Blake Mitchell
3/22/2019

6.945 - Draft Project Proposals

Overview

In general, I am interested in concurrency and in domain modeling, and so those are things I have tried to address in my ideation for potential projects. I have had some trouble honing in on a project and so had a bit of an idea explosion. I apologize about the length! Any thoughts on the more interesting projects are welcome, and I will keep refining my thoughts on these. In general, I would like to learn more about building DSLs, concurrency, continuations (this is a new topic for me that I have heard about but not studied before), execution flow, and macros.

Proposal 1: Concurrency Substrate

PUNCHLINE: Build a substrate of concurrency primitives such that more complex concurrent and parallel processes can be built from these.
INSPIRATION: *Concurrent Programming in ML* by Reppy, Concurrent ML, async workflows in F#, GNU Guile Fibers, actor model

Not many languages are great at concurrency, but many languages have been slowly adding primitives for building up concurrent processes, normally underneath the name of "async", for example async/await in C# or Python and async workflows in F#. As discussed in the dataflow project proposal, it is very helpful to have concurrency primitives at your disposal so that you don't have to constantly deal with the how of concurrency but rather on the what and your specific application of some concurrent processes. Taking inspiration from various languages, one could potentially implement the following syntax, behavior, and library in Scheme:

```
(define example-process ...)
(define async-example-process (async example-process)) ;; wrap a process as an async process
(define example-processes (map async (list ...)))      ;; wrap multiple processes
(start-async async-example-process) ;; starts and does not wait for completion
(define result (wait-async async-example-process)) ;; starts and waits for async process to
                                                   ;; return a value
(define result-list
  (wait-async (start-parallel example-processes))) ;; starts multiple async processes in
                                                   ;; parallel and waits for return values and
                                                   ;; puts them in a list named result-list
```

Other constructs could be added to the library to help manage concurrent processes such as returning results, sending messages to running concurrent processes, stopping concurrent processes, etc. There could be more granular procedures that allow you to build up custom concurrent processes. As far as I can tell, this is what Concurrent ML does.

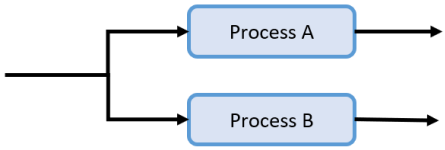
IMPLEMENTATION STRATEGY: This seems fun and doable. The above was heavily inspired by async workflows in F#, but I think the above makes sense almost as is in terms of user interaction with such a library. Other procedures and macros could be added to help manage the asynchronous processes, includes ways in to run them both synchronously and asynchronously. It's possible such a library could be used to build up the dataflow or actor systems described in other projects.

Blake Mitchell
3/22/2019

Proposal 2: Dataflow language

PUNCHLINE: Build a dataflow system that could be used as the backend of a visual or diagram language or simply used to represent a system of dataflow processes (dataflow computation, digital circuits, etc.) such that dataflow processes are automatically concurrent when possible.
INSPIRATIONS: SICP (Section 3.3.4), VHDL, LabVIEW, digital circuit diagrams, vvvv, Pure Data, Max

In textual languages, I feel there is a dimension missing. When programming in a text-based language, it feels linear in a way, such that it is difficult to see the shapes and data flow of the program. This is particularly apparent when doing concurrent programming. Visual languages make concurrent programming nearly trivial as it is easy to describe concurrent processes in a visual manner:



In such a diagram, one would imagine Process A and Process B operating in parallel such that once data arrives at the processes, they are immediately executed independently of the other process. This can be done in text-based languages, but I find it a little lacking in terms of how it's typically presented. This type of thing also becomes important in domains in which parallel processing, such as on FPGAs, is inherent to the execution hardware.

However, one still needs to be able to encode such visual dataflow processes. A language like Scheme is ready-made to encode and compute this type of dataflow. I think we could build a dataflow language within Scheme that would be intended to represent a visual dataflow language that has natural concurrency properties. For example, the above might be encoded in Scheme as:

```
(define (dataflow-process x)
  (define incoming-terminal (new-terminal x))
  (define outgoing-terminal-a (new-terminal))
  (define outgoing-terminal-b (new-terminal))
  (branch incoming-terminal => a b)
  (dataflow ((process-a a) => outgoing-terminal-a)
            ((process-b b) => outgoing-terminal-b)))
```

(Or something like this. I'm not completely happy with or convinced by this particular setup, but it gets the idea across and could be honed.)

IMPLEMENTATION STRATEGY: This could be implemented on top of the concurrency substrate described in another project. I would love to embark on a project like this, but some more work is needed in defining the surface of this dataflow language and other examples of dataflow processes it is meant to be able to describe. Although a nice end goal would be to actually build a visual dataflow language that could be edited, this requires a significant amount of GUI work. So instead, this project could be such that visual dataflow processes are meant as inspiration and representations of the textual dataflow language such that the language could later serve as the backend of a visual dataflow language.

Blake Mitchell
3/22/2019

Proposal 3: Teachable Actors

PUNCHLINE: Create an actor system such that actors can teach other actors with new message processors.
INSPIRATIONS: Actor model, CANopen, Erlang, Smalltalk

The actor model is a common solution applied to problems of distribution, concurrency, and modularity. By actor I mean a *thing* that is able to launch other actors and send and receive messages from other actors. The *thing* that embodies an actor could be many things, a common one being an object in the OOP sense. But an actor mustn't necessarily be an object.

One problem of ridigity in actor systems is that the messages an actor knows how to process must be defined ahead of time. When an actor receives a message it does not have a handler for, a common solution is to simply ignore the message (sometimes called dropping the message) or to send a message back to the sender stating it doesn't know what to do with that information. Another solution is to stop the actor upon receiving an invalid message and possibly generate an error, but this is overly harsh and generally fragile, in my opinion.

So actors are fun, important, and useful. On top of building an actor system, what if we had an actor system that allowed us to *teach* actors? That is, we could provide existing and running actors with new message processing behavior. I think this wouldn't be too hard for message processors that don't update an actor's state. So if the message processor just does something and then maybe notifies another actor, that would be doable. It might get a little tricky if the message being taught requires teaching the actor about some new state and/or data.

In general, we could build an actor system within Scheme. I have applications in mind such that an actor system would make sense, although I have struggled to make practical sense out of the teachable actors. One thing that comes to mind are what I have read about Common Lisp and Smalltalk in terms of being able to update running code. This seems similar to the teachable actor concept. Another application is that of simulation or artificial intelligence such that actors can talk to each other in a deeper manner than just sending pre-defined messages. However, these two latter applications are outside the scope of a small term project.

IMPLEMENTATION STRATEGY: This is one of the more interesting projects and could potentially be an extension of the concurrency substrate project, since the actor system could be built using the concurrency library, but it is the least defined in a practical sense. (For example, I did not include any Scheme code in how the library might work, but I have a few ideas.) I think implementing an actor system would be doable as I have experience with making systems that make use of the actor model. But the teachable actor thing needs more exploration.

Blake Mitchell
3/22/2019

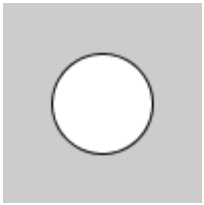
Proposal 4: 2D Graphics the Way I Think

PUNCHLINE: Generate 2D graphics and animation declaratively.
REFERENCES: Processing, Quil, vvvv

This project proposal is inspired by both the benefits and detriments of Processing, which is an environment that makes 2D and 3D graphics accessible to artists and designers. It can be used as a Java library, but by itself, it can be considered a language and IDE. It is a nice system, but it leaves me wanting. For example, I have long wanted to get more seriously into learning relativity, both special and general, and in doing so I picked up some books on special relativity that focus on learning how to properly interpret and create spacetime diagrams. When going over these, it seemed like a fun project to create a spacetime diagram library in a programming language such that not only could they be made easier programmatically, but they could also be animated to better illustrate some property or concept. I feel Processing has the lowest barrier to such a library, but the way graphical objects and actions are represented in the language doesn't match up with the way I think of them. There is a cognitive disconnect.

For example, in Processing, if I want a circle, I need to supply both properties of the circle, such as its radius, and where to place it. This also couples the shape itself with its position.

```
size(100,100);
circle(50,50,50);
```



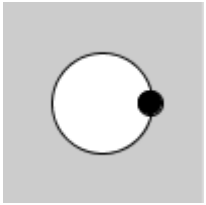
Note that there are properties not captured here, such as the fill and stroke colors. (Processing's terminology for the color of the interior and boundary is fill and stroke, respectively.) There are default colors, but to change them, you need to have a stroke and/or fill command prior to drawing and placing the shape.

Now, let's say I wanted another circle to move around the edge of this circle. In Processing, you have to annoyingly parameterize the circle and then update the position over time. This would be done like this:

```
int centerX = 50; int centerY = 50; int radius = 25; float t = 0;

void setup() {
  size(100, 100);
}

void draw() {
  background(204);
  int x = int(centerX+radius*cos(t));
  int y = int(centerY+radius*sin(t));
  t = t + 0.01;
  fill(255);
  circle(centerX, centerY, 2*radius);
  fill(0);
  circle(x, y, radius/2);
}
```



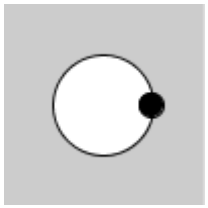
To me, this is horrid! All I wanted to do was animate a black disk circling around my previously drawn circle, but I had to do a bunch of stuff that has to do with *how* to do with what I want and not *what*. That is, the Processing interaction is highly imperative, and I would rather something more declarative.

Let's see how one might do this in a proposed Scheme system that knows how to draw these things.

```
(define centerX 50)
(define centerY 50)
(define radius 50)
(define speed (hz (/ 1 10)))
(define disk1 (disk radius ‘interior "white” 'boundary "black"))
(define disk2 (disk (/ radius 2) ‘interior "black" 'boundary "black"))

(setup
  (size 100 100)
  (background (grayscale 204))
  (place-by-center disk1 centerX centerY))

(draw
  (move-along-path disk2 (boundary disk1) speed))
```



Here, we define things much more clearly and declaratively, where anyone who reads this, whether they know Scheme or not could guess as to what it might do. The expression `(hz (/ 1 10))` tags the number 0.1 with Hz such that it describes a speed of "one time every 10 seconds". So when applied to movement along a path this means that the object moving moves at a speed such that it completes the path once every 10 seconds. The procedure `disk` simply creates a disk shape, which we can optionally color the interior and boundary. The procedure `place-by-center` takes an object and x,y coordinates and places the object at (x,y) by centering the object at (x,y). Other similar procedures might be something like `place-by-left` or `place-by-top`. Then the procedure `move-along-path` takes an object and moves it along a path, in this case provided by `(boundary disk1)`, at a given speed. The `boundary` procedure acts as the mathematical boundary operator such that it selects the boundary of a shape. There could be an optional argument to `move-along-path` that dictates what the shape does when it reaches the end of the path. Does it start back at the beginning or does it "reflect" such that it starts backwards from the end towards the beginning? The words `setup` and `draw` might make sense more as macros rather than procedures.

IMPLEMENTATION STRATEGY: I think this could be done within MIT/GNU Scheme, but the graphics system is a little spartan. It would probably be easier to use Racket, which has a substantial graphical library with cross-platform support, but is still a Scheme dialect. Another option would be to code the above described System in MIT/GNU Scheme but then emit Processing code. (This would be similar to the regular expression language.) Then this emitted code could be run by Processing, which would handle all of the graphics.

Blake Mitchell
3/22/2019

Proposal 5: Scheme Type System

PUNCHLINE: Add types to Scheme but don't let them get in the way.
RELATED: Software Design for Flexibility (Section 4.5.3), Typed Racket

Lisp, Scheme, and their descendants are wonderful languages, but *sometimes* one does wish for a static type system to help. (Not all the time. See below.) Such a wish surfaces when interacting with some new library or when refactoring. A static type system helps one know what is expected and what goes where. It helps *connect the dots*. A possible syntax for this in Scheme might look like:

```
(:type Number -> Number => Number)
(define (multiply x y) (* x y))

(:type List of Integer => Integer)
(define (sum-list lst) (reduce + 0 lst))
```

The usage is that types are declared with an S-expression starting with `:type`, argument types are separated by `->`, and the output type follows the `=>` designation. This is a nice start to a type system' user interaction design. However, I think it is still possible we relax the type system. For example, in an ML language like F#, you can have lists:

```
let listExample1 = [1; 2; 3] // int list
let listExample2 = [1; 2.0]
error FS0001: All elements of a list constructor expression must have the same type. This
expression was expected to have type 'int', but here has type 'float'.
```

but they are required to be homogeneous as can be seen by the error. A key property of Lisps and Schemes is that great use is made of heterogeneous lists. I don't see a reason why a type system shouldn't allow for that. A syntax for this might be:

```
(:type (Anything => Boolean) -> List of Anything => List of Anything)
(define (filter predicate lst) ...)
```

Once we introduce `Anything` and then use it like in `List of Anything`, one might wonder if there's some balance between say `List of Int` and `List of Anything`. For example, why can't I have a procedure that accepts a list consisting of either numbers or strings but nothing else? How about:

```
(:type List of String|Number => String)
(define (create-email-address list-of-strings-and-numbers) ...)
```

In this case we treat the type `String|Number` as an anonymous union type such that something of type `String|Number` is either a String or Number. This leads us to the introduction of union types to the typed language. In fact, union types, or algebraic types, are some of the most powerful features, in my opinion, found in some static languages, mostly descendants of SML. Union types are very useful for domain modeling and work well for pattern matching. We could introduce the ability to define a new type that represents a union:

```
(union-type 'Boolean ('True | 'False))
```

All this being said about adding types to Scheme, we have to face the fact that both statically and dynamically typed languages have their benefits. However, when working in one, the type system often gets in the way, in the case of static typing, or is too out of the way, in the case of dynamic typing. When working in a statically typed languages, there are sometimes cases where you have to stop making progress and start wrestling with the type system. In dynamically typed languages, it is often the case that you wish the type system could have told you something sooner than it did.

I have seen some solutions to this, namely the gradual typing found in Racket. However, that is even still too limiting. You must ahead of time decide whether you are going to use #lang racket or #lang typed/racket. You can of course call code written in the other language between Racket modules, but as far as I know, you cannot intermix them within a single module. There is also no way, again that I know of, to easily toggle the type system that you are using.

So, what if we *could* toggle the type system we are using, on the fly? For example, in a dynamically typed language, it is a common idiom to use consistent naming and documentation to notify users what types are expected for a procedure's arguments. What if we could toggle this scheme between documentation and actual type declarations? So for the above, the S-expressions starting with `:type` would become just documentation under the hood in the case of the type system being turned off.

IMPLEMENTATION STRATEGY: The above seems like a nice design and is similar to Typed Racket. However, I have no experience doing this, and this is probably hard. Maybe there's some type system theory that says it won't work or isn't possible. Although I do think it would be an interesting project, and maybe one way to make it doable is to restrict the type system to a subset of Scheme. Maybe there's also a way to scope the project, potentially reducing it down to implementing just the union type and some pattern matching based upon it. Also, the above syntax may be difficult to do as presented, since to my knowledge, macros can't look outside the S-expressions they are in. So one compromise would be to make `:type` either a regular macro or procedure to avoid having to parse special syntax. Like this:

```
(:type Number -> Number => Number
  (define (multiply x y) (* x y)))
```

Blake Mitchell
3/22/2019

Proposal 6: Differential Forms Library

PUNCHLINE: Build a library to handle differential forms.
INSPIRATION: scmutils, SICM, differential geometry

Earlier in the semester, I built my own automatic differentiation library after we covered it in class since this is something I had done before in LabVIEW but I wanted to try it in Scheme. I took a slightly different implementation approach by explicitly introducing dual numbers. Some code snippets:

```
(define-record-type <dual-number>
  (make-dual-number real dual) ; constructor
  dual-number?                 ; predicate
  (real real-part)             ; accessor
  (dual dual-part))            ; accessor

(define (dual*-binary a+b*e c+d*e)
  (let ((a (real-part a+b*e)) (b (dual-part a+b*e))
        (c (real-part c+d*e)) (d (dual-part c+d*e)))
    (make-dual-number (* a c) (+ (* b c) (* a d)))))
(define (dual* . dual-numbers)
  (reduce-left dual*-binary 1 dual-numbers))

(define-print-method dual-number?
  (standard-print-method "<dual-number>" get-dual-number-parts))

(define (f-of-dual-number f derivative-of-f dual-number)
  (let ((a (real-part dual-number))
        (b (dual-part dual-number)))
    (make-dual-number (f a) (* b (derivative-of-f a)))))

(define (dual-sin dual-number)
  (f-of-dual-number sin cos dual-number))

(define (differentiate-f f derivative-of-f real-number)
  (dual-part (f-of-dual-number f
                               derivative-of-f
                               (make-dual-number real-number 1))))
```

A multi-linear algebra library could be combined with this plus the symbolic arithmetic we have worked on to combine into a library for handling differential forms. Thus, a library could make sense of $\omega = f\,dx + g\,dy$ on $\mathbb{R}^2$ and $d\omega = \left(\frac{\partial g}{\partial x} - \frac{\partial f}{\partial y}\right) dx \wedge dy$ in a way similar to (the dimension of the ambient space isn't specified and would need to be):

```
(define omega (+ (* (literal-function f) (1-form x)) (* (literal-function g) (1-form y))))
; -> omega = (+ (* (literal-function f) 'dx) (* (literal-function f) 'dy)
(define domega (d omega))
; -> (* (- (partial (literal-function g) x) (partial (literal-function f) y))
;       (wedge 'dx 'dy))
```