WIKIPEDIA

Arrow (computer science)

In computer science, **arrows** or **bolts** are a type class used in programming to describe computations in a pure and declarative fashion. First proposed by computer scientist John Hughes as a generalization of monads, arrows provide a referentially transparent way of expressing relationships between *logical* steps in a computation.[1] Unlike monads, arrows don't limit steps to having one and only one input. As a result, they have found use in functional reactive programming, point-free programming, and parsers among other applications.[1][2]

Contents

Motivation and history

While arrows were in use before being recognized as a distinct class, it wasn't until 2000 that John Hughes first published research focusing on them. Until then, monads had proven sufficient for most problems requiring the combination of program logic in pure code. However, some useful libraries, such as the Fudgets library for graphical user interfaces and certain efficient parsers, defied rewriting in a monadic form.[1]

The formal concept of arrows was developed to explain these exceptions to monadic code, and in the process, monads themselves turned out to be a subset of arrows.[1] Since then, arrows have been an active area of research. Their underlying laws and operations have been refined several times, with recent formulations such as arrow calculus requiring only five laws.[3]

Relation to category theory

In category theory, the Kleisli categories of all monads form a proper subset of Hughes arrows.[1] While Freyd categories were believed to be equivalent to arrows for a time,[4] it has since been proven that arrows are even more general. In fact, arrows are not merely equivalent, but directly equal to enriched Freyd categories.[5]

Definition

Like all type classes, arrows can be thought of as a set of qualities that can be applied to any data type. In the Haskell programming language, arrows allow functions (represented in Haskell by `->` symbol) to combine in a reified form. However, the actual term "arrow" may also come from the fact that some (but not all) arrows correspond to the morphisms (also known as "arrows" in category theory) of different Kleisli categories. As a relatively new concept, there is not a single, standard definition, but all formulations are logically equivalent, feature some required methods, and strictly obey certain mathematical laws.[6]

Functions

The description currently used by the Haskell standard libraries *requires* only two basic operations:

- A type constructor `arr` that takes functions `->` from any type `s` to another `t`, and lifts those functions into an arrow `A` between the two types.[7]

```
arr (s -> t)        ->   A s t
```

- A piping method `first` that takes an arrow between two types and converts it into an arrow between tuples. The first elements in the tuples represent the portion of the input and output that is altered, while the second elements are a third type `u` describing an unaltered portion that bypasses the computation.[7]

```
first (A s t)       ->   A (s,u) (t,u)
```

Although only these two procedures are strictly necessary to define an arrow, other methods can be derived to make arrows easier to work with in practice and theory. As all arrows are categories, they can inherit a third operation from the class of categories:

- A composition operator >>> that can attach a second arrow to a first as long as the first function's output and the second's input have matching types.[7]

```
A s t  >>>  A t u   ->   A s u
```

One more helpful method can be derived from a combination of the previous three:

- A merging operator `***` that can take two arrows, possibly with different input and output types, and fuse them into one arrow between two compound types. Note that the merge operator is *not* necessarily commutative.[7]

```
A s t  ***  A u v   ->   A (s,u) (t,v)
```

Arrow laws

In addition to having some well-defined procedures, arrows must obey certain rules for any types they may be applied to:

- Arrows must always preserve all types' identities `id` (essentially the definitions of all values for all types within a category).[7]

```
arr id              ==   id
```

- When connecting two functions `f` & `g`, the required arrow operations must distribute over compositions from the left.[7]

```
arr (f >>> g)       ==   arr f  >>>  arr g
first (f >>> g)     ==   first f  >>>  first g
```

- In the previous laws, piping can be applied directly to functions because order must be irrelevant when piping & lifting occur together.[7]

```
arr (first f)       ==   first (arr f)
```

The remaining laws restrict how the piping method behaves when the order of a composition is reversed, also allowing for simplifying expressions:

- If an identity is merged with a second function to form an arrow, attaching it to a piped function must be commutative.[7]

```
arr (id *** g)  >>>  first f       ==   first f  >>>  arr (id *** g)
```

- Piping a function before type simplification must be equivalent to simplifying type before connecting to the unpiped function.[7]

```
first f  >>>  arr ((s,t) -> s)     ==   arr ((s,t) -> s)  >>>  f
```

- Finally, piping a function twice before reassociating the resulting tuple, which is nested, should be the same as reassociating the nested tuple before attaching a single bypass of the function. In other words, stacked bypasses can be flattened by first bundling together those elements unchanged by the function.[7]

```
first (first f)  >>>  arr ( ((s,t),u) -> (s,(t,u)) )   ==
  arr ( ((s,t),u) -> (s,(t,u)) )  >>>  first f
```

Applications

Arrows may be extended to fit specific situations by defining additional operations and restrictions. Commonly used versions include arrows with choice, which allow a computation to make conditional decisions, and arrows with feedback, which allow a step to take its own outputs as inputs. Another set of arrows, known as arrows with application, are rarely used in practice because they are actually equivalent to monads.[6]

Utility

Arrows have several benefits, mostly stemming from their ability to make program logic explicit yet concise. Besides avoiding side effects, purely functional programming creates more opportunities for static code analysis. This in turn can theoretically lead to better compiler optimizations, easier debugging, and features like syntax sugar.[6]

Although no program strictly requires arrows, they generalize away much of the dense function passing that pure, declarative code would otherwise require. They can also encourage code reuse by giving common linkages between program steps their own class definitions. The ability to apply to types generically also contributes to reusability and keeps interfaces simple.[6]

Arrows do have some disadvantages, including the initial effort of defining an arrow that satisfies the arrow laws. Because monads are usually easier to implement, and the extra features of arrows may be unnecessary, it is often preferable to use a monad.[6] Another issue, which applies to many functional programming constructs, is efficiently compiling code with arrows into the imperative style used by computer instruction sets.

Limitations

Due to the requirement of having to define an `arr` function to lift pure functions, the applicability of arrows is limited. For example, bidirectional transformations cannot be arrows, because one would need to provide not only a pure function, but also its inverse, when using `arr`[8]. This also limits the use of arrows to describe push-based reactive frameworks that stop unnecessary propagation. Similarly, the use of pairs to tuple values together results in a difficult coding style that requires additional combinators to re-group values, and raises fundamental questions about the equivalence of arrows grouped in different ways. These limitations remain an open problem, and extensions such as Generalized Arrows[8] and N-ary FRP[9] explore these problems.

References

1. Hughes, John (May 2000). "Generalising Monads to Arrows". *Science of Computer Programming*. **37** (1–3): 67–111. doi:10.1016/S0167-6423(99)00023-4 (https://doi.org/10.1016%2FS0167-6423%2899%2900023-4). ISSN 0167-6423 (https://www.worldcat.org/issn/0167-6423).
2. Paterson, Ross (27 March 2003). "Chapter 10: Arrows and Computation" (http://www.soi.city.ac.uk/~ross/papers/fop.ps.gz) (PS.GZ). In Gibbons, Jeremy; de Moor, Oege (eds.). *The Fun of Programming*. Palgrave Macmillan. pp. 201–222. ISBN 978-1403907721. Retrieved 10 June 2012.
3. Lindley, Sam; Wadler, Philip; Yallop, Jeremy (January 2010). "The Arrow Calculus" (http://homepages.inf.ed.ac.uk/wadler/papers/arrows-jfp/arrows-jfp.pdf) (PDF). *Journal of Functional Programming*. **20** (1): 51–69. doi:10.1017/S095679680999027X (https://doi.org/10.1017%2FS095679680999027X). hdl:1842/3716 (https://hdl.handle.net/1842%2F3716). ISSN 0956-7968 (https://www.worldcat.org/issn/0956-7968). Retrieved 10 June 2012.
4. Jacobs, Bart; Heunen, Chris; Hasuo, Ichiro (2009). "Categorical Semantics for Arrows". *Journal of Functional Programming*. **19** (3–4): 403–438. doi:10.1017/S0956796809007308 (https://doi.org/10.1017%2FS0956796809007308). hdl:2066/75278 (https://hdl.handle.net/2066%2F75278).
5. Atkey, Robert (8 March 2011). "What is a Categorical Model of Arrows?" (https://doi.org/10.1016/j.entcs.2011.02.014). *Electronic Notes in Theoretical Computer Science*. **229** (5): 19–37. doi:10.1016/j.entcs.2011.02.014 (https://doi.org/10.1016%2Fj.entcs.2011.02.014). ISSN 1571-0661 (https://www.worldcat.org/issn/1571-0661).
6. Hughes, John (2005). "Programming with Arrows" (http://www.cse.chalmers.se/~rjmh/afp-arrows.pdf) (PDF). *Advanced Functional Programming*. 5th International Summer School on Advanced Functional Programming. 14–21 August 2004. Tartu, Estonia. Springer. pp. 73–129.

doi:10.1007/11546382_2 (https://doi.org/10.1007%2F11546382_2). ISBN 978-3-540-28540-3. Retrieved 10 June 2012.

7. Paterson, Ross (2002). "Control.Arrow" (https://web.archive.org/web/20060213173453/http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Arrow.html). *base-4.5.0.0: Basic libraries*. haskell.org. Archived from the original (http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Arrow.html) on 13 February 2006. Retrieved 10 June 2012.
8. Joseph, Adam Megacz (2014). "Generalized Arrows" (https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-130.pdf) (PDF). *Technical Report No. UCB/EECS-2014-130*. EECS Department, University of California, Berkeley. Retrieved 20 October 2018.
9. Sculthorpe, Neil (2011). *Towards safe and efficient functional reactive programming* (http://eprints.nottingham.ac.uk/11981/1/thesis.pdf) (PDF) (PhD). University of Nottingham.

External links

- Arrows: A General Interface to Computation (http://www.haskell.org/arrows)
- A New Notation for Arrows (http://www.soi.city.ac.uk/~ross/papers/notation.html), Ross Paterson, in ICFP, Sep 2001.
- Arrow notation (https://web.archive.org/web/20071014174133/http://haskell.org/ghc/docs/latest/html/users_guide/arrow-notation.html) ghc manual