First-order interactive programming

Roly Perera

School of Computer Science, University of Birmingham
roly.perera@dynamicaspects.org

Abstract. *Interactive programming* is a method for implementing programming languages that supports an interactive, exploratory style of program development and debugging. The basic idea is to reify the steps of a computation into a persistent data structure which can be explored interactively, and which reacts to changes to inputs like a spreadsheet. Reifying the computation associates the computed value with *provenance* information, which is essential to effective program comprehension and debugging. Making the data structure persistent means that it can evolve *incrementally*, preserving existing structure where possible, allowing the programmer to apply fixes to a program in the middle of a complex debugging activity without having to restart the program and lose browsing context. Interactive programming lies at the intersection of incremental computation, software visualisation and reactive programming.
Keywords: functional debugging, reactive programming, incremental computation, software visualisation

1 Programming as an interactive dialogue

One model of programming is as an ongoing dialogue between programmer and development environment. Programming activities – tweaking code, writing new test cases, stepping through a computation in a debugger, and so on – play the role of *questions*, and the feedback provided by the programming environment the role of *responses*. The questions fall into two basic categories. *What if* questions concern computations other than the "current" one, and thus involve a hypothetical change in either code or data. What value would the program produce for a different input? What value would a different program produce for the same input? *Provenance* questions, on the other hand, concern the current computation. How did that get to be zero? Why was that true rather than false? The emphasis of the ongoing dialogue often shifts between constructive, diagnostic and remedial.

This "interactive" model of programming may be appealing its simplicity, but is poorly supported by traditional programming environments. In this paper we present a technique for implementing programming languages that supports this model directly. Our contribution is to show how combining some well-understood notions in a novel way can yield a substantially different end-user experience. §2 motivates our work by describing a common programming scenario poorly supported in existing tools. §3 introduces the main technical concepts of our approach and develops a formal example based on the scenario presented earlier. §4 mentions some related work. An appendix contains two additional figures.

2 The whys and what ifs of programming

Provenance information is essential to debugging and program comprehension. Unfortunately, by the time we observe a result, the provenance data associated with it has usually been discarded. If we want it we must reconstruct it: by manually instrumenting the code with trace statements, stepping through the execution interactively in a breakpoint debugger, or using a visualisation tool to inspect a previously-generated trace. Yet all we want is access to what has just taken place in the interpreter.

Nevertheless, it is certainly possible to obtain provenance information with current tools, even if it is somewhat laborious. The problem gets more complicated when we consider that we rarely ask a single provenance question in isolation. Instead, the system's answer to our first question invites another, and so on. This is what is going on when we step through a complex execution in a debugger or obtain a very specific view in a tracing tool. The result is a complex tree of provenance questions and answers that "explains" the result of interest. This is a problem because what we are typically most interested in, once we have obtained this intensional view of a computation, is *what happens to that chain of reasoning* if we change something. Does this function behave correctly *for the right reasons* if I give remove an element from the list? Does this change to a base case of this recursive function fix it *in the way I expected*? The intensional view remains important for as long as I am interested not only in what my program does, but in how it does it.

In a nutshell, traditional debuggers cannot address a "what if" question in the middle of a complex provenance-related enquiry. Instead, debugging sessions are restricted to exploring *single executions*. Posing a "what if" question requires restarting the debugging session and effectively forgetting the carefully constructed chain of provenance questions. This is the problem we want to solve.

2.1 Motivating example

We have put this in rather abstract terms so far. To make things more concrete, we shall consider a programming scenario in a hypothetical development tool where this kind of interwoven activity is explicitly provided for. This will motivate the particular technical solution we discuss in §3. The four main steps of the scenario are shown in Fig. 2; the annotations in yellow boxes explain various aspects of the UI required to understand the example.

The conceptual model presented to the user is that of a *nested spreadsheet*, in which formulae are themselves spreadsheets. To frame a provenance question is to browse into a formula and observe the intermediate computations; to pose a "what if" question is to modify a value or a formula and observe how the structure changes. (Like a spreadsheet, one *navigates* between computations by *editing*; each step in Fig. 2 corresponds to one such edit.) The unique feature of our system is that the programmer can navigate whilst in a complex view state, allowing testing, debugging and bug-fixing to be interleaved efficiently.

The language used in the example (described more formally in §3.1) is a pure, first-order, call-by-value (CBV) functional language with inductive data types. We will investigate the applicability of our approach to higher-order, lazy, polymorphic languages in later work.

The scenario starts with the programmer loading the source code from Fig. 1, which compares two lists of natural numbers for equality, into the tool. The program contains two bugs. However, the UI initially shows the program computing the value `False`, which leads the programmer to suppose that the program is correct. (In Fig. 2, the visualisation of this initial state is the small rectangular cell in the top-left corner of the figure.) The scenario then progresses through four transitions from this initial state.

```
define
    equal(x,y) = case x of
        Nil -> case y of
            Nil -> False
            Cons(a,b) -> False
        Cons(x',y') -> case y of
            Nil -> False
            Cons(a,b) -> case equal_nat(a,x') of
                True -> equal(b,y')
                False -> False
    equal_nat(x,y) = case x of
        Zero -> case y of
            Zero -> True
            Succ(a) -> False
        Succ(x') -> case y of
            Zero -> True
            Succ(a) -> equal_nat(a,x')
in
    equal(Nil,Cons(Zero,Nil))
```

Fig. 1: Buggy program comparing two lists

Step 1. Test & diagnose. The programmer tests her initial hypothesis that the program is correct by trying it out at another data point, changing the first list to be `Cons(Zero,Nil)`. The UI updates to show the new computation also having the value `False`, which is incorrect, so she starts tracking down the source of the error. Observing that `equal_nat` correctly determines that the heads of the two lists are equal, she deduces that the fault lies within the `True` branch of the `case` expression which switches on the result of `equal_nat`.[1] In this branch, the computation recurses into `equal` with the tails of both lists, which in this

[1] This diagnostic method, which can be partly automated, is called *algorithmic debugging* [1]. The idea is to locate *incorrect* sub-computations whose child computations are *correct*. These necessarily contain errors.

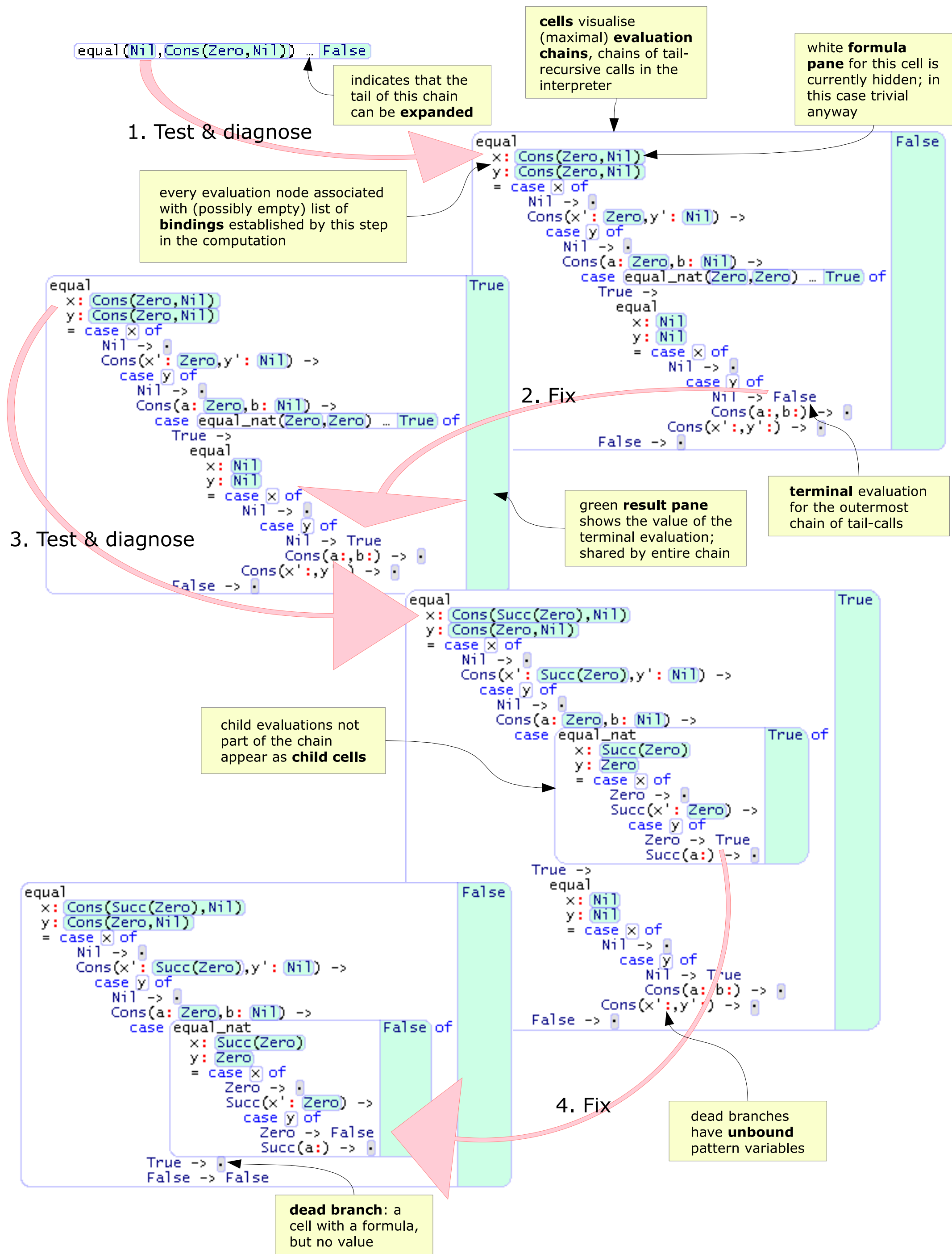

Fig. 2: Interwoven test-diagnose-fix scenario

case are `Nil`. Browsing further, she can see that the problem lies in the definition of `equal`, where `equal(Nil,Nil)` is defined to be the constant `False`.

Step 2. Fix. She fixes the problem by changing the appropriate constant in the body of `equal` from `False` to `True`. (We suppose that this edit is possible from any invocation of `equal`, but otherwise we will not concern ourselves with the details of how this happens.) Having fixed this problem, the computation updates automatically, and again the programmer sees what appears to be a correct result. The update of the computation does not affect the particular view configuration obtained in Step 1.

Step 3. Test & diagnose. Supposing again that her program is correct, the programmer selects another test case, this time changing the first list so that the two lists differ only in their head values. Again the structure of the view is unchanged. But the computed value does not change to `False` as she expects, inviting another diagnostic foray. Visual inspection reveals the source of the problem this time to be `equal_nat`, since `equal_nat(Succ(Zero),Zero)` is `True`. By expanding this sub-computation, she can see a similar problem to the first one, except that this time the constant `True` in the body of `equal_nat` should be `False`.

Step 4. Fix. She makes the required fix to `equal_nat`, which this time has some effect on the view. The recursive call to `equal` has become a dead branch, confirming her intuition that if the heads of the two lists differ, no comparison of the tails is required. The other branch of the `case`, which simply consists of the literal `False`, has become active. The structure of the `equal_nat` call is unchanged, except that it now computes `False`, as expected, as does the overall computation.

3 Execution as a persistent, reactive data structure

The scenario just described is a simple one, but one we take to be representative of programming "in the large". Supporting such an interactive form of programming means dispensing with the traditional notion of execution as a transient process. The approach we propose combines three notions. The first is *reified execution*. Reification is based on the observation that "what happens" during execution – the information required to answer a provenance question – can be precisely characterised by a suitably chosen big-step operational semantics. The semantics in effect determines a *data type* of executions: the data type whose inhabitants are the derivation trees for the evaluation judgment. Supporting provenance questions is simply a matter of making such data structures available for interactive inspection.

We achieve this by deriving from our chosen big-step semantics a *reifying semantics* which assigns to each evaluated term not a plain value, but an *evaluation*, an instance of the data type characterised by the original big-step semantics. The reifying semantics makes evaluation with respect to the original interpreter "transparent", by transcribing its dynamic behaviour into a static

data structure. By choosing the "right" semantics to reify, we can make observable those aspects of the evaluation we wish to expose, such as the substitution of arguments for formal parameters, and keep others hidden.

So reification is how we support provenance questions. For "what if" questions, we add a second notion: *reactivity*, where the evaluation (*qua* data structure), reacts to changes in inputs by adjusting into a new configuration, like a spreadsheet. The intention is to permit the modification of code as well as data, thus providing what Tanimoto calls "level 4 liveness" [2], and thereby accommodating the full generality of "what if" questions. For now we restrict ourselves to modification of data, in a first-order setting only.[2]

But the combination of reification and reactivity alone is insufficient to allow the programmer to explore alternative executions without having to discard a potentially complex browsing context. For this we need *persistence*, the property a mutable data structure has when prior versions are retained rather than discarded when it changes state [3]. Persistence allows nodes from the previous configuration to be reused if they recur in subsequent states. It requires representing the reified evaluation not as a simple tree-structured value, but as a graph of mutable nodes that can vary independently. Preserving the identity of evaluation nodes across edits allows the identity of views (with their associated browsing state) to be preserved too, as our motivating example requires.

We now develop these concepts in more formal detail in the setting of the toy language we used in the example. We start by giving a standard (non-interactive) formalisation of the language (§3.1), as a baseline. We then develop an "interactive" treatment (§3.2), incorporating the three concepts of reification, reactivity and persistence. Finally, we discuss the relationship between the baseline language and its interactive counterpart (§3.3), and conclude that the interactive implementation fully supports our motivating example, minus the UI aspect.

3.1 Baseline language

Our language lacks function types and polymorphism, but is otherwise straightforward. The rather unusual treatment of environments (and indeed our decision to use environments at all, over substitution), reflects the specific role played by the evaluation relation in this approach. Recall that the inductive definition of the relation is taken to characterise a *data type*, the data type of *evaluations* to be presented in a graphical UI. Figure 2 states quite clearly that this data type associates, to every sub-evaluation, a (possibly empty) list of *bindings* introduced by that step in the evaluation. This UI requirement directly translates into a feature of our evaluation relation: that it use environments rather than substitution, and that environments are maintained as lists of *frames*, with bindings in inner frames hiding bindings in outer frames, and empty frames explicitly represented. Semantically, of course, this is redundant: hidden bindings are never used, and empty frames are simply ignored.

[2] The advantage is that we maintain a clean separation of code and data: code can have an immutable representation, and data a mutable one.

As a general convention, we write _ for an anonymous meta-variable.

Definition 1 (Type). *Fix a finite set of* types A, B.

Write $\overrightarrow{A}$ for a finite sequence of types; assume a similar convention throughout. To each type A, we associate a family of sequences of types $A : \{\overrightarrow{B_c}\}_{c \in C}$ where C is a finite set of *constructors* and $\overrightarrow{B_c}$ is the arity of any constructor $c \in C$. Our example program assumes the types `Bool` and `Nat`, with the usual inductive definitions, plus the type `List`, representing lists of `Nat`, with constructors `Nil` and `Cons(Nat,List)`.

Definition 2 (Function). *Fix a finite set of* functions f. *To each f, we associate the pair $f(\overrightarrow{A}) : B$ of a finite sequence of types $\overrightarrow{A}$ and a type B.*

Definition 3 (Identifier). *Fix a set of* identifiers x, y.

Definition 4 (Context). *Define a* context frame $\overrightarrow{x : A}$ *to be a finite sequence of distinct identifiers each associated with a type. Define a* context Γ *to be either the* empty *context* $\cdot$ *or the pair* $\Gamma, \overrightarrow{x : A}$ *of a context* Γ *and a context frame* $\overrightarrow{x : A}$.

Write $x : A \in \Gamma$ to mean that A is the type associated with x in the rightmost frame of Γ which has an entry for x.

Definition 5 (Term). *The terms of the language are defined by:*

$$M, N ::= \mathsf{idf}_x \mid \mathsf{ctr}_c(\overrightarrow{M}) \mid \mathsf{case}_A(M, \{\overrightarrow{x_c}.M_c\}_{c \in C}) \mid \mathsf{app}_f(\overrightarrow{M})$$

The judgment $\Gamma \vdash M : A$ states that M has type A in context Γ. The typing rules are straightforward and are omitted.

Definition 6 (Function environment). *Define a* function environment *to be any function which assigns to every* $f(\overrightarrow{A}) : B$ *the pair* $(\overrightarrow{x}, M)$ *of a sequence of identifiers* $\overrightarrow{x}$ *of the same length as* $\overrightarrow{A}$ *and a term* $\cdot, \overrightarrow{x : A} \vdash M : B$.

Definition 7 (Program). *Define a* program (δ, M) *of type* A *to be a pair of a function environment* δ *and a term* $\cdot \vdash M : A$.

Definition 8 (Value). *Define a* value $V : A$ *to be a term of type* A *built entirely out of constructors.*

Definition 9 (Environment). *Define an* environment frame $\overrightarrow{x \mapsto V : A}$ to be the pair of a context frame $\overrightarrow{x : A}$ *and a sequence of values* $\overrightarrow{V : A}$. *For any context* Γ, *define an* environment ρ for Γ *to be the* empty *environment* $\cdot$, *if* Γ *is empty, or the pair* $(\rho, \overrightarrow{x \mapsto V : A})$ *of an environment* ρ *for* Γ' *and an environment frame* $\overrightarrow{x \mapsto V : A}$, *if* Γ *is of the form* $\Gamma', \overrightarrow{x : A}$.

Write $\rho(x)$ for the value associated with x in the rightmost frame of ρ which has an entry for x.

$$\overline{\rho, \mathsf{idf}_x \Downarrow \rho(x)}$$

$$\frac{(\rho, \epsilon), \overrightarrow{M} \Downarrow \overrightarrow{V}}{\rho, \mathsf{ctr}_{A,c}(\overrightarrow{M}) \Downarrow \mathsf{ctr}_{A,c}(\overrightarrow{V})}$$

$A : \{\overrightarrow{B_c}\}_{c \in C}$:

$$\frac{(\rho, \epsilon), M \Downarrow \mathsf{ctr}_{A,c}(\overrightarrow{V}) \quad (\rho, \overrightarrow{x_c \mapsto V : B_c}), N_c \Downarrow V'}{\rho, \mathsf{case}_A(M, \{\overrightarrow{x_c}.N_c\}_{c \in C}) \Downarrow V'}$$

$f(\overrightarrow{A}) : B$ and $\delta(f) = (\overrightarrow{x}, N)$:

$$\frac{(\rho, \epsilon), \overrightarrow{M} \Downarrow \overrightarrow{V} \quad (\rho, \overrightarrow{x \mapsto V : A}), N \Downarrow V'}{\rho, \mathsf{app}_f(\overrightarrow{M}) \Downarrow V'}$$

Fig. 3: Evaluation judgment

Definition 10 (Evaluation relation). *The evaluation relation is defined in Figure 3. The judgment $\rho, M \Downarrow V$ states that the term $\Gamma \vdash M : A$ in environment ρ for Γ* evaluates to *the value $V : A$.*

An evaluation step which introduces no bindings must nonetheless push the empty environment frame ϵ, as explained earlier. We note that the relation is deterministic. The judgment $\rho, \overrightarrow{M} \Downarrow \overrightarrow{V}$ states that the sequence of terms $\Gamma \vdash \overrightarrow{M : A}$ in environment ρ for Γ evaluates to the sequence of values $\overrightarrow{V : A}$ (definition omitted); we assume a similar convention for other relations.

3.2 Interactive treatment

Interactive syntax We now develop the interactive treatment of the language just defined. The first part of the interactive treatment deals with syntax. We define an *interactive term* (Def. 13), a representation of terms that allows them to contain references to mutable storage cells called *locations* (Def. 11). Locations represent the parts of terms which can change as the result of an edit. A syntactic transformation called *lifting* (Def. 15) transforms a term into an interactive term.

Definition 11 (Location). *Fix a countably infinite set of* locations *u, v.*

Definition 12 (World). *Define a* world *w to be a finite partial function from locations to types.*

Write $v : A \in w$ to mean $v \in \mathrm{dom}(w)$ and $w(v) = A$.

Definition 13 (Interactive term). *We extend the abstract syntax defined in §5 with references to locations:*
$M, N ::= ... \mid \mathsf{val}_v : A$

Again the typing rules are straightforward and are omitted. The judgment $w, \Gamma \vdash M : A$ states that *interactive term* M has type A in world w and context Γ.

We obtain an interactive term from a non-interactive term by *lifting* (Def. 15), which replaces the top-level *values* in the term by references to locations in a suitably typed world, producing a *store* (Def. 14) where those locations are set to the extracted values.

Definition 14 (Store). *For a world w, define a* store σ for w *to be a function which assigns to every location $v : A \in w$, where $A : \{\overrightarrow{B_c}\}_{c \in C}$, the pair $(c, \overrightarrow{u})$ of a constructor $c \in C$ and a sequence of locations $\overrightarrow{u : B_c} \in w$, where the relation induced on locations is acyclic.*

Informally, if $\sigma(v) = (c, \overrightarrow{u})$ then the location $v : A \in \sigma$ holds a value built by constructor c of type A, with argument values held in locations $\overrightarrow{u}$ in σ.

Definition 15 (Lifting relation). *The definition of the lifting relation is straightforward; for completeness it is given in Figure A-1 of the appendix. The judgment $M \triangleright w, \sigma, M'$ states that* lifting *a term $\Gamma \vdash M : A$ can yield a world w, an interactive term $w, \Gamma \vdash M' : A$, and a store σ for w.*

The behaviour of lifting should be obvious given an example. Lifting the program in Fig. 1 could yield the following world w and store σ:

$v : A \in w$	$\sigma(v) = (c, \overrightarrow{u})$	$v : A \in w$	$\sigma(v) = (c, \overrightarrow{u})$
Location & type	*Contents*	*Location & type*	*Contents*
v_0 : `Bool`	`False`	v_6 : `Bool`	`True`
v_1 : `Bool`	`False`	v_7 : `List`	`Nil`
v_2 : `Bool`	`False`	v_8 : `Nat`	`Zero`
v_3 : `Bool`	`False`	v_9 : `List`	`Nil`
v_4 : `Bool`	`True`	v_{10} : `List`	`Cons(v8, v9)`
v_5 : `Bool`	`False`		

with the definition of `equal` in the resulting program replaced by the following interactive version:

```
equal(x,y) = case x of
    Nil -> case y of
        Nil -> val 3
        Cons(a,b) -> val 2
    Cons(x',y') -> case y of
        Nil -> val 1
        Cons(a,b) -> case equal_nat(a,x') of
            True -> equal(b,y')
            False -> val 0
```

(`equal_nat` lifted similarly), and the program body by `equal(val 7,val 10)`.

Interactive semantics The second part of the interactive treatment defines the mutable structures which represent reified evaluations, and the relations which build and maintain these structures.

Interactive terms are interpreted in *interactive environments* (Def. 16), which associate identifiers with locations rather than values. An *evaluation node* (Def. 17) pairs an interactive environment and an interactive term. An *evaluation trace* (Def. 18) is a directed acyclic graph of evaluation nodes, where each node is annotated with the location where the value computed at that step is stored. An *evaluation store* (Def. 19) pairs an evaluation trace with a suitably typed store.

Definition 16 (Interactive environment). *Define an* interactive environment frame $\overrightarrow{x \mapsto v : A}$ in w *to be the pair of a context frame* $\overrightarrow{x : A}$ *and a sequence of locations* $\overrightarrow{v : A} \in w$*. For any context* Γ *and any world* w*, define an* interactive environment for Γ in w *to be the* empty *interactive environment* $\cdot$*, if* Γ *is empty, or the pair* $(\rho, \overrightarrow{x \mapsto v : A})$ *of an interactive environment* ρ *for* Γ' *in* w *and an interactive environment frame* $\overrightarrow{x \mapsto v : A}$ *in* w*, if* Γ *is of the form* $\Gamma', \overrightarrow{x : A}$*.*

Analogously to regular environments, we write $\rho(x)$ for the location associated with x in the rightmost frame of ρ which has an entry for x.

Definition 17 (Evaluation node). *For any world* w*, define an evaluation node* $e = (\Gamma, \rho, A, M)$ *in* w *as the quadruple of a context* Γ*, an interactive environment* ρ *for* Γ *in* w*, a type* A*, and an interactive term* $w, \Gamma \vdash M : A$*.*

Where the context permits, we write simply (ρ, M) for the evaluation node (Γ, ρ, A, M).

Definition 18 (Evaluation trace). *For any world* w*, define an* evaluation trace χ in w *to be the pair of a finite set* U *of evaluation nodes in* w*, and a function* χ *which assigns to every* $e = (\Gamma, \rho, A, M)$ *in* U *the pair* $(\overrightarrow{e}', v)$ *of a finite sequence* $\overrightarrow{e}'$ *of elements of* U *and a location* $v : A \in w$*, where the relation on evaluation nodes induced by* χ *is acyclic.*

Informally, $e \mapsto (\overrightarrow{e}', v) \in \chi$ states that, in evaluation trace χ in w, the evaluation node e in w has as immediate sub-evaluations the sequence of evaluation nodes $\overrightarrow{e}'$ in w, and stores its result in location $v \in w$.

Definition 19 (Evaluation store). *Define an* evaluation store w, σ, χ *to be the triple of a world* w*, a store* σ *for* w *and an evaluation trace* χ *in* w*.*

We now present the judgments of the interactive semantics. The two principle relations co-operate in a mutually inductive fashion. A new evaluation node is constructed for the first time by *reifying evaluation* $\Downarrow_{\square}$ (Fig. 5). When constructing the sub-evaluations of the node, reification may find that a sub-evaluation already exists in the current trace, in which case it will be reused. However, the node being reused may not be consistent with the current store: it may represent a branch of the computation which became "dead" at a previous state

and which has now become “live” again. To make it consistent, reification must deploy *synchronisation* $\Downarrow_{\boxslash}$ (Fig. 6) on the reused node.

In turn, synchronisation, while processing a `case` expression, may select a different `case`-clause to evaluate or may evaluate it in a different environment. If the selected evaluation does not already exist in the current trace, synchronisation must deploy reification to construct it afresh. To simplify the definitions of these two relations, a third relation, *persistent evaluation* $\Downarrow_{\blacksquare}$ (Fig. 4), ensures that a consistent evaluation node exists in the trace, regardless of whether the node was previously present in the trace.

Each relation is deterministic up to permutation of fresh locations, and relates an evaluation store w, σ, χ and evaluation node e in w to an evaluation store w', σ', χ' with $e \in \chi'$, where $w \subseteq w'$ and $\mathrm{dom}(\chi) \subseteq \mathrm{dom}(\chi')$.

Definition 20 (Persistent evaluation). *Mutually inductively define the following relations:*

$w, \sigma, \chi, e \Downarrow_{\blacksquare} w', \sigma', \chi'$	*as given in Figure 4*
$w, \sigma, \chi, e \Downarrow_{\square} w', \sigma', \chi'$, *where* $e \notin \chi$;	*as given in Figure 5*
$w, \sigma, \chi, e \Downarrow_{\boxslash} w', \sigma', \chi'$, *where* $e \in \chi$;	*as given in Figure 6*

Persistent evaluation has the effect of *memoising* [4] the interpreter. But whereas memoisation is typically used to improve performance, here it has observable consequences: it ensures a unique evaluation node is associated with any given evaluation, allowing the state of any dependent views to persist across edits.

$e \in \chi$:

$$\frac{w, \sigma, \chi, e \Downarrow_{\boxslash} w', \sigma', \chi'}{w, \sigma, \chi, e \Downarrow_{\blacksquare} w', \sigma', \chi'}$$

$e \notin \chi$:

$$\frac{w, \sigma, \chi, e \Downarrow_{\square} w', \sigma', \chi'}{w, \sigma, \chi, e \Downarrow_{\blacksquare} w', \sigma', \chi'}$$

Fig. 4: Persistent evaluation judgment

Reification allocates fresh locations in two situations. In the `ctr` rule, a fresh location is allocated to hold the value being constructed. In the `case` rule, a fresh location is allocated to hold a *copy* of the contents of the location associated with the evaluation of the selected clause. This location is then permanently associated with the `case`-node, although its contents may change during synchronisation.

$e = (_, \mathsf{val}_v)$:

$$\frac{}{w, \sigma, \chi, e \Downarrow_\Box w, \sigma, \chi + \{e \mapsto (\epsilon, v)\}}$$

$e = (\rho, \mathsf{idf}_x)$:

$$\frac{}{w, \sigma, \chi, e \Downarrow_\Box w, \sigma, \chi + \{e \mapsto (\epsilon, \rho(x))\}}$$

$e = (\rho, \mathsf{ctr}_{A,c}(\overrightarrow{M}))$, $v \notin w'$, and $e'_i = ((\rho, \epsilon), M_i)$ and $e'_i \mapsto (_, u_i) \in \chi'$ for each $i \in \mathrm{dom}(\overrightarrow{M})$:

$$\frac{w, \sigma, \chi, \overrightarrow{e}' \Downarrow_\blacksquare w', \sigma', \chi'}{w, \sigma, \chi, e \Downarrow_\Box w' \cup \{v : A\}, \sigma' \cup \{v \mapsto (c, \overrightarrow{u})\}, \chi' \cup \{e \mapsto (\overrightarrow{e}', v)\}}$$

$e = (\rho, \mathsf{case}_A(M, \{\overrightarrow{x_c}.N_c\}_{c \in C}))$, $v \notin w''$, $e' = ((\rho, \epsilon), M)$, $e' \mapsto (_, v') \in \chi'$, $A : \{\overrightarrow{B_c}\}_{c \in C}$, $\sigma'(v') = (c, \overrightarrow{u})$, $N_c : A'$, $e'' = ((\rho, \overrightarrow{x_c \mapsto u : B_c}), N_c)$ and $e'' \mapsto (_, v'') \in \chi''$:

$$\frac{w, \sigma, \chi, e' \Downarrow_\blacksquare w', \sigma', \chi' \quad w', \sigma', \chi', e'' \Downarrow_\blacksquare w'', \sigma'', \chi''}{w, \sigma, \chi, e \Downarrow_\Box w'' \cup \{v : A'\}, \sigma'' \cup \{v \mapsto \sigma''(v'')\}, \chi'' \cup \{e \mapsto (\langle e', e'' \rangle, v)\}}$$

$e = (\rho, \mathsf{app}_f(\overrightarrow{M}))$, $e'_i = ((\rho, \epsilon), M_i)$ and $e'_i \mapsto (_, u_i) \in \chi'$ for each $i \in \mathrm{dom}(\overrightarrow{M})$, $\delta(f) = (\overrightarrow{x}, N)$, $f(\overrightarrow{A}) : B$, $e'' = ((\rho, \overrightarrow{x \mapsto u : A}), N)$ and $e'' \mapsto (_, v) \in \chi''$:

$$\frac{w, \sigma, \chi, \overrightarrow{e}' \Downarrow_\blacksquare w', \sigma', \chi' \quad w', \sigma', \chi', e'' \Downarrow_\blacksquare w'', \sigma'', \chi''}{w, \sigma, \chi, e \Downarrow_\Box w'', \sigma'', \chi'' \cup \{e \mapsto (\overrightarrow{e}' \frown \langle e'' \rangle, v)\}}$$

Fig. 5: Reification judgment

Synchronisation utilises a relational override operator $\oplus$ to perform non-increasing store and evaluation trace updates. Synchronisation preserves the identity both of its input evaluation node, and of the location with which that node is associated in the trace. This allows the rules to remain simple. In fact, the only non-trivial part of the definition is the rule for case expressions, where a dead clause may be spliced out of the graph and replaced by another.

Synchronisation has another essential role to play, in providing the reactivity required for our example. When the programmer poses a "what if" question, in the form of a mutation to a location representing a constant in the program, synchronisation is used to restore consistency to the evaluation store. Further work is required to make this notion of "consistency" precise: for example to show that synchronisation is idempotent and that reified evaluation produces an evaluation store which is *stable*, i.e. in which synchronisation has nothing to do.

$e \neq (_, \mathsf{case}_A(_, \{\overrightarrow{x_c}.N_c\}_{c \in C}))$ and $e \mapsto (\overrightarrow{e}', _) \in \chi$:

$$\frac{w, \sigma, \chi, \overrightarrow{e}' \Downarrow_{\boxslash} w', \sigma', \chi'}{w, \sigma, \chi, e \Downarrow_{\boxslash} w', \sigma', \chi'}$$

$e = (\rho, \mathsf{case}_A(_, \{\overrightarrow{x_c}.N_c\}_{c \in C}))$, $e \mapsto (\langle e', _\rangle, v) \in \chi$, $e' \mapsto (_, v') \in \chi'$,
$\sigma'(v') = (c, \overrightarrow{u})$, $A : \{\overrightarrow{B_c}\}_{c \in C}$, $e'' = ((\rho, \overrightarrow{x_c \mapsto u : B_c}), N_c)$ and $e'' \mapsto (_, v'') \in \chi''$:

$$\frac{w, \sigma, \chi, e' \Downarrow_{\boxslash} w', \sigma', \chi' \qquad w', \sigma', \chi', e'' \Downarrow_{\blacksquare} w'', \sigma'', \chi''}{w, \sigma, \chi, e \Downarrow_{\boxslash} w'', \sigma'' \oplus \{v \mapsto \sigma''(v'')\}, \chi'' \oplus \{e \mapsto (\langle e', e''\rangle, v)\}}$$

Fig. 6: Synchronisation judgment

3.3 Relationship between interactive treatment and baseline

Unlifting (Def. 21) inverts lifting; this will allow us to state the relationship between the baseline language and its interactive counterpart.

Definition 21 (Unlifting). *For any world w, any store σ for w, and any location $v : A \in w$ with $\sigma(v) = (c, \overrightarrow{u})$, define the* unlifting $v \triangleleft \sigma$ of v in σ *as the value $\mathit{ctr}_{A,c}(\overrightarrow{u} \triangleleft \sigma)$ of type A, generalising to sequences $\overrightarrow{v} \triangleleft \sigma$ in the obvious way.*

Unlifting relies on the acyclicity of the store. Using the definition of unlifting on locations, we define the unlifting $M \triangleleft \sigma$ of an interactive term (omitted), with $\mathsf{val}_{v,A} \triangleleft \sigma \stackrel{\text{def}}{=} v \triangleleft \sigma$ being the only non-trivial case. We then use this to define the unlifting $\rho \triangleleft \sigma$ of an interactive environment (also omitted).

For the moment we consider only computations which do not diverge. We want to obtain the following theorems; this is work to be done but the proofs should be straightforward:

Theorem 1. *For any evaluation stores w, σ, χ and w', σ', χ' and any evaluation node $e = (\rho, M)$ in w with $e \notin \chi$ and $e \mapsto (_, v) \in \chi'$,*

$$w, \sigma, \chi, (\rho, M) \Downarrow_{\square} w', \sigma', \chi' \quad \Rightarrow \quad \rho \triangleleft \sigma, M \triangleleft \sigma \Downarrow v \triangleleft \sigma'$$

Theorem 2. *For any evaluation store w, σ, χ, any evaluation node $e = (\rho, M)$ in w with $e \notin \chi$, and any value V,*

$$\begin{aligned} \rho \triangleleft \sigma, M \triangleleft \sigma \Downarrow V \quad \Rightarrow \quad & \exists w', \sigma', \chi'. \exists v \in w'. w, \sigma, \chi, e \Downarrow_{\square} w', \sigma', \chi' \\ & \wedge e \mapsto (_, v) \in \chi' \wedge v \triangleleft \sigma' = V \end{aligned}$$

Another important practical consideration that we have ignored is incremental performance. As presented, synchronisation proceeds top-down and thus must traverse the entire evlauation graph. An efficient implementation would exploit the dependency information in the graph and proceed bottom-up, ignoring unaffected parts of the graph.

It remains to informally demonstrate that our interactive treatment does indeed support the motivating example of §2.1. We have implemented[3] the interactive version of the language in F#. Figure A-2 shows a dump of the evaluation graph, unravelled into a tree, corresponding to each of the five states of Fig. 2. The initial state was generated by persistent evaluation; each subsequent state was generated by programmatically editing the store and then synchronising. The parts of the graph that changed relative to the previous state are highlighted in red. Each evaluation node is preceded by a (possibly empty) environment frame, rendered in square brackets `[...]`. It should be clear that the evaluation graph both contains the information and exhibits the behaviour required to support our scenario. Dead branches are not explicitly represented, but their presentation can be derived easily, and so we suggest that it would be relatively straightforward to implement a UI like the one proposed, where the view state is preserved across edits, on top of this reactive data structure.

Our derivation of the reifying semantics is currently a manual process; we hope to explore a more "semantics-directed" approach in the future, as for example used by Kishon and Hudak [5] for execution monitoring.

4 Related work

Reified evaluation arises, in a restricted form, in Acar's *self-adjusting computation* [6] (SAC), as well as in program visualisation and debugging tools. Reactivity is central to visual programming, spreadsheet languages, functional reactive programming (FRP) [7], and again, SAC. Persistence also arises in SAC, which is therefore of special importance, since it is the only prior work which combines all three concepts. A detailed analysis of prior work on spreadsheet languages remains to be done. Subtext [8] is also similar, although based on copying rather than sharing.

Self-adjusting computation. SAC is a language and runtime system for incremental computation. After an initial evaluation, the inputs of a program can be repeatedly modified, and the resulting changes to the output observed. During the initial evaluation, the runtime records a *trace* identifying how parts of the computation depend on other parts. When an input is modified, the output is re-calculated by a bottom-up *change propagation* algorithm, which exploits the information in the trace to perform the update efficiently. The main differences are in the extent and nature of the reification. SAC only captures the aspects of evaluation relevant to efficient incremental update, whereas our system reifies the entire evaluation. Partial reification means that SAC must rely on the *re-execution* of code fragments to synchronise the state of adaptive computations when the modifiables they read have changed. Re-execution interacts poorly with imperative features such as I/O and memory allocation, since effects may be re-executed during change propagation. On the other hand, it is unclear how to recover traditional imperative features at all with our approach. Our system is also potentially very inefficient.

[3] F# source code is available at http://code.google.com/p/interactive-programming/.

Tracing debuggers. A common debugging technique is to augment the interpreter to produce a *trace* or reification of the interpreter's behaviour alongside the original behaviour. Tracing interpreters are often used with functional languages, where there is a requirement to deal with call-by-need in a user-friendly way. An example is Nilsson and Sparud's *evaluation dependence tree* (EDT) [9]. The EDT represents sharing explicitly, but omits details of when particular redexes were demanded. The authors only informally relate their data structure to a semantics, noting in passing that it resembles a proof tree for a "pseudo-CBV" interpreter able to determine whether arguments are eventually needed or not. "Time-travel" debuggers for imperative languages [10], which allow the programmer to debug backwards in time, use a similar trace-based approach. As we mentioned in §2, the main difference between these efforts and ours is that they are not *reactive*: they do not allow online modification of data or code. Instead, experimenting with a different initial configuration requires regenerating the trace and re-loading it into the offline browser.

Acknowledgments Thanks to Sam Davis, Jeff Foster, Kevlin Henney, Paul Levy, Robin Message, Tom Stuart and John Zabroski for helpful comments on earlier drafts.

References

1. Shapiro, E.Y.: Algorithmic program debugging. ACM Distinguished Dissertations. MIT Press, Cambridge, MA, USA (1983)
2. Tanimoto, S.L.: VIVA: A visual language for image processing. Journal of Visual Languages and Computing **1**(2) (1990)
3. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. In: STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing, New York, NY, USA, ACM Press (1986) 109–121
4. Michie, D.: Memo functions and machine learning. Nature **218** (1968) 19–22
5. Kishon, A., Hudak, P.: Semantics directed program execution monitoring. Journal of Functional Programming **5**(04) (1995) 501–547
6. Acar, U.A.: Self-Adjusting Computation. Phd thesis, Department of Computing Science, Carnegie Mellon University (2005)
7. Elliott, C., Hudak, P.: Functional reactive animation. In: ICFP '97: Proceedings of the Second ACM SIGPLAN International Conference on Functional programming, New York, NY, USA, ACM (1997) 263–273
8. Edwards, J.: Subtext: uncovering the simplicity of programming. In: OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2005) 505–518
9. Nilsson, H., Sparud, J.: The evaluation dependence tree: an execution record for lazy functional debugging. Research Report LiTH-IDA-R-96-23, Department of Computer and Information Science, Linkping University, S-581 83, Linkping, Sweden (August 1996)
10. Lewis, B.: Debugging backwards in time. In Ronsse, M., De Bosschere, K., eds.: Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003). (September 2003)

APPENDIX

$$\frac{}{\mathsf{idf}_x \triangleright \varnothing, \varnothing, \mathsf{idf}_x}$$

$M'_i = \mathsf{val}_{u_i}$, for all $i \in \mathrm{dom}(\overrightarrow{M'})$, and $v \notin w$:

$$\frac{\overrightarrow{M} \triangleright w, \sigma, \overrightarrow{M'}}{\mathsf{ctr}_{A,c}(\overrightarrow{M}) \triangleright w \cup \{v \mapsto A\}, \sigma \cup \{v \mapsto (c, \overrightarrow{u})\}, \mathsf{val}_v}$$

$M'_i \neq \mathsf{val}_{u_i}$, for some $i \in \mathrm{dom}(\overrightarrow{M'})$:

$$\frac{\overrightarrow{M} \triangleright w, \sigma, \overrightarrow{M'}}{\mathsf{ctr}_{A,c}(\overrightarrow{M}) \triangleright w, \sigma, \mathsf{ctr}_{A,c}(\overrightarrow{M'})}$$

$\mathrm{dom}(w_0) \cap \mathrm{dom}(w_1) = \varnothing$:

$$\frac{M \triangleright w_0, \sigma_0, M' \quad \{N_c\}_{c \in C} \triangleright w_1, \sigma_1, \{N'_c\}_{c \in C}}{\mathsf{case}_A(M, \{\overrightarrow{x_c}.N_c\}_{c \in C}) \triangleright w_0 \cup w_1, \sigma_0 \cup \sigma_1, \mathsf{case}_A(M', \{\overrightarrow{x_c}.N'_c\}_{c \in C})}$$

$$\frac{\overrightarrow{M} \triangleright w, \sigma, \overrightarrow{M'}}{\mathsf{app}_f(\overrightarrow{M}) \triangleright w, \sigma, \mathsf{app}_f(\overrightarrow{M'})}$$

Fig. A-1: Lifting judgment

The analogous judgments for sequences of terms and families of case clauses (omitted) require a side-condition asserting disjointness of worlds similar to the one for the case rule above. We extend the definition of lifting to programs (δ, M) in the obvious way.

```
equal(...) = v12: False
+- [] val-v7 = v7: Nil
+- [] val-v10 = v10: Cons(Zero,Nil)
`- [x: v7, y: v10] case_List = v12: False
   +- [] x = v7: Nil
   `- [] case_List = v11: False
      +- [] y = v10: Cons(Zero,Nil)
      `- [a: v8, b: v9] val-v2 = v2: False
```

(a) State 1

```
equal(...) = v12: False
+- [] val-v7 = v7: Cons(Zero,Nil)
+- [] val-v10 = v10: Cons(Zero,Nil)
`- [x: v7, y: v10] case_List = v12: False
   +- [] x = v7: Cons(Zero,Nil)
   `- [x': v13, y': v14] case_List = v20: False
      +- [] y = v10: Cons(Zero,Nil)
      `- [a: v8, b: v9] case_Bool = v19: False
         +- [] equal_nat(...) = v16: True
         |  +- [] a = v8: Zero
         |  +- [] x' = v13: Zero
         |  `- [x: v8, y: v13] case_Nat = v16: True
         |     +- [] x = v8: Zero
         |     `- [] case_Nat = v15: True
         |        +- [] y = v13: Zero
         |        `- [] val-v6 = v6: True
         `- [] equal(...) = v18: False
            +- [] b = v9: Nil
            +- [] y' = v14: Nil
            `- [x: v9, y: v14] case_List = v18: False
               +- [] x = v9: Nil
               `- [] case_List = v17: False
                  +- [] y = v14: Nil
                  `- [] val-v3 = v3: False
```

(b) State 2

```
equal(...) = v12: True
+- [] val-v7 = v7: Cons(Zero,Nil)
+- [] val-v10 = v10: Cons(Zero,Nil)
`- [x: v7, y: v10] case_List = v12: True
   +- [] x = v7: Cons(Zero,Nil)
   `- [x': v13, y': v14] case_List = v20: True
      +- [] y = v10: Cons(Zero,Nil)
      `- [a: v8, b: v9] case_Bool = v19: True
         +- [] equal_nat(...) = v16: True
         |  +- [] a = v8: Zero
         |  +- [] x' = v13: Zero
         |  `- [x: v8, y: v13] case_Nat = v16: True
         |     +- [] x = v8: Zero
         |     `- [] case_Nat = v15: True
         |        +- [] y = v13: Zero
         |        `- [] val-v6 = v6: True
         `- [] equal(...) = v18: True
            +- [] b = v9: Nil
            +- [] y' = v14: Nil
            `- [x: v9, y: v14] case_List = v18: True
               +- [] x = v9: Nil
               `- [] case_List = v17: True
                  +- [] y = v14: Nil
                  `- [] val-v3 = v3: True
```

(c) State 3

```
equal(...) = v12: True
+- [] val-v7 = v7: Cons(Zero,Nil)
+- [] val-v10 = v10: Cons(Succ(Zero),Nil)
`- [x: v7, y: v10] case_List = v12: True
   +- [] x = v7: Cons(Zero,Nil)
   `- [x': v13, y': v14] case_List = v20: True
      +- [] y = v10: Cons(Succ(Zero),Nil)
      `- [a: v8, b: v9] case_Bool = v19: True
         +- [] equal_nat(...) = v16: True
         |  +- [] a = v8: Succ(Zero)
         |  +- [] x' = v13: Zero
         |  `- [x: v8, y: v13] case_Nat = v16: True
         |     +- [] x = v8: Succ(Zero)
         |     `- [x': v21] case_Nat = v22: True
         |        +- [] y = v13: Zero
         |        `- [] val-v4 = v4: True
         `- [] equal(...) = v18: True
            +- [] b = v9: Nil
            +- [] y' = v14: Nil
            `- [x: v9, y: v14] case_List = v18: True
               +- [] x = v9: Nil
               `- [] case_List = v17: True
                  +- [] y = v14: Nil
                  `- [] val-v3 = v3: True
```

(d) State 4

```
equal(...) = v12: False
+- [] val-v7 = v7: Cons(Zero,Nil)
+- [] val-v10 = v10: Cons(Succ(Zero),Nil)
`- [x: v7, y: v10] case_List = v12: False
   +- [] x = v7: Cons(Zero,Nil)
   `- [x': v13, y': v14] case_List = v20: False
      +- [] y = v10: Cons(Succ(Zero),Nil)
      `- [a: v8, b: v9] case_Bool = v19: False
         +- [] equal_nat(...) = v16: False
         |  +- [] a = v8: Succ(Zero)
         |  +- [] x' = v13: Zero
         |  `- [x: v8, y: v13] case_Nat = v16: False
         |     +- [] x = v8: Succ(Zero)
         |     `- [x': v21] case_Nat = v22: False
         |        +- [] y = v13: Zero
         |        `- [] val-v4 = v4: False
         `- [] val-v0 = v0: False
```

(e) State 5

Fig. A-2: Evolution of persistent state (changes from previous state in red)