Improv: Live Coding for Robot Motion Design

Alexandra Q. Nilles
Computer Science Department
University of Illinois at Urbana-Champaign
nilles2@illinois.edu

Chase Gladish
Computer Science Department
University of Illinois at Urbana-Champaign
webmaster@marysville-ohio.com

Mattox Beckman
Computer Science Department
University of Illinois at Urbana-Champaign
larst@affiliation.org

Amy LaViers
Mechanical Science and Engineering Department
University of Illinois at Urbana-Champaign

ABSTRACT

This paper introduces the Improv system, a programming language for high-level description of robot motion with immediate visualization of the resulting motion on a physical or simulated robot. The intended users of this tool are anyone looking to quickly generate robot motion, such as educators, artists, and researchers. The system includes a domain-specific language, inspired by choreographic techniques, which allows for several ways of composing and transforming movements such as reversing movements in space and time and changing their relative timing. Instructions in the Improv programming language are then executed with roshask, a Haskell client for ROS ("Robot Operating System"). ROS is an open-source robot software framework which is widely used in academia and industry, and integrated with many commercially available robots. However, the ROS interface can be difficult to learn, especially for people without technical training. This paper presents a "live coding" interface for ROS compatible with any text editor, by executing whenever the user saves changes. Currently, Improv can be used to control any robot compatible with the 'Twist' ROS message type (which sets linear and rotational velocity). This paper presents Improv implementations with the two-dimensional simulator Turtlesim, as well as three-dimensional TurtleBots in the Gazebo simulation engine.

KEYWORDS

robotics, choreography, live coding, ROS, Haskell, roshask, HRI

ACM Reference Format:
Alexandra Q. Nilles, Chase Gladish, Mattox Beckman, and Amy LaViers. 2018. Improv: Live Coding for Robot Motion Design. In *Proceedings of ACM Woodstock conference (MOCO '18).* ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

write introduction

MOCO '18, July 1997, El Paso, Texas USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

2 RELATED WORK

One closely related project is *Improv* is *Dance*, a domain-specific language built in Haskell [6]. The project included a DSL inspired by Labanotation, as well as a reactive layer that allowed the robot to respond to sensor events. The project targeted humanoid robots, while *Improv* has so far targeted mobile robots, and *Dance* would generate the necessary 3D simulation code, though did not include the live-coding interface of this work and predates ROS. *Improv* has incorporated and adapted some of the data structures from *Dance*, namely the `Action` and `Dance` data types.

This work is also influenced by live coding interfaces and programming languages for generating music, often part of the *Algorave* performance movement [3]. In particular, the programming language TidalCycles [8] has had a strong influence on this work, both syntactically and in how relative timing of events is managed.

Al Jazari is a live-coding installation which uses a simple graphical language to allow people to control robots (in simulation) [7]. The language includes conditionals based on external state and communication between the bots. The program state of the robot is also visualized.

Especially when used with the two-dimensional Turtlesim, *Improv* is reminiscent of *Logo* [9], an educational, interpreted dialect of Lisp that is often used in conjunction with a simulation of a two-dimensional turtle. Our programming language has different features than *Logo*, does not support recursion as *Logo* does, and most importantly is integrated with ROS and thus able to be used with three-dimensional simulators and actual robots.

We know of two other projects have addressed the problem of the complex development cycle in ROS by creating tools for interactive or "live" programming. One such project [1] created a DSL in Python which allows for wrapping and modifying existing ROS nodes, using the Python REPL. However, by using the Python REPL, the user is only able to experiment with commands in a shell and is not able to save the commands they have tried in a file. Additionally, since the DSL is implemented as a library in Python, it inherits some of the opaque syntax of the Python ROS client. Improv has a simpler, albeit less powerful, programming language and models movement explicitly. Another closely related work to *Improv* is the Live Robot Programming (LRP) language [2] and its integration with PhaROS [5], a client library for ROS written is Pharo, a dynamic programming language specialized for live updating and hot recompilation. This project allows for live-coding of ROS nodes

and reconfiguration of the ROS network with a much shorter development cycle than traditional ROS programming. However, the aims of these projects and Improv are slightly different - the DSL, while more high-level than most robot programming languages, was not designed around modelling movement itself and instead model state machines that transition on events. Thus, both of these related projects are better suited for applications which involve reactivity and sensing of the environment, while Improv is better suited to applications where the user wishes to quickly generate certain movements and creatively explore movement patterns and transformations. We also have designed the Improv language with accessibility and ease-of-use in mind, especially for inexperienced programmers, and future work will focus on testing and measuring the usability of the system in user studies.

3 ARCHITECTURE OVERVIEW

The *Improv* system consists of the following components:

- a live-coding interface, where the user writes a program in any text editor of their choosing. When they save changes to the file, they observe the resulting movement pattern on a simulated robot
- a shell script which monitors the user's file for changes, at which point it resets the simulator, and executes a compiled Haskell program which interprets the user code and starts a corresponding ROS node
- the ROS node, created with roshask [4], which sends messages to the simulator. Currently, only `Twist` messages can be created, which contain linear and rotational velocity information.
- A simulator (or actual robot), which recieves ROS messages and executes movement on a robot platform. Currently, we have tested the system with:
  - TurtleSim: a two dimensional simulator with limited physics, where velocity commands are nearly exactly executed on an animated turtle.
  - Gazebo with a TurtleBot robot model: a three-dimensional simulator with more realistic physics, where velocity commands control simulated motors.

See Figure 1 for a diagrammatic representation of information flow in the system.

4 EXAMPLES

Figures 2 and 3 show examples of the system in use with two different text editors and two different simulators. Note that the choice of editor and the choice of simulator are decoupled, and *Improv* is absolutely editor-agnostic, relying only on an operating-system level script which monitors a text file for changes to execute the interpretation and simulation process. *Improv* is somewhat simulator-agnostic, although currently it is only possible to control robots which use a certain ROS message type for control (the Twist message, which sets desired linear and rotational velocity).

5 LIVE-CODING INTERFACE

One important design decision for developers of interactive text-based programming tools is whether to tie their tool to a specific editor. For example, the algorave live-coding tool TidalCycles was originally developed for emacs, a powerful editor which has a notoriously steep learning curve. Many people prefer to use simpler editors, so new live-coding plug-ins have been developed for editors such as Atom and Sublime Text. This editor-based approach has advantages, such as a large degree of customizability and extensibility using the features of the editors. However, it also introduces challenges such as maintaining feature parity between editors, as well as the up-front investment needed to interface with new editors. In our case, we are developing a tool that should be usable by

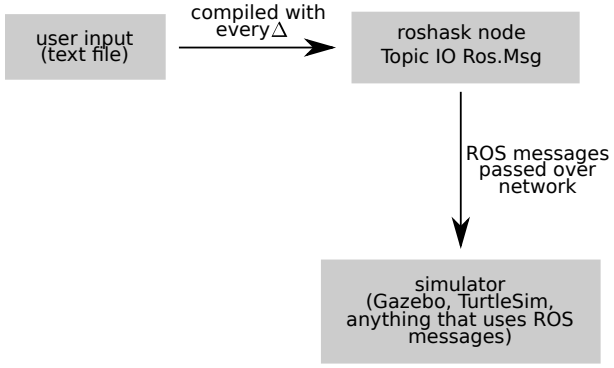

Figure 1: An illustration of how user input, written to a text file, is converted into a ROS node which publishes messages to a simulator. fix this diagram, is old from 598

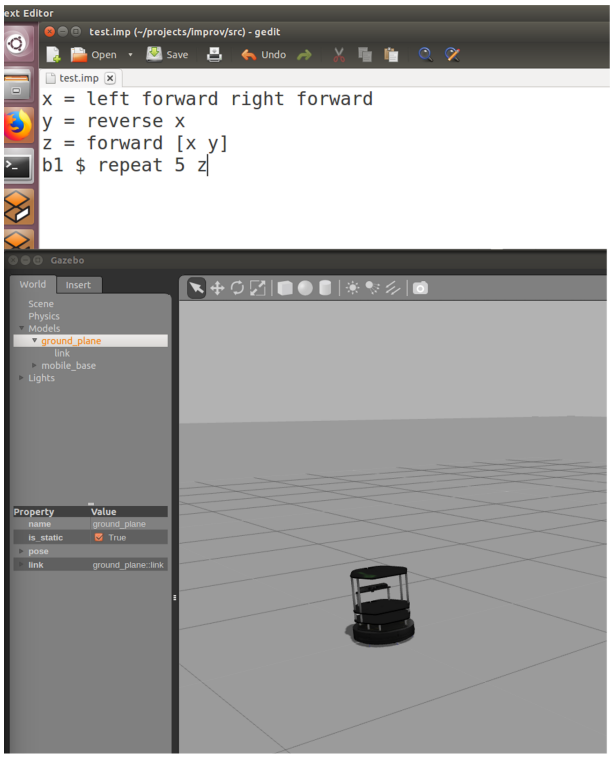

Figure 2: A program written in Gedit, a simple graphical text editor, with Gazebo and a simulated Turtlebot.

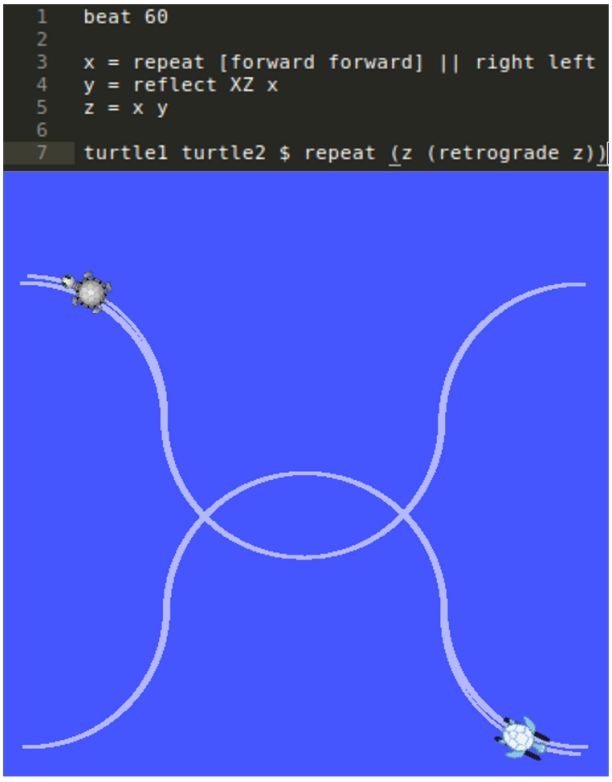

Figure 3: A program written in vim, a terminal-based text editor, with TurtleSim and two simulated turtles.

artists, children, and programming novices, as well as experienced roboticists. Thus, we wish to allow users flexibility to use their editor of preference.

To accomplish this, instead of creating an interface for each desired editor, we use a shell script which monitors the file that the user is editing for changes. Every time the user saves changes to the file, the program detects a change, interprets the user's new program, and resets the simulator and ROS node. This design choice circumvents the need to interface with specific editors. Additionally, many editors have keyboard shortcuts for saving files. Thus, executing programs contributes minimally to the overall workload of using the system, especially when compared to the many steps required to test changes to traditional ROS programs. The delay between saving the file and observing the changes in the simulator is very short - while we have not done a formal timing analysis, the delay is a fraction of a second and not noticeably longer than the time it takes to shift attention from the text editor to the simulator.

6 DOMAIN-SPECIFIC LANGUAGE DESIGN

Many domain-specific languages for robot programming exist; for example, a review in 2016 identified 137 relevant publications

justify making another one... or just take this number out and add other motivation

The base type of the *Improv* language is a movement. Movements are discretized and can be combined with each other in various ways, forming different movements. The precise way in which this is interpreted on a robot platform is defined by the language's translation to Haskell and the commands sent to ROS for the particular robot in question.

```
movement = prim
         | movement movement
         | transformer movement
         | [movement]
         | (movement)
         | movement || movement

exp = rs $ movement
    | var = movement
```

Figure 4: The grammar of Improv programs. `exp` represents top-level expressions, which execute movements on robot(s), or store movements in variables. `movement`s are converted into ROS message streams and can be composed and grouped in multiple ways. run past Mattox, see if he has an example I can emulate

The terminals in *Improv*, such as `forward` or `right`, are mapped to streams of ROS messages. In our implementation so far, we have mapped to the `Twist` ROS message, which contains two three-dimensional vectors representing the robot's linear and angular velocity. Controllers, which often are included with commercial robots, are required to perform the low-level motor control to acheive and maintain the desired velocities.

Here, we have simplified the language by varying only three of these values, the robots x, y–velocity in the plane and its angular velocity in the plane. Many ROS- and Gazebo-integrated robots have available velocity controllers. We have also discretized velocities, though users control velocity only relative to other movements, by specifying their relative timing.

Movements are organized in time into units, where each unit is performed in one "beat." The base timing of beats (units per minute) can be specified by the user, and is only limited by the maximum publishing frequency of ROS and the physical constraints of the robot platform.

Users can specify a series of commands such as *move forward for one beat, turn right for one beat, move forward for one beat* with the command `forward right forward`. Movements separated by white space will occur in different "beats."

The user can also use brackets to compress a sequence of movements into one beat, such as `forward right forward`, which will cause these three movements to happen in the same amount of time as the first movement in the previous example. This syntax and behavior is directly inspired by *TidalCycles*, which has a similar mechanism for grouping sounds. In our implementation, bracketing n movements causes each movement to be performed n times faster, but for $1/n$ times as long, so the movement has the same extent but is performed faster.

Movements can also be performed in parallel, such as `forward || right`, which will cause the robot to curve to the right as it moves forward. In our implementation, velocities in parallel are simply added, so `forward || backward` would result in no

movement. A movement which is two movements in parallel will terminate when the "shorter" movement ends, so the program `(forward right) || forward` will never turn right (parenthesis are used to group movements without changing timing).

We have also implemented several transformations which map a function over a movement. For example, `repeat` takes an integer and a movement as arguments and causes that movement to repeat for the specified number of times.

In space, we have `reflect`, which takes a plane and a movement as arguments and returns the reflected movement (`reflect YZ right` yields `left`, where the `YZ` plane is body-centered and would be the saggital plane on a biological organism). For transforming movements in time, `reverse` is a unary operator which will reverse the order of a series of movements: `reverse (forward right left)` is equivalent to `left right forward`. To reverse the trajectory itself, we use `retrograde`, which uses spatial reflections and reverses time; for example, `retrograde (forward right left)` is equivalent to `right left backward`. Both `retrograde` and `reverse` are their own inverses: applying them twice returns the original movement, as would be expected.

While we have only implemented these combinators for simple and very symmetric mobile robots, one could imagine making more complicated types of symmetry for other robot platforms. We have included a typeclass `Symmetric a` which is defined by a function `refl :: Plane -> a -> a` which is parameterized by a body type `a`. By defining this function once for a new robot platform, detailing all the different symmetries of the body, the functionality of these spatial transformers can be extended to new platforms.

6.1 Multiple Robots

The *Improv* system has the capapbility to control multiple robots at once, using a syntax which mirrors how *TidalCycles* allows for multiple tracks to be composed simultaneously. Each robot is given a unique name in the shell script which launches ROS and the *Improv* system, and this is also where the initial location of each robot is specified. Then, in the user's program, they specify which movement sequence should be associated with each robot. For example, to make robot `r1` move forward and robot `r2` move backward, the user would write

```
r1 $ forward
r2 $ backward
```

This syntax, along with assigning movements to variables, can make it easy to specify relationships between how different robots are moving, such as

```
x = left right [forward right]
r1 $ x
r2 $ retrograde x
```

which would cause robot `r2` to perform the same movement as robot `r1`, but in retrograde. It is also possible to command two robots to do the same movement with a program such as

```
r1 r2 $ forward backward
```

As *Improv* is extended to other platforms in the future, this could be an interesting mechanism for studying how the same high-level choreographic commands are perceived when executed on different platforms with different interpretations of the commands.

7 MODELLING MOVEMENT IN HASKELL

add motivation: choregraphic operations, technologies. source to cite? Labanotation? Really I learned this stuff in Catie's class. Can probably just cite that and arXiv paper?

Programs in the *Improv* DSL are interpreted by a compiled Haskell program into an ADT we call a `Dance`, which can be thought of as a tree that holds all movement primitives (the terminals in the *Improv* language, such as `forward`). The operators of the type encode the parallel/series structure of the user's program. To execute a `Dance` as a series of ROS messages, we must flatten the tree while maintaining this relative timing information, which will be discussed in Section 7.2.

`Dances` are defined as

```
data Dance b = Prim Action Mult b
             | Rest Mult
             | Skip
             | Dance b :+: Dance b
             | Dance b :||: Dance b
```

where `Prim` is a motion primitive type, holding the `Action` (direction and spatial extent of the movement), `Mult` which stores timing information, and `b`, a parameterized type describing the part of the robot to move. `Rest` indicates that the robot part is not moving for some period of time (and is a terminal in the *Improv* language). `Skip` is the identity dance, having no effect on the robot for no time duration, and is necessary for the monoidal structure of the parallel (`:||:`) and series (`:+:`) operators.

7.1 Algebraic Structure of Dances

maybe nix this section - or make less pretentious, more clear benefits

This algebraic structure helps enforce the timing behavior that we expect; namely, associativity. If `d1`, `d2`, and `d3` are all `Dances` (with an arbitrary number and structure of movement primitives in each), then the order that they are sequenced together shouldn't matter:

```
(d1 :+: d2) :+: d3 = d1 :+: (d2 :+: d3)
```

And similarly, if they are all three in parallel, arbitrary groupings should not change the meaning of the program. This is exactly the behavior that the algebraic objects *monoids* have, namely associativity and an identity element. See [11] for a much more detailed discussion on the usefulness of monoids in modelling and programming languages, in the context of *Diagrams*, a Haskell DSL for creating vector graphics.

We can create monoid instances in Haskell for these operators on `Dance` data types, which allow for lists of `Dances` (read in by the parser as all elements between square brackets or separated by the `||` operator) to be combined in sequence or parallel. The functions for doing so, `seqL` and `parL`, have the type `(Parts a) => [Dance a] -> Dance a`: they combine movements using the monoidal operator.

7.2 Relative Timing

As programs are parsed, we must enforce the timing semantics - movements inside square brackets, such as `[forward right`

forward], must occur within one "beat." To accomplish this, the parser reads in the elements inside brackets as a list of Dances.

Then we call a function seqL which uses the length of the initial list to determine how much to speed up each individual dance before composing the movements. This is accomplished with a function changeTiming which takes a multiplier m and a Dance, and propagates the multiplier through the Dance recursively. This allows for nested sequential movements: for example, the program [forward [left left] forward] would translate to the Dance

```
Prim (A f 3 r) :+: Prim (A l 6 r) :+:
Prim (A l 6 r) :+: Prim (A f 3 r)
```

where f is the Action corresponding to forward, l is the Action corresponding to left, and r is the robot body. Note that the two primitives inside the nested brackets have Mults of 6, since they must occur six times as fast as normal to allow the whole movement to occur in one "beat."

8 INTERFACING WITH ROS

benefits of interfacing with ROS

Roshask is a client library for ROS, written in Haskell [4]. It treats streams of values (such as those published and subscribed to by ROS nodes) as first class values, which allows for them to be combined and transformed in more natural ways than traditional ROS client libraries. For example, when we put 'Dance's in parallel, we wish to combine two list of motion commands with some function for parallel execution - whether this is averaging commands which affect the same body part, or additively combining them, or whatever interpretation the designer wishes for a specific robot platform. In Haskell, this is accomplished easily with the 'zipWith' function, which takes two lists and a function for combining values in those lists. Roshask extends this type of expressivity to the combination and transformation of ROS message streams.

To integrate a robot with *Improv*, one must specify how to convert the Dance data structure to a list of ROS messages. For example, for a non-articulated symmetric mobile robot (such as a Roomba or TurtleBot), this is accomplished by two functions: moveBase and danceToMsg. Since the robot has only one body part, moveBase takes an Action (direction and extent of movement) and converts it to a single ROS velocity command (which sets the desired linear and rotational velocity in the plane). Since our discretization of movements is relational (for example, you can have Full, Half, and Quarter extent, and directions are related through symmetries), only a small number of these translations from Actions to velocities need to be explicitly defined and the rest can be derived through their relations. For example, moveBase (A dir Half) is defined as fmap (*2) (moveBase (A dir Quarter)), where fmap maps the function *2 over the values in the velocity command returned by moveBase (A dir Quarter). This encodes the relationship that a Half extent is twice as far as a Quarter, thus the robot must move twice as fast in the same direction to travel twice the distance in the same amount of time.

Once this conversion (from a 'Dance' data structure to a list of ROS messages) has been completed, the list of ROS messages is given as an argument to a ROS node defined in roshask. This node publishes the commands over the ROS network. Even if multiple robots are controlled, the system still only uses one ROS node which publishes to multiple topics. Work is ongoing on whether to extend the system to control multiple ROS nodes, and if so, how best to implement this feature.

With all of these implementation details, we would like to emphasize that the design decisions for how *Improv* programs are realized on robot platforms are relatively arbitrary and a single robot could have a multitude of different implementations. "It is not technological constraints that hold us back from using technology in new ways; technology changes at a tremendous rate. Our willingness to explore beyond the constraints of our imagination has the greatest effect" [10].

9 CONCLUSIONS AND FUTURE WORK

This paper has presented a working implementation of a small domain-specific language which is interpreted and executed as a stream of ROS messages published by a ROS node. Due to fast interpretation of the language and a program which monitors a user's program file for changes, this system allows for a very fast feedback loop while programming robots. We hope that this short feedback loop, along with a programming language which models movement directly instead of the abstract state control flow typical of robotics languages, would decrease the cognitive load associated with robot programming. We aim to make the creative development of robot motion patterns faster, easier, and more accessible to a broader swath of potential users.

Future work will include systematic studies of people's qualitative assesment of the usability of the system, as well as quantitative measures on how quickly people iterate on programs in the *Improv* language and how much the robot moves as a result. As far as we know, no similar usability studies have been performed on the more mainstream C++ and Python ROS clients. We plan to include a range of participants in our study, including people with limited programming experience and no ROS experience, as well as people familiar with ROS.

The main limitation of *Improv*, as compared to other live-coding tools for ROS, is that it includes no features for controlling the robots based on sensor observations or interactions with the environment.

Another limitation of *Improv* is the complexity of defining the conversion from 'Dance' data structures to ROS messages, especially for robots with many body parts and degrees of freedom. Additionally, we have only implemented the system for robots which have velocity controllers, and several aspects of the movement transformation model depend on this assumption. For some robots, especially commercially available models, position-based controllers may be the only option available.

add uplifting good things, future work to address these limitations

10 ACKNOWLEDGEMENTS

Alli grant? Should ask Steve

REFERENCES

[1] Sorin Adam and Ulrik Pagh Schultz. 2014. Towards Interactive, Incremental Programming of ROS Nodes. *arXiv* (2014). http://arxiv.org/abs/1412.4714

[2] Miguel Campusano and Johan Fabry. 2017. Live robot programming: The language, its implementation, and robot api independence. *Science of Computer Programming* 133 (2017), 1–19.

[3] Nick Collins and Alex McLean. 2014. Algorave: Live performance of algorithmic electronic dance music. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. 355–358.
[4] Anthony Cowley and Camillo J Taylor. 2011. Stream-oriented robotics programming: The design of roshask. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*. IEEE, 1048–1054.
[5] Pablo Estefó, Miguel Campusano, Luc Fabresse, Johan Fabry, Jannik Laval, and Noury Bouraqad. 2014. Towards live programming in ROS with PhaROS and LRP. *arXiv preprint arXiv:1412.4629* (2014).
[6] Liwen Huang and Paul Hudak. 2003. *Dance: A Declarative Language for the Control of Humanoid Robots.* Technical Report YALEU/DCS/RR-1253. Yale University.
[7] Alex McLean, Dave Griffiths, Nick Collins, and Geraint A Wiggins. 2010. Visualisation of live code.. In *EVA*.
[8] Alex McLean and Geraint Wiggins. 2010. Tidal–pattern language for the live coding of music. In *Proceedings of the 7th sound and music computing conference*.
[9] Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA.
[10] Thecla Schiphorst. 1986. *A Case Study of Merce Cunningham's Use of the Lifeforms Computer Choreographic System in the Making of Trackers.* Master's thesis. Simon Fraser University.
[11] Brent A Yorgey. 2012. Monoids: theme and variations (functional pearl). In *ACM SIGPLAN Notices*, Vol. 47. ACM, 105–116.