Pearl: Diagrams for Composing Compilers

John Wickerson
Imperial College London, UK
j.wickerson@imperial.ac.uk

Paul Brunet
University College London, UK
p.brunet@ucl.ac.uk

Abstract

T-diagrams (or ‘tombstone diagrams’) are widely used in teaching for explaining how compilers and interpreters can be composed together to build and execute software. In this Pearl, we revisit these diagrams, and show how they can be redesigned for better readability. We demonstrate how they can be applied to explain compiler concepts including bootstrapping and cross-compilation. We provide a formal semantics for our redesigned diagrams, based on binary trees. Finally, we suggest how our diagrams could be used to analyse the performance of a compilation system.

2012 ACM Subject Classification Software and its engineering → Compilers

Keywords and phrases compiler networks, graphical languages, pedagogy

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.1

1 Introduction

In introductory courses on compilers across the globe, students are taught about the interactions between compilers using ‘tombstone diagrams’ or ‘T-diagrams’. In this formalism, a compiler is represented as a ‘T-piece’. A T-piece, as shown in Fig. 1, is characterised by three labels: the *source* language of the compiler, the *target* language of the compiler, and the language in which the compiler is *implemented*.

Complex networks of compilers can be represented by composing these basic pieces in two ways. *Horizontal* composition means that the output of the first compiler is fed in as the input to the second. *Diagonal* composition means that the first compiler is itself compiled using the second compiler. Figures 2 and 3 give examples of both forms.

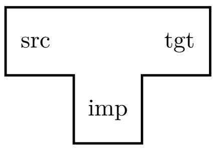


Figure 1 A compiler from source language ‘src’ to target language ‘tgt’ implemented in language ‘imp’.

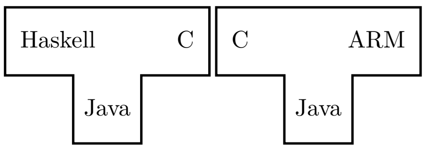


Figure 2 Horizontal composition of T-pieces. Haskell is compiled to C using a compiler written in Java, and thence to ARM assembly using another compiler that is also written in Java.

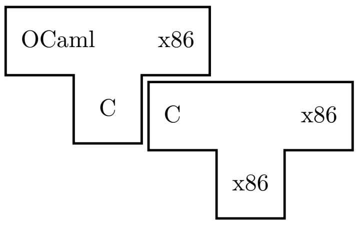


Figure 3 Diagonal composition of T-pieces. An OCaml-to-x86 compiler written in C is being compiled using a C-to-x86 compiler written in x86 assembly.

In this paper, we investigate the foundations of these diagrams. What are the rules that govern how they can be composed (§2)? Can we redesign them to make them more understandable (§3)? What do they mean in general (§4)? And what do they tell us about the compilers they depict (§5)?

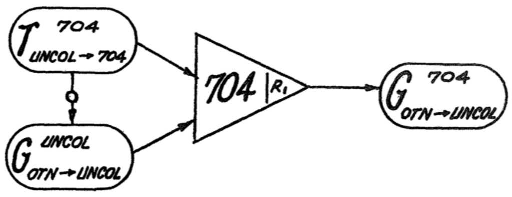


Figure 4 A precursor to the T-diagram [8]. An UNCOL-to-704 compiler (top left) running on an IBM 704 machine (middle) is used to compile an OTN-to-UNCOL compiler implemented in UNCOL (bottom left), which results in an OTN-to-UNCOL compiler implemented in 704 machine code (right).

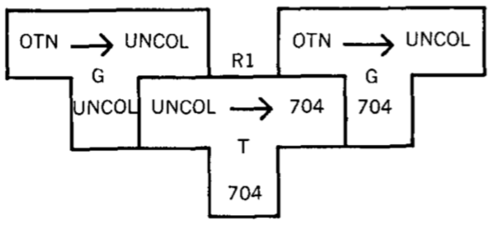


Figure 5 The first T-diagram [1]. This T-diagram depicts the same information as Fig. 4.

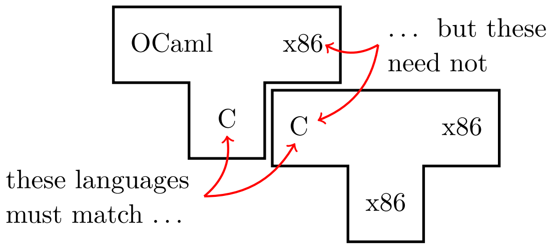


Figure 6 Only one of the two joining interfaces requires its languages to match (**Problem 1**)

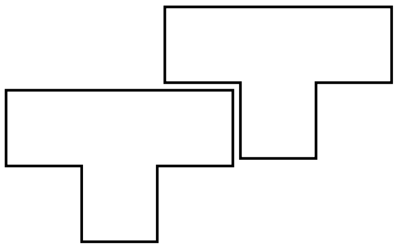


Figure 7 This form of composition looks legal but is not (**Problem 3**)

2 Background

The usefulness of diagrams for explaining the connectivity between compilers can be traced back at least as far as the UNCOL project in the 1950s [8], which was an early effort to provide a 'Universal Computer-Oriented Language' that could interface with many different front-ends and back-ends. Diagrams such as the one reproduced in Fig. 4 were used to show how compilers producing UNCOL could be composed with those that consume it. Shortly thereafter, T-diagrams were proposed by Bratman [1] as an improvement on the UNCOL diagrams. As an example, Fig. 5 shows Bratman's reimagining of Fig. 4.

Three problems with T-diagrams are immediately apparent.

Problem 1 When two T-pieces are diagonally composed, they meet at *two* interfaces, and it is not clear that only *one* of these interfaces is required to have matching languages. As shown in Fig. 6, diagonal composition only requires the 'implementation' language of the left piece to match the 'source' language of the right piece. (There are, of course, ways around this. For instance, McKeeman et al. [5] draw their T-pieces with an additional grey banner showing the name of the compiler, as illustrated later in Fig. 21a.)

Problem 2 T-diagrams typically do not distinguish the *operands* to the composition from the *result* of the composition. For example, in Fig. 5, the right T-piece is actually the result of composing the left T-piece with the middle T-piece, but this relationship is not made clear by the diagram – all three pieces appear on an equal footing. Figure 21a suffers from the same problem to such an extent that it becomes almost unreadable. (Of course, the obvious way to avoid this problem is simply not to show the result of the composition in the same diagram.)

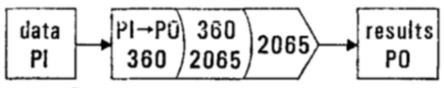


Figure 8 A Rosin diagram [6]. The input data is written in the 'PI' language; this is compiled to the 'PO' language by a compiler implemented in IBM 360 machine code. That compiler is running in a interpreter for 360 machine code that is implemented in 2065 machine code, which itself is running on an IBM 2065 machine.

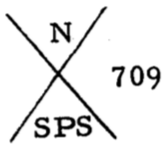


Figure 9 A Burkhardt diagram [2] of a compiler that accepts programs in language 'N', generates programs in language 'SPS', and is implemented in IBM 709 machine code.

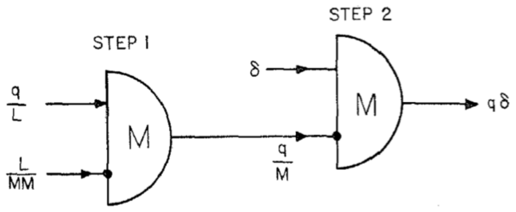


Figure 10 A diagram by Sklansky et al. [7]. Step 1 shows an algorithm q written in language L ($\frac{q}{L}$) being compiled by an L-to-M compiler implemented in M ($\frac{L}{MM}$) on a machine M, to produce an implementation of q in M ($\frac{q}{M}$). In Step 2, this implementation is executed on machine M with data δ, to produce the result $q(\delta)$.

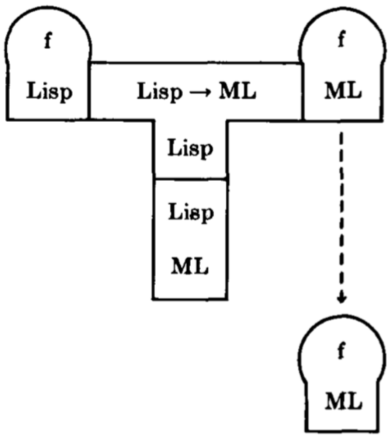


Figure 11 An Earley–Sturgis diagram [3]. The f function is written in Lisp, then translated to machine language (ML) by a compiler written in Lisp. That compiler is running inside a Lisp interpreter written in ML. The dashed line seems to be an attempt to clarify that the right-hand piece is the *result* of the composition (cf. **Problem 2**).

Problem 3 The symmetric shape of the T-pieces invites a third mode of composition, presented in Fig. 7, that is not actually meaningful. (Note that this form of composition does *seem* to appear in Fig. 5, but this is only because the operands and the result of composition are being conflated, as a result of **Problem 2**.)

Over the next couple of decades, several other diagrammatic systems were proposed. For example, Burkhardt [2] proposed replacing Bratman's T-pieces with X-shaped pieces (Fig. 9), Sklansky et al. [7] suggested D-shaped pieces (Fig. 10), Rosin [6] used pieces with a variety of shapes (Fig. 8), and Earley and Sturgis [3] extended Bratman's T-pieces with I-pieces that represent interpreters (Fig. 11). However, none of them satisfactorily resolved all three problems identified above.

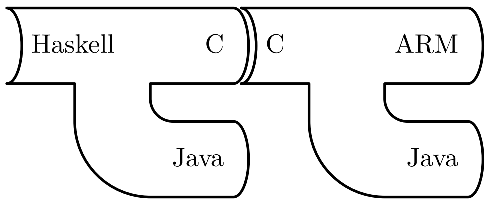


Figure 12 Horizontal composition of J-pieces

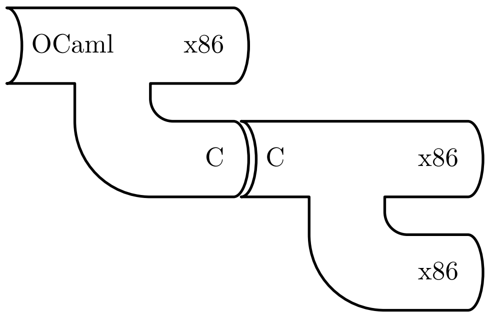


Figure 13 Diagonal composition of J-pieces

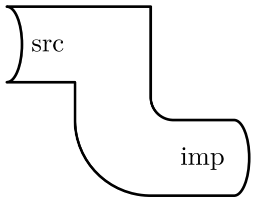


Figure 14 An I-piece

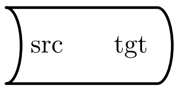


Figure 15 A tube

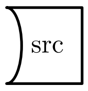
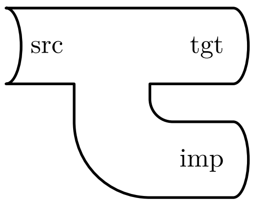


Figure 16 A stopper

3 Our proposal: J-diagrams

We propose a redesign in which the basic piece has the shape of a backwards 'J':



This design makes it clear that there are two (and only two) modes of composition, as shown in Fig. 12 and Fig. 13 respectively. Thus **Problem 1** and **Problem 3** are immediately addressed. To address **Problem 2**, we simply propose that J-pieces that are the *result* of the composition are not shown in the same diagram.

We also propose two special cases of the J-piece that omit one interface. An I-piece (Fig. 14) omits the 'target' interface; it represents an interpreter, following Earley and Sturgis [3]. A 'tube' (Fig. 15) omits the 'implementation' interface; it represents a compilation step that does not require an implementation, such as compiling Clight [4] to its superset, C. A 'stopper' (Fig. 16) omits both the 'target' and 'implementation' interface; it represents a machine that can directly execute programs in the 'source' language.

Over the next few pages, we illustrate J-diagrams using several examples: Java compilation (Fig. 17), verified C compilation using CompCert (Fig. 18), a compiler testing framework (Fig. 19), cross-compilation (Fig. 20), and the history of the XPL programming language (Fig. 21). In each case, we argue that the J-diagram is intuitive and clear, especially when compared to an equivalent T-diagram or ad-hoc diagram.

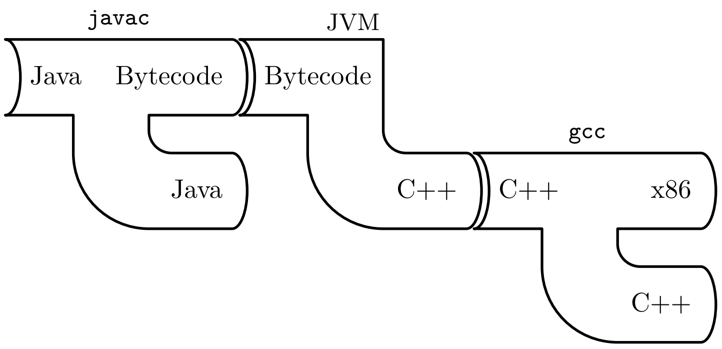


Figure 17 Java compilation as a J-diagram. A Java program is compiled to Java Bytecode using the `javac` compiler, which is written in Java. The JVM is an interpreter for Java Bytecode written in C++. The JVM itself is compiled using a C++ compiler such as `gcc`. Both `javac` and `gcc` must themselves be compiled somehow, but this diagram elects to leave those steps unspecified.

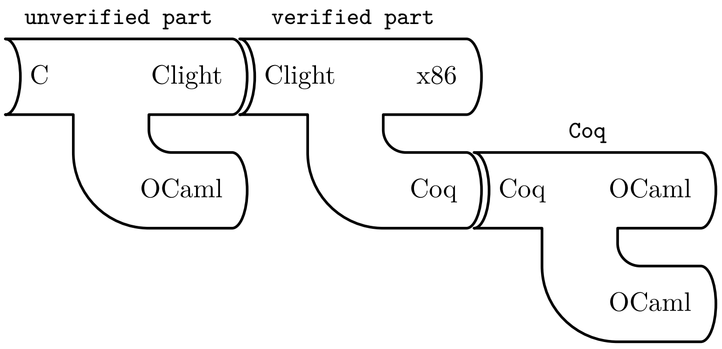


Figure 18 A J-diagram showing the high-level architecture of the CompCert compiler [4]. The initial translation from C into the Clight sublanguage is not verified, but the rest of the compiler is written in Coq. The Coq tool is used to extract executable OCaml code.

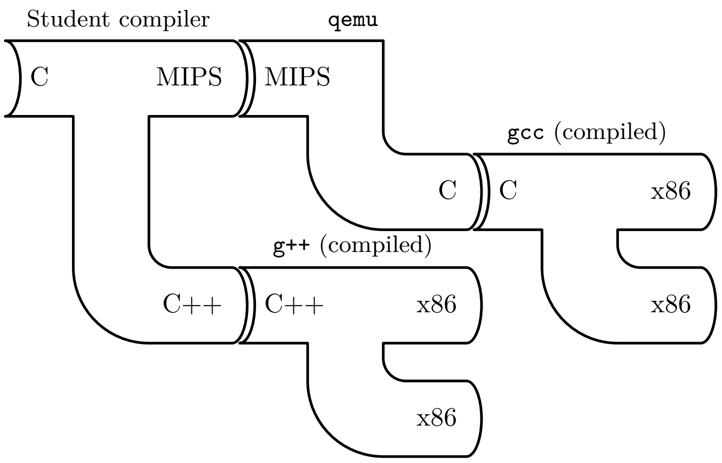
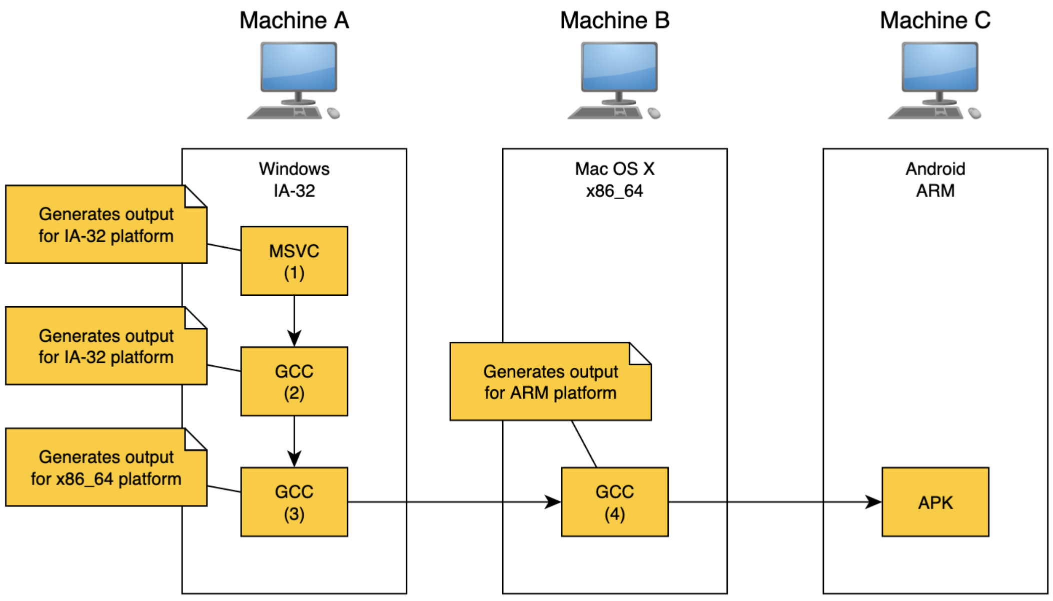
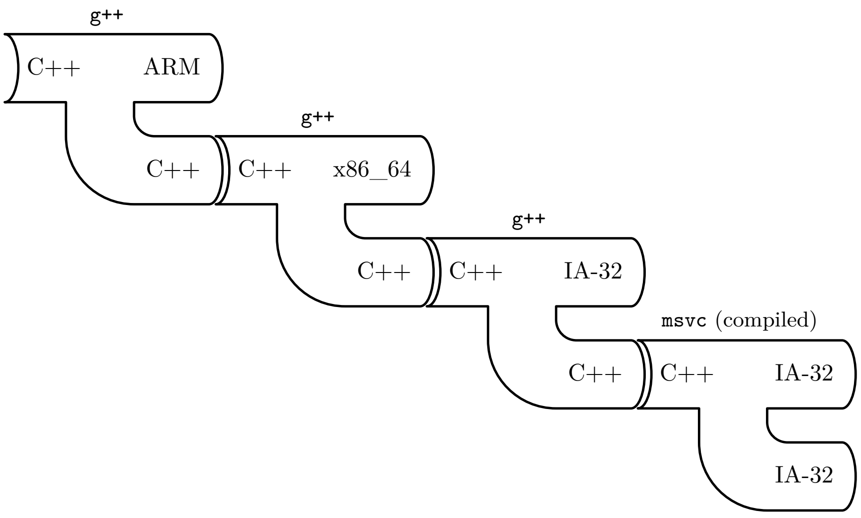


Figure 19 A J-diagram showing how compilers written by students are assessed. Students submit a C-to-MIPS compiler implemented in C++, which is built using `g++`. The output of this compiler is tested by running it on the `qemu` MIPS emulator, which is implemented in C and built using `gcc`. In this diagram, we assume that `g++` and `gcc` have both been compiled for an x86 machine.

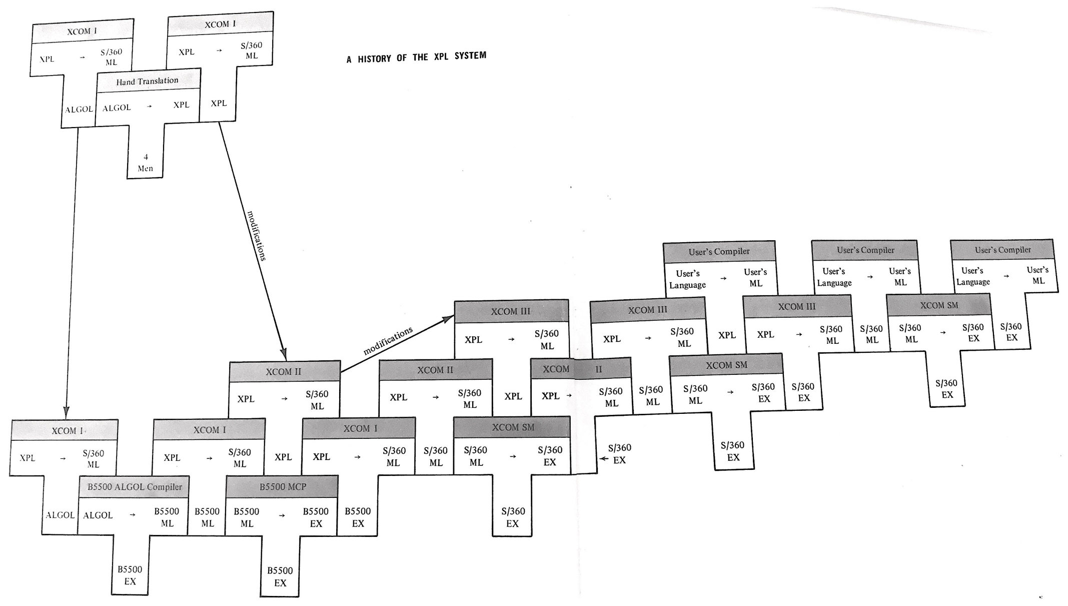

Machine A
Machine B
Machine C
Windows
IA-32
Mac OS X
x86_64
Android
ARM
Generates output for IA-32 platform
MSVC (1)
Generates output for IA-32 platform
GCC (2)
Generates output for x86_64 platform
GCC (3)
Generates output for ARM platform
GCC (4)
APK

(a) A diagram from Wikipedia explaining cross-compiling (`https://en.wikipedia.org/wiki/Cross_compiler`).

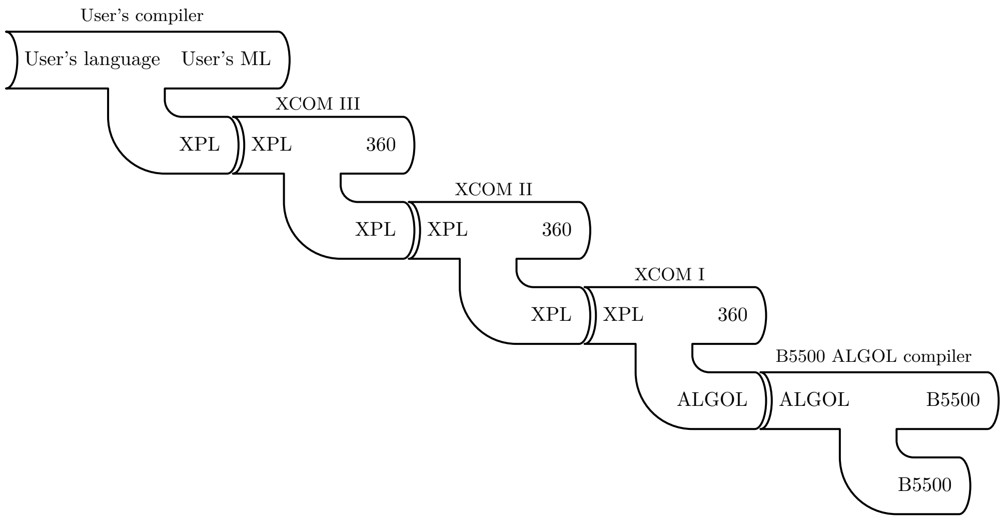


(b) A J-diagram explaining cross-compiling (best read right-to-left).

Figure 20 Explaining cross-compilation using **(a)** an ad-hoc diagram and **(b)** a J-diagram. The aim here is to use one machine (say, running Windows on an IA-32 processor) to build a compiler that can execute on another machine (say, running Mac OS on an x86_64 processor) and whose output that be executed on a third machine (say, running Android on an ARM processor). In both cases, the sequence is as follows. We first use the native Microsoft Visual C++ (`msvc`) compiler on the Windows machine to build the native `g++` compiler for that machine. We then use this compiler to build a `g++` compiler that targets the Mac. This compiler, in turn, is used to build the final compiler, which will execute on the Mac but will generate programs that execute on the Android device.



(a) The history of the XPL language as a T-diagram [5]. The large number of T-pieces, together with the ambiguity about which connections are meaningful, mean that this diagram is hard to understand.



(b) As a J-diagram, the history of the XPL language becomes much easier to read than in Fig. 21a. The first XPL compiler (XCOM I) targeted IBM 360 machine code, and was written in ALGOL, which itself was compiled and executed on a Burroughs B5500 computer (far right). The second XPL compiler (XCOM II) was written in XPL, and was compiled using XCOM I, via a bootstrapping process. Similarly, the third version (XCOM III) was compiled using XCOM II. At the far left of the diagram, we see XPL being used to produce a compiler for a new user.

Figure 21 Explaining the history of the XPL language using **(a)** a T-diagram and **(b)** a J-diagram.

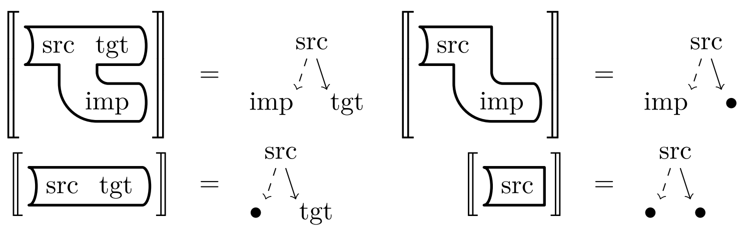


Figure 22 Interpreting J-pieces, I-pieces, tubes, and stoppers as vertex-labelled binary trees.

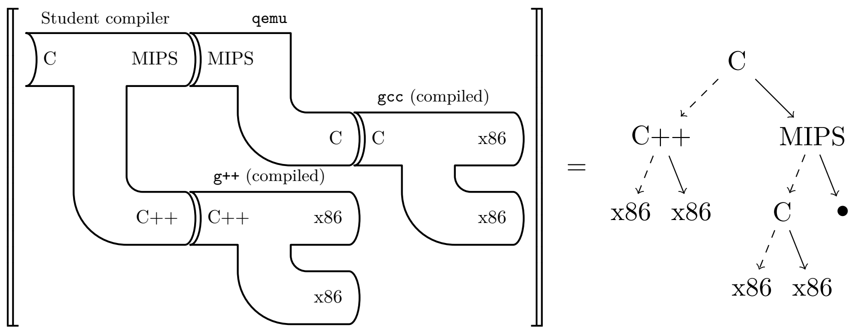


Figure 23 The J-diagram in Fig. 19 interpreted as a tree.

4 Interpreting J-diagrams

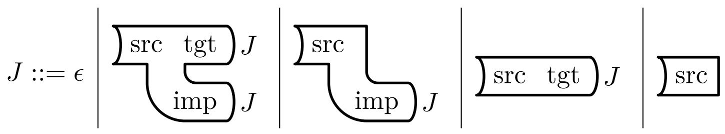
Our J-diagrams can be interpreted as vertex-labelled binary trees. As shown in Fig. 22, each J-piece represents a tree with three vertices, with each vertex labelled either with a language or with the distinguished '•' symbol. Composition of pieces corresponds to gluing trees together so that a leaf of the first tree has the same label as the root of the second tree. Figure 23 provides an example of a tree obtained in this way.

We claim that interpreting a J-diagram as a tree in this way is natural. At the root of the tree is the 'source' language we wish to execute. At the leaves of the tree are all the languages that we need to be able to execute in order to be able to execute the language at the root of the tree. There is actually no particular need to distinguish between the 'implementation' language and the 'target' language – both are simply languages that we need to be able to execute. In other words, a compiler can be seen as a scheme for reducing the problem of executing programs in its 'source' language into two different problems:

- the problem of executing programs in its 'target' language (the compiler's output) and
- the problem of executing programs in its 'implementation' language (the compiler itself).

For example, Fig. 19 can be thought of as a method for executing C programs that depends upon a method for executing x86 programs.

We now provide some more details about how J-diagrams can be interpreted as trees. The (two-dimensional) syntax of J-diagrams is defined by the following grammar.



For every J-diagram J except ϵ, we define *root*(J) as the language at its root. We are then in a position to define *well-formed* J-diagrams; that is, those where the languages at

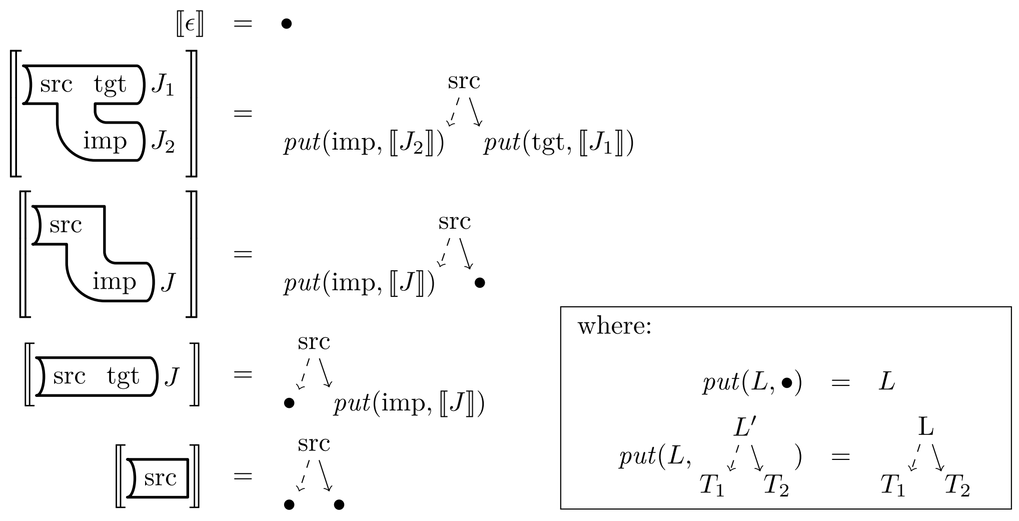
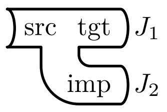

$[\![\epsilon]\!] = \bullet$
src tgt J_1
imp J_2
$put(\text{imp}, [\![J_2]\!])$ $put(\text{tgt}, [\![J_1]\!])$
src imp J
$put(\text{imp}, [\![J]\!])$
src tgt J
src
where:
$put(L, \bullet) = L$
$put(L, L'(T_1, T_2)) = L(T_1, T_2)$

Figure 24 The semantics of J-diagrams

each interface match. A J-diagram is well-formed if and only if for every J-piece



that it contains, we have both $root(J_1) = \text{tgt}$ (or $J_1 = \epsilon$) and $root(J_2) = \text{imp}$ (or $J_2 = \epsilon$). Well-formedness of I-pieces and tubes is defined similarly.

Next, for representing vertex-labelled binary trees, we require the following simple syntax:

$$T ::= \bullet \;\Big|\; L \;\Big|\; \begin{tikzcd}[column sep=tiny] & L \arrow[dl, dashed] \arrow[dr] & \\ T & & T \end{tikzcd}$$

where L is any language.

Ultimately, we can define the semantics of (well-formed) J-diagrams as shown in Fig. 24.

▶ Remark 1 (An alternative diagrammatic system). Having interpreted our diagrams as trees, another way to draw our diagrams presents itself, which we present as an alternative. The basic piece, shown in Fig. 25, returns to the original 'T' shape, but loses the 'blockiness' of the original diagrams. Composition is then done by matching the bottom or right-hand vertex of the first tree with the left-hand vertex of the second tree. Figure 26 gives a larger example of this alternative style of diagram.

$$\begin{tikzcd} \text{src} \arrow[r, dash] & {} \arrow[r] \arrow[d, dashed] & \text{tgt} \\ & \text{imp} & \end{tikzcd}$$

Figure 25 An alternative representation of a compiler.

$$\begin{tikzcd} \text{C} \arrow[r, dash] & {} \arrow[rrrr] \arrow[d, dashed] & & & & \text{MIPS} \arrow[d, dashed] & & \\ & \text{C++} \arrow[r, dash] & {} \arrow[r] \arrow[d, dashed] & \text{x86} & & \text{C} \arrow[r, dash] & {} \arrow[r] \arrow[d, dashed] & \text{x86} \\ & & \text{x86} & & & & \text{x86} & \end{tikzcd}$$

Figure 26 An alternative to the J-diagram in Fig. 19.

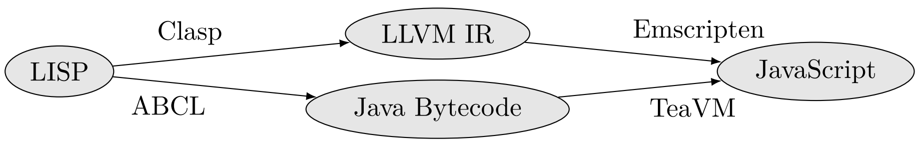
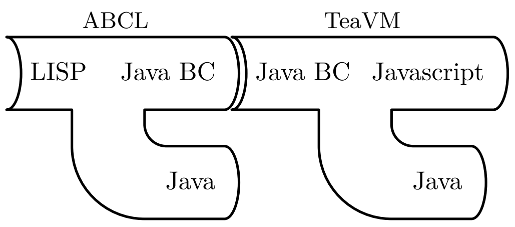
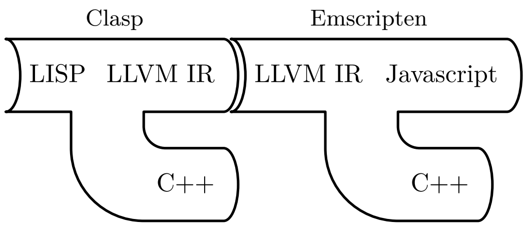


Figure 27 A graph of possible compilation strategies from LISP to JavaScript.



(a) First option: LISP is compiled to Javascript via Java Bytecode using ABCL (`https://abcl.org`) and TeaVM (`http://teavm.org`), both of which are implemented in Java.



(b) Second option: LISP is compiled to Javascript via LLVM IR using Clasp (`https://github.com/clasp-developers/clasp`) and Emscripten (`http://emscripten.org`), both implemented in C++.

Figure 28 Designing a compilation strategy from LISP to JavaScript with the help of J-diagrams.

5 Using J-diagrams to analyse compilers

In this section, we suggest how J-diagrams can be used to aid the analysis and comparison of compilation strategies.

5.1 Comparing compilation strategies using J-diagrams

Figure 27 shows a graph of possible compilation strategies from LISP to JavaScript. This is the kind of graph generated by Akram's tool for discovering all ways – whether direct or indirect – of compiling between any two given languages (`https://akr.am/languages`). Each vertex represents a language, and each edge represents a compiler from one language to another.

What is missing from this graph is information about how each compiler is implemented, and hence which further compilers may be needed in order to build them. To address this shortcoming, Fig. 28 shows both possible compilation strategies as a pair of J-diagrams. By using this representation, it becomes clear that if we compile via Java Bytecode we shall also need the ability to execute Java code, and if we compile via LLVM IR we shall also need the ability to execute C++ code. Thus, more information is made available for the designer to make an informed choice.

5.2 Performance analysis using J-diagrams

If we have additional information about each J-piece beyond simply the languages involved, then we can use J-diagrams as an aid for reasoning about the performance of compilation strategies, as outlined in this subsection.

For any single J-piece, it is natural to ask:

- How good is the generated output? Or, to coin a phrase, how *well-targeted* is the compiler? As a simple example, a compiler that inserts unnecessary `sleep()` instructions into its generated output is probably *not* well-targeted.

- How long does it take to run? Or, to coin another phrase, how *well-implemented* is the compiler? As a simple example, a compiler whose source code includes unnecessary `sleep()` instructions is probably *not* well-implemented.

Then, for a J-diagram with source language S, it is natural to ask: if I use this compilation strategy to execute programs in language S, how quickly will I get the result?

A straightforward answer is that it depends upon how well-targeted and well-implemented all the compilers in the network are, since all of the compilers must be run before the final result can be obtained. For instance, in Fig. 18, in order to calculate the result of running the input C program, we need to run Coq to obtain an OCaml implementation of the verified compiler, then we need to run both the unverified compiler and the verified compiler. If the Coq compiler is well-targeted, then the verified compiler will be well-implemented.

A more nuanced answer involves distinguishing the time taken to obtain an executable ("compilation time") from the time taken to run that executable ("running time"). In fact, there are several compilation times here: the time taken to compile the source program, the time taken to compile that compiler, the time taken to compile *that* compiler, and so on. Traditionally, one is more concerned with improving the running time than the compilation time, since an executable can be compiled once and then run many times. In turn, the time taken to compile the executable tends to be more important than the time taken to compile the compiler, since a compiler can be compiled once and then used many times.

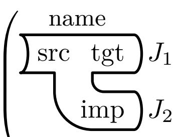
These various compilation times correspond to the different rows of a J-diagram. The running time depends upon how well-targeted the top row of J-pieces are, and how efficient is the machine or interpreter that runs this executable. The time taken to obtain this executable, on the other hand, depends upon how well-implemented the top row of J-pieces are, which in turn depends upon how well-targeted are the J-pieces (in the second row) that compile them. The time taken to obtain the compiler that obtains this executable depends upon how well-implemented the second row of J-pieces are, which in turn depends upon how well-targeted are the J-pieces in the third row, and so on.

For instance, in Fig. 18, the time taken to obtain the CompCert compiler depends upon how well-implemented the Coq compiler is. The time taken to use CompCert obtain an executable from a user-provided C program depends upon how well-implemented the unverified and verified compilers are, and how well-targeted the Coq compiler is. The running time of this executable depends upon how well-targeted the unverified and verified compilers are.

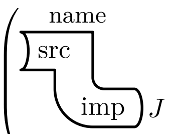
We can formalise this using the *runningTime* and *compileTime* functions defined in Fig. 29, which operate on the formal syntax for J-diagrams laid out in §4. As an example, we can see by unfolding those definitions that the 'compile time' for the J-diagram in Fig. 19 depends upon *well-implementedness*(Student compiler) and *well-targetedness*(`g++`), and that the 'running time' for that J-diagram depends upon *well-targetedness*(Student compiler), *well-implementedness*(`qemu`), and *well-targetedness*(`gcc`), all of which is as expected.

6 Conclusion

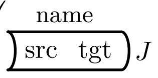
We have investigated the foundations of diagrams that describe how compilers can be composed. To conclude, let us return to the four questions posed in the introduction. First, we explained in §2 that existing graphical systems, chiefly T-diagrams, are unsatisfactory because the rules about how compilers can be composed are not intuitive; for instance, one form of diagonal composition is legal (cf. Fig. 3) but the symmetric one is not (cf. Fig. 7). Second, we presented in §3 a new visual language, based on 'backwards J'-shaped pieces, that *runningTime*(ϵ) :− *(no dependencies)*.

runningTime(

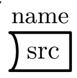
) :− *well-targetedness*(name), *runningTime*(J_1)

runningTime(

) :− *well-implementedness*(name), *runningTime*(J).

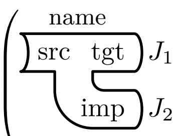
runningTime(

) :− *runningTime*(J).

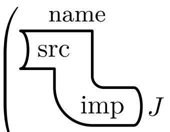
runningTime(

) :− *well-implementedness*(name).

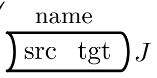
compileTime(ϵ) :− *(no dependencies)*.

compileTime(

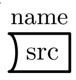
) :− *well-implementedness*(name), *runningTime*(J_2)

compileTime(

) :− *(no dependencies)*.

compileTime(

) :− *compileTime*(J).

compileTime(

) :− *(no dependencies)*.

Figure 29 How the running time and compile time of a program depend upon the well-implementedness and well-targetedness of the various compilers involved. Following a Prolog-like notation, we write ':−' to mean 'depends upon'. Compared to the formal syntax for J-diagrams laid out in §4, we have added a 'name' label to each piece so we can easily refer to it.

addressed the shortcomings of previous systems. Third, in answer to the question of what these diagrams 'mean', we showed in §4 how they can be interpreted quite straightforwardly as binary trees. A J-diagram can thus be thought of as a strategy for problem-reduction: it describes how the problem of executing programs in the language at its root vertex can be reduced to the problems of executing programs in all the languages at its leaf vertices. Fourth, in answer to the question of what these diagrams tell us about the compilers they depict, we suggested in §5 how they could be a useful aid when comparing different compilation strategies, or when analysing which parts of the compilation process are on the 'critical path' regarding compiling time or running time.

Ultimately, we expect J-diagrams will prove most valuable in teaching settings, because they are – we believe – an improvement upon the T-diagrams that are already in widespread use.

References

1 Harvey Bratman. An alternate form of the "UNCOL diagram". *Communications of the ACM*, 4(3):142, 1961.

2 Walter H. Burkhardt. Universal programming languages and processors: A brief survey and new concepts. In *American Federation of Information Processing Societies (AFIPS)*, pages 1–21, 1965.

3 Jay Earley and Howard Sturgis. A formalism for translator interactions. *Communications of the ACM*, 13(10):607–617, 1970.

4 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

5 William Marshall McKeeman, James J. Horning, and David B. Wortman. *A compiler generator*. Englewood Cliffs, 1970.

6 Robert F. Rosin. A graphical notation for describing system implementation. *Software: Practice and Experience*, 7:239–250, 1977.

7 Jack Sklansky, M. Finkelstein, and E.C. Russell. A formalism for program translation. *Journal of the ACM*, 15(2):165–175, 1968.

8 J. Strong, Joseph Henry Wegstein, Alan L. Tritter, J. Olsztyn, Owen R. Mock, and T. Steel. The problem of programming communication with changing machines: A proposed solution (part 2). *Communications of the ACM*, 1(9):9–15, 1958.