Cables, Trains and Types

Simon J. Gay*

School of Computing Science, University of Glasgow, UK.
Simon.Gay@glasgow.ac.uk

Abstract. Many concepts of computing science can be illustrated in ways that do not require programming. *CS Unplugged* is a well-known resource for that purpose. However, the examples in *CS Unplugged* and elsewhere focus on topics such as algorithmics, cryptography, logic and data representation, to the neglect of topics in programming language foundations, such as semantics and type theory.
This paper begins to redress the balance by illustrating the principles of static type systems in two non-programming scenarios where there are physical constraints on forming connections between components. The first scenario involves serial cables and the ways in which they can be connected. The second example involves model railway layouts and the ways in which they can be constructed from individual pieces of track. In both cases, the physical constraints can be viewed as a type system, such that typable systems satisfy desirable semantic properties.

1 Introduction

There is increasing interest in introducing key concepts of computing science in a way that does not require writing programs. A good example is *CS Unplugged* [2], which provides resources for paper-based classroom activities that illustrate topics such as algorithmics, cryptography, digital logic and data representation. However, most initiatives of this kind focus on "Theoretical Computer Science Track A" [4] topics (algorithms and complexity), rather than "Track B" topics (logic, semantics and theory of programming). To the extend that logic is covered, the focus is on gates and circuits rather than deduction and proof.

In the present paper, we tackle Track B by describing two non-programming scenarios illustrating the principles of static type systems. The first scenario involves serial cables, and defines a type system in which the type of a cable corresponds to the nature of its connectors. The physical design of the connectors enforces the type system, and this guarantees that the semantics (electrical connectivity) of a composite cable is determined by its type.

The second scenario is based on model railway layouts, where there is a desirable runtime safety property that if trains start running in the same direction, there can never be a head-on collision. Again, the physical design of the pieces of track enforces a type system that guarantees runtime safety. The situation

* Supported by the UK EPSRC grant EP/K034413/1, "From Data Types to Session Types: A Basis for Concurrency and Distribution (ABCD)".

Fig. 1. A serial cable with 25-pin female (left) and male (right) connectors.

here is more complicated than for serial cables, and we can also discuss the way in which typability is only an approximation of runtime safety.

We partially formalise the cables example, in order to define a denotational semantics of cables and prove a theorem about the correspondence between types and semantics. A fully formal treatment would require more machinery, of the kind that is familiar from the literature on semantics and type systems, but including it all here would distract from the key ideas. We treat the railway example even less formally; again, it would be possible to develop a more formal account.

I am only aware of one other non-technical illustration of concepts from programming language foundations, which is Victor's *Alligator Eggs* [12] presentation of untyped λ-calculus. When I have presented the cables and trains material in seminars, audiences have found it novel and enjoyable. I hope that these examples might encourage other such scenarios to be observed — and there may be a possibility of developing them into activities along the lines of *CS Unplugged*.

2 Cables and Types

The first example involves serial cables. These were widely used to connect computers to peripherals or other computers, typically using the RS-232 protocol, until the emergence of the USB standard in the late 1990s. Figure 1 shows a serial cable with 25-pin connectors, and illustrates the key point that there are two polarities of connector, conventionally called *male* and *female*. Figure 2 shows a serial cable with 9-pin connectors, both female. The physical design is such that two connectors can be plugged together if and only if they are of different male/female polarity and have the same number of pins. From now on we will

ignore the distinction between 9-pin and 25-pin connectors, and assume that we are working with a particular choice of size of connector.

For our purposes, the interesting aspect of a serial cable is that it contains two wires for data transmission. These run between the *send* (SND) and *receive* (RCV) pins of the connectors. There are other wires for various power and control signals, but we will ignore them.

There are two ways of connecting the send/receive wires. If SND is connected to SND and RCV is connected to RCV, then the cable is called a *straight through* cable (Figure 3). This is just an extension cable. Alternatively, if SND is connected to RCV and RCV is connected to SND, then the cable enables two devices to communicate because the SND of one is connected to the RCV of the other. This is called a *null modem* cable (Figure 4).

Fig. 2. A serial cable with 9-pin female connectors.

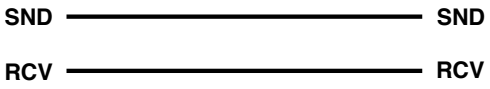

Fig. 3. A straight through cable.

With two ways of wiring SND/RCV, and three possible pairs of polarities for the connectors, there are six possible structures for a serial cable. They have different properties in terms of their electrical connectivity and their physical pluggability. When choosing a cable with which to connect two devices, clearly it is important to have the correct connectors and the correct wiring. Because the wiring of a cable is invisible, there is a conventional correspondence between the choice of connectors and the choice of wiring.

Fig. 4. A null modem cable.

- A straight through cable has different connectors at its two ends: one male, one female.
- A null modem cable has the same connectors at its two ends: both male, or both female.

It is easy to convince oneself that this convention is preserved when cables are plugged together to form longer cables. By thinking of the electrical connectivity of a cable as its semantics, and the nature of its connectors as its type, we can see the wiring convention as an example of a type system that guarantees a semantic property. In the rest of this section, we will sketch a formalisation of this observation.

Figure 5 gives the definitions that we need. Syntactically, a *Cable* is either one of the fundamental cables or is formed by plugging two cables together via the $\cdot$ operator. The fundamental cables are the straight through cable, $\mathsf{straight}$, and two forms of null modem cable, null_1 and null_2. Recalling that a null modem cable has the same type of connector at both ends, the forms null_1 and null_2 represent cables with two male connectors and two female connectors. It doesn't matter which cable is male-male and which one is female-female.

To define the type system, we use the notation of classical linear logic [8]. Specifically, we use *linear negation* $(-)^\perp$ to represent complementarity of connectors, and we use *par* ($\parr$) as the connective that combines the types of connectors into a type for a cable. This is a special case of a more general approach to using classical linear logic to specify typed connections between components [6]. We use $\mathbb{B}$ to represent one type of connector, and then $\mathbb{B}^\perp$ represents the other type. As usual, negation is involutive, so that $(\mathbb{B}^\perp)^\perp = \mathbb{B}$. The notation $\mathbb{B}$ is natural because we will use boolean values as the corresponding semantic domain. It doesn't matter whether $\mathbb{B}$ is male or female, as long as we treat it consistently with our interpretation of null_1 and null_2. The typing rule PLUG, which is a special case of the *cut* rule from classical linear logic, specifies that cables can be plugged together on complementary connectors. In this rule, A, B and C can each be either $\mathbb{B}$ or $\mathbb{B}^\perp$.

Example 1. The cable $\mathsf{straight} \cdot \mathsf{straight}$ represents two straight through cables connected together. It is typable by

$$\frac{\mathsf{straight} : \mathbb{B} \parr \mathbb{B}^\perp \qquad \mathsf{straight} : \mathbb{B} \parr \mathbb{B}^\perp}{\mathsf{straight} \cdot \mathsf{straight} : \mathbb{B} \parr \mathbb{B}^\perp}\ \textsc{Plug}$$

Syntax

$$\mathit{Cable} ::= \mathsf{straight} \mid \mathsf{null}_1 \mid \mathsf{null}_2 \mid \mathit{Cable} \cdot \mathit{Cable} \qquad \text{cables}$$

$$A, B, C ::= \mathbb{B} \mid \mathbb{B}^{\perp} \qquad \text{types}$$

Type equivalence

$$(\mathbb{B}^{\perp})^{\perp} = \mathbb{B}$$

Typing rules

$$\mathsf{straight} : \mathbb{B} \parr \mathbb{B}^{\perp} \qquad \mathsf{null}_1 : \mathbb{B} \parr \mathbb{B} \qquad \mathsf{null}_2 : \mathbb{B}^{\perp} \parr \mathbb{B}^{\perp}$$

$$\frac{c : A \parr B \qquad d : B^{\perp} \parr C}{c \cdot d : A \parr C}\ \textsc{Plug}$$

Semantics

Writing $\mathbb{B}^{[\perp]}$ to represent either $\mathbb{B}$ or $\mathbb{B}^{\perp}$, the denotational semantics of $c : \mathbb{B}^{[\perp]} \parr \mathbb{B}^{[\perp]}$ is

$$[\![c]\!] \subseteq \{\mathtt{true}, \mathtt{false}\} \times \{\mathtt{true}, \mathtt{false}\}$$

defined inductively on the syntactic construction of c by:

$$[\![\mathsf{straight}]\!] = \{(\mathtt{false}, \mathtt{false}), (\mathtt{true}, \mathtt{true})\} \qquad \text{identity, } \mathsf{id}$$

$$[\![\mathsf{null}_1]\!] = \{(\mathtt{false}, \mathtt{true}), (\mathtt{true}, \mathtt{false})\} \qquad \text{inversion, } \mathsf{inv}$$

$$[\![\mathsf{null}_2]\!] = \{(\mathtt{false}, \mathtt{true}), (\mathtt{true}, \mathtt{false})\} \qquad \text{inversion, } \mathsf{inv}$$

$$[\![c \cdot d]\!] = [\![c]\!] \circ [\![d]\!] \qquad \text{relational composition}$$

Fig. 5. Formalisation of cables.

This composite cable has the same type as a single straight through cable, and we will see that it also has the same semantics.

Example 2. The cable $\mathsf{null}_1 \cdot \mathsf{null}_2$ is two null modem cables connected together, which will also be semantically equivalent to a straight through cable. It is typable by

$$\frac{\mathsf{null}_1 : \mathbb{B} \parr \mathbb{B} \qquad \mathsf{null}_2 : \mathbb{B}^{\perp} \parr \mathbb{B}^{\perp}}{\mathsf{null}_1 \cdot \mathsf{null}_2 : \mathbb{B} \parr \mathbb{B}^{\perp}}\ \textsc{Plug}$$

Example 3. The cable $\mathsf{straight} \cdot \mathsf{null}_1$ is a null modem cable extended by plugging it into a straight through cable. Semantically it is still a null modem cable. It is typable by

$$\frac{\mathsf{straight} : \mathbb{B} \parr \mathbb{B}^{\perp} \qquad \mathsf{null}_1 : \mathbb{B} \parr \mathbb{B}}{\mathsf{straight} \cdot \mathsf{null}_1 : \mathbb{B} \parr \mathbb{B}}\ \textsc{Plug}$$

To complete the formalisation of the syntax and type system, we would need some additional assumptions, at least including commutativity of $\parr$ so that we can flip a straight through cable end-to-end to give $\mathsf{straight} : \mathbb{B}^\perp \parr \mathbb{B}$. However, the present level of detail is enough for our current purposes.

We define a denotational semantics of cables, to capture the electrical connectivity. We interpret both $\mathbb{B}$ and $\mathbb{B}^\perp$ as $\{\mathtt{true}, \mathtt{false}\}$ so that we can interpret a straight through cable as the identity function and a null modem cable as logical inversion. Following the framework of classical linear logic, we work with relations rather than functions. Plugging cables corresponds to relational composition.

Example 4. Calculating the semantics of the cables in Examples 1–3 (for clarity, including the type within $[\![-]\!]$) gives

$$[\![\mathsf{straight} \cdot \mathsf{straight} : \mathbb{B} \parr \mathbb{B}^\perp]\!] = \mathsf{id} \circ \mathsf{id} \quad = \mathsf{id} \quad = [\![\mathsf{straight}]\!]$$

$$[\![\mathsf{null}_1 \cdot \mathsf{null}_2 : \mathbb{B} \parr \mathbb{B}^\perp]\!] = \mathsf{inv} \circ \mathsf{inv} = \mathsf{id} \quad = [\![\mathsf{straight}]\!]$$

$$[\![\mathsf{straight} \cdot \mathsf{null}_1 : \mathbb{B} \parr \mathbb{B}]\!] = \mathsf{id} \circ \mathsf{inv} \quad = \mathsf{inv} = [\![\mathsf{null}_1]\!]$$

This illustrates the correspondence between the type of a cable and its semantics.

The following result is straightforward to prove.

Theorem 1. *Let A be either $\mathbb{B}$ or $\mathbb{B}^\perp$ and let c be a cable.*

1. *If $c : A \parr A$ then $[\![c]\!] = \mathsf{inv}$.*
2. *If $c : A \parr A^\perp$ then $[\![c]\!] = \mathsf{id}$.*

Proof By induction on the typing derivation, using the fact that $\mathsf{inv} \circ \mathsf{inv} = \mathsf{id}$. □

This analysis of cables and their connectors has several features of the use of static type systems in programming languages. The semantics of a cable is its electrical connectivity, which determines how it behaves when used to connect devices. The type of a cable is a combination of the polarities of its connectors. There are some basic cables, which are assigned types in a way that establishes a relationship between typing and semantics. The physical properties of connectors enforce a simple local rule for plugging cables together. The result of obeying this rule is a global correctness property: for *every* cable, the semantics is characterised by the type.

It is possible, physically, to construct a cable that doesn't obey the typing rules, by removing a connector and soldering on a complementary one. For example, connecting $\mathsf{straight} : \mathbb{B} \parr \mathbb{B}^\perp$ and $\mathsf{straight} : \mathbb{B}^\perp \parr \mathbb{B}$, by illegally joining $\mathbb{B}^\perp$ to $\mathbb{B}^\perp$, gives a straight through cable with connectors $\mathbb{B} \parr \mathbb{B}$. Such cables are available as manufactured components, called *gender changers*. Usually they are very short straight through cables, essentially two connectors directly connected back to back, with male-male or female-female connections. They are like type casts: sometimes useful, but dangerous in general. If we have a cable that has

been constructed from fundamental cables and gender changers, and if we can't see exactly which components have been used, then the only way to verify that its connectors match its semantics is to do an electrical connectivity test — i.e. a runtime type check.

Typically, a programming language type system gives a *safe approximation* to correctness. Every typable program should be safe, but usually the converse is not true: there are safe but untypable programs. Cable gender changers are not typable, so the following typing derivation is not valid.

$$\dfrac{\dfrac{}{\mathsf{changer}_1 : \mathbb{B} \parr \mathbb{B}}\ \textit{untypable} \qquad \dfrac{}{\mathsf{changer}_2 : \mathbb{B}^\perp \parr \mathbb{B}^\perp}\ \textit{untypable}}{\mathsf{changer}_1 \cdot \mathsf{changer}_2 : \mathbb{B} \parr \mathbb{B}^\perp}\ \textsc{Plug}$$

However, the semantics is defined independently of typing, and

$$\begin{aligned} [\![\mathsf{changer}_1 \cdot \mathsf{changer}_2]\!] &= [\![\mathsf{changer}_1]\!] \circ [\![\mathsf{changer}_2]\!] \\ &= \mathsf{id} \circ \mathsf{id} \\ &= \mathsf{id} \end{aligned}$$

so that the typing $\mathsf{changer}_1 \cdot \mathsf{changer}_2 : \mathbb{B} \parr \mathbb{B}^\perp$ is consistent with Theorem 1.

3 Trains and Types

The second example of a static type system is based on model railway layouts. Specifically, the simple kind that are aimed at young children [1, 5], rather than the elaborate kind for railway enthusiasts [3]. The examples in this paper were constructed using a "Thomas the Tank Engine" [7] set.

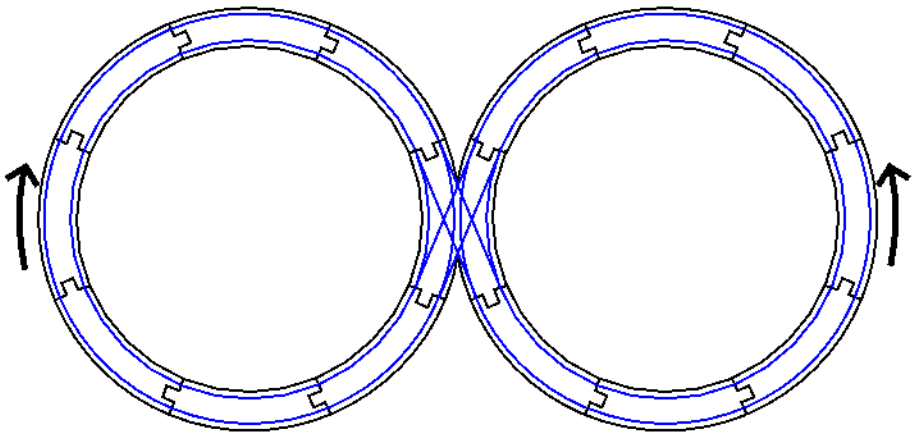

Fig. 6. A figure eight layout.

Figure 6 shows a simple figure eight layout consisting of two circles linked by a crossover piece. The blue lines (coloured in the electronic version of the paper) show the guides for the train wheels — in these simple sets, they are grooves rather than raised rails. Notice that there are multiple pathways through the crossover piece. It would be possible for a train to run continuously around one of the circles, but in practice the tendency to follow a straight path means that it always transfers through the crossover piece to the other circle.

It's clear from the diagram that when a train runs on this layout, it runs along each section of track in a consistent direction. If it runs clockwise in the left circle, then it runs anticlockwise in the right circle, and this never changes. Consequently, if two trains run simultaneously on the track, both of them in the correct consistent direction, there can never be a head-on collision. For example, if one train starts clockwise in the left circle, and the other train starts anticlockwise in the right circle, they can never move in opposite directions within the same circle. They might side-swipe each other by entering the crossover section with bad timing, or a faster train might rear-end a slower train, but we will ignore these possibilities and focus on the absence of head-on collisions as the safety property that we want to guarantee.

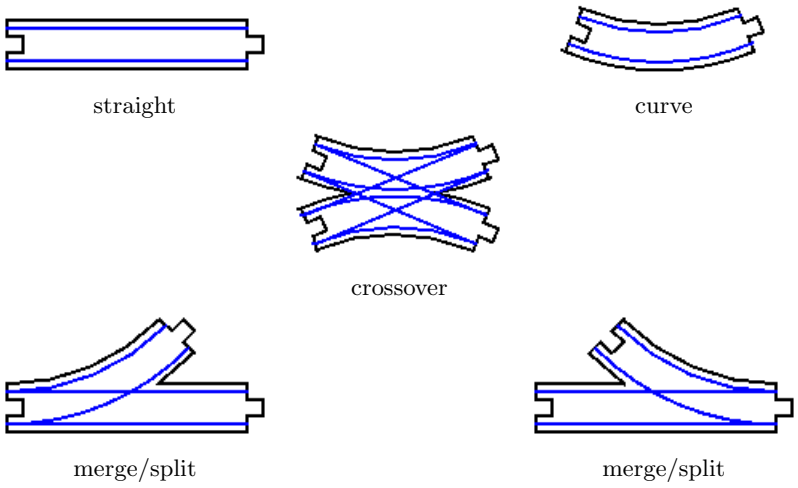

Fig. 7. Basic track pieces.

Figure 7 shows a collection of basic track pieces. They can be rotated and reflected (the pieces are double-sided, with grooves on the top and bottom), which equivalently means that the merge/split pieces (bottom row) can be used with inverted connectors. When a merge/split piece is used as a split (i.e. a train enters at the single endpoint and can take either the straight or curved branch), there is a lever that can be set to determine the choice of branch. We

will ignore this feature, because we are interested in the safety of layouts under the assumption that any physically possible route can be taken.

The pieces in Figure 7 can be used to construct the figure eight layout (Figure 6) as well as more elaborate layouts such as the one in Figure 8. It is easy to see that the layout of Figure 8 has the same "no head-on collisions" property as the figure eight layout.

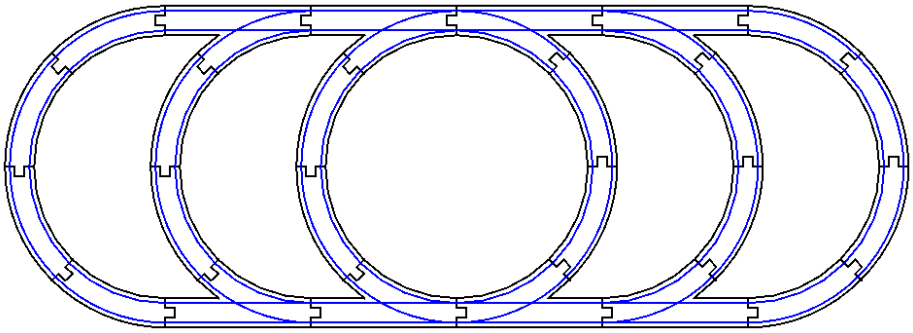

Fig. 8. A layout with multiple paths.

Each track piece has a number of endpoints, where it can be connected to other pieces. We will refer to each endpoint as either positive (the protruding connector) or negative (the hole). The pieces in Figure 7 have the property that if a train enters from a negative endpoint, it must leave from a positive endpoint. This property is preserved inductively when track pieces are joined together, and also when a closed (no unconnected endpoints) layout is formed. This inductively-preserved invariant is the essence of reasoning with a type system, if we consider the type of a track piece or layout to be the collection of polarities of its endpoints. If we imagine an arrow from negative to positive endpoints in each piece, the whole layout is oriented so that there are never two arrowheads pointing towards each other. This is exactly the "no head-on collisions" property. It is possible to use the same argument in the opposite direction, with trains running from positive to negative endpoints, to safely orient the layout in the opposite sense.

This argument could be formalised by defining a syntax for track layouts in the language of traced monoidal categories [9, 11] or compact closed categories [6, 10] and associating a directed graph with every track piece and layout.

The track pieces in Figure 7 are not the only ones. Figure 9 shows the Y pieces, which violate the property that trains run consistently from negative to positive endpoints or *vice versa*. They can be used to construct layouts in which head-on collisions are possible. In the layout in Figure 10, a train can

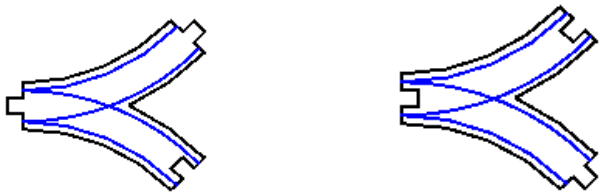

Fig. 9. The Y pieces.

run in either direction around either loop, and independently of that choice, it traverses the central straight section in both directions.

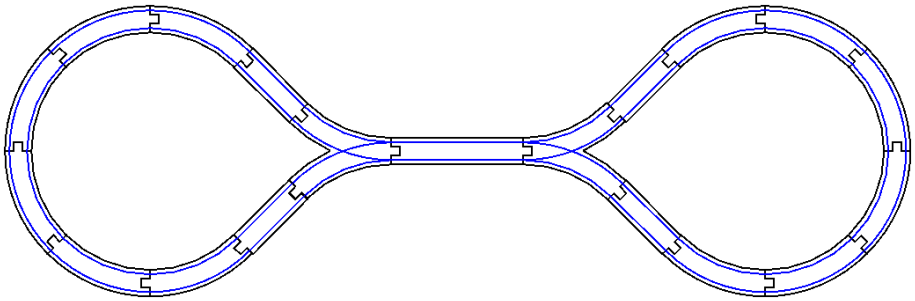

Fig. 10. An unsafe layout using Y pieces.

It is possible to build safe layouts that contain Y pieces. Joining two Y pieces as in Figure 11 gives a structure that is similar to the crossover piece (Figure 7) except that the polarities of the endpoints are different. This "Y crossover" can be used as the basis for a safe figure eight (Figure 12). However, safety of this layout cannot be proved by using the type system. If a train runs clockwise in the circle on the right, following the direction from negative to positive endpoints, then its anticlockwise journey around the circle on the left goes against the polarities. To prove safety of this layout, we can introduce the concept of logical polarities, which can be different from the physical polarities. In the circle on the left, assign logical polarities so that the protruding connectors are negative and the holes are positive, and then the original proof works.

A more exotic layout is shown in Figure 13. This layout is safe for one direction of travel (anticlockwise around the perimeter and the upper right loop) but unsafe in the other direction. More precisely, if a train starts moving clockwise around the perimeter, there is a path that takes it away from the perimeter and

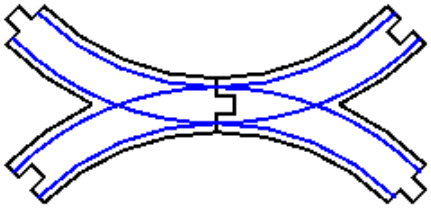

Fig. 11. Joining Y pieces to form a crossover.

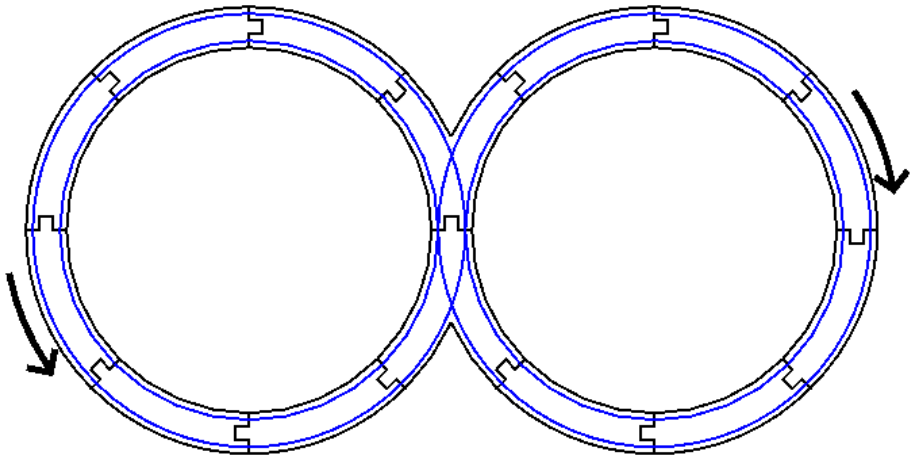

Fig. 12. A safe figure eight using Y pieces. In the circle on the right, the direction of travel follows the physical polarity, but in the circle on the left, the direction of travel is against the physical polarity. To prove safety, assign logical polarities in the circle on the left, which are opposite to the physical polarities.

then back to the perimeter but moving anticlockwise, so that it could collide with another clockwise train.

Safety of the anticlockwise direction cannot be proved by physical polarities, because of the Y pieces. Figure 14 shows that it cannot be proved even by using logical polarities. This is because the section with dashed lines, where the arrows diverge, would require a connection between two logically negative endpoints. To prove safety we can observe that for the safe direction of travel, the section with dashed lines is unreachable. Therefore we can remove it (Figure 15) to give an equivalent layout in which safety can be proved by logical polarities. In fact the layout of Figure 15 is safe in both directions.

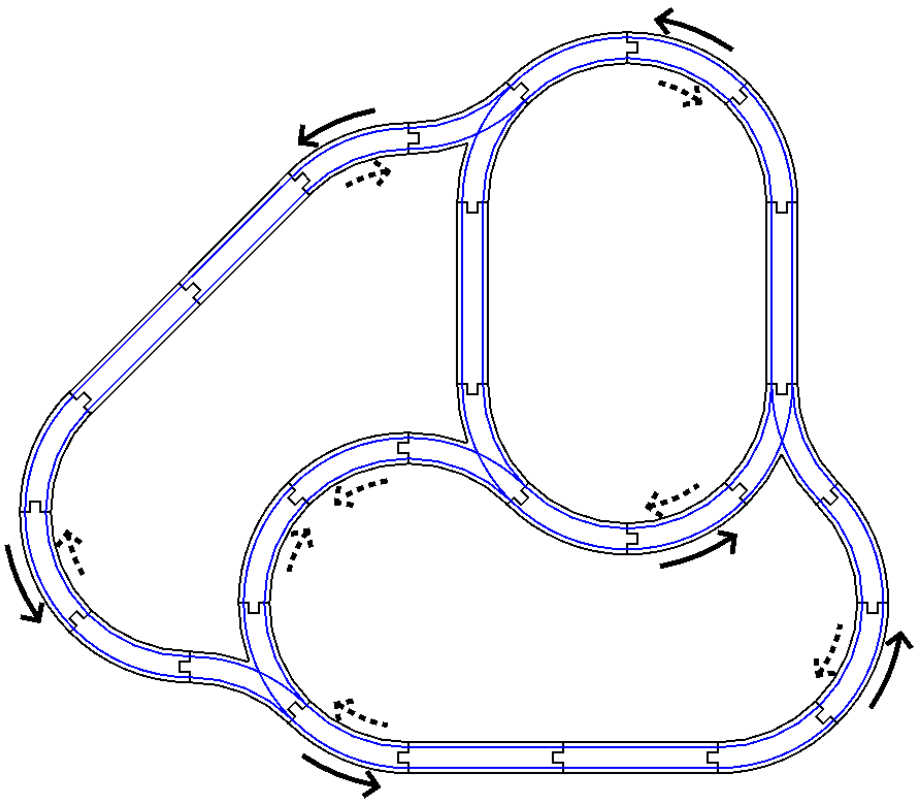

Fig. 13. A layout using Y pieces that is safe in one direction (solid arrows) but not the other (dashed arrows).

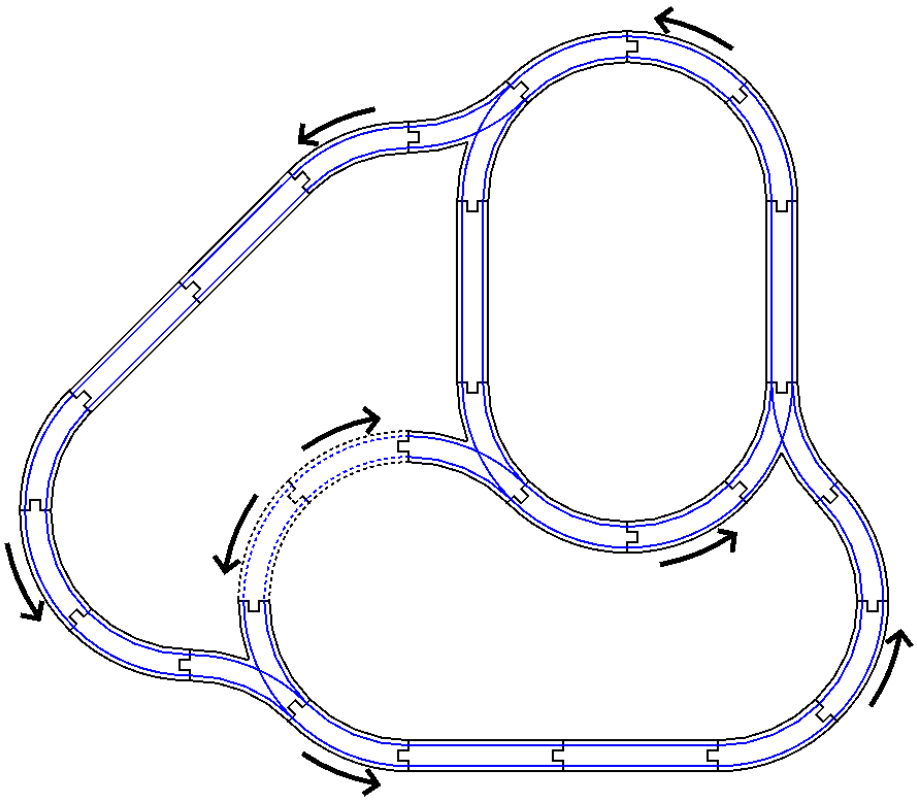

Fig. 14. This layout is safe for travel in the direction of the arrows, because the dashed section of track is unreachable. However, the divergent arrows in the dashed section mean that logical polarities cannot be used to prove safety.

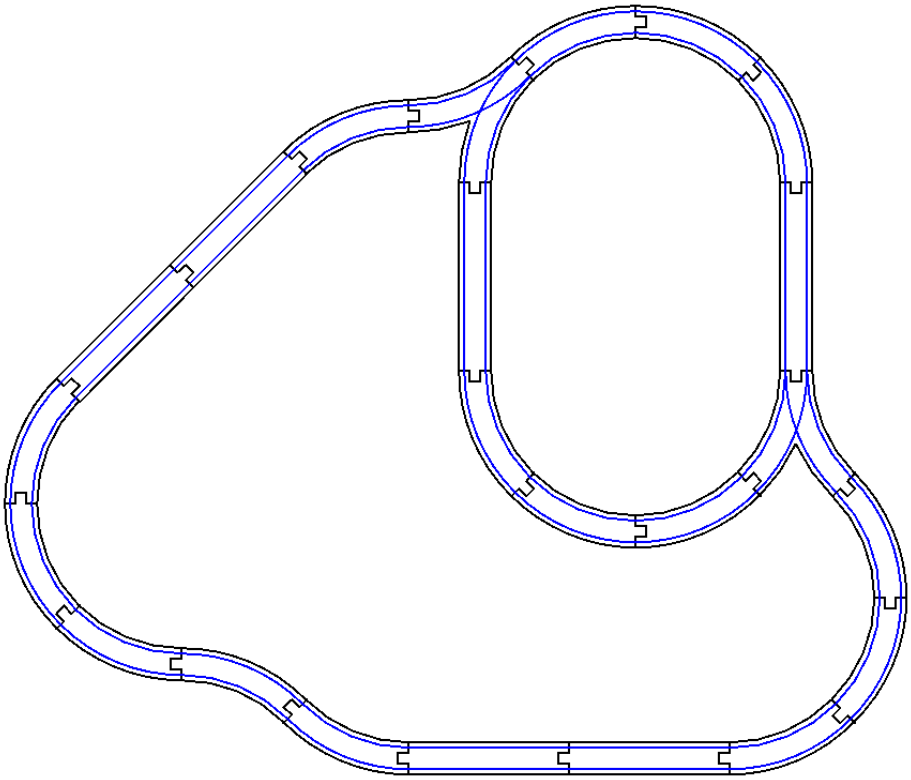

Fig. 15. The layout of Figure 13 with the problematic section of track removed. This layout is safe in both directions. For clockwise travel around the perimeter, following the physical polarities, logical polarities are assigned to the inner loop.

4 Conclusion

I have illustrated the ideas of static type systems in two non-programming domains: serial cables, and model railways. The examples demonstrate the following concepts.

- Typing rules impose local constraints on how components can be connected.
- Following the local typing rules guarantees a global semantic property.
- Typability is an approximation of semantic safety, and there are semantically safe systems whose safety can only be proved by reasoning outside the type system.
- If a type system doesn't type all of the configurations that we know to be safe, then a refined type system can be introduced in order to type more configurations (this is the step from physical to logical polarities in the railway example).

As far as I know, the use of a non-programming scenario to illustrate these concepts is new, or at least unusual, although I have not systematically searched for other examples.

There are several possible directions for future work. One is to increase the level of formality in the analysis of railway layouts, so that the absence of head-on collisions can be stated precisely as a theorem, and proved. Another is to elaborate on the step from physical to logical polarities, again in the railway scenario. Finally, it would be interesting to develop teaching and activity materials based on either or both examples, at a similar level to CS Unplugged.

Acknowledgements

I am grateful to Ornela Dardha, Conor McBride and Phil Wadler for comments on this paper and the seminar on which it is based; to João Seco for telling me about the *Alligator Eggs* presentation of untyped λ-calculus; and to an anonymous reviewer for noticing a small error.

References

1. Brio. `www.brio.uk`.
2. CS Unplugged. `csunplugged.org`.
3. Hornby. `www.hornby.com`.
4. Theoretical Computer Science. `www.journals.elsevier.com/theoretical-computer-science`.
5. Thomas & Friends. `www.thomasandfriends.com`.
6. S. Abramsky, S. J. Gay, and R. Nagarajan. Interaction categories and the foundations of typed concurrent programming. In Manfred Broy, editor, *Proceedings of the NATO Advanced Study Institute on Deductive Program Design*, pages 35–113, 1996.
7. W. Awdrey. *Thomas the Tank Engine*. Edmund Ward Ltd., 1946.
8. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

9. A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447468, 1996.
10. G. M. Kelly and M. L. Laplaza. Coherence for compact closed categories. *Journal of Pure and Applied Algebra*, 19:193–213, 1980.
11. G. Ştefănescu. *Network Algebra*. Springer, 2000.
12. Bret Victor. Alligator eggs. `worrydream.com/AlligatorEggs`.