Chapter 1

Braincurry: A Domain-Specific Language for Integrative Neuroscience

Tom Nielsen[1], Tom Matheson[2], Henrik Nilsson[3]
Category: Research

Abstract: This paper describes *Braincurry*, a domain-specific, declarative language for describing and analysing experiments in neuroscience. Braincurry has three goals: to allow experiments and data analysis to be described in a way that is sufficiently abstract to serve as a *definition*; to facilitate carrying out experiments by *executing* such descriptions; and to be directly *usable by end users*: neuroscientists. We adopted an experimental and incremental approach to the design and implementation of Braincurry, focusing on the neurophysiological response to visual stimuli in locusts as a test case. Braincurry is currently implemented as an embedding in Haskell, which is a highly effective tool for this kind of exploratory language design. The declarative nature of Haskell and its flexible syntax fitted with our goals. We discuss the requirements for a realistic language meeting the above goals, describe the current Braincurry design and how it may be generalised, and explain how some particularly challenging hard real-time requirements were met.

1.1 INTRODUCTION

Repeatable experiments are the very core of empirical science. Not until scientists can repeat an experiment in a controlled manner can this experiment be used to

[1]Department of Biology, University of Leicester, UK, and School of Computer Science, University of Nottingham, UK; `tanielsen@gmail.com`

[2]Department of Biology, University of Leicester, UK; `tm75@le.ac.uk`

[3]School of Computer Science, University of Nottingham, UK; `nhn@cs.nott.ac.uk`

properly test a hypothesis. And not until an experiment can be repeated independently by *other* scientists, permitting them to check the premises of the experiment, are the outcomes of the experiment considered by the scientific community to be proper evidence for or against various explanatory models. Experiments must thus be repeated, potentially many times, as the success rate can be very low or the outcome variable.

This raises two issues:

- How can an experiment be described in sufficient detail to make it repeatable?
- To what extent can an experiment be mechanised to reduce variability in experimental conditions and to save work?

Traditionally, these issues are addressed independently: published experimental results are accompanied by a detailed description of the experiment in natural language, and experimental paradigms are simplified and preprogrammed to permit consistency. In this case the description is separate from the experiment itself.

If a *single* description could address *both* issues, at least to some extent, this would bring a number of advantages. For example:

- A description that is part of a working experimental setup guarantees that the description is complete and precise.
- There may be a reduced need for auxiliary experts such as programmers as the experiment description developed by researchers would also serve as the program controlling the experiment.
- Ultimately, this could lead to a shared, precise language for describing experiments in a particular area.

Braincurry is being developed to explore the feasibility of an experiment description language of the kind outlined above for integrative neuroscience. This is a challenging task. For example, an experiment could involve showing a visual stimulus to a living animal, recording electrical activity from neurons, and measuring motor behaviour, all at the same time; see Figure 1.1. Another experiment might involve direct patterned electrical stimulation of a neuron while monitoring the responses of several other neurons recorded optically using a CCD camera mounted on a microscope. Such experimental complexity and variability necessitates the configuration and synchronisation of a potentially large set of heterogeneous devices, as well as the generation of stimuli and recording of responses subject to hard real-time constraints.

Experiments may be performed to distinguish between competing models or to estimate free parameters in a given model. Where an explicit computational model exists, it would be advantageous to be able to directly compare recorded responses to simulations. For an experiment description language, this means that a description should, where possible, be executable both on data acquisition hardware recording from living neural tissue, and on appropriate models.

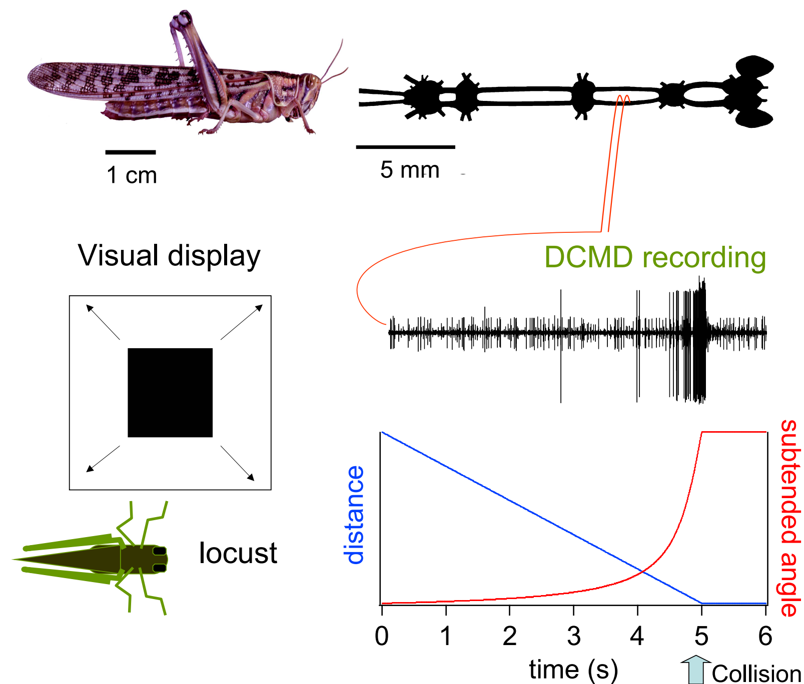

FIGURE 1.1. Top left, the adult desert locust *Schistocerca gregaria* with its central nervous system (top right; modified from [1]) consisting of the brain and a series of ganglia with nerves ("connectives") running between them. We record from the descending contralateral movement detector (DCMD) neuron using a pair of silver hook electrodes wrapped around one of the connectives. The bottom panels illustrate the result of running an experiment using Braincurry. Left, the visual stimulus is an expanding black square, giving the impression of a cube approaching on collision course with constant velocity. Right, the recording from the hook electrodes shows electrical responses from several distinct neurons, of which the largest amplitude is the DCMD. On the same timescale below, two calculated parameters of the visual stimulus: apparent distance and the observed angle formed by the visual stimulus on the retina.

It is not clear what an experiment description language should look like to best serve the diverse needs of neuroscientists. Conventionally, the description of experiments is limited to configuration of pre-written software through a graphical user interface, or by direct editing of configuration files. This approach is limiting because any fundamentally new need has to be addressed by modifying the underlying software. For flexibility, a proper language is needed. But how to make such

a language flexible enough for general use without making it too complicated for domain experts who may not possess conventional programming skills?

In our experimental and incremental approach to the design of Braincurry, the initial version as presented here is focused on experiments on insect vision [4]. This allows us to test the feasibility of the idea of a language like Braincurry and to more easily experiment with various language designs. To this end, Braincurry is at present realised as an embedding in the functional language Haskell [7]. Using a functional language allows us to use composition and abstraction to control the complexity of experiment descriptions, and has the further advantage that the functional program gives a concise, declarative description of what has been done in the experiment in line with our overall goals introduced above.

The specific contributions of the present paper are as follows.

- We develop general requirements for an experiment description language for integrative neuroscience.
- We outline a flexible implementation of such a language based on composable interpreters that has proved to be sufficient for our present experiments on insect vision.
- Our particular research on insect vision required a high-throughput and predictable system for visual animations, typifying the kind of hard real-time signal generation problem common in the application domain. As an example of how to address this kind of requirement, we have implemented a hard real-time 3D analogue of the Pan language [2]. We outline this implementation and describe how it fits in with the rest of our framework.
- We sketch how the ideas presented in this paper could be extended to a general-purpose experiment description language for integrative neuroscience.

1.2 REQUIREMENTS

Currently, in the field of neuroscience, experiments are conducted with varying degrees of automation. One common approach is to use one of a range of systems that are specifically intended for automating certain aspects of the experiment, such as stimulus generation. These systems typically provide a graphical user interface allowing the experimenter to define protocols within a domain restricted to the intended target application. Alternatively, this may be done by amending some kind of configuration file using a text editor. Examples include systems like AxoGraph[4], NeuroMatic[5], or the sequence editor in Spike2[6] (Spike2 does in addition provide a simple programming language).

A key advantage of this approach is the relatively shallow learning curve: scientists without any programming background, including junior graduate students

[4] http://axograph.com/

[5] http://www.neuromatic.thinkrandom.com/

[6] http://www.ced.co.uk/pru.shtml

who are only beginning to learn about neuroscience, can very easily design and conduct experiments. The drawback is the limited flexibility: while some of these systems can be extended, this does require programming in whatever language the system is implemented in.

For increased flexibility, an experiment can be programmed in an imperative language with bindings to the data acquisition and stimulation devices. This definition will delineate the steps necessary in preparing and running the experiments. Common choices here are to use Matlab, Spike2 or Igor Pro[7]. However, the result is usually a monolithic application, fit for only a single purpose and hard if not impossible to reuse as a *component* of other experiments. Moreover, the high-level description of the experiment will be deeply hidden behind imperative programming constructs, making it impossible to reflect on the experiment in any meaningful way.

Ideally, a system for supporting experiments in neuroscience needs to be both *flexible*, to allow experimenters to do what they need to do, and *declarative*, to lower the learning curve and allow for composition and reflection. The above considerations suggest the following principal requirements for Braincurry:

Synchronised analog and digital data acquisition and output: Typical integrative neuroscience experiments require accurately synchronised generation of stimuli and recording of responses. In our own experiments on locusts (Figure 1.1), these experiments involve at the very least simultaneous presentation of visual stimuli and recording of the electrical responses of cells, followed by signal detection and sorting. This means that Braincurry must be able to meet hard real-time constraints if it is to be a practically useful tool.

Composable experiment descriptions: Experiment descriptions should be *composable*: given a description d_1 with input i_1 and output o_1, and another description d_2 with input i_2 and output o_2, it should, using an *experiment composition operator* $\oplus$, be possible to form a composite experiment $d_1 \oplus d_2$ with a combined input of i_1 and i_2, and a combined output of o_1 and o_2. This allows for the development of libraries of "small experiments" that can be combined to describe more complex experiments. The experiment description gives a clear record of what has been done on each experiment. (Of course, some combination of experiments may not make sense. This must be detected and reported.)

One language for simulations and experiments: Two common modes of research in integrative neuroscience are of particular interest to us:

- derivation of a computational model from experimental data;
- investigation of whether such a model can accurately predict experimental observations.

In both of these cases, we are trying to match the response of two systems (one, a living organism; the other, a numerical simulation) to a common set of inputs. This task is greatly facilitated if these inputs are described in the same way for

[7] `http://www.wavemetrics.com`

both systems. Thus, we require that the experiment description can be executed both on *physical* data acquisition systems interfaced to *real* neural tissue, and on arbitrary *simulations* of such systems. Apart from the clarity that such a unified approach gives in terms of comparing experimental data with models, it also opens up exciting possibilities that would be hard to achieve with separate experiment description languages. For example, it may be possible to automatically estimate free parameters of a model to fit given experimental data, or it may be possible to automate experiments to search for a difference with a given model.
Accommodation of diversity: Neuroscience is a highly interdisciplinary field where new experimental possibilities constantly emerge. At the same time, no one lab can embrace all of these experimental possibilities. An ideal data acquisition system based on experiment descriptions thus needs to be very flexible.

Structured, practical storage: We would like to store the results of running experiment descriptions (*in vivo* or *in silico*) so that they can be easily retrieved and indexed. The experiment description itself and as much relevant information as possible should be stored along with the raw data. We may also like the results of analyses to be stored in a similar way to raw data, while clearly indicating that it was obtained after the experiment. The data should be retrieved by posing questions that are as natural as possible to a neuroscientist.

1.3 ARCHITECTURE OF BRAINCURRY

To accommodate the many different techniques used in experimental neuroscience, we took a minimal approach in designing Braincurry. As a result, deploying Braincurry in a new setting requires the addition of quite a bit of functionality to create a working data acquisition and simulation system. However, these definitions do not have to be written on a blank slate: the Braincurry implementation provides considerable conceptual and concrete infrastructure in the form of *combinators for composable interpreters* and a *scheduler* that provides a small core necessary for coordinating the experiments. In practice, a group of users with similar requirements would define a part of their own description language and one or more interpreters to link the final, customised experiment description language to hardware or numerical solvers.

As a first step in customising Braincurry, the user defines the *trial option datatype*. This is a custom datatype for which variants denote individual components of a *trial*; i.e., a finite period of time with fixed configuration during which observation takes place. For instance, for our experiments on locusts, we have defined an algebraic datatype *LocustOptions* with (parametrised) constructors for playing an animation, playing sounds, and recording from an electrode. Values of this type convey information about a particular aspect of a trial such that a list of trial options completely specifies the lab-specific aspects of a trial in an experiment. Figure 1.2 shows the Braincurry code specifying the experiment of Figure 1.1, including the specification of the animation and the list of trial options. The animation part of this code is explained in detail in Section 1.5.

```
black = Col 0 0 0
centreCube l = Translate (Vec (−l / 2) (−l / 2) 0)
                         (Box (Vec l l l))
loomAnim lov =
  let l = 0.298
      v = l / (lov ∗ 2)
  in
    [LetN "distance" (minN (v ∗ (Time − 5)) (−0.17)),
     MkShape $ WithColour black
              $ Translate (Vec 0 0 (VarN "distance"))
              $ centreCube l
    ]
loomExperiment = concat $ do
    ivl ← [15,30]
    lov ← [0.01,0.02]
    take 10 (repeatInterval ivl
                    [PlayAnimation (loomAnim lov),
                     RecordEC])
```

FIGURE 1.2. Complete Braincurry specification for an experiment involving the generation of a visual stimulus that gives the impression of a cube-shaped object approaching with a length-to-velocity ratio *lov* (in seconds). The experiment description *loomExperiment* specifies that the experiment should be repeated 10 times for two different values of *lov* and with two different intervals *ivl* (in seconds) between repetitions, and that the neural responses should be recorded concurrently (*RecordEC*).

The second step in customising Braincurry is to create one or more *apparatuses*: interpreters that execute experiments or simulations defined by the trial options datatype. We stress that the value of the trial options datatype does not have any inherent meaning in Braincurry until the user also defines an apparatus that consumes a trial description and produces the results of running such a trial.

As discussed in section 1.2, one of our goals is that it should be possible to run an experiment description *both* on a physical experimental setup and as a simulation. One way to achieve this would be to define *two* interpreters for the *same* options type. However, this solution is problematic in at least two ways:

- Writing an interpreter is not an easy task and is fundamentally at odds with Braincurry's stated aim of being accessible to scientists with little programming experience.
- Two interpreters, even when written for the same language (option type), may have different structure and present different interfaces.

We have addressed these issues by providing the facilities to *compose* the nec-

essary interpreters, rather than building them from scratch. These facilities consist of a series of *base interpreters*, independent of the trial option type, for carrying out experiments on data acquisition hardware or in simulated integrate-and-fire neurons [6], and of the combinators necessary to *modify* these interpreters to respond to the values of the trial option type. Although composing an interpreter in this way still requires some knowledge of Haskell, such as the syntax for function application, it is much more straightforward than starting from scratch. The resulting interpreters could likely also be shared between labs using similar techniques. Furthermore, the approach ensures a uniform interface for our interpreters such that these can be used interchangeably.

1.4 IMPLEMENTATION OF BRAINCURRY

An apparatus is thus an options interpreter for either experimental setups or simulations. How shall they be represented? The chosen data structure must be parametric on the trial options datatype. In order to accommodate different kinds of side effects of experiments (that must interact with the real world) and simulations (that may not have any side effects at all), it should also be parametric on the monad in which functions are run. In addition, the apparatus must present a uniform interface to the experiment scheduler. We chose to represent apparatuses as a type-parametrised module [11]; i.e., a record of functions (representing the uniform interface) parametric on a monad and the trial options datatype. Each field is a monadic action corresponding to a *step* involved in running a trial. The step division is quite fine-grained so as to be appropriate for the configuration and operation of a broad range of data acquisition devices:

```
data Apparatus m o = Apparatus{
  initialise    :: m (),
  newTrial      :: [o] → m (),
  prepare       :: [o] → m (),
  wait          :: Double → m (),
  run           :: [o] → m [(String,AnyResult)],
  finaliseTrial :: m (),
  finalise      :: m ()
}
```

In more detail, the meaning of each step is as follows. The step *initialise* is run at the beginning of a long experiment consisting of many trials. It is typically be used to initialise screens and to open devices. Correspondingly, *finalise* can be used to close these devices when the experiment is over. The remaining five steps are invoked once for each trial in the order in which they are listed:

- *newTrial* and *prepare* set up any devices and prepare them for an imminent triggering signal (the exact specification of which is dependent on the device)
- *wait* counts down a specified number of seconds to the trigger that will start

the trial (although, for simulation purposes, an apparatus can customise this step to avoid spending real time waiting)

- *run* runs over the duration of the trial and returns outcomes of the trial
- *finaliseTrial* cleans up after a trial; for example, it may free memory allocated for storing the trial results.

Note that the complete list of trial options are passed to each step where needed.

Some apparatuses may not require all of these steps. As new apparatuses are defined from a default record where all fields are defined as the empty action (i.e. return (), or, in the case of *run*, return [ ]), only steps that are required need to be defined.

An individual apparatus is thus a value that can be manipulated and reconfigured at run-time. To facilitate this, we have written a series of combinators for generic modification of apparatuses. For example, there are combinators for insertion of monadic actions before or after a specific step, like the following:

```
beforeRun :: Monad m ⇒
             ([o] → m ()) → Apparatus m o → Apparatus m o
beforeRun brun app = app{ run = brun ≻→ run app }
```

where

```
(≻→) :: Monad m ⇒ (a → m c) → (a → m b) → a → m b
f ≻→ g = λx → f x ≫ g x
```

The composition of an apparatus begins with a base apparatus. This is provided as part of the initial Braincurry installation and is agnostic about the meaning of the trial options datatype. For instance, we provide a base apparatus for controlling data acquisition devices through the Comedi[8] library:

```
type ComediApparatusM = StateT ComediState IO
comediApparatus :: Double → Apparatus ComediApparatusM o
comediApparatus acquisitionFreqInHz =
  Apparatus{
    ...
    run = (λopts → do
      hasOut ← anyScheduledOutputs
      when hasOut (liftIO $ do
        forkIO start_cont_output
        return ())
      liftIO $ start_cont_acq
      getAllWaves),
    ...
  }
```

[8]http://www.comedi.org

Note how a Comedi-specific monad is constructed by adding a suitable state component to the IO monad. The step operations are here exemplified with the *run* step, the definition of which is rather typical: if there are scheduled outputs, like sound, a thread is spawned whose task it is to keep the output buffers associated with the outputs filled. Then data acquisition is initiated.

A base apparatus is supplied as a Haskell module, containing a value of the apparatus type parametric on the trial options datatype, as well as combinators for modifying this apparatus. These combinators are intended to tie together this apparatus with the options datatype.

The final apparatus is thus formed by layering modifying combinators around a base apparatus, instantiated with a particular trial options type. For example:

```
dcmdRig :: Apparatus ComediApparatusM LocustOptions
dcmdRig =
  prepareMap dcmdRigPrepare $
  triggerWith dcmdTrigger   $
  comediApparatus 20000 {-Hz -}
```

In configuring an apparatus using specific values of the options datatype, we found it helpful to write a combinator that iterates over the experiment description (presented as a list of trial options), thus allowing us to pattern-match and add for instance a preparatory action for specific trial options:

```
dcmdRigPrepare :: LocustOptions → ComediApparatusM ()
dcmdRigPrepare RecordEC =
  readWave silverElectrode "ecVoltage"
```

Ultimately, the apparatus is passed to the scheduler that is responsible for coordinating the experiment. The run method returns a collection of named results for each trial, wrapped in the existential data constructor *AnyResult*:

```
data AnyResult = ∀ r . Result r ⇒ AnyRes r
```

The *Result* class details how to store and retrieve data types into a relational database (see Section 1.6).

These wrapped data types are collected by the scheduler as the sequence of trials is executed. No storage or display is performed automatically because the person conducting the experiment might want to do different things with these data depending on the purpose of the experiment. For instance, they might create a log file, display the results on screen, or store the results in complete form for later analysis. To give the experimenter a choice of how to process results and to keep the design as modular as possible, we have parametrised the scheduler on a results handler, i.e. an action that consumes the named results. We have also implemented an operator ⤖ that composes two results handlers into a new handler:

```
(⤖) :: Monad m ⇒ (a → b → m c) → (a → b → m d) → a → b → m d
f ⤖ g = λa b → f a b ≫ g a b
```

1.5 VISUAL STIMULATION

Many neuroscience experiments investigating vision require the display of shapes of differing colours, contrasts or textures that move or appear and disappear. Such stimuli must be presented with a sufficiently high screen refresh rate and no dropped frames to ensure a natural neural response. For example, insect vision has high temporal resolution necessitating a screen refresh rate above 100 Hz [5].

To address the need to generate visual stimuli, we include a trial option describing animations. Animations are represented as a function from time to a list of shapes (in the style of Functional Reactive Animation (Fran) [3]), where each shape is composed of arbitrarily scaled or rotated geometric primitives.

To turn such descriptions into a live display on a monitor, the apparatus controlling the experiment is extended with a mechanism taking care of the rendering. However, it is very difficult to fulfil the requirement of a refresh rate of at least 100 Hz in a garbage-collected language like Haskell as the garbage collector can add unpredictable delays in execution[9]. Due to these hard real-time requirements, we do not use Haskell functions directly to represent these animations. Instead the user constructs a symbolic expression, effectively an *Abstract Syntax Tree* (AST), detailing the shapes to be rendered and how they evolve over time. This expression is then *compiled* into C code for execution in a different process. The implementation of our animation component is thus similar to the language Pan [2], but our language is much simpler.

Besides shapes, our language provides primitive values, variables, expressions over these, a conditional construct, and a variable binding mechanism. The language does not itself contain any mechanism for functional abstraction, relying instead on the host language to provide this facility. The types of the language are restricted to scalars and vectors of floating point numbers, Booleans, and colours. Everything is simply typed to facilitate compilation into C. Moreover, the binding mechanism is limited to scalar numbers for the same reason. The special numeric expression *Time* gives the number of seconds since the trial start. This is the key to describing animations as time-varying shapes, or, thanks to the conditional, even more drastic changes.

As an example, consider the following code fragment from Figure 1.2:

$$\mathit{LetN}\ \texttt{"distance"}\ (\mathit{minN}\ (v * (\mathit{Time} - 5))\ (-0.17))$$

Here, the *N* suffix indicates we are working on scalar numbers. The *LetN* construct binds a variable *at the C level* for use later in the program. In this case, the variable `distance` is bound to a time-varying value that represents the distance between the object and the observer in real-world coordinates. Note that *all* expressions in our animation language potentially yield values that vary over time, i.e. they really represent signals.

[9]Particular *implementations* of garbage-collected language may provide some hard real-time guarantees, but the problem remains unless the language *specification* provides such guarantees (and all implementations conform).

Shapes are composed in a functional manner starting from geometric primitives. The latter can be translated or rotated as necessary by vectors specified in real-world 3D coordinates relative to an observer at a known location viewing the display. For instance, consider the following fragment, again from Figure 1.2:

```
MkShape $ WithColour black
        $ Translate (Vec 0 0 (VarN "distance"))
        $ centreCube l
```

Here, a time-varying shape is made up from a centred cube by first translating it by a time-varying vector and then colouring it black.

Finally, an animation is a list of declarations that introduce new variables or create shapes (as above). The function *loomAnim* in Figure 1.2 is an example of an animation of a looming object. As we have already seen, the object is a cube with constant dimensions that is translated with respect to the viewer. As the subject-object distance decreases linearly with time, it will appear as if the object is approaching at a constant speed, until it intersects the location of the physical screen surface (in our case, 0.17 m from the observer). Note how the animation is made up of a list of declarations: first one that defines the variable `distance` bound to a decreasing value, then one that declares the cube shape, coloured in black and translated by the previously defined variable `distance`.

A consequence of this simple design is that whole animations are themselves composable by concatenating the lists of declarations, so that more animations can be built up from simpler fragments that will be superimposed. These fragments can be named and parametrised as Haskell values or functions.

The abstract syntax tree is translated to C source code, then compiled by GCC into a dynamic library. The generated source code uses OpenGL to display computed geometric shapes at each frame. The graphics card handles the translation of 3D coordinates to screen output, and in fact does so much more efficiently than we would be able to achieve by calculating a screen buffer with the CPU and then transferring that to the graphics card for display. In our measurements, on a Pentium 4 computer with an nVidea GeForce 5200 graphics card, it took up to 9 ms to transfer a frame from the computer memory to the graphics card, but only a few hundred microseconds to instruct the graphics card to draw equivalent primitives in 3D coordinates. Ensuring that these coordinates correspond to the real world is a matter of setting up the correct viewing angle and distance, which is a single instruction in OpenGL code.

The dynamic library containing the animation is loaded by an independent process that controls the screen between animations. This process receives a triggering signal from Braincurry when a trial involving animation starts, and the animation is then executed. We use POSIX signals to communicate between these processes.

1.6 PERSISTENCE

One important advantage of using a language to define experiments is that we have a clear record of the procedures that took place during a particular experimental session, i.e. a collection of trials from (for instance) the same animal. To take full advantage of this possibility, it is desirable to store not only the experiment description, but also the results and any available metadata (such as time and circumstances of the experiments) in a structured manner to facilitate searching, indexing, annotation, and further analysis later on.

The backend of Braincurry's structured storage system is provided by the relational database PostgreSQL. Using an off-the-shelf solution for storage, searching and indexing meant that we only had to write two components in order to provide structured storage for experimental results: a results handler (see Section 1.4) targeting the database, and a tailored query language to retrieve results[10].

Implementing composable results handlers turned out to be straightforward and will not be discussed further. For querying the database, users could be asked to use SQL. This would save us from (most of) the trouble of implementing a query language, but would also make Braincurry considerably harder to use: First, the end-user may not know SQL, which is a large and complex language. Second, formulating queries in SQL necessitates revealing the relational structure of the underlying database. This structure may not be obvious, and even if it were, committing to some specific structure could unnecessarily restrict the implementation.

Instead, we chose to build a customised domain-specific query language which encapsulates the underlying structure of the database such that we can hide context queries including joins and sub-selects from the end user. For simplicity, we implemented the domain-specific query language as a data term that is translated to an SQL statement at runtime. To regain some of the type safety of an SQL statement, we implemented this term as a Generalised Algebraic Data Type (GADT) [10] parametrised on the type of the result of the query:

```
data Query r where
  Values     :: Result a ⇒ Name → Query [a]
  Trials     :: Query [TrialInfo]
  Sessions   :: Query [SessionInfo]
  InTrials   :: Result a ⇒ [Int] → Query [a] → Query [a]
  InSession  :: TrialQuery a ⇒ Int → Query [a] → Query [a]
  HasResult  :: TrialQuery a ⇒ Name → Query [a] → Query [a]
  Where      :: (TrialQuery a, Result b) ⇒
                Name → Oper → b → Query [a] → Query [a]
  SpecLike   :: String → Query [a] → Query [a]
data Oper = LessThan | GreaterThan | Equals | Like
```

[10] However, this implementation was facilitated by having the outcomes of an experiment wrapped in a type *AnyResults* that we know how to serialise to a relational database. In that sense, Braincurry is not as loosely coupled as we would have liked.

Queries of this type permit the retrieval of either stored, named results (the *Values* constructor), metadata pertaining to the trial (the *Trials* constructor: trigger time, experiment description), or the session (the *Sessions* constructor: start time, number of trials, name). Queries formed by these base constructors may then be restricted to results that belong to specific sessions (*InSession*, parametrised on the session identifier), a subset of the trials in a session (*InTrials*, that takes a list of indices of the trials in session), trials during which a result with a certain name has been stored (*HasResults*), trials where a named result fulfils a predicate (*Where*), or trials in which the serialised experiment description matches a certain regular expression (*SpecLike*).

Query terms are translated directly into SQL and results cast back to Haskell through methods in the *Result* class. This process is entirely hidden from the user: we simply provide a top-level function $ask :: Query\ a \rightarrow IO\ (Maybe\ a)$.

Our typed, domain-specific query language gives a simple mechanism for retrieving results from a moderately complex database without requiring the end user to understand sub-selects or joins. The main limitation is that each result must be retrieved separately; i.e., there is no term for retrieving a pair of results. A possible extension, at the expense of a more complicated implementation, would be a constructor $Zip :: Query\ [a] \rightarrow Query\ [b] \rightarrow Query\ [(a,b)]$.

1.7 EVALUATION AND STATUS

The system as presented has been implemented and used to run successful experiments in our laboratory. Thus far, Braincurry has not imposed any constraints on the experiments we have wanted to do, performance has been adequate, and we have been able to set up entirely new experiments very quickly. We have implemented five different non-trivial experiments in Braincurry that include visual and auditory stimulation and numerical simulations. These experiment definitions are *written* by one person but have been *read* by several non-Haskell programmers. In addition, we have used Braincurry definitions to present new experimental results to neuroscientists [9], thus validating the use of Braincurry as a medium for scientific communication. All in all, Braincurry has proved to be a practically very useful tool for describing and running experiments and simulations. The source code is available at `http://github.com/glutamate/braincurry/`.

A significant limitation in the design of Braincurry is the necessity of building new interpreters for every model or experimental setup. Although we have sought to decrease the amount of work required to do this, setting up Braincurry in a new laboratory does require nontrivial knowledge of Haskell. This difficulty can be alleviated by sharing interpreters between labs. For example, code for frequently used apparatuses could be made available on-line. Relying entirely on prefabricated apparatuses does remove a significant amount of power from the experimenter. However, such apparatuses can be made very general, even if this means making the trial options datatype fixed. The advantages offered by using combinators and functional abstraction in defining experiments remain.

Braincurry is a very general system for scheduling experiments, which can be

adapted to many different types of experiments both in neuroscience and in other fields. However, this generality comes at the cost that the experiment description does not reveal the semantics of the experiment, which relies on the implementation of the interpreters on which the experiment description is intended to run. This makes it quite hard to choose an analysis method based on the described experiment. Our future work will focus on adding some structure to the experiment description while retaining generality.

1.8 RELATED WORK

There is substantial overlap between our approach and the "robot scientist" investigating yeast genomics [8]. The principal difference is that the robot scientist includes assumptions about the underlying biology that allow it to autonomously form and test hypotheses, although limited to relatively simple experimental paradigms. Our system aims to be more general by accommodating many different experimental techniques. However, this generality makes it much more difficult to generate new hypotheses from previous data. Our future research is likely to address the need for a compromise between generality and functionality. We note that large parts of the robot scientist are also written in a declarative language (Prolog). However, it does not appear to be the goal of the robot scientist project itself to develop new notations for describing experiments. For this purpose, separate semantic web ontologies have been proposed [12]. But it is not clear that the subject-predicate-object triples underlying the semantic web are suitable for describing the experiments conducted in neurophysiology. In the approach taking by the Braincurry language, the ability to describe experimental *processes* is critical. If one views experiments and simulations as *programs*, they may be described more concisely and with greater modularity in a *programming language*, where it is possible to give a name to recurrent patterns of scientific procedure.

1.9 CONCLUSIONS

The ease by which Haskell supports experimental language design and the usability of the resulting language have shortened the cycle for implementing and assessing new language features. Although Braincurry is at present somewhat limited in scope, we have a design which is sufficiently high-level to serve as a descriptive specification of the parts of an experiment that can be mechanised, yet is much more flexible than what could be achieved by just selecting options from a menu and providing specific values for parameters. Moreover, thanks to the compositional design, it should be relatively straightforward to enlarge the scope of Braincurry towards experimental techniques that we have not used in the present study. Future work will address some of the limitations of Braincurry including the difficulty in describing new components of an experiment, and will address post-acquisition analysis.

In common with embedded domain-specific languages in general, there is arguably a bit of "syntactic embedding noise" that could limit the appeal of Brain-

curry to users who are chiefly interested in using a ready-configured Braincurry instance in a specific lab setting. However, with some implementation effort, it should be relatively straightforward to support such users by providing a more polished surface layer. What is important, and the focus of this paper, is that Braincurry is a proper language, the design principles of this language, and the flexibility and generality that ensues from this and that ultimately benefits all users.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their thorough and constructive feedback that helped improve the paper, and Steve Rogers for the use of the picture of a locust. This research was supported by a Human Frontier Science Program Long-Term Fellowship and the BBSRC.

REFERENCES

[1] M. Burrows. *The Neurobiology of an Insect Brain.* Oxford University Press, 1996.

[2] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, 2003.

[3] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of ICFP'97: International Conference on Functional Programming*, pages 163–173, June 1997.

[4] F. Gabbiani, H. G. Krapp, and G. Laurent. Elementary computation of object approach by a wide-field neuron. *Science*, 270:1000–1003, 1995.

[5] F. Gabbiani, H. G. Krapp, and G. Laurent. Computation of object approach by a wide-field, motion-sensitive neuron. *Journal of Neuroscience*, 19(3):1122–1141, 1999.

[6] W. Gerstner and W. M. Kistler. *Spiking Neuron Models.* Cambridge University Press, 2002.

[7] P. Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

[8] R. D. King, K. E. Whelan, F. M. Jones, P. G. K. Reiser, C. H. Bryant, S. H. Muggleton, D. B. Kell, and S. G. Oliver. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, 427:247–252, Jan. 2004.

[9] T. A. Nielsen, H. Nilsson, and T. Matheson. New eyes on visual habituation in locust: an experiment description language for integrative neuroscience. Poster T14-7C presented at Göttingen Meeting of the German Neuroscience Society, Mar. 2009.

[10] S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389, July 2004.

[11] T. Sheard and E. Pasalic. Two-level types and parameterized modules. *Journal of Functional Programming*, 14(5):547–587, 2004.

[12] L. N. Soldatova, W. Aubrey, R. D. King, and A. Clare. The EXACT description of biomedical protocols. *Bioinformatics*, 24(13):295–303, 2008.