Programming languages for interactive computing

Roly Perera
Dynamic Aspects, Bristol, U.K.

March 12, 2007

Abstract

Traditional programming languages are *algorithmic*: they are best suited to writing programs that acquire all their inputs before executing and only produce a result on termination. By contrast most applications are *interactive*: they maintain ongoing interactions with their environments. Modern systems address this incompatibility by manually extending the execution model of the host language to support interaction, usually by embedding an event-driven state management scheme which executes fragments of imperative code in response to interactions, the job of each executed fragment being to restore the internal consistency of the computation. The downside of this approach to interaction is that it relies heavily on mutable stores and side-effects and mixes application logic in with behaviour which is more properly the responsibility of an execution model. I describe a programming model called **declarative interaction** which supports interaction directly. The distinguishing feature of the model is its *modal* construal of state and interaction.

1 Declarative interaction

Almost all software systems today are *interactive*, in that they maintain ongoing interactions with their environments, rather than simply producing a result on termination [GSW06]. Indeed, a consistent trend since the beginning of the digital era has been towards increasingly interactive systems. The trend has progressed on at least two fronts: enhancements to end-user interactivity, and increasingly integrated systems. The trend began with the first teletype and textual interfaces and continued through early GUIs and LAN-based operating systems. It continues with today's 3D virtual worlds and applications deployed over the wide-area network.

With the Internet now emerging as the "global operating system", the pressure on our software to be interactive is greater than ever before. Consider how the following requirements can be understood in terms of enhanced interactivity:

- Ability to reconfigure or repair applications without taking them offline → *interaction with code as well as data*
- Long-running, continously-available applications → *interaction must be robust*
- Sessions resumable from wherever we happen to be located → *persistence of interactions*
- Transparent recovery from latency problems and intermittent connectivity → *interaction should not be semantically sensitive to the network*
- Mobile code whose behaviour depends on execution context → *dynamically scoped interactions*

A variety of process algebras and other formalisms have been developed for modelling and reasoning about interactive systems. Yet despite the trend towards greater interactivity, we continue to lack a simple and coherent paradigm for building robust interactive systems. The main obstacle has been what we might characterise as an "impedance mismatch" between traditional algorithmic programming languages and the way interactive systems abstractly work. Whereas an algorithmic language treats a program as a black box which produces a final value on termination, an interactive system allows other systems to observe and influence its behaviour as it executes, and must adjust its internal state in response to each interaction to maintain the consistency of the computation.

In current desktop systems, the mismatch is usually resolved by representing the state of the system as a set of mutable stores and then employing a notification scheme to maintain the consistency of the state. Rather than the host language being used to execute a single sequential program to termination, it is employed to execute fragments of imperative code as interactions occur. Each executed fragment must produce exactly the side-effects required to synchronise the state of the system correctly.

Unfortunately this near-universal reliance on the imperative to support interaction has come at an enormous cost in complexity and reliability. It will be useful here to recall Brooks' distinction between *essential* and *accidental* (or *inessential*) complexity [Bro87]. Complexity inherent in the problem itself (from a user's perspective), or which can be attributed to human factors, is essential; what remains is accidental or inessential. Interactive systems as currently implemented are dominated by inessential complexity, the main culprits being this *ad hoc* management of state, explicit concern with flow of control, and unnecessary use of non-deterministic concurrency [MM06]. Imperative languages [ab]used in this way are the "Turing tar-pit in which everything is possible but nothing of interest is easy" [Per82]. Web applications only complicate things further, by adding a variety of ill-defined staging policies and interaction models into the mix.[1]

Ironically, what unifies much of this inessential imperative complexity is that it exists in the service of *essentially declarative ends*. Indeed, I suggest that the reason the current paradigm works at all is that systems implemented in this way approximate a much simpler, declarative model of interactive computation. Experienced software designers rely tacitly on something like this declarative model in order to make decisions about what counts as reasonable behaviour. Rather than being ill-suited to interaction, as is sometimes assumed, perhaps because of the somewhat arcane feel of techniques such as *monads* [JW93] for modelling effectful computation, the declarative paradigm – with a simple orthogonal extension to support interactivity – fits the abstract behaviour of interactive, stateful systems surprisingly well. But today, because this declarative behaviour is achieved only indirectly, interactive systems are significantly less reliable and harder to reason about than they need to be, despite the widespread use of design patterns such as *Observer* [GHJV95] to manage some of the inessential complexity. State-related dysfunctions in particular, such as the gradual degradation of a system over time, spurious sensitivities to the order in which two operations take place, deadlocks and race conditions, are common.

I propose that the best way to address the impedance mismatch between algorithmic languages and interactive systems is not to wallow in the tar-pit of imperative languages, but to lift declarative languages into a computational model which supports interaction directly. The aim of this paper is to set out the conceptual foundations of such a model, which I shall refer to as the **declarative interaction** model. No new results are presented; instead the paper attempts provide the philosophical framework and motivation for further research. Future work will formalise various aspects of the model.

Like some other declarative approaches to interaction, the proposed model maintains a clean separation of (stateless, deterministic) *declarative* computation and (stateful, non-deterministic)

[1] Several projects, such as Links [CLWY06], aim to address this impedance mismatch specifically for Web applications.

reactive computation. The model is distinguished by its *modal* construal of interaction, whereby an interactive system is taken to be a space of canonically represented "possible worlds", each expressing a purely declarative computation, along with an index into that space indicating the "actual world", which represents the system's current *state*. To interact with such a system is to select a new *actual* state from this space of possible states. Interactions (effectful operations) are lifted to *meta-programs* that manipulate values of modal type; crucially, they are unable to interfere with the purely declarative semantics of each possible state, and in any case are only required when they are essential to application logic. Non-determinism arises from concurrent interactions, which are handled transactionally.

The declarative interaction model relates to many active research areas, including modal type systems, incremental computation, meta-programming, declarative concurrency, transactional concurrency, dataflow computing, interactive computing, and wide-area computing. As we shall see, four closely related concepts in particular are central to the model: modality (§2), incrementality (§3), concurrency (§4), and persistence (§5). Although envisaged as a multi-paradigm framework within which to study various aspects of interactive programming, rather than a single programming language, a key validation of the model will be the development of an **interactive programming language** (§6) which supports the model directly. Such languages will make it possible to build software systems which are intrinsically interactive.

2 Modality

The central feature of the declarative interaction model is its *modal* interpretation of state and interaction: it construes states as possible worlds and interaction as navigation between possible worlds. I will now try to make good my claim that most interactive systems can be understood as abstractly realising such a model, by considering two kinds of interaction.

The first kind of interaction is typified by the act of typing a single word into a word processor by simply typing the individual letters which make up the word, in the correct order – such as entering the word "at" by first pressing the *a* key, and then the *t* key. What is significant here is that the *first-order state* of the system evolves as a monotonic function of the interaction history. By "first-order" state, I mean the *least sequence of interactions* required to construct the system from the null structure. The importance of this algebraic view of state will hopefully become clear, but for now we need only note that what characterises this kind of interaction is *monotonicity*: new interactions are never interpreted as the undoing or replacement of earlier actions.

Monotonicity allows the system to have a straightforward declarative interpretation. Although the system certainly has something of a stateful "feel" – inputs are provided and the system responds – over the total input sequence, the system is stateless. *Abstractly* this is so, even though such a system would today probably be implemented using mutable stores, non-deterministic concurrency and explicit synchronisation. This simple kind of interaction, and its declarative interpretation, is the basis of the *dataflow* model of declarative concurrency [Col04], and can be used, for example, to implement a simple stream-based producer-consumer style interactive system purely declaratively. The interactive nature of the system is simply a property of the execution model, which allows the computation to proceed in a stepwise fashion, before the input is fully known, and intermediate states of the computation to be observed. These intermediate states of the system are sometimes known as *partial terminations* [VH04].

The class of interactions we have considered so far is rather trivial, being restricted to just those for which the state of the system evolves monotonically. To describe the richness of interactive systems in general, we need to allow *retroactive modification*, rather than just monotonic evolution, of the first-order state. Here, the system is able to interpret new interactions as the deletion or insertion of actions at (up to consistency constraints) arbitrary positions in the canonical sequence

representing the first-order state. The effect of such an edit is that of undoing back to the point of modification, deleting or inserting actions as required, and then redoing all the undone actions.

The significance of this more general kind of interaction is that systems typically have both invertible (undoable) and commutative operations, in particular when their first-order state can be partitioned into orthogonal components. Under such circumstances there are multiple interaction sequences that lead to the same first-order state. Returning to our word processing example, consider the various sequences of key presses that can result in the insertion of the word "at" (leaving the caret just to the right of the t): $\langle a, t\rangle$, $\langle t, leftArrow, a, rightArrow\rangle$, $\langle a, backspace, a, t\rangle$, and so on. Each of these interaction sequences must produce the same first-order state. But unless its first-order state can be retroactively modified, the system cannot maintain a canonical representation of that state when invertible operations are undone or commutative operations are re-ordered.

It is here that a *modal* construal of state allows the system to retain a declarative interpretation. A retroactive modification is interpreted thus: as a sideways step into a *counterfactual* state of affairs or possible world where a different canonical sequence of actions obtains, namely the nearest one which accommodates the required modification. The new state is identical with the one which would have resulted had the interaction sequence simply been different in the first place, and the declarative purity of individual states remains isolated from the interactions which navigate between states. Retroactive edits actually include the first kind of interaction as a subcase, since monotonically extending the input is just one way of arbitrarily editing it.

The declarative interaction model in effect treats an interactive system as what is sometimes called a *retroactive data structure* [DIL04]. A retroactive data structure is similar to a *fully persistent* data structure, a mutable data structure which provide access to, and modification of, any version (state) of the structure [DSST86]. Whereas modification of a prior version of a fully persistent data structure always creates a new branch of history, retroactive data structures allow *alternate* histories to be created for the current version through the insertion or deletion of past actions, such edits being construed modally as sideways steps into nearby states. Retroactive data structures are closely related to dynamisation and adaptive computations; for example no general technique exists for turning any data structure into an efficiently retroactive counterpart.

The effect of the modal view of interaction is to separate interaction from purely declarative computation by lifting it into an earlier *stage* of the computation. Interactions are, in effect, *meta-programs* which treat declarative computations as data. A key consequence of this staging is *non-interference*: it is not possible to infer the precise history of interactions from the first-order state, since it only represents their net effect. This is a strong desirable for many security-sensitive applications [HHM+02].

A close affinity between interaction, meta-programming and modality is also suggested by recent work in modal type systems. Modal types, which were first studied in relation to staged computation [DP96], can, like monads, be used to capture properties of the interactions a computation has with its environment. They typically feature two type constructors that correspond to the necessity and possibility operators of modal logic. The constructor $\Box$ (box) is a universal quantifier: $\Box A$ holds in the current world iff A is *necessary*, i.e. holds in *all* the worlds. The constructor $\Diamond$ (diamond) is an existential quantifier: $\Diamond A$ holds in the current world iff A is *possible*, i.e. holds in *some* world. One can understand *worlds* as standing for *states* of the environment; the relation to meta-programming is that the substitution of code into a syntactic *context* corresponds to interaction with an environment, with the syntactic context as the environment, and the capture of free variables as the interaction [Nan02]. A computation with precondition P can be assigned a bounded universal type $\Box_P A$, indicating that it can execute in all states satisfying P (in all contexts in which the name bindings described by P are available). Dually, a computation with postcondition Q can be assigned a bounded existential type $\Diamond_Q A$, indicating that it yields some state satisfying Q (some context in which the name bindings described by Q are introduced). Although closely related

to monads [Kob97], modal types are able to classify certain kinds of interaction more precisely [Nan03], and will therefore probably provide a suitable type-theoretic foundation for the proposed model.

3 Incrementality

The job of an interactive system is to reflect a consistent interpretation of the interaction history at all times. This emphasis on *maintenance of an interpretation*, rather than *computation of a final value*, suggests that we think of interactive systems as inherently *incremental* computations. Rather than running from the beginning through to termination to produce a final result, such systems move between *partial* terminations representing their interpretation of particular interaction sequences. We can legitimately describe this behaviour as incremental because – abstractly at any rate – the system merely adjusts its state, rather than re-computing it *ab initio* (from scratch).

Concretely of course things are not so simple. Modern interactive systems realise this kind of abstractly incremental behaviour by utilising a complex "state patching" scheme to synchronise the internals of the computation, usually only updating a small subset of system state, rather than refreshing the entire system *ab initio*. But such manual schemes, which are reliant on mutable stores and side-effects, are major sources of complexity and programmer error. Care must be taken to patch the state correctly, and, when the effect is intended to have some kind of dynamic extent, to restore the previous state afterwards. Crucially, essential application logic is lost in a morass of state-management machinery.

This points to the real problem with the state patching approach: its reliance on imperative code to achieve manually what is more properly the responsibility of an incremental execution model which directly supports interaction. The foundation of such a model is the modal view of interaction outlined in the previous section, where the *states* of interactive systems are construed as individual declarative computations, and *interaction* (mutation of state) as navigation between these computations. When interactions navigate between "nearby" states, many sub-computations of those states can clearly be shared. The execution model therefore need only evaluate those parts of the new state which are not shared with the old.

Utilising such an execution model directly frees us of the need to write the state management code. Interaction causes the state of the computation to adjust automatically in the manner of a dataflow computation or spreadsheet. Perhaps most significantly, we can write programs directly in a declarative style, and rely on the runtime to provide interactivity.

The problem of efficiently adjusting a computation to input changes presents several challenges, and has been studied extensively. One of the more interesting recent efforts, due to Acar, Blelloch and Harper, is so-called *self-adjusting computation* [ABH04], which combines two techniques: function caching or *memoisation* [Mic68], and change propagation based on dynamic dependence graphs, a development also due to Acar *et al.* Self-adjusting computation is interesting because both memoisation and change propagation align closely with our modal notion of incrementality. Memoisation corresponds to the persistence, across changes, of declarative computations, allowing the parts of a computation which are unaffected by a change to be re-used. Change propagation is the dual concept, corresponding to navigation between those declarative computations, causing those parts of the computation which are affected by a change to be evaluated. Memoisation is "top-down" and handles "shallow" interactions well; change propagation is "bottom-up" and handles "deep" interactions well. Combined correctly they can handle both shallow and deep changes efficiently. The main shortcoming of self-adjusting computation is that it falls short of being a purely run-time transformation; non-trivial effort on the part of the programmer is required to transform a standard declarative program into a self-adjusting program. Strict and non-strict arguments, for example, must be explicitly distinguished.

4 Concurrency

Interactive systems process multiple inputs and generate multiple outputs simultaneously, and are therefore inherently concurrent. However, the dominant concurrency paradigm today – sequential threads sharing mutable state – is the source of much of the inessential complexity in modern software systems. One of the main problems is the widespread use of non-determinism in the service of essentially deterministic ends [Lee06][Col04]. This is more of a concern than ever because current trends in microprocessor design suggest that the next major advances in hardware performance will come from increasingly parallel architectures – "multicore" or chip multiprocessors (CMPs) [Cre05]. We face a difficult challenge: translating this increased CPU bandwidth into practical performance, whilst simultaneously making interactive systems less complex. This will only be feasible if we can make languages more inherently concurrent, shifting the complexity burden from application developers to language implementors.

To achieve this our computational model needs to distinguish, and support, two kinds of concurrency[2]:

1. *Deterministic* or *declarative* concurrency [VH04] which has no observable non-determinism and preserves referential transparency.

2. *Non-deterministic* concurrency, which involves the deliberate and controlled mutation of shared state, and which is semantically visible, in the form of non-determinism.

Both kinds of concurrency are critical to the delivery of robust, concurrent interactive systems. Modality once again offers a unifying conceptual framework.

The simplest kind of declarative concurrency is that offered by the standard producer-consumer style dataflow model described in §2. The model allows the execution of the code which consumes the list to be interleaved with the execution of the code which computes the tail of the list. Whereas a typical implementation today would probably involve explicitly synchronised threads accessing a shared mutable list, the synchronisation is *implicit* in the declarative model, and the "mutability" of the list is nothing more than the visibility of intermediate states. Data dependencies ensure a deterministic outcome regardless of how the computation is initiated.

Declarative concurrency is not restricted to a single producer or single consumer. Graphical user interfaces for example in effect display multiple output streams simultaneously, each driven by an independent demand [HC94]. Such systems can be understood as the *parallel composition* of a number of (perhaps similar) declarative computations. Each concurrent computation conceptually induces a distinct demand chain. When these computations overlap, such as when the computation of the colour of one pixel overlaps the corresponding computation for an adjacent pixel, the composite process executes incrementally, albeit in a concurrent rather than serial sense. Once again data dependencies ensure a deterministic outcome.[3]

Non-deterministic concurrency is of a quite different character, involving the mutation of shared state by concurrent interactions. Specifically, non-determinism arises from the need to *merge* concurrent interaction streams. Recall that in the proposed model, a stateful or "reactive" process is a mutable slot storing an index into the space of possible declarative computations. The value of the index indicates the current state of the process. Such processes represent *versions* or *views* of interactive systems which are deemed "authoritative" from the vantage point of some external agent.

[2]The distinction is roughly the one drawn by Peyton Jones *et al.* between *implicit* and *explicit* concurrency [JGF96].

[3]This generalised dataflow model is closely related to so-called *intensional* programming languages like Indexical Lucid and Multidimensional Lucid [Pla00], which also have a "possible worlds" semantics and a demand-driven execution model based on concurrent dataflow.

To interact with a stateful process is to navigate through the space of declarative computations and select a new one as its current state. When *multiple* interaction streams arrive simultaneously at the same reactive process, these concurrent streams in effect represent alternative "future histories" of the reactive process: hypothetical ways in which its state might subsequently evolve. Merging these alternate timelines into a single authoritative history sometimes requires a mechanism for arbitrarily interleaving them. This process of arbitration is the only respect in which non-determinism features in the declarative interaction model.

Sometimes there are consistency constraints on the allowable states of a reactive process which restrict the ways in which concurrent interaction streams can non-deterministically interleave. In particular, certain interaction sequences may only be applied atomically. This is of course the familiar notion of a *transaction*. A contribution of declarative interaction is that the execution model naturally supports transactions. Transactional concurrency [ST95] is already simpler than traditional lock-based approaches to concurrency, avoiding a number of problems usually associated with locks, such as non-compositionality and the possibility of deadlock and priority inversion [HMPJH05]. But modality offers an even simpler conceptual framework for transactions than the standard one, as well as a potentially more efficient implementation. Transactional memory standardly employs a so-called "optimistic" synchronisation policy. Rather than acquiring a lock, a transaction accumulates a local log of its state reads and writes. When the transaction completes, its log is *validated*, which involves checking that any state which was read by the transaction has not been modified in the interim by another transaction. The transaction is only committed if validation succeeds; otherwise it is re-executed *ab initio*, generating a new transaction log.

Because the ontology of "possible worlds" already supports multiple branching histories, there is no need to generate transaction logs or verify transactions. Instead one simply selects the right version or state of the interactive process – the one in which the desired sequence of events took place. "Committing" or applying a particular transaction is implicit in the act of navigating to the intended version. The challenge for implementors is to devise efficient strategies for speculatively generating alternative hypothetical "future histories" of the system, preferring timelines that are most likely to result from actual interactions, and uncaching those deemed unlikely to be revisited.

The potential efficiency gain comes from incrementality. Although each version or state has the advantage of being a purely declarative computation which executes in semantic isolation from the others, when some sub-sequence of one of the competing timelines is "committed" (selected as the authoritative version of events), it does not generally invalidate the other pending transactions, although they now apply to an obsolete version of the system. They need only be incrementally adjusted to reflect the version of the system to which they are eventually applied, rather than re-executed *ab initio*.

So under the proposed model, the essence of reactive computation is this higher-order activity of merging the alternate timelines implied by concurrent interaction streams into consistent authoritative histories. Since declarative computation is isolated from reactive computation through *staging*, as we saw in §2, the semantics of the language can be partitioned into deterministic and non-deterministic components, and the separation enforced through a suitable type system.[4] Such a stratified approach has been taken in languages like Concurrent Haskell [JGF96] and O'Haskell [NC97], albeit with monadic, rather than modal, type systems enforcing the separation.

[4]In the distributed computing literature, this is sometimes described as the separation of *computation* from *coordination*.

5 Persistence

Transactions are more than just about safely co-ordinating changes to shared state: they are also about managing the *granularity* of those changes. The significance of this should become clear if we consider what Cardelli has called *wide-area* computing [Car99], interactive computation on wide-area networks. In a truly "connected" world, any computation is potentially wide-area, as software can be connected to other systems located anywhere in the global network. Wide-area computing is distinguished by the *unbounded latency* problem. Since there is no practical upper bound to communication delays, a client system cannot in general distinguish a long delay from a network failure or indeed from voluntary disconnection.

Voluntary disconnection in particular is important because it also occurs in the local-area network. Think of a query or view that is invalidated as soon as the user makes a change to the system, and must be manually refreshed. Refreshing such a view is like reconnecting, synchronising the state of the view, and then disconnecting again. Such disconnected views are common in part because of the very problem the declarative interaction model addresses: the need to manually write the code responsible for synchronisation. But there are in fact good reasons why some views *require* coarse-grained synchronisation.

For a start there are usability considerations: often we want to look at a snapshot rather than a continuously evolving view. But also recall that interactive systems are (adopting Wegner's terminology) *open*; external systems can observe and influence their behaviour [Weg98]. It is legitimate therefore to think of interactive systems as *proper parts* of larger systems, which may themselves be open or closed.[5] Not only can single systems be arbitrarily large, but they can grow and shrink dynamically as dependent computations – observers – are attached and detached. And since these observers can be arbitrarily complex, it clearly must be possible to temporarily disconnect observers and resynchronise them when required, rather than require them to continuously update.

These concerns suggest that even in a local-area environment, flexible synchronisation policies will be required long after we stop having to write the code to do the actual synchronisation. Performance advances will not make this issue go away either, since it seems safe to assume that as computers become more powerful, we will simply invent more computationally demanding uses for them. There is therefore a general phenomenon that our computational model for interactive systems has to contend with: *arbitrarily long disconnection* of interacting subsystems, whether due to local factors such as a user-specified synchronisation schedule, or global circumstances such as network outage or long latency. Whatever the reasons for disconnection, the effect is the same.

Computation which can transparently recover from temporary disconnection or failure is sometimes called *persistent computation* [BR01]. Information about the evolution of the observed system that occurred while the client was disconnected must not be lost, and when the client reconnects, it must automatically "catch up" with the current state. Reconnecting must not force the client to recompute all its state from scratch; it should only have to synchronise. The execution model must therefore retain information about which versions of dependent systems were last seen, so that synchronisation only has to incorporate what has happened since.

Together, these requirements suggest another role for transactions beyond concurrency: managing the granularity of synchronisation of (potentially widely-distributed) state. Let us understand the *total first-order information content* in a system as exhausted by the states of all the reactive processes in the system; in particular that there is *no extra information* (other than historical data) in the interactions themselves. It follows that rather than processes *receiving messages* from other processes, they may only *observe state changes* in them. The communications between processes

[5] Interaction in this sense grows the *scope* of our computations to leave us inside them, as both passive witnesses and active participants. The dual principle, again due to Wegner, is that all systems can be understood as composed of interacting parts, or equivalently, that non-interactivity is not preserved under the operation of taking subsystems.

are *deltas* (transactional descriptions of changes) in observed processes; to respond to such a communication is to *synchronise* (maintain the consistency of) one's interpretation of the state of the dependee process, incorporating the *corresponding* delta into one's own state. Change propagates through a large system of interacting processes in order to maintain the consistency of these interpretations.[6] Partial termination is the state a reactive process is in whenever it reflects the most recently observed states of the processes it is connected to.

Crucially, clients are unable to distinguish *disconnection* from an observed system, from the observed system deciding to batch up changes before publishing them. In fact the synchronisation policy can lie anywhere on the continuum between fully incremental and full batch-mode. The upshot is that transactions serve not only as a concurrency mechanism, but also as a mechanism for tolerating unbounded disconnection times by allowing micro-changes to batch up into larger deltas. How this approach relates to other asynchronous interaction models suited to distributed computing is a topic for future study. In allowing only declarative inter-process communication, and containing state within processes, for example, it may bear some relation to the *join calculus* [FG02], in which concurrent processes can only interact via declarative pattern-matching.

The Holy Grail here is *scale invariance*: a single execution model which serves the needs of small-scale, local-area interactive systems as well as it does those of large-scale, wide-area interactive systems. Locally, there will always be some views too expensive to maintain incrementally, and which need to be synchronised on an as-needed basis instead; globally, there will always be issues of latency and intermittent connectivity which require essentially the same solution. Scale invariance would allow us to add distribution and complexity to a system without having to fundamentally change the underlying computational model.[7]

6 Interactive programming languages

Declarative approaches to building interactive systems like graphical user interfaces are not new; Haggis [FJ96], Fudgets [CH98], Brisk [HC94], Clean [AP98] and Fruit [CE01] are but a few examples of such efforts. I suggest that at least one further innovation will be required before declarative languages are likely to be widely adopted: the development of *interactive programming languages*. The best way to build an interactive system is, after all, interactively – at least part of the time.

We shall rather informally understand an interactive programming language (IPL) as follows. An IPL allows programmers to build interactive systems that conform to the declarative interaction model – that is to say systems which have a modal notion of state, that separate stateful reactive computation from a declarative core language, and that are incremental, concurrent, transactional and persistent – in an "immediate" or direct way which is qualitatively different *both* from the batch-style edit-compile-execute cycle *and* from the read-eval-print loop of traditional interactive programming environments such as Lisp. An IPL is neither compiled nor interpreted in the traditional sense: *it is itself an interactive system* that conforms to the declarative interaction model, although it may not have been built using it. The "state" at any point in time of an implementation of an IPL is a partially terminated program in the declarative core language: "programming" is just interacting with its state. Its initial state is just the null program; the user can edit this into any program in the space of possible programs of the declarative core language.

[6] Since it construes inter-process communication as synchronisation of an *interpretation*, I have previously called this idea *semantic computing* [PFK05]. When the synchronisation policy is batch-mode, synchronisation behaves like a *compiler*, translating batches of source changes into batches of target changes. When the synchronisation policy is incremental, synchronisation behaves like an *interpreter*, incrementally applying changes to the target as the source changes. The notion of state is algebraic; a state is just the least batch of changes required to build it from the null structure.

[7] The current interest in using transactions to handle concurrency at all scales is a move in this direction.

IPLs thus conflate the standard distinctions between programming, and interaction at large; between compile-time, and run-time; between editing code, and editing data; and between languages, and applications. In principle at least, every application built using an IPL exposes the full power of the IPL; and conversely every IPL implementation is manifested concretely as a running application.

It appears that the notion of an IPL is largely independent of the choice of the declarative core language. An early experiment [PFK05] showed how the untyped lambda calculus can be made interactive; a more recent investigation [unpublished] suggests that object-oriented languages can be treated similarly. Interactive logic and relational programming languages are presumably feasible.[8]

As presented, the declarative interaction model is little more than a preliminary sketch. Several concretisation steps will be required before its practical utility can be evaluated. One such step will be the implementation of a general-purpose IPL suitable for local-area as well as wide-area interactive computing. In reversing the usual situation in which declarative programs are regarded as guests within a primarily procedural environment, such languages should offer significant simplicity and robustness benefits. Moreover, because their programming model reflects how interactive systems *behave*, rather than how they have been implemented until now, the impedance mismatch between languages and applications should be greatly reduced.

Acknowledgements

I thank Kevlin Henney, Jeff Foster and the four anonymous referees for valuable comments.

References

[ABH04] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive memoization. Technical Report CMU-CS-03-208, School of Computer Science, Carnegie Mellon University, 2004.

[AP98] Peter Achten and Marinus J. Plasmeijer. Interactive functional objects in Clean. In *IFL '97: Selected Papers from the 9th International Workshop on Implementation of Functional Languages*, pages 304–321, London, UK, 1998. Springer-Verlag.

[BR01] Ewa Z. Bem and John Rosenberg. An approach to implementing persistent computations. In *POS-9: Revised Papers from the 9th International Workshop on Persistent Object Systems*, Lecture Notes in Computer Science, pages 189–200, London, UK, 2001. Springer-Verlag.

[Bro87] Frederick P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[Car99] Luca Cardelli. Wide area computation. In Jiri Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *Automata, Languagese and Programming, 26th International Colloquium, ICALP'99 Proceedings*, volume 1644 of *Lecture Notes in Computer Science*, pages 10–24. Springer, 1999.

[CE01] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *2001 Haskell Workshop*, September 2001.

[8] So construed, interactive programming languages are not to be confused with "interaction protocols" and other high-level interaction languages for multi-agent systems. Rather, an IPL would be a suitable medium in which to *implement* an interaction protocol or agent-oriented language.

[CH98] Magnus Carlsson and Thomas Hallgren. *Fudgets - Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Chalmers University of Technology, Göteborg University, 1998.

[CLWY06] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. http://groups.inf.ed.ac.uk/links/papers/links-esop06.pdf, 2006.

[Col04] Raphaël Collet. Laziness and declarative concurrency. In *2nd Workshop on Object-Oriented Language Engineering for the Post-Java Era: Back to Dynamicity, 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, Oslo, Norway, June 2004.

[Cre05] Mache Creeger. Multicore CPUs for the masses. *ACM Queue*, 3(7), September 2005.

[DIL04] Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 281–290, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[DP96] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 258–270, New York, NY, USA, 1996. ACM Press.

[DSST86] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, New York, NY, USA, 1986. ACM Press.

[FG02] Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *Applied Semantics: Advanced Lectures*, volume 2395/2002 of *Lecture Notes in Computer Science*, pages 268–332. Springer Berlin/Heidelberg, 2002.

[FJ96] Sigbjorn Finne and Simon L. Peyton Jones. Composing the user interface with Haggis. In *Advanced Functional Programming, Second International School-Tutorial Text*, pages 1–37, London, UK, 1996. Springer-Verlag.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Boston, MA, USA, 1995.

[GSW06] Dina Goldin, Scott A. Smolka, and Peter Wegner, editors. *Interactive Computation: The New Paradigm*. Springer, New York, NY, USA, 2006.

[HC94] Ian Holyer and David Carter. Deterministic concurrency. In K. Hammond and J.T. O'Donnell, editors, *Proceedings of the 1993 Glasgow Workshop on Functional Programming*, Berlin, Germany, 1994. Springer Verlag.

[HHM+02] Jason D. Hartline, Edwin S. Hong, Alexander E. Modht, William R. Pentney, and Emily C. Rocke. Characterizing history independent data structures. In *Proceedings of Thirteenth International Symposium on Algorithms and Computation (ISAAC)*, volume 2518 of *Lecture Notes in Computer Science*. Springer, 2002.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[JGF96] Simon L. Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.

[JW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84, New York, NY, USA, 1993. ACM Press.

[Kob97] Satoshi Kobayashi. Monad as modality. In *NSL '94: Proceedings of the first workshop on Non-standard logics and logical aspects of computer science*, pages 29–74, Amsterdam, The Netherlands, The Netherlands, 1997. Elsevier Science Publishers B. V.

[Lee06] Edward A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, January 10 2006.

[Mic68] Donald Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[MM06] Ben Moseley and Peter Marks. Out of the tar pit. http://web.mac.com/ben_moseley/frp/paper-v1_01.pdf, 2006.

[Nan02] Aleksandar Nanevski. Meta-programming with names and necessity. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 206–217, New York, NY, USA, 2002. ACM Press.

[Nan03] Aleksandar Nanevski. From dynamic binding to state via modal possibility. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 207–218, New York, NY, USA, 2003. ACM Press.

[NC97] Johan Nordlander and Magnus Carlsson. Reactive objects in a functional language - an escape from the evil "I". In *Proceedings of the Third Haskell Workshop*, 1997.

[Per82] Alan J. Perlis. Epigrams on programming. *SIGPLAN Notices*, 17(9):7–13, 1982.

[PFK05] Roly Perera, Jeff Foster, and György Koch. A delta-driven execution model for semantic computing. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 63–71, New York, NY, USA, 2005. ACM Press.

[Pla00] John Plaice. Multidimensional Lucid: Design, semantics and implementation. In *Distributed Communities on the Web: Third International Workshop, DCW 2000*, volume 1830/2000 of *Lecture Notes in Computer Science*. Springer, 2000.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM Press.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, MA, USA, 2004.

[Weg98] Peter Wegner. Interactive foundations of computing. *Theoretical Computer Science*, 192(2):315–351, 1998.