

Towards a system for directional types for Ruby

Alex Ferguson

University College Cork, National University of Ireland *

Abstract

A modified type system for the Ruby VLSI design language is described, which adds directional information to the types of Ruby relations. Reasons for wishing to do so are discussed, and a realisation of the refined typing scheme is outlined. In order to deal with one otherwise troublesome case, constraints are added to the type system to maintain principal types in the presence of direction information.

1 Introduction

It might be said that inside Ruby was a functional language, trying to get out — but in the syntax of the language, and its type system, it shows no imminent signs of escape. This paper takes a step towards expressing this knowledge in a refinement of the Ruby type system.

Is it really worthwhile to capture this directional information explicitly? The author believes it is, for a number of reasons, as it:

- is a step towards concrete layout of circuits.
- helps the compiler translate relations into their characteristic, underlying functions, when simulating, compiling, or co-synthesising.
- gives the user a better handle on operational behaviour.
- enables the user to add (and have verified) ‘direction declarations’.

This does not settle the question of *how* directional information should best be added or obtained. For example, rather than using a type system, why not perform, say, abstract evaluation of directions? See for example causality analysis, the approach of the DTU Ruby group, which is such an interpretation [7], as is the approach suggested in the first Ruby paper [9]. Although abstract interpretation is the more notoriously intractable of the two, the author is aware of no applicable estimates of the relative computational expense of these approaches. The benefit it is hoped to achieve by this method is that types are more ‘transparent’ to the programmer than a supposedly purely internal abstract analysis, for both reading and writing.

2 A first attempt

One approach which immediately suggests itself is to write directions as ‘mode’-style annotations on types, in the style of say, Ada subprogram parameters [1], or logic programming modes [4]. This would give, for example for the (left-to-right) inverter circuit (shown in Figure 1), the following Ruby type:

inv : in *Bit* ~ out *Bit*

*email: abf@cs.ucc.ie

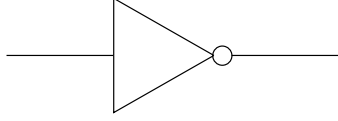


Figure 1: An inverter circuit

Now consider, however, the type rule for serial composition, $(;)$, illustrated in Figure 2:

$$\frac{R : A \sim B \quad S : B \sim C}{R; S : A \sim C}$$

and take the simple Ruby expression $inv; inv$. Treating modes as if they were simply type elements, this is not well-typed with respect to the directions, as the ‘out’ on the range of the first ‘ inv ’ fails to match the ‘in’ of the domain of the second. A rule for composition of such moded types would have to *reverse* the sense of one of the intermediate directions, as further indicated in Figure 2.

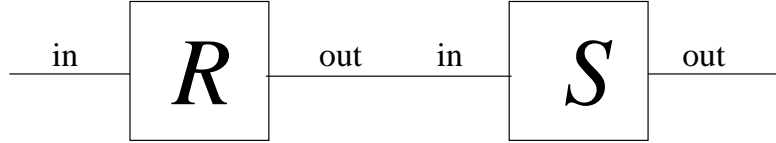


Figure 2: Serial composition $R; S$, with direction annotations

Therefore, in order for such a scheme to work, we would have to use a modified rule for directional $;$ -composition, such as:

$$\frac{R : (d)A \sim (d')B \quad S : (d'')B \sim (d''')C}{R; S : (d)A \sim (d''')C} d' = op \ d''$$

where $op \ in = out$, $op \ out = in$. For simplicity, this rule assumes that each side of a relation has a single, top-level direction on each side of either relation (which would be a quite impractical restriction), and already the rule is somewhat inelegant.

There is then the further problem of circuits which are genuinely bidirectional, all essentially elaborations of id , which is in physical terms a wire, bus, or such connector: see Figure 3(a). (A significant exception to this is *fork*, which it becomes necessary to treat differently; this is addressed in Section 4.) Although some other relations, such as inv , are semantically equivalent to their inverses, they are nevertheless treated as distinct due to the need to map them on to distinct layouts.

We could give a directed type to a left-to-right restriction of id , corresponding to that in Figure 3(b):

$$id_l : in \ Bit \sim out \ Bit$$

or similarly, a right to left one:

$$id_r : out \ Bit \sim in \ Bit$$

requiring that a ‘polymorphic’ id would have to be of a form such as:

$$id : (d) \ Bit \sim (op \ d) \ Bit$$

where op is now playing the role of a ‘type constructor’. This is syntactically clumsy, at best. Also, it is not *a priori* clear that this should be the canonical form of the principal type, as opposed to:

$$id : (op \ d) \ Bit \sim (d) \ Bit$$

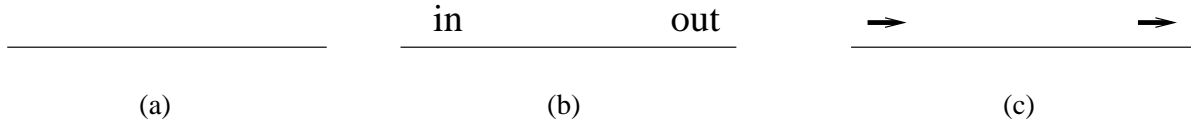


Figure 3: id with: a) no directions, b) ‘modes’, and c) senses

without some further restriction such as a syntactic left-to-right clause in the rule for type variable introduction. An approach of this sort would also complicate and make less intuitive arranging for directed types to be subtypes of undirected types, due to this feature that the ‘same’ type has different directions in different occurrences.

3 A proposed solution

To address the above difficulties, it is instead proposed to describe both inputs and outputs by their directional ‘sense’: that is, either (lexically) left-to-right, or right-to-left. We will write these as respectively \rightarrow and \leftarrow . (Note that Sheeran [9] also suggested such directions, though in the context of an abstract interpretation.)

As ultimately all circuits can be expressed in terms of bits, it is necessary only to augment all occurrences of the type *Bit* with some direction, *d*:

(*d*)*Bit*

The directed type for *inv* now becomes:

$$\text{inv} : (\rightarrow)\text{Bit} \sim (\rightarrow)\text{Bit}$$

while a two-input *and* gate (Figure 4) would be typed:

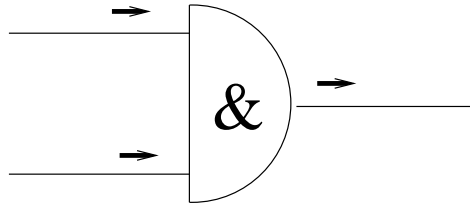
$$\text{and} : \langle (\rightarrow)\text{Bit}, (\rightarrow)\text{Bit} \rangle \sim (\rightarrow)\text{Bit}$$


Figure 4: and gate, with directional annotations

However, for convenience these directional annotations will be allowed at any point in a type structure, and we will consider these to be synonyms for the canonical type where only simple types are annotated. We do this by distributing annotations over all type constructors; e.g., we will write $(d)\langle A, B \rangle$ synonymously with $\langle (d)A, (d)B \rangle$, allowing us to abbreviate the above as:

$$\text{and} : (\rightarrow)\langle \text{Bit}, \text{Bit} \rangle \sim (\rightarrow)\text{Bit}$$

This scheme has the important benefit that these ‘senses’ compose in the same manner as conventional types. Thus, *inv*; *inv* typechecks, complete with directional refinements, using the usual rule for ‘;’, simply treating the directions and variables ranging over directions as if they were types to be unified.

Identity over bits could now be given either the type:

$$\text{id} : (\rightarrow)\text{Bit} \sim (\rightarrow)\text{Bit}$$

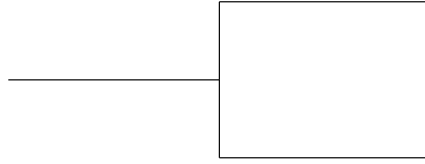


Figure 5: The fork circuit

as per Figure 3(c), or:

$$id : (\leftarrow)Bit \sim (\leftarrow)Bit$$

suggesting the obvious bidirectional, or ‘polymorphic’ generalisation:

$$id : (d)Bit \sim (d)Bit$$

for any direction d . This more clearly follows the normal pattern of polymorphic types, applied to directions.

In order to do this, we must introduce a class of ‘polymorphic direction variables’, δ_i , which range not over types, but over directions, of which there are evidently only two ‘monomorphic’ instances. Following the Hindley-Milner scheme [5], these are quantified (over this range) at the outermost level of the type, thus for example, we can give id the type:

$$id : \forall \delta \in \{\leftarrow, \rightarrow\} . (\delta)Bit \sim (\delta)Bit$$

Because of this restriction as to where direction quantifiers may appear, types could sensibly be written without them, regarding the quantification as being implicit over variables free in a type expression, as is usual in the Hindley-Milner system.

These variables may then be instantiated in a similar way to conventional polymorphic type unification:

$$\frac{e : \forall \delta \in \{\leftarrow, \rightarrow\} . T}{e : T[d/\delta]} DSPEC$$

where d is some direction. Otherwise, the usual scheme is followed.

Note that there is no corresponding ‘*DGEN*’ rule: generalisation may not (and need not) be carried out as a separate step, but instead such quantifiers are only introduced at id (and at any similarly-typed constructs added to the language).

4 The fork problem

A limitation of the scheme described so far is that it cannot fully describe relations such as ‘*fork*’ (Figure 5).

$$\begin{aligned} fork & : \alpha \sim \langle \alpha, \alpha \rangle \\ x \text{ fork } (x, x), & \text{ s.t. } x \in D_\alpha \end{aligned}$$

This can be given three different ‘monodirectional’ types, as follows:

$$fork : \forall \alpha . (\rightarrow)\alpha \sim \langle (\rightarrow)\alpha, (\rightarrow)\alpha \rangle$$

$$fork : \forall \alpha . (\leftarrow)\alpha \sim \langle (\rightarrow)\alpha, (\leftarrow)\alpha \rangle$$

$$fork : \forall \alpha . (\leftarrow)\alpha \sim \langle (\leftarrow)\alpha, (\rightarrow)\alpha \rangle$$

but no single polydirectional scheme can be written which generalises all of them. In order to express this type, it is necessary to extend the language of types to capture the idea that a function can have three (or in general, more) arguments, any one (and only one) of which may be an input, with the remainder being required to be outputs.

Whether it is necessary or desirable to be able to find such a principal type is arguable, and other approaches might be considered. Firstly, one might simply require that all uses of *fork* be suitably annotated, say by an index, specifying which monodirectional instance is required at a given occurrence. Alternatively, one might try to infer which instance is needed from the context of the usage, and where this is not possible due to an insufficiently tight typing restriction, either report a type error, or require that an instance be specified by the user, as in the first observation. (Compare this with the *monomorphism restriction* on overloaded contexts in Haskell [8].)

For the purposes of this paper, however, it will be assumed that the principal type property is required, and a solution will be outlined. The approach that will be taken is to add a small language of constraints to types, in a manner similar to schemes more usually used for subtyping, such as the work of Fuh and Mishra [2], and of Mitchell [6]. Constrained types will be written in the form $\forall\alpha_1 \dots \forall\alpha_n \mid C_1, \dots, C_k \cdot T$, where $C = \{C_1, \dots, C_k\}$ is a set of constraints on the variables $\alpha_1 \dots \alpha_n$. Furthermore, we will consider a term to be well-typed only when it has a type of this form, such that at least one solution of the given constraints exists. A constrained type is considered to denote the same set of terms as that denoted by all monotypes T' such that $T' = \sigma T$, for some substitution σ satisfying σC . The typing rules will require to be modified accordingly, in large part simply to collect and propagate the constraints, except as discussed in the following.

We can introduce a constrained polymorphic type for *fork*, as follows:

$$\frac{}{\mathit{fork} : \forall\alpha . \forall\delta_1 \forall\delta_2 \forall\delta_3 \mid \mathit{Fork}(\delta_1, \delta_2, \delta_3) . (\delta_1)\alpha \sim \langle (\delta_2)\alpha, (\delta_3)\alpha \rangle} \mathit{FORK}$$

where the δ_i are a family of direction variables, and *Fork* is a type constraint requiring the given variables to match those possible in any single instance of *fork*, and is defined as the 3-ary relation:

$$\mathit{Fork} = \{(\rightarrow, \rightarrow, \rightarrow), (\leftarrow, \rightarrow, \leftarrow), (\leftarrow, \leftarrow, \rightarrow)\}$$

Because it is feasible for a fork to contain portions which have different directions, it is sensible to generalise this beyond a single constraint on the direction variables of each. In order to do this, the *Fork* constraint can be extended over the whole of a (directed) type, allowing directions in different subcomponents to be instantiated separately. Thus the type for *fork* becomes:

$$\frac{}{\mathit{fork} : \forall\alpha_1 \forall\alpha_2 \forall\alpha_3 \mid \mathit{Fork}(\alpha_1, \alpha_2, \alpha_3) . \alpha_1 \sim \langle \alpha_2, \alpha_3 \rangle} \mathit{FORK}$$

and correspondingly, we extend the constraint *Fork* over types as follows:

$$\mathit{Fork}((\delta_1)\alpha, (\delta_2)\alpha, (\delta_3)\alpha) = \mathit{Fork}(\delta_1, \delta_2, \delta_3)$$

where α is any non-directed type; this reduces to the earlier case, on direction variables only; and:

$$\mathit{Fork}((a, b), (c, d), (e, f)) = \mathit{Fork}(a, c, e) \wedge \mathit{Fork}(b, d, f)$$

reducing a constraint on structured values to separate constraints on their components, at a simpler type. Cases for any other structured types may be defined similarly. Note that in all cases, the undirected portion of the type is always instantiated to the same type in each limb; applying the constraint to the whole type is simply a device to facilitate access to direction variables nested within.

This scheme can be adapted to cover the (very few) other cases which exhibit this difficulty. These are: variations of *fork* with permutations of the arguments, which can be dealt with by a similar permutation of the type constraint, and/or the use of the $()^{-1}$ combinator; the ‘end-around identity’ relation, which relates two arguments presented on a single side of the relation (physically, a U-shaped wire or bus), and can be

dealt with by the fork rule and a dummy wire on the other side of the circuit; and *forks* with higher degrees of ‘fan-out’, which can be expressed as combinations of the atomic, two-way *fork*.

For example, a 1:3 fork (one wire on the left, three on the right) can be defined by:

$$\mathit{fork}3 = \mathit{fork}; [\mathit{id}, \mathit{fork}]$$

This will be given the type:

$$\forall \alpha . \forall \delta_1 \forall \delta_2 \forall \delta_3 \forall \delta_4 \forall \delta_0 \mid \mathit{Fork}(\delta_1, \delta_2, \delta_0), \mathit{Fork}(\delta_0, \delta_3, \delta_4) . (\delta_1)\alpha \sim \langle (\delta_2)\alpha, \langle (\delta_3)\alpha, (\delta_4)\alpha \rangle \rangle$$

Note the ‘extra’ direction variable, δ_0 , which appears in the quantifiers and the constraints, but not in the body of the type. This corresponds to the intermediary wire between the right-hand-side of the first *fork*, and the left-hand-side of the second. Instantiating any of the outermost variables of the circuit will immediately allow simplification of the constraints: for example, if we choose δ_3 to be \leftarrow , the type will become, after substitution, and a call to a suitable simple constraint solver:

$$\forall \alpha . \forall \delta_1 \forall \delta_2 \forall \delta_4 \mid \mathit{Fork}(\delta_1, \delta_2, \delta_4) . (\delta_1)\alpha \sim \langle (\delta_2)\alpha, \langle (\leftarrow)\alpha, (\delta_4)\alpha \rangle \rangle$$

5 Conclusions

5.1 The use of polymorphism, and constraints

It might reasonably be thought that to employ the mechanism of polymorphism over a two-valued class was somewhat excessive. Alternative approaches would be to use some sort of class-based or subtyping system, or simple overloading. However, as existing Ruby type systems do not utilise such mechanisms, adding these for this specific purpose would likely be just as much additional machinery. It seems safe to assume that in any case, the underlying type-checking (or inference) mechanism need not be as involved as that for full polymorphism.

Similarly, it may seem unfortunate to have to introduce a constraint-based system simply to type the single operation *fork*, but this appears necessary to obtain a most general typing, and is reasonable in the sense that *fork* is the primitive which captures the essence of this sort of multi-directionality.

5.2 Relationship to layout considerations

Ruby expressions may be interpreted as either two-faced or four-faced ‘tiles’, with in the latter case each side of a relation being split across two faces of its tile. It might therefore be thought that it would be desirable to factor this into the directions of the types calculated for such circuits, or to express such considerations as further refinements of such a type system, say, by adding \uparrow and \downarrow as additional types (or subtypes) in the direction scheme. We believe that this turns out to not be the case, and the two-directional system outlined above captures equational properties of the Ruby program which are essentially orthogonal to the question of physical layout. Witness that a parameter bearing a \rightarrow -directed type could be laid out in such a way that it falls on *any* of the four sides of a tile, depending in whether it is an input or an output, and on which scheme for physical layout is chosen.

In fact, this ‘sidedness’ is purely a heuristic matter, as any given circuit can be laid out in either the two-faced or four-faced style. Accordingly, it could be argued that it would be inappropriate to express this consideration in the type system at all, but rather that it would be better to deal with it by such methods as ‘pragmas’ in the program text, or by analysis of the circuit to determine a suitable layout.

5.3 Directions and relations

A possible objection to the notion of adding directional types to Ruby is that it is after all a *relational* language, and it seems often useful to write relational specifications of programs, and refine them using relational equivalences, frequently introducing (and subsequently eliminating) constructs which are well

defined relations, but which correspond to no implementable circuit. Such specifications may well include equations which are not ‘well-directed’. The author believes that it is reasonable to expect Ruby tools (such as compilers, proof-checkers, etc.) to allow *both* styles of interaction: a type system that allows undirected specifications, as well as programs with types giving fully-specified directions. In the event that polydirectional types were given to circuits other than simple ‘wires’, such as a bidirectional type for an inverter, then a third layer would become appropriate: program specifications could be given types polymorphic in their directional variables, but only circuits which had eliminated such overloading would actually be implementable.

5.4 Future work

Firstly, a formal description of the type extension proposed is yet to be completed. This first requires establishing which version of the (directionless) Ruby type system is to be used as a base. Then whether type inference is (still) possible will be investigated, and indeed whether it is desirable. Next, an implementation would be useful, to assess the benefits of obtaining this information, and as a means of comparison with other directionality schemes. It is envisaged that this scheme be added to the Glasgow Ruby compiler.

6 Acknowledgements

The author would first of all like to thank Satnam Singh, for providing the motivation for this paper in the form of his musings about a suitable type system for the Glasgow Ruby Compiler, and also a number of useful discussions on the subject. David N. Turner and Jan Nicklisch each offered advice and commentary on my proposed type system which were particularly helpful in recognising and attempting to deal with the case of *fork*. The Workshop referees made a number of useful comments, particularly John Hughes who suggested a more general typing technique for the *fork* case.

References

- [1] *Reference Manual for the Ada Programming Language*. United States Department of Defense (ANSI/MIL-STD-1815A), Washington D.C., January 1983.
- [2] You-Chin Fuh, Prateek Mishra. Type Inference with Subtypes. *TCS* 73(2): 155-175, 1990.
- [3] Geraint Jones, Mary Sheeran. Circuit design in Ruby. Lecture notes on Ruby from a summer school in Lyngby, Denmark, September 1990. in *Formal Methods for VLSI Design*, (J. Staunstrup ed.), North Holland, 1990.
- [4] C. S. Mellish. The automatic generation of mode declarations for Prolog programs. DAI Research Paper 163, Department of Artificial Intelligence, University of Edinburgh, 1981.
- [5] A. J. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, Vol. 17, 1978.
- [6] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3): 245-286, 1991.
- [7] Robin Sharp. T-Ruby, A Tool for Handling Ruby Expressions, 2nd. edition. Technical Report ID-TR: 1994-154, Department of Computer Science, Technical University of Denmark, December 1994.
- [8] John Peterson, Kevin Hammond (eds). Report on the Programming Language Haskell. Yale University Research Report YALEU / DCS / RR-1106.
- [9] Mary Sheeran. Describing and reasoning about circuits using relations, *Theoretical foundations of VLSI design* (McEvoy and Tucker eds). Cambridge Tracts in Theoretical Computer Science 10, 1990.