathit{of}\ \ l_1 : PC_1(S), \ldots, l_n : PC_{\text{n}}(S)$$

3.4 Functor

Given functions f : source $\rightarrow$ source$'$ and g : target$'$ $\rightarrow$ target, we define the partial compiler

$$PC_{\text{Func}}(f, g) : (\text{source}, \text{target}) \multimap (\text{source}', \text{target}')$$

to be:

$$\begin{array}{l} \mathit{Compiler}\ \ S : \text{source}. \\ \ \ \mathit{Reduction}\ \ f(S), \\ \ \ \mathit{Generation}\ \ T' : \text{target}'.\, g(T') \end{array}$$

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

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

We describe two useful applications in which g is the identity $\text{id}_{\text{target}}$ on target.

Traditional compilers usually include a sequence of optimizing passes, given by optimizing transformation functions Opt : source $\rightarrow$ source. Such passes correspond to partial compilers of the form $PC_{\text{Func}}(\text{Opt}, \text{Id}_{\text{target}})$.

Staged compilers (e.g., [13,24]) are frequently built from a sequence of transformations between (progressively lower-level) intermediate representations, followed by a final compilation step:

$$\text{source}_1 \xrightarrow{\text{Trans}_1} \ldots \xrightarrow{\text{Trans}_{n-1}} \text{source}_n \xrightarrow{C} \text{target}$$

One can model this structure by composing partial compilers $PC_{\text{Func}}(\text{Trans}_i, \text{Id}_{\text{target}})$, obtaining a partial compiler $PC_{\text{Stage}} : (\text{source}_1, \text{target}) \multimap (\text{source}_n, \text{target})$, where

$$PC_{\text{Stage}} =_{\text{def}} PC_{\text{Func}}(\text{Trans}_1, \text{Id})\langle\!\langle \ldots \langle\!\langle PC_{\text{Func}}(\text{Trans}_{n-1}, \text{Id})\rangle\!\rangle \ldots \rangle\!\rangle$$

The final compiler is then $PC_{\text{Stage}}\langle\!\langle C \rangle\!\rangle$. This integrates staged compilation into our framework in a straightforward way.

3.5 Iteration

The iteration operation iterates a partial compiler

$$PC : (\text{source}, \text{target}) \multimap (\text{source}, \text{target})$$

up to n times, stopping if a given predicate $Pred : \text{source} \to \text{bool}$ becomes true. We define

$$H_{PC} : \text{nat} \to ((\text{source}, \text{target}) \multimap (\text{source}, \text{target}))$$

to be:

$$\begin{aligned} H_{PC}(0) &= \text{Id} \\ H_{PC}(n+1) &= \textit{IF Pred THEN } \text{Id } \textit{ELSE PC}\langle\!\langle H_{PC}(n) \rangle\!\rangle \end{aligned}$$

(We assume the λ-calculus has a facility for primitive recursion.) Applying H_{PC} to $Num : \text{nat}$, one obtains the partial compiler

$$\text{DO } PC \text{ UNTIL } \textit{Pred} \text{ FOR } Num \text{ TIMES}$$

This could be used to repeatedly apply an optimizing compiler PC until a fixed-point is reached, as detected by $Pred$.

4 Application to Query Processing

In this section we return to our motivating problem: 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 μLINQ compilers. In the LINQ terminology, inherited from databases, source programs are called "queries" and target programs are called "plans".

4.1 LINQ and μLINQ

LINQ 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 relational algebra) and comprehension calculi [3]. Much as in 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 to other collections. Queries (source programs) are (syntactic) compositions of LINQ operators. For example, the query `C.Select(e => f(e))`, where `e => f(e)` is the LINQ syntax for the lambda expression λ`e.f(e)`, uses the `Select` operator (called *map* in other programming languages) to apply `f` to every element `e` of a collection `C`. The result is a collection of the same size as the input collection. The elements `e` can have any .Net type, and `f(e)` can be any .Net computation returning a value. The core LINQ operators are named after SQL. All LINQ operators are second-order, as their arguments include functions.

μLINQ Syntax. The *basic datatypes* are ranged over by `I`, `O`, and `K` (which stand for "input", "output" and "key"); they are given by the grammar:

$$\mathtt{I} ::= \mathtt{B} \mid \mathtt{I}^*$$

where `B` ranges over a given set of *primitive* datatypes, such as `int`, the type of integers. The type $\mathtt{I}^*$ stands for the type of collections (taken to be finite lists) of elements of type `I`. The corresponding .NET type is $\mathtt{IEnumerable}\langle\mathtt{I}\rangle$.

μLINQ queries (source programs) consist of sequences of operator applications; they are not complete programs as the syntax does not specify the input data collection (in contrast to LINQ). They are specified by the grammar

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

Here `Exp` and `FExp` range over given sets of *expressions* and *function expressions*, of respective given types `I` or $\mathtt{I}_1 \times \ldots \times \mathtt{I}_n \to \mathtt{O}$. The details of the given primitive types, expressions, and function expressions are left unspecified.

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

$$\begin{array}{l}\mathtt{SelectMany{<}I,O{>}(FExp)} : \mathtt{I}^* \to \mathtt{O}^* \;\; (\text{if } \mathtt{FExp} \text{ has type } \mathtt{I} \to \mathtt{O}^*)\\ \mathtt{Aggregate{<}I{>}(FExp,Exp)} : \mathtt{I}^* \to \mathtt{I}^*\\ \qquad\qquad\qquad (\text{if } \mathtt{FExp} \text{ has type } \mathtt{I} \times \mathtt{I} \to \mathtt{I}, \text{ and } \mathtt{Exp} \text{ has type } \mathtt{I})\\ \mathtt{GroupBy{<}I,K{>}(FExp)} : \mathtt{I}^* \to \mathtt{I}^{**} \quad (\text{if } \mathtt{FExp} \text{ has type } \mathtt{I} \to \mathtt{K})\end{array}$$

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

μLINQ Semantics. We begin with an informal explanation of the semantics. A query of type $\mathtt{I}^* \to \mathtt{O}^*$ denotes a function from `I` collections to `O` collections. We begin with operator applications and then consider composite queries.

`SelectMany<I,O>(FExp)` applied to a collection returns the result of applying `FExp` to all its elements and concatenating the results. So, for example, the query `SelectMany<int,int>(n => [n,n+1])` applied to $C =_{\text{def}} [1, 2, 3, 4, 5]$ results in the list $[1, 2, 2, 3, 3, 4, 4, 5, 5, 6]$.

`Aggregate<I>(FExp,Exp)` applied to a collection produces a singleton list containing the result of a fold operation [11] performed using `FExp` and `Exp`. So, for example, `Aggregate<int,int>((m,n) => m+n,6)` applied to C results in the list $[1+(2+(3+(4+(5+6))))] = [21]$. Some of the compilers we construct require that such aggregations are *(commutatively) monoidal*, i.e., that `FExp` is associative (and commutative) with unit `Exp`.

`GroupBy<I,K>(FExp)` groups all the elements of a collection into a collection of sub-collections, where each sub-collection consists of all the elements in the original collection sharing a common key; the key of a value is computed using `FExp`. The sub-collections in the result 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 C results in the list [[1,3,5],[2,4]].

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 is completed by giving it a denotational semantics. We only show the semantics for a language fragment; it is easy, if somewhat tedious, to spell it out for the full language. First we assign a set $[\![\mathtt{I}]\!]$ to every μLINQ type `I`, assuming every primitive type already has such a set assigned:

$$[\![\mathtt{I}_1 \times \ldots \times \mathtt{I}_n]\!] =_{\mathrm{def}} [\![\mathtt{I}_1]\!] \times \ldots \times [\![\mathtt{I}_n]\!]$$

Next, to any well-typed operator application $\mathtt{OpApp} : \mathtt{I} \to \mathtt{O}$ we assign a function $[\![\mathtt{OpApp}]\!] : [\![\mathtt{I}]\!] \to [\![\mathtt{O}]\!]$, given a denotation $[\![\mathrm{Exp}]\!] \in [\![\mathtt{I}]\!]$ for each expression $\mathtt{Exp} : \mathtt{I}$. For example:

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

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

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

μLINQ and MapReduce. The popular MapReduce [6] distributed computation programming model can be succinctly expressed in μLINQ:

$$\mathtt{MapReduce(map,\ reduceKey,\ reduce)} : \mathtt{I}^* \to \mathtt{O}^*$$

is the same as

```
SelectMany(map);GroupBy(reduceKey);SelectMany(l => [reduce(l)])
```

where $\mathtt{map} : \mathtt{I} \to \mathtt{O}$ is the map function, $\mathtt{reduceKey} : \mathtt{O} \to \mathtt{K}$ computes the key for reduction, and $\mathtt{reduce} : \mathtt{O}^* \to \mathtt{O}$ is the reduction function. (Since we use `SelectMany` for applying the reduction function, the result of `reduce` is embedded into a list with a single element.)

4.2 Compiling μLINQ

A Single-Core Compiler. We start by defining the types for sources (queries) and targets (plans). Let us assume we are given a type FExp corresponding to the set of function expressions, and a type Exp for constants. Then we define types OpAp and MLSource, corresponding to the sets of μLINQ operator applications and queries by setting:

$$\begin{aligned}\text{OpAp} \quad &= \mathit{SelectMany} : \text{FExp} + \\ &\quad\ \mathit{Aggregate} : \text{FExp}\times\text{Exp} + \\ &\quad\ \mathit{GroupBy} : \text{FExp} \\ \text{MLSource} &= \text{OpAp}^*\end{aligned}$$

We assume we have a type MLTarget of μLINQ targets (plans) T with semantics $[\![T]\!]$: $\text{MLData} \longrightarrow \text{MLData}$, where MLData consists of lists of items, where items are either elements of (the semantics of) a basic μLINQ type B, or lists of such items.

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:

$$\begin{aligned}C_{\text{SelectMany}} &: \text{FExp} \longrightarrow \text{MLTarget} \\ C_{\text{Aggregate}} &: \text{FExp}\times\text{Exp} \longrightarrow \text{MLTarget} \\ C_{\text{GroupBy}} &: \text{FExp} \longrightarrow \text{MLTarget}\end{aligned}$$

The denotational semantics of μLINQ operators (Section 4.1) gives 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:

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

Finally, we use the generalized sequential partial compiler PC_{SEQ} and the star operation, both introduced in Section 3.1, to construct a compiler

$$C_{\mu\text{LINQ}} : \text{MLSource} \to \text{MLTarget}$$

for arbitrary μLINQ queries, where

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

A Multi-core Compiler. In this example we construct a partial compiler PC_{MC} to allow our single-core compiler to target a multi-core machine whose cores can execute plans independently. The most obvious way to take advantage of the available parallelism is to decompose the work 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.

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

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

A partial compiler $PC_{\mathrm{MC}} : (\mathrm{OpAp}, \mathrm{MLTarget}) \multimap (\mathrm{OpAp}, \mathrm{MLTarget})$ for operator applications for multi-core machines with cores c_1 and c_2 can be given by:

$$\begin{array}{l} \textit{Compiler} \;\; S : \mathrm{OpAp}. \\ \;\; \textit{Reduction} \;\; S, \\ \;\; \textit{Generation} \;\; T : \mathrm{MLTarget}.\, \mathrm{G_{MC}}(S, T) \end{array}$$

where, for any OpAp S, MLTarget T:

$$\begin{array}{rl} [\![\mathrm{G_{MC}}(S,T)]\!](d) = \lambda d : \mathrm{MLData}.\, \textit{let } d' \textit{ be } \mathrm{part}(S,d) \textit{ in} \\ \mathrm{collate}(S, [\![\mathrm{run}_{c_1}(T)]\!](d'), [\![\mathrm{run}_{c_2}(T)]\!](d \backslash d')) \end{array}$$

The definition of the semantics of $\mathrm{G_{MC}}$, which we now explain, provides a blueprint for its intended parallel implementation. First, the functions $\mathrm{run}_{c_1}, \mathrm{run}_{c_2}$ ensure that their argument $\mathrm{MLTarget}$ is run on the specified core; they act as the identity on the semantics. Next, for any list d, $\mathrm{part}(S,d)$ and $d \backslash \mathrm{part}(S,d)$ constitute a division of d into two parts in a query-dependent manner; here $d \backslash d'$ is chosen so that $d = d' \cdot (d \backslash d')$, if possible (we use $\cdot$ for list concatenation). The function "collate" assembles the results of the computations together, also in a query-dependent manner.

There are many possible ways to define part and $\mathrm{collate}$ and one reasonable specification is shown in Table 2. There, $\mathrm{prefix}(d)$ gives a prefix of d, $\mathrm{head}_{\mathtt{Exp}}(d)$ is the first element of d, assuming d is non-empty, and $[\![\mathtt{Exp}]\!]$ otherwise, and $\mathrm{setr}(d)$, which is used to ensure that a given key is in only one partition, consists of d with all repetitions of an element on its right deleted.

The `SelectMany` operator is homomorphic w.r.t. concatenation. It can be computed by partitioning the collection d into an arbitrary prefix and suffix, applying `SelectMany` recursively on the parts, 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 from both parts will also be disjoint.

The complete multi-core μLINQ compiler is given by

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

It is straightforward to generalize this to machines with n cores by suitably modifying part and $\mathrm{collate}$.

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. The functionality it provides is essentially that of PLINQ [7], the parallel LINQ implementation.

Compilation for Distributed Execution. The strategy employed for 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 including resource allocation and scheduling in the plan language. Consider an example of a cluster of machines, and suppose we are dealing with a large input collection, stored on a distributed filesystem (e.g., [9]) 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. In the following simple example we just use two machines.

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

$$\begin{array}{l} \mathit{Compiler}\ \ S : \mathrm{OpAp}. \\ \ \ \mathit{Reduction}\ \ S, \\ \ \ \mathit{Generation}\ \ T : \mathrm{MLTarget}.\, \mathrm{G_{CL}}(S,T) \end{array}$$

where, for any OpAp S and MLTarget T,

$$\begin{array}{rl} [\![\mathrm{G_{CL}}(S,T)]\!](d) = \lambda d : \mathrm{MLData}. & \\ & \mathit{let}\ \ m_1, m_2 : \mathrm{Machine}\ \mathit{be}\ \mathrm{getm}, \mathrm{getm}\ \mathit{in} \\ & \mathit{let}\ d'\ \mathit{be}\ \mathrm{mpart}(S, d, m_1, m_2)\ \mathit{in} \\ & \ \ \mathrm{collate}(S, [\![\mathrm{run}(m_1, T)]\!](d'), [\![\mathrm{run}(m_2, T)]\!](d \backslash d')) \end{array}$$

Here, $\mathrm{Machine}$ is the type of cluster machines and the constant $\mathrm{getm} : \mathrm{Machine}$ nondeterministically schedules a new machine. When applied to S, d, m_1, m_2, the function mpart returns the first part of a partition of d into two, using a policy not detailed here; as in the case of part, when S is a `GroupBy` the two parts should contain no common keys. Note that the run functions are now parametrized on machines. The relative location of data and machines on the cluster is important. In particular, the partition policy for mpart may depend on that; we also assume that the code $\mathrm{run}(m, T)$ first loads remote data onto m. As before, the semantics of $\mathrm{G_{CL}}$ provides a blueprint for a parallel implementation.

Formally we assume given a set Sch of scheduler states, and as run-time monad T_{run} take $\mathcal{F}^+(\mathrm{Sch} \times X)^{\mathrm{Sch}}$, the standard combination of side-effect and nondeterminism monads ($\mathcal{F}^+(X)$ is the collection of non-empty finite subsets of X); for $[\![\mathrm{getm}]\!]$ we assume an allocation function $\mathrm{Sch} \to \mathcal{F}^+(\mathrm{Sch} \times [\![\mathrm{Machine}]\!])$.

The cluster-level operator application compiler is then obtained by composing the cluster partial compiler with the multi-core compiler described previously

$$PC_{\mathrm{Cluster}}\langle\!\langle PC_{\mathrm{MC}}\langle\!\langle C_{\mathrm{OO}}\rangle\!\rangle\,\rangle\!\rangle$$

and then the complete compiler is:

$$PC_{\mathrm{SEQ}}\langle\!\langle PC_{\mathrm{Cluster}}\langle\!\langle PC_{\mathrm{MC}}\langle\!\langle C_{\mathrm{OO}}\rangle\!\rangle\,\rangle\!\rangle^*\rangle\!\rangle$$

The cluster-level compiler is structurally similar to the multi-core compiler, but the collections themselves are already partitioned and the compiler uses the collection structure to allocate the computation's run-time resources.

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 across both cores and machines. With a tiny change we obtain a compiler that only parallelizes across machines:

$$PC_{\mathrm{SEQ}}\langle\!\langle PC_{\mathrm{Cluster}}\langle\!\langle C_{\mathrm{OO}}\rangle\!\rangle^{*}\rangle\!\rangle.$$

With a little more work one can also add the only important missing MapReduce optimization, namely early aggregation in the map stage.

5 Mathematical Foundations

We now turn to a semantical account of partial compilers in terms of a category of *tactics*. We then discuss the categorical correlates of our polymorphic operations on compilers and partial compilers, and the relationships with the Dialectica category and Milner's tactics. We work with a cartesian closed category **K** with a strong monad T. This supports Moggi's computational lambda calculus [20]: each type σ denotes an object $[\![\sigma]\!]$ of **K**, and every term

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash M : \tau$$

denotes a morphism of **K**

$$[\![M]\!] : [\![\sigma_1]\!] \times \ldots \times [\![\sigma_n]\!] \to [\![\tau]\!]$$

As is common practice, we may confuse terms and their denotations, writing M instead of $[\![M]\!]$; in particular we make free use of the definitions and notation of Section 2. In doing so, we can use types and terms as notations for objects and morphisms, and treat objects x as type constants denoting themselves and morphisms $f : x \to y$ as constants denoting elements of the corresponding function type $x \to y$. We can also use the proof rules of the computational lambda calculus to establish relations between morphisms.

The objects of our category of tactics are pairs (P, S) of objects of **K**; we call P and S (objects of) *problems* and *solutions*, respectively. The morphisms from (P, S) to (P', S') are morphisms of **K** of the form

$$f : P \longrightarrow \mathrm{T}(P' \times (S' \Rightarrow \mathrm{T}(S)))$$

and it is these that are called tactics.

The identity on (P, S) is $\mathrm{Id}_{(P,S)} = \lambda x : P. (x, \lambda y : S. y)$ and the composition $(P, S) \xrightarrow{gf} (P'', S'')$ of $(P, S) \xrightarrow{f} (P', S')$ and $(P', S') \xrightarrow{g} (P'', S'')$ is $gf = f\langle\!\langle g\rangle\!\rangle$ (note the order reversal), making use of the definition in Section 2. Using Moggi's laws for the computational lambda calculus, one can show that composition is associative with the identity as unit, and so this does indeed define a category.

Rather than speaking of sources, targets and partial compilers, we have chosen here to speak more neutrally of problems, solutions and tactics. We follow Blass [2] for problems and solutions, and Milner for tactics: one can think of tactics as tactics for reducing problems to subproblems. Compilers are simply modelled as Kleisli morphisms $P \to \mathrm{T}(S)$.

We now consider the categorical operations corresponding to some of the operations on partial compilers and compilers that we defined above. We define the action of a given tactic $f : (P, S) \longrightarrow (P', S')$ on a Kleisli morphism $h : P^{'} \longrightarrow \mathrm{T}(S^{'})$ by:

$$h \cdot f = f\langle\!\langle h \rangle\!\rangle : P \longrightarrow S$$

using the composition operation of partial compilers with compilers of Section 2. In terms of this “right action” notation the action equations of Section 2 become:

$$(h \cdot g) \cdot f = h \cdot gf \qquad h \cdot \mathrm{Id} = \mathrm{Id}$$

We define tensors of Kleisli morphisms and tactics similarly, again making use of the definitions in Section 2. The expected functorial laws

$$\mathrm{Id} \otimes \mathrm{Id} = \mathrm{Id} \qquad (f' \otimes g')(f \otimes g) = (f'f \otimes g'g)$$

for the tensors of tactics hold if the monad is *commutative* [14], for example when there are no compile-time effects, or for nondeterminism, probabilistic choice, or nontermination (so having recursion is fine); typical cases where they fail are exceptions or side-effects. When they hold, so too do the expected associativity, action, and unit laws for the n-ary compositions defined in Section 2.

In general one obtains only a premonoidal structure [23] with weaker laws:

$$\mathrm{Id} \otimes \mathrm{Id} = \mathrm{Id} \qquad (f \otimes g) = (g \otimes \mathrm{Id})(\mathrm{Id} \otimes f)$$

$$(f' \otimes \mathrm{Id})(f \otimes \mathrm{Id}) = (f'f \otimes \mathrm{Id}) \qquad (\mathrm{Id} \otimes g')(\mathrm{Id} \otimes g) = (\mathrm{Id} \otimes g'g)$$

The “twist” in the definition of the tensor in Section 2 of two tactics is needed to obtain these laws. The weaker laws yield correspondingly weaker laws for the n-ary compositions.

Turning to Section 3, the cases operation arises from the fact that categorical sums exist when the solution objects are the same, i.e., $(P_1, S) + (P_2, S) = (P_1 + P_2, S)$, and the functorial operation arises from the evident functor from $\mathbf{K}_{\mathrm{T}}^{\mathrm{op}} \times \mathbf{K}_{\mathrm{T}}$ to the category of tactics ($\mathbf{K}_{\mathrm{T}}$ is the Kleisli category of T). The literature on Dialectica categories contains further functorial constructions that may also prove useful—for example, the sequential construction of Blass [2] is intriguing.

The Dialectica category has the same objects as the tactics category. A morphism $(f, g) : (P, S) \longrightarrow (P', S')$ consists of a *reduction* function $f : P \longrightarrow P'$ and a *solution* function $g : P \times S' \longrightarrow S$. This is essentially the same as a tactic, in the case of the identity monad, and the Dialectica category is then equivalent to the category of tactics. To incorporate compile-time effects in the Dialectica category, one might alternatively try $f : P \longrightarrow \mathrm{T}(P')$ and $g : P \times S' \longrightarrow \mathrm{T}(S)$. However this does not give a category: the evident composition is not associative.

As we have said, partial compilers also arose by analogy with Milner's tactics. Milner cared about sequents and theorems, whereas we care about sources and targets. His tactics produce lists and have the form:

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

But these are nothing but 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, e.g., we both use a composition operation, though his is adapted to lists, and the composition of two tactics may fail. He also makes use of an OR tactical, which tries a tactic and, if that fails (by raising a failure exception), tries an alternate; we have replaced that by our conditional partial compiler.

6 Implementations

Section 4 describes a compiler for a stylized language. We used the compiler forest architecture to implement two proof-of-concept compilers for (essentially) functional languages targeting a computer cluster: one for LINQ and one for matrix computations. The implementations reuse multiple partial compilers.

Our compiler forest implementations closely parallel the examples in this paper. The lowest layer implements "tactics" (see Section 5): computations on abstract problems and solutions that provide the basic composition operation. On top of this we build a partial compiler abstraction, where problems are source programs and solutions are targets. We then implement a combinator library for the operations described in Sections 2 and 3. A set of abstract base classes for partial compilers, programs, data, optimization passes, and execution engines provide generic useful operations. A set of libraries provides support for manipulating .Net `System.Linq.Expressions` objects, which are the core of the intermediate representation used by all our compilers. To implement partial compilers one writes source reduction functions R and target generation functions G, exactly as described in Section 2.

Compiling LINQ. The LINQ compiler structure closely parallels the description from Section 4, but handles practically the entire LINQ language, with a cluster-level compiler (PC_{Cluster}), a machine multi-core compiler (PC_{MC}), and a core-level compiler based on native LINQ-to-objects. We also implemented a simple GPU compiler C_{GPU} based on Accelerator [26]. A conditional partial compiler steers queries to either C_{GPU} or PC_{MC}, since C_{GPU} handles only a subset of LINQ, and operates on a restricted set of data types.

While our implementation is only preliminary, it performs well and has served to validate the architectural design. For example, when running MapReduce queries, our multi-core compiler produces a speed-up of 3.5 using 4 cores. We tested our compiler on a cluster with 200 machines; at this size the performance of MapReduce computations is essentially the same as with DryadLINQ, since I/O is the dominant cost in such applications.

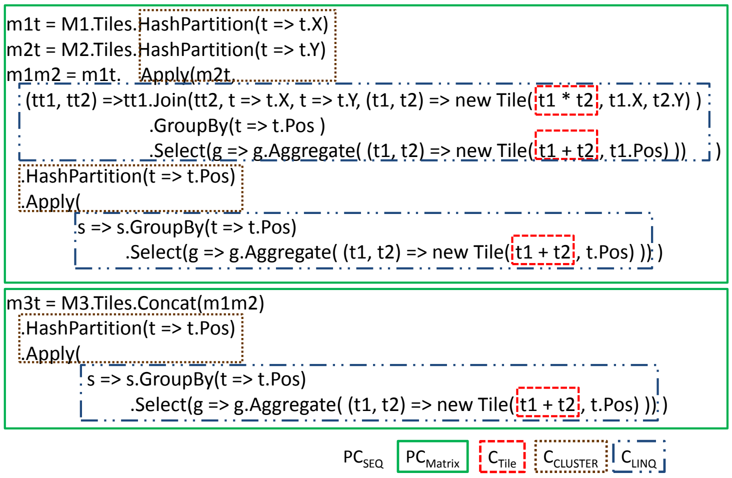
Compiling Matrix Algebra. We have defined a simple functional language for computing on matrices, with operations such as addition, multiplication, transposition, solving linear systems, Cholesky factorization, and LU decomposition. All these operations are naturally parallelizable. The matrices are modeled as two-dimensional collections of tiles, where the tiles are smaller matrices. Large-scale matrices are distributed collections of tiles, each of which is a matrix composed of smaller tiles. This design is useful for dense matrices; by making tiles very small it can also accommodate sparse matrices.

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 on collections of tiles, where the functions FExp applied to the elements are also tile/matrix operations. The collection computations are then passed to the distributed LINQ compiler of Section 6 to generate code running on a cluster. The basic distributed matrix compiler is:

$$PC_{\mathrm{SEQ}}\langle\langle PC_{\mathrm{Matrix}}\langle\langle C_{\mathrm{Tile}}, C_{\mathrm{Cluster}}\rangle\rangle^{*}\ \rangle\rangle$$

where PC_{Matrix} is a binary partial compiler that rewrites an operation on matrices in terms of a LINQ computation (compiled by its second child) applying functions to a set of tiles (compiled by its first child), and C_{Cluster} is the distributed LINQ compiler described previously.

Figure 2 illustrates how the work of compiling the expression $M1 \times M2 + M3$ is partitioned between the compilers involved. In this example we do not use a multi-core LINQ compiler as part of C_{Cluster}.



```
m1t = M1.Tiles.HashPartition(t => t.X)
m2t = M2.Tiles.HashPartition(t => t.Y)
m1m2 = m1t.  Apply(m2t,
  (tt1, tt2) =>tt1.Join(tt2, t => t.X, t => t.Y, (t1, t2) => new Tile(t1 * t2, t1.X, t2.Y) )
                    .GroupBy(t => t.Pos )
                    .Select(g => g.Aggregate( (t1, t2) => new Tile(t1 + t2, t1.Pos) ))   )
  .HashPartition(t => t.Pos)
  .Apply(
        s => s.GroupBy(t => t.Pos)
             .Select(g => g.Aggregate( (t1, t2) => new Tile(t1 + t2, t.Pos) )) )

m3t = M3.Tiles.Concat(m1m2)
  .HashPartition(t => t.Pos)
  .Apply(
         s => s.GroupBy(t => t.Pos)
              .Select(g => g.Aggregate( (t1, t2) => new Tile(t1 + t2, t.Pos) )) )
```

Fig. 2. Intermediate result produced when compiling the expression `M1 * M2 + M3` using the distributed matrix compiler. The colored dotted lines indicate how various parts of the program are generated or assigned to various compilers; PC_{SEQ} is responsible for the complete program. We show the logical program state just before the leaf compilers C_{Tile} and C_{LINQ} (which is a part of C_{Cluster}) are invoked. `HashPartition` implements the "part" partitioning construct, while `Apply` corresponds to the "run$_m$" construct that executes a program on one partition, and `Concat` is concatenation.

7 Related Work

Federated and heterogeneous distributed databases also decompose computations between multiple computation engines. In the former, queries are converted into queries against component databases using wrappers [25,15], and most work concentrates on optimizations. Partial compilers serve a similar, but more general, role as they can have multiple children while wrappers operate on a single database. Regarding the latter, systems such as Dremel [18] that use a tree of databases to execute queries could be implemented in a principled way using a hierarchy of partial compilers.

The authors of [16] use graph transformations to allow multiple analyses to communicate. In [4] cooperating decompilers are proposed, where individual abstract interpretations share information. Our approach supports these applications using the iteration operation.

As we have seen, multistage compilers, e.g., [13,24], fit within our framework. However our formalism is more general than standard practice, as non-unary partial compilers enable branching partial multistage compilation, dividing sources between different engines, or parallelizing data computations.

8 Discussion and Conclusions

We made several simplifications so as to concentrate on the main points: partial compilers and their compositions. For example, μLINQ does not have a join operator, and function expressions were left unspecified; in particular they did not contain nested queries. Adding join leads to tree-shaped queries rather than lists, and nested queries lead to DAG's: indeed DryadLINQ plans are DAG's. (There seems to be no natural treatment of operator-labeled DAG's for functional programming in the literature, though there is related work on graphs [8].) There is a version of the star operator of Section 3.1 for trees, which enables the compiler of Section 4.2 to be extended to joins; there should also be a version for DAG's.

A well-known shortcoming of modularity is that it hides information that could potentially be useful across abstraction boundaries thereby impacting performance (see for example the micro-kernel/monolithic operating system debate); in our context, it may prevent cooperating partial compilers from sharing analysis results. A way to "cheat" to solve this problem is to use a partial compiler whose source language is the same as the intermediate language of its parent — a much richer language than the source alone. Whether this approach is practical remains to be validated by more complex compiler implementations.

The benefits of structuring compilers as we do may extend beyond modularity: since partial compilers are now first-class values, operations for compiler creation, composition and extensibility can be exposed to users, allowing compilers to be customized, created and invoked at run-time.

Partial compilers were motivated by the desire to discover the "right" interface between a set of cooperating compilers (the components of DryadLINQ described in the introduction). We were surprised when we stumbled on the partial compiler methodology, because it is extremely general and very simple. A partial compiler *provides a compilation service* to the upper layers (as do traditional compilers), but also *invokes the same, identical service* from the lower layers. While this structure looks overly simple, it is surprisingly powerful; one reason is that the objects that cross the interface between compilers are quite rich (source and target programs).

Acknowledgements. We are grateful to Martı̀n Abadi, Gavin Bierman, Valeria de Paiva, Robert Harper, Martin Hyland, Michael Isard, Frank McSherry, and Phil Scott for their comments and suggestions.

References

1. Benton, N., Hughes, J., Moggi, E.: Monads and Effects. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 42–122. Springer, Heidelberg (2002)
2. Blass, A.: Questions and answers – a category arising in linear logic, complexity theory, and set theory. In: Advances in Linear Logic. London Math. Soc. Lecture Notes, vol. 222, pp. 61–81 (1995)
3. Buneman, P., et al.: Comprehension syntax. SIGMOD Record 23(1), 87–96 (1994)
4. Chang, B.-Y.E., Harren, M., Necula, G.C.: Analysis of low-level code using cooperating decompilers. In: Proc. 13th SAS, pp. 318–335. ACM (2006)
5. de Paiva, V.: The Dialectica categories. In: Proc. Cat. in Comp. Sci. and Logic, 1987. Cont. Math., vol. 92, pp. 47–62. AMS (1989)
6. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Proc. 6th OSDI, pp. 137–150. ACM (2004)
7. Duffy, J.: Concurrent Programming on Windows. Addison Wesley (2008)
8. Erwig, M.: Inductive graphs and functional graph algorithms. J. Funct. Program. 11(5), 467–492 (2001)
9. Ghemawat, S., Gobioff, H., Leung, L.: The Google file system. In: Proc. 19th SOSP, pp. 29–43. ACM (2003)
10. Gordon, M.J., Milner, A.J., Wadsworth, C.P.: Edinburgh LCF. LNCS, vol. 78. Springer, Heidelberg (1979)
11. Hutton, G.: A tutorial on the universality and expressiveness of fold. J. Funct. Program. 9(4), 355–372 (1999)
12. Hyland, J.M.E.: Proof theory in the abstract. APAL 114(1-3), 43–78 (2002)
13. Kelsey, R., Hudak, P.: Realistic compilation by program transformation. In: Proc. 16th POPL, pp. 281–292. ACM (1989)
14. Kock, A.: Commutative monads as a theory of distributions. Theory and Applications of Categories 26(4), 97–131 (2012)
15. Kossmann, D.: The state of the art in distributed query processing. ACM Comput. Surv. 32, 422–469 (2000)
16. Lerner, S., et al.: Composing dataflow analyses and transformations. In: Proc. 29th POPL, pp. 270–282. ACM (2002)
17. Meijer, E., et al.: LINQ: reconciling object, relations and XML in the .NET framework. In: Proc. SIGMOD Int. Conf. on Manage. Data, p. 706. ACM (2006)
18. Melnik, S., et al.: Dremel: interactive analysis of web-scale datasets. Proc. VLDB Endow. 3, 330–339 (2010)
19. Milner, R., Bird, R.: The use of machines to assist in rigorous proof. Phil. Trans. R. Soc. Lond. A 312(1522), 411–422 (1984)
20. Moggi, E.: Computational lambda-calculus and monads. In: Proc. 4th LICS, pp. 14–23. IEEE Computer Society (1989)
21. Moggi, E.: Notions of computation and monads. Inf. Comput. 93(1), 55–92 (1991)
22. Pierce, B.C.: Types and programming languages. MIT Press (2002)
23. Power, J., Robinson, E.: Premonoidal categories and notions of computation. MSCS 7(5), 453–468 (1997)
24. Sarkar, D., Waddell, O., Dybvig, R.K.: Educational pearl: A nanopass framework for compiler education. J. Funct. Program. 15(5), 653–667 (2005)
25. Sheth, A., Larson, J.: Federated database systems for managing distributed, heterogeneous, and autonomous databases. ACM Comput. Surv. 22, 183–236 (1990)
26. Tarditi, D., Puri, S., Oglesby, J.: Accelerator: using data parallelism to program GPU's for general-purpose uses. In: Proc. 12th. ASPLOS, pp. 325–335. ACM (2006)
27. Yu, Y., et al.: DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In: Proc. 8th OSDI, pp. 1–14. ACM (2008)