def Gribber = (Grace + Gibber)

Timothy Jones
Victoria University of Wellington, NZ
tim@ecs.vuw.ac.nz

James Noble
Victoria University of Wellington, NZ
kjx@ecs.vuw.ac.nz

ABSTRACT

Grace is a new object-oriented education programming language that we are designing. One of the Grace implementations, Hopper, is an interpreter that runs on top of JavaScript. Gibber is series of libraries that support real-time audio processing, and also a simple livecoding interactive development environment, also on top of JavaScript. In this demonstration, we will present Gribber, a web-based IDE that incorporates the Hopper interpreter into the Gibber IDE, and so gives Grace live access to the Gibber audio libraries. Because of Hopper's continuation-passing design, Gribber programs can appear multi-threaded, apparently running one or more loops in parallel, each pseudo-thread generating sound and graphics in simultaneously. We will demonstrate how Gribber can be used for *BALSA*-style algorithm animation, and *Sorting Out Sorting* style algorithm races.

1. Introduction to Grace

We are engaged in the design of Grace, a new object-oriented open source programming language aimed at instructors and students in introductory programming courses (Black et al. 2012; Black et al. 2013; Homer et al. 2014). Grace aims to include features that have been found useful in software practice, while allowing multiple different teaching approaches without requiring that concepts be introduced to students before they are ready.

While many aspects of Grace's design will be familiar to most object-oriented programmers — we hope Grace's syntax, declarations, control structures, and method semantics appear relatively conventional — there are some aspects of Grace's design that are unique. In particular, Grace tries to separate the concepts of *object*, *class*, and *type*. In Grace, objects can be created without classes (like JavaScript or Self), or with classes (like most other object-oriented languages). Classes in turn can be built *by hand* out of objects (like Emerald, if anyone's counting). Grace types are optional (gradual): types in Grace programs may be left out all together (like Python or JavaScript), always included (like Java or C♯), or used in some mixture (like Dart or Dylan).

To give the flavour of the language, we show the Grace code for an imperative Insertion Sort, as may be found in many introductory programming course sequences. Hopefully this example is comprehensible to readers from a wide range of programming backgrounds:

```
method insertionSort(array) {
  var i := 2
  while {i <= array.size} do {
    var j := i
    while {(j > 2) && (array.at(j-1) > array.at(j))} do {
       array.swap(j) and(j-1)
       j := j - 1
       }
    i := i + 1
  }
  print "done insertion"
}
```

The first Grace implementation, Minigrace (Homer 2014) was a self-hosted prototype compiler to C and JavaScript. More recently, Jones has built Hopper (Jones 2015), a continuation-passing style interpreter written in JavaScript.

2. Livecoding in Grace

As well as being artistically and philosophically interesting in its own right (Blackwell et al. 2014), livecoding promises to increase engagement and learning in Computer Science pedagogy — indeed pedagogy has motivated much of the early work in live programming of many kinds (Papert 1980; Goldberg and Robson 1983). More recently, projects such as SonicPi (Aaron 2015), for example, use livecoding both to increase student engagement in programming, and also to support learning; indeed there is some evidence that livecoding can increase both engagement and learning even when it replaces static slide shows of the same content (Rubin 2013).

Unfortunately, our existing first Grace programming environment prototypes did not support livecoding. While Minigrace could self-host in JavaScript in a browser, it ran only in batch mode: code could not be modified on the fly, and loops (e.g. sort methods) would lock up the browser until they completed (Homer 2014). This is a problem for introductory programming courses where sorting is a common topic.

3. Grace in Gibber

We were able to solve these problems by combining our second-generation Hopper interpreter (Jones 2015) with the Gibber livecoding environment (Roberts and Kuchera-Morin 2012; Roberts et al. 2014). Gibber provides powerful audio and video libraries, and an IDE particularly suited to livecoding that can be easily extended to support different languages. JavaScript livecoding in Gibber unfortunately suffers from the problem that loops in programs will effectively delay the browser until they complete — this is a problem of JavaScript's single-threaded event-based programming model: each event must run to completion before the next event can begin execution. For e.g. livecoding and visualising a sorting algorithm, we need to visualise each interesting event in the algorithm as it happens, rather that just flick from an unsorted to a sorted array in a single time step.

The Hopper interpreter solves this problem by running Grace programs in a continuation-passing style. Each Grace subexpression (*method request* in Grace terminology) is executed one step, and then control is returned to the main loop, with the next step reified as a continuation (Abelson and Sussman 1985). After a timeout to allow other events, browser (or Gibber tasks), or indeed other Grace expressions to run, the delayed continuation will be picked up, again executed one step, again reified as a continuation and control yielded again. The effect is quite similar to SonicPi's use of Ruby's threading (Aaron and Blackwell 2014) except implemented solely in an interpreter written in JavaScript: the continuation passing and time slicing is not visible to the Grace programs. This allows Gribber to support *BALSA*-style algorithm animation (Brown 1988; Brown and Hershberger 1991), and *Sorting Out Sorting* style algorithm races (Baecker and Sherman 1981), which cannot be supported in Gibber in naked JavaScript. Figure 1 shows a Grace insertion sort running under Gribber in the Gibber IDE.

4. Conclusion

We have combined the Grace Hopper interpreter with the Gibber livecoding environment to produce Gribber — a livecoding environment for the Grace educational programming language. We hope Gribber will help communicate the advantages of livecoding to the educational programming languages community, and the advantages of a clean educational programming language design to the livecoding community.

5. Acknowledgements

We thank the reviewers for their comments. This work is supported in part by the Royal Society of New Zealand Marsden Fund and James Cook Fellowship.

References

Aaron, Sam. 2015. "π))): Sonic Pi." `http://sonic-pi.net`.

Aaron, Sam, and Alan Blackwell. 2014. "Temporal Semantics for a Live Coding Language." In *FARM*.

Abelson, Harold, and Gerald Jay Sussman. 1985. *Structure and Interpretation of Computer Programs.* MIT Press; McGraw-Hill.

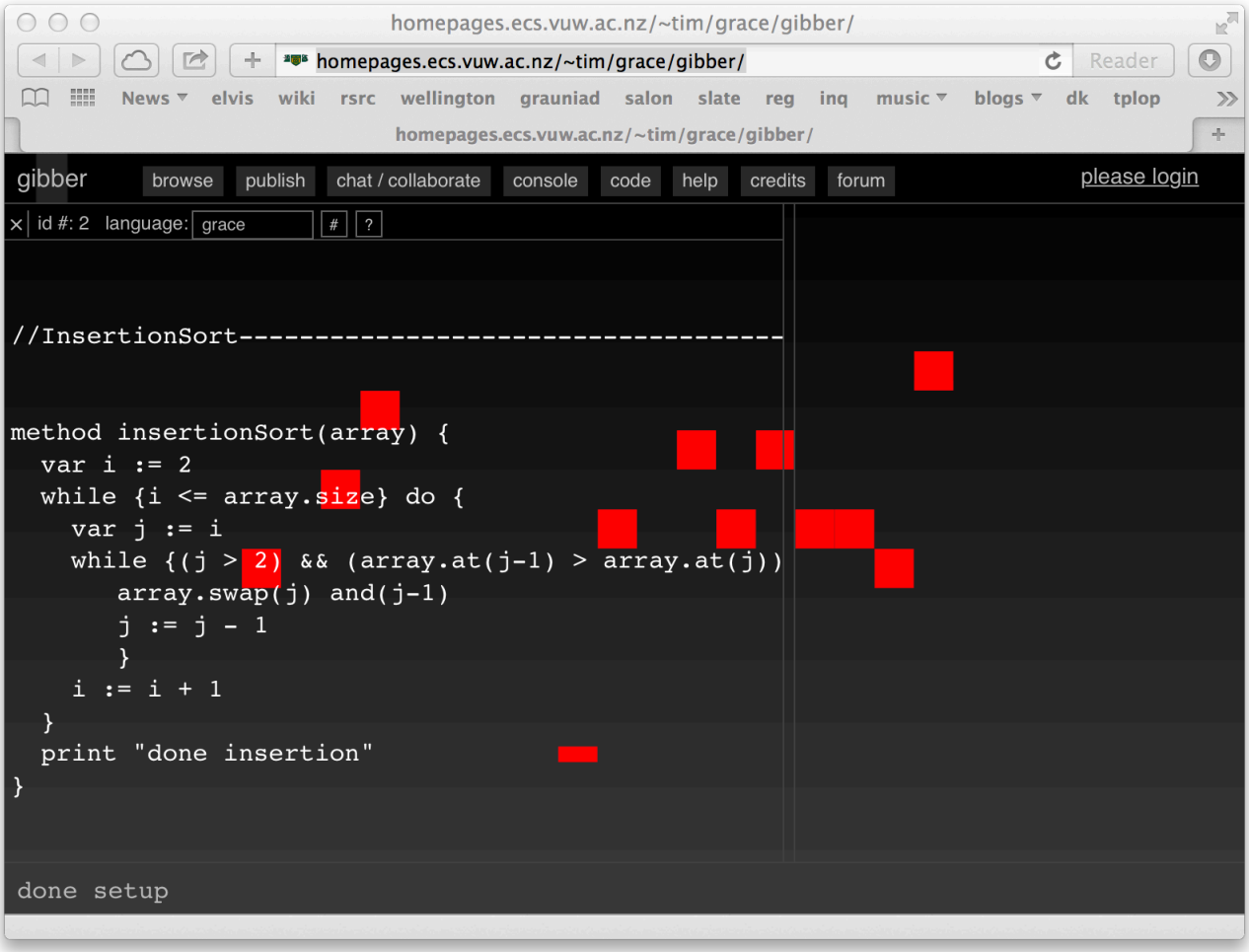

Figure 1: *A Grace insertion sort running under Gribber in the Gibber IDE. Note the translucent squares visualising element values: these are updated as the algorithm runs.*

Baecker, Ronald M., and David Sherman. 1981. “Sorting Out Sorting.” 16 mm colour sound film.

Black, Andrew P., Kim B. Bruce, Michael Homer, and James Noble. 2012. “Grace: the Absence of (Inessential) Difficulty.” In *Onward!*, 85–98.

Black, Andrew P., Kim B. Bruce, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. 2013. “Seeking Grace: a New Object-Oriented Language for Novices.” In *SIGCSE*, 129–134.

Blackwell, Alan, Alex McLean, James Noble, and Julian Rohrhuber. 2014. “Collaboration and Learning Through Live Coding (Dagstuhl Seminar 13382).” *Dagstuhl Reports* 3 (9): 130–168.

Brown, Marc H. 1988. *Algorithm Animation.* ACM Distinguished Dissertation. MIT Press.

Brown, Marc H., and John Hershberger. 1991. “Color and Sound in Algorithm Animation.” *IEEE Computer* 25 (12) (December).

Goldberg, Adele, and David Robson. 1983. *Smalltalk-80: the Language and Its Implementation.* aw.

Homer, Michael. 2014. “Graceful Language Extensions and Interfaces.” PhD thesis, Victoria University of Wellington.

Homer, Michael, Timothy Jones, James Noble, Kim B. Bruce, and Andrew P. Black. 2014. “Graceful Dialects.” In *ECOOP*, 131–156.

Jones, Timothy. 2015. “Hop, Skip, Jump: Implementing a Concurrent Interpreter with Promises.” Presented at `linux-conf.au`.

Papert, Seymour. 1980. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, Inc.

Roberts, C., and J. Kuchera-Morin. 2012. “Gibber: Live Coding Audio in the Browser.” In *ICMC*.

Roberts, C., M. Wright, J. Kuchera-Morin, and T Höllerer. 2014. “Gibber: Abstractions for Creative Multimedia Programming.” In *ACM Multimedia*.

Rubin, Marc J. 2013. “The Effectiveness of Live-Coding to Teach Introductory Programming.” In *SIGCSE*, 651–656.