First-order interactive programming

Roly Perera

School of Computer Science, University of Birmingham
`rnp@cs.bham.ac.uk`

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 in 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, using the scenario to illustrate the various notions. §4 mentions some related work. An accompanying technical report [1] formalises the toy language used in the example and its interactive counterpart.

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 now behave correctly *for the right reasons* if I 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 inquiry. 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 in our hypothetical tool is that of a *nested spreadsheet*, in which formulae are themselves spreadsheets. This UI concept is not itself part of our proposal, but is intended to be suggestive of what one might build on top of such a system. 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.

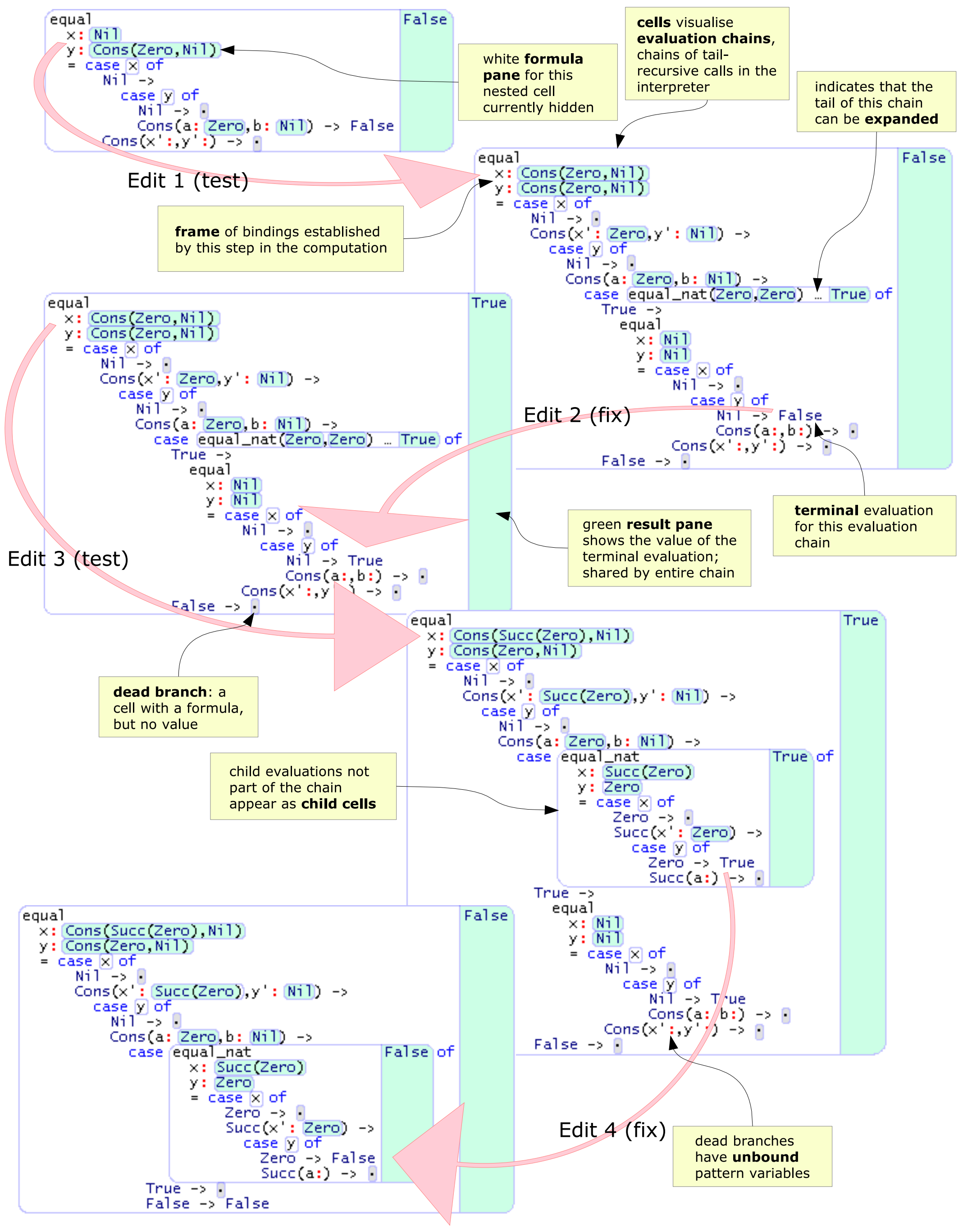
The language used in the example, and described more formally in [1], is a pure, first-order, call-by-value (CBV) functional language with inductive data types. Our example program assumes the types `Bool` and `Nat`, with the usual inductive definitions, plus the type `List`, representing lists of `Nat`, with constructors `Nil` and `Cons(Nat,List)`. 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, this is the state shown 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

Edit 1. Test. 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


white **formula pane** for this nested cell currently hidden
cells visualise **evaluation chains**, chains of tail-recursive calls in the interpreter
indicates that the tail of this chain can be **expanded**
Edit 1 (test)
frame of bindings established by this step in the computation
Edit 2 (fix)
green **result pane** shows the value of the terminal evaluation; shared by entire chain
terminal evaluation for this evaluation chain
Edit 3 (test)
dead branch: a cell with a formula, but no value
child evaluations not part of the chain appear as **child cells**
Edit 4 (fix)
dead branches have **unbound** pattern variables

```
equal                                                        False
  x: Nil
  y: Cons(Zero,Nil)
  = case x of
      Nil ->
        case y of
          Nil -> ·
          Cons(a: Zero,b: Nil) -> False
      Cons(x':,y':) -> ·
```

```
equal                                                        False
  x: Cons(Zero,Nil)
  y: Cons(Zero,Nil)
  = case x of
      Nil -> ·
      Cons(x': Zero,y': Nil) ->
        case y of
          Nil -> ·
          Cons(a: Zero,b: Nil) ->
            case equal_nat(Zero,Zero) … True of
              True ->
                equal
                  x: Nil
                  y: Nil
                  = case x of
                      Nil -> ·
                        case y of
                          Nil -> False
                          Cons(a:,b:) -> ·
                      Cons(x':,y':) -> ·
              False -> ·
```

```
equal                                                        True
  x: Cons(Zero,Nil)
  y: Cons(Zero,Nil)
  = case x of
      Nil -> ·
      Cons(x': Zero,y': Nil) ->
        case y of
          Nil -> ·
          Cons(a: Zero,b: Nil) ->
            case equal_nat(Zero,Zero) … True of
              True ->
                equal
                  x: Nil
                  y: Nil
                  = case x of
                      Nil -> ·
                        case y of
                          Nil -> True
                          Cons(a:,b:) -> ·
                      Cons(x':,y':) -> ·
              False -> ·
```

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

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

Fig. 2: Interwoven test-diagnose-fix scenario

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 the two lists, which in this 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`.

Edit 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 after Edit 1.

Edit 3. Test. 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`.

Edit 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". To support this interactive form of programming, we dispense with the traditional conception of execution as a transient process. The approach we propose combines three notions. The first is *reified execution*. This 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 proof trees for the evaluation judgment. Supporting provenance inquiries is simply a matter of making such data structures available for interactive inspection.

[1] This diagnostic method, which can be partly automated, is called *algorithmic debugging* [2]. The idea is to locate *incorrect* sub-computations whose child computations are *correct*. These necessarily contain errors.

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 a suitable 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, while keeping others hidden. Our derivation of the reifying semantics is currently a manual process. In the future, we hope to explore a "semantics-directed" approach, such as the one used by Kishon and Hudak [3] for execution monitoring.

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. Our eventual intention is to permit the modification of code as well as data, thus providing what Tanimoto calls "level 4 liveness" [4], and thereby accommodating the full generality of "what if" questions. For now, we restrict ourselves to modification of data, which naturally suggests a first-order, CBV setting, where there is a clear distinction between code and data. Lifting this restriction is another topic for future study.

To allow the programmer to explore alternative executions without having to discard a potentially complex browsing context, we need a third notion: *persistence*. This means that prior versions of the data structure are retained rather than discarded when it changes state [5], allowing 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 evaluations across edits allows the identity of views (with their associated browsing state) to be preserved too, as our motivating example requires.

We now apply the three notions of reification, reactivity and persistence to the toy language introduced earlier, and show how they can be used to derive an "interactive" version of the language. The interactive language fully supports the interwoven test-diagnose-fix scenario of Fig. 2, except for the UI aspect. The formal treatment in [1] is rather detailed, and consists mainly of definitions; here we focus how the key technical ideas fit together. We have implemented the interactive version of the language in F$^\sharp$.[2]

3.1 Interactive syntax

To allow changes to program data to be made after the program has executed, we use a mutable representation of a term called an *interactive term*. In our restricted first-order setting, "data" just means constants occurring in the original program, namely terms built purely out of constructors of inductive data types. An interactive term is simply a term where each constant has been replaced by

[2] Source code available at http://code.google.com/p/interactive-programming/.

a reference to a *location* of the same type in a *value store*, which holds a representation of that constant. Edits are then modelled as external changes to the value store. As the program itself has no write access to the value store, the pure functional semantics of the language is preserved in the "interactive" version.

To represent constants, a value store associates each location of type A in its domain to a constructor `c` of A and a sequence of appropriately typed *child locations* in the same store, one for each parameter of `c`. Value stores are acyclic; we exploit this acyclicity (and the smallness of our examples) in a notation which omits the types of locations and flattens descendant locations into a string. Thus we write `v10:Cons(v8:Zero,v9:Nil)` to indicate that (in some value store) `v10` is set to `Cons` and has children `v8:Zero` and `v9:Nil`.

The process of producing an interactive term and a suitable initial value store from a term is called *lifting*. Lifting our example program might yield the following value store:

```
v0:False     v4:True      v8:Zero
v1:False     v5:False     v9:Nil
v2:False     v6:True      v10:Cons(v8:Zero,v9:Nil)
v3:False     v7:Nil
```

and the following interactive version of `equal`:

```
equal(x,y) = case x of
    Nil -> case y of
        Nil -> val v3
        Cons(a,b) -> val v2
    Cons(x',y') -> case y of
        Nil -> val v1
        Cons(a,b) -> case equal_nat(a,x') of
            True -> equal(b,y')
            False -> val v0
```

with `equal_nat` lifted similarly, and the program body lifted to `equal(val v7,val v10)`. (We say *might*, because lifting is deterministic only up to permutation of locations.) The syntactic construct `val v` appears only in interactive terms, and is used to refer to the location `v` of a lifted constant. Lifting is inverted by *unlifting*: the unlifting of the interactive term `val v10` recovers the constant `Cons(Zero,Nil)`.

3.2 Interactive semantics

By referring to a set of locations, an interactive term is associated with a *family* of evaluations, indexed by value stores which assign constants to those locations: for each such value store, there is a potentially different evaluation for that interactive term. Evaluating the lifted program with its initial value store does not simply reduce it to a final result, but instead yields a *transition system* which allows the programmer to explore interactively the family of evaluations for that program by making changes to the value store. Roughly speaking, each state contains a *reification*, viz. a representation of the proof tree, of the evaluation of the program's unlifted counterpart at that state.

Reification We start by defining the evaluation relation whose derivations we intend to reify. Taking ρ to range over environments, $\mathtt{x}$ over identifiers, M and N over terms, V over constants, $\mathtt{c}$ over constructors, and $\mathtt{f}$ over functions, the definition is given in Fig. 3. The notation $\cdot$ means the empty environment, and $(\rho, \overrightarrow{\mathtt{x} \mapsto V})$ the environment ρ extended with a *frame* $\overrightarrow{\mathtt{x} \mapsto V}$ which binds each identifier in $\vec{\mathtt{x}}$ to the corresponding constant in $\vec{V}$. We use environments, rather than substitution, to preserve the occurrences of identifiers in the source code. The pair ρ, M of a term M and a closing environment ρ we call an *evaluand*.[3]

$$\frac{}{\rho, \mathtt{x} \Downarrow \rho(\mathtt{x})}$$

$$\frac{\rho, \vec{M} \Downarrow \vec{V}}{\rho, \mathtt{c(}\vec{M}\mathtt{)} \Downarrow \mathtt{c(}\vec{V}\mathtt{)}}$$

The type of M has a set C of constructors:

$$\frac{\rho, M \Downarrow \mathtt{c(}\vec{V}\mathtt{)} \quad (\rho, \overrightarrow{\mathtt{x}_\mathtt{c} \mapsto V}), N_\mathtt{c} \Downarrow V'}{\rho, \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{c(}\vec{\mathtt{x}_\mathtt{c}}\mathtt{)}\ \mathtt{->}\ N_\mathtt{c}\}_{\mathtt{c} \in C} \Downarrow V'}$$

$\mathtt{f}$ has parameters $\vec{\mathtt{x}}$ and body N:

$$\frac{\rho, \vec{M} \Downarrow \vec{V} \quad (\cdot, \overrightarrow{\mathtt{x} \mapsto V}), N \Downarrow V'}{\rho, \mathtt{f(}\vec{M}\mathtt{)} \Downarrow V'}$$

Fig. 3: Traditional (non-interactive) evaluation judgment

To a first approximation, a reified evaluation is just a proof tree for the $\Downarrow$ relation of Fig. 3. However, the reification must record the final result of the computation in a *location*, rather than simply represent it as a constant, so that it can be updated when a location the result depends on changes. For a similar reason, we need a notion of environment, called an *interactive environment*, which maps identifiers to locations, rather than constants. The pair ρ, M of an interactive term M and a closing interactive environment ρ we call an *interactive evaluand*. From now on, by "environment" we shall generally mean "interactive environment", and similarly for evaluands, unless otherwise stated.

[3] We can think of an evaluand as the tuple of arguments that uniquely identifies a call to the interpreter. Child evaluands identify recursive calls. Thus an alternative interpretation of the proof tree is as a *call tree* for the interpreter. This was the intuition we used to explain Fig. 2.

Not only may an interactive term compute different results in different value stores, but the sub-computations used to compute it may change also. These considerations mean we cannot store reified evaluations as simple tree-structured values, but must maintain them in a map which associates to every evaluand in its domain an *output location*, where the result of the computation is stored, plus a sequence of *children*, evaluands in the same trace identifying immediate sub-computations. We call such a map an *evaluation trace*, and the pair of an evaluation trace and a suitably typed value store an *evaluation store*. The output location is fixed, once allocated, whereas the child evaluands and the contents of the output location may change as a direct or indirect result of an edit.

The structure of an evaluand in the evaluation store is constrained in certain ways that make it conform to the structure of the evaluation relation being reified. For a reference `val v`, it must output to `v` and have no children. For an identifier `x`, it must output to the location to which `x` is bound, and again have no children. For a constructor `c`, it must have a child for each sub-expression, and must output to a fresh location $\texttt{v:c(}\overrightarrow{\texttt{u}}\texttt{)}$, where $\overrightarrow{\texttt{u}}$ are the output locations of its children. For a function `f` applied to n arguments, its first n children must be evaluands for the arguments, and its last child an evaluand for the body of `f` in an environment extended by a frame $\overrightarrow{\texttt{x} \mapsto \texttt{u}}$, where $\overrightarrow{\texttt{x}}$ are the parameters of f and $\overrightarrow{\texttt{u}}$ the output locations of the first n children. It must share an output location with its last child, corresponding to the tail-call optimisation. For a case analysis, the constraints are somewhat weaker. It must have exactly two children: one for the expression being pattern-matched, and one corresponding to one of the case-clauses (which we call the *active clause*), in an environment that binds the associated pattern variables to locations of the right type. It must output to a fresh location of the right type.

From the definition in Fig. 3, we then derive a procedure called *reification*, which for an evaluation store and an evaluand ρ, M fresh in the trace, builds a record of the evaluation of the unlifted counterpart of ρ, M in the evaluation store. We do not give a definition of reification here, since the constraints on the structure of the evaluation store just given for the most part determine its behaviour. The only exception is for a case analysis evaluand, which the evaluation store permits to be *unstable*, in a sense which will be made precise later, but which reification ensures is initially stable. Permitting the evaluation store to contain unstable evaluands allows states which are intermediate between two reifications.

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

(a) State 1 (stable)

```
equal = v12:False
+- val v7 = v7:Cons(v13:Zero,v14:Nil)
+- val v10 = v10:Cons(v8:Zero,v9:Nil)
+- [x: v7, y: v10] case_List = v12:False
   +- x = v7:Cons(v13:Zero,v14:Nil)
   +- [] case_List = v11:False
      +- y = v10:Cons(v8:Zero,v9:Nil)
      +- [a: v8, b: v9] val v2 = v2:False
```

(b) State 1′ (unstable)

Fig. 4: Evaluation store, states 1 & 1′

States 1 through 5 in Figs. 4 onwards show an evaluation store as it transitions through the five states of our original scenario. The contents of any locations referenced by the trace are shown "inline", using the flattened notation for locations we introduced in §3.1. State 1 is the initial state of the store, corresponding to the initial state of Fig. 2. The trace contains a single root evaluand $\cdot$, `equal(val v7,val v10)`, called the *program root*. In the figure this is elided to `equal`. The program root outputs to location `v12:False`, and has three children: two for the evaluation of the arguments in $\cdot$, and one for the body of `equal` evaluated in $\cdot$ extended by the frame `[x: v7, y: v10]`.

The evaluation of the body of `equal` is a case-analysis with children for the identifier `x`, bound to `v7:Nil`, and for the clause for `Nil`. There are no pattern variables to bind, and so the clause is evaluated in an environment extended by the empty frame `[]`. The evaluation of the clause involves another case analysis, with children for the identifier `y`, bound to `v10:Cons(v8:Zero,v9:Nil)`, and for the `Cons` clause, which consists simply of the lifted constant `val v2`. The inner clause is evaluated in an environment extended by the frame `[a: v8, b: v9]`, which binds the pattern variables to the child locations of `v10`, although neither of these bindings happens to be used.

It should be clear how the information in the evaluation store supports provenance inquiries. We can see that there are precisely three facts which contribute to the call to `equal` having the value `False`: that the first argument is `Nil`; that the second is a `Cons`; and that `equal` of `Nil` and a `Cons` is defined to be `False`.

Reactivity We support "what if" questions by having the system react to changes to locations corresponding to the lifted constants. Locations where intermediate results are stored cannot be modified directly. When a change occurs, the program root may become *unstable*, in that program has a different unlifting, for which the trace may no longer represent a valid reified evaluation. We define an idempotent procedure called *synchronisation* which transitions to a configuration that incorporates the change. Here we give only an informal definition. To synchronise an evaluand ρ, M which has output location `v`:

- if M is a `case` expression:
  - synchronise the first child of ρ, M;
  - let `v':c(`$\overrightarrow{\texttt{u}}$`)` be the output location of the first child;
  - let M' be the clause for `c`, with pattern variables $\overrightarrow{\texttt{x}}$;
  - let ρ' be $(\rho, \overrightarrow{\texttt{x} \mapsto \texttt{u}})$;
  - if ρ', M' is fresh in the trace, synchronise ρ', M'; otherwise, reify ρ', M'.
  - copy `u` to `v`, where ρ', M' has output location `u`.
- otherwise, synchronise each child of ρ, M.

This definition exploits the constraints on the evaluation store given earlier. For example, the evaluand for a function application shares an output location with the evaluation of the body. Thus to synchronise the evaluation of an application it suffices to synchronise the evaluation of the body.

We can now make precise our notion of stability by defining a *stable* evaluand to be one for which synchronisation leaves the evaluation store unchanged. Stability and synchronisation are illustrated by the transition from state $1'$ in

```
equal = v12:False
+- val v7 = v7:Cons(v13:Zero,v14:Nil)
+- val v10 = v10:Cons(v8:Zero,v9:Nil)
+- [x: v7, y: v10] case_List = v12:False
   +- x = v7:Cons(v13:Zero,v14:Nil)
   +- [x’: v13, y’: v14] case_List = v20:False
      +- y = v10:Cons(v8:Zero,v9: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
[] case_List = v11:False
+- y = v10:Cons(v8:Zero,v9:Nil)
+- [a: v8, b: v9] val v2 = v2:False
```

(a) State 2 (stable)

```
equal = v12:False
+- val v7 = v7:Cons(v13:Zero,v14:Nil)
+- val v10 = v10:Cons(v8:Zero,v9:Nil)
+- [x: v7, y: v10] case_List = v12:False
   +- x = v7:Cons(v13:Zero,v14:Nil)
   +- [x’: v13, y’: v14] case_List = v20:False
      +- y = v10:Cons(v8:Zero,v9: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: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
[] case_List = v11:False
+- y = v10:Cons(v8:Zero,v9:Nil)
+- [a: v8, b: v9] val v2 = v2:False
```

(b) State 2′ (unstable)

Fig. 5: Evaluation store, states 2 & 2′

Fig. 4 to state 2 in Fig. 5. “Accepted” changes to be incorporated by synchronisation are shown in green; the unstable fringes of the evaluation are shown in red. State 1′ is identical to state 1 except that the first argument to `equal` has been modified as per Edit 1 of our scenario. The program root is unstable because the outer case analysis requires synchronisation: the first child of the case analysis outputs to `v7`, which is now set to `Cons` instead of `Nil`, and yet the `Nil` clause is still active. We say that an evaluand is *reachable* if it is a descendant of the program root. To accommodate the change into a new stable state, we must activate the `Cons` clause instead, and splice the evaluation of the `Nil` clause out of the reachable trace. The evaluation of the `Cons` clause is a fresh region of the trace which did not arise in an earlier state, so synchronisation must invoke reification to build it. State 2 shows the resulting stable state, with all the incorporated changes shown in green. In particular, we can see that the evaluation of the `Cons` clause makes a call to `equal_nat` and also a recursive call to `equal`, as per state 2 in our UI mock-up.

The transition to state 2 was straightforward, as resolving the first instability immediately yielded a stable state. Most of the work was done by reification, in creating the fresh trace for the `Cons` clause. The transition from state 2 to state 3 shows a more involved synchronisation. The transition is initiated by an edit corresponding to the first bug-fix of our scenario, where `equal(Nil,Nil)` is corrected to `True`. The resulting unstable state (not shown) is identical to state 2 except that `v3` is `True` rather than `False`. The first synchronisation step involves setting `v17` to `True` as well, to stabilise the parent case analysis. Incorporating this change in turn requires setting `v18` to `True`, resulting in state 2′, which is still unstable. Then we must set `v19` to `True`, to synchronise the case analysis for `Bool`, and so on. The “wavefront” of synchronisation proceeds bottom-up until

```
equal = v12:True
+- val v7 = v7:Cons(v13:Zero,v14:Nil)
+- val v10 = v10:Cons(v8:Zero,v9:Nil)
+- [x: v7, y: v10] case_List = v12:True
   +- x = v7:Cons(v13:Zero,v14:Nil)
   +- [x': v13, y': v14] case_List = v20:True
      +- y = v10:Cons(v8:Zero,v9: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
[] case_List = v11:False
+- y = v10:Cons(v8:Zero,v9:Nil)
+- [a: v8, b: v9] val v2 = v2:False
```

(a) State 3 (stable)

```
equal = v12:True
+- val v7 = v7:Cons(v13:Zero,v14:Nil)
+- val v10 = v10:Cons(v8:Succ(v21:Zero),v9:Nil)
+- [x: v7, y: v10] case_List = v12:True
   +- x = v7:Cons(v13:Zero,v14:Nil)
   +- [x': v13, y': v14] case_List = v20:True
      +- y = v10:Cons(v8:Succ(v21:Zero),v9:Nil)
      +- [a: v8, b: v9] case_Bool = v19:True
         +- equal_nat = v16:True
         |  +- a = v8:Succ(v21:Zero)
         |  +- x' = v13:Zero
         |  +- [x: v8, y: v13] case_Nat = v16:True
         |     +- x = v8:Succ(v21: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
[] case_List = v11:False
+- y = v10:Cons(v8:Succ(v21:Zero),v9:Nil)
+- [a: v8, b: v9] val v2 = v2:False
[] case_Nat = v15:True
+- y = v13:Zero
+- [] val v6 = v6:True
```

(b) State 4 (stable)

Fig. 6: Evaluation store, states 3 & 4

it reaches an evaluand which is already stable, at which point all reachable parts of the trace must be stable and all changes incorporated (state 3). States 4 and 5 are obtained similarly, by introducing an instability in the form of a change to a location corresponding to a lifted constant, and then synchronising.

Persistence After synchronisation, there may be fragments of the trace which are no longer reachable. We see this in state 5, where the recursive call to `equal` on the tails of the lists is no longer required, as the heads of the lists differ. These fragments will be *reused* whenever a state arises in which they become reachable again. This is the mechanism via which states share sub-structure with prior states whenever possible.

Over time, these unreachable fragments can become unstable, since only the reachable parts of the trace are kept in sync. This arises in state 6, which extends our original scenario. This is a stable state, created by modifying the tail of one of the lists to include an extra element, and then synchronising. The recursive call to `equal` is unstable, since the wrong case clause is active. But this part of the trace is unreachable, and therefore allowed to remain unstable. Modifying state 6 to put the head of the second list back to `Zero` introduces a reachable instability (state 6′) which when synchronised forces a comparison of the tails. The recursive call is incorporated again and re-synchronised, resulting in stable state 7.

Synchronisation and reification thus enjoy a mutually recursive relationship. When recursively constructing a child evaluand, reification may determine that it already exists and need only be synchronised. When activating the child evaluand for a case clause, synchronisation may determine that it is fresh and needs to be reified. The trace thus serves as something like a *memo-table* [6] for reification,

```
equal = v12:False
+- val v7 = v7:Cons(v13:Zero,v14:Nil)
+- val v10 = v10:Cons(v8:Succ(v21:Zero),v9:Nil)
+- [x: v7, y: v10] case_List = v12:False
   +- x = v7:Cons(v13:Zero,v14:Nil)
   +- [x': v13, y': v14] case_List = v20:False
      +- y = v10:Cons(v8:Succ(v21:Zero),v9:Nil)
      +- [a: v8, b: v9] case_Bool = v19:False
         +- equal_nat = v16:False
         |  +- a = v8:Succ(v21:Zero)
         |  +- x' = v13:Zero
         |  +- [x: v8, y: v13] case_Nat = v16:False
         |     +- x = v8:Succ(v21:Zero)
         |     +- [x': v21] case_Nat = v22:False
         |        +- y = v13:Zero
         |        +- [] val v4 = v4:False
         +- [] val v0 = v0:False
[] case_List = v11:False
+- y = v10:Cons(v8:Succ(v21:Zero),v9:Nil)
+- [a: v8, b: v9] val v2 = v2:False
[] 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
```

(a) State 5 (stable)

```
equal = v12:False
+- val v7 = v7:Cons(v13:Zero,v14:Nil)
+- val v10 = v10:Cons(v8:Succ(v21:Zero),v9:Cons(v23:Zero,v24:Nil))
+- [x: v7, y: v10] case_List = v12:False
   +- x = v7:Cons(v13:Zero,v14:Nil)
   +- [x': v13, y': v14] case_List = v20:False
      +- y = v10:Cons(v8:Succ(v21:Zero),v9:Cons(v23:Zero,v24:Nil))
      +- [a: v8, b: v9] case_Bool = v19:False
         +- equal_nat = v16:False
         |  +- a = v8:Succ(v21:Zero)
         |  +- x' = v13:Zero
         |  +- [x: v8, y: v13] case_Nat = v16:False
         |     +- x = v8:Succ(v21:Zero)
         |     +- [x': v21] case_Nat = v22:False
         |        +- y = v13:Zero
         |        +- [] val v4 = v4:False
         +- [] val v0 = v0:False
[] case_List = v11:False
+- y = v10:Cons(v8:Succ(v21:Zero),v9:Cons(v23:Zero,v24:Nil))
+- [a: v8, b: v9] val v2 = v2:False
[] case_Nat = v15:True
+- y = v13:Zero
+- [] val v6 = v6:True
[] equal = v18:True
+- b = v9:Cons(v23:Zero,v24:Nil)
+- y' = v14:Nil
+- [x: v9, y: v14] case_List = v18:True
   +- x = v9:Cons(v23:Zero,v24:Nil)
   +- [] case_List = v17:True
      +- y = v14:Nil
      +- [] val v3 = v3:True
```

(b) State 6 (stable)

Fig. 7: Evaluation store, state 6

but one where the reused result may be stale and require refreshing. Rather than being used to improve performance within a single computation, the role of memoisation here is to force the sharing of computational structure between states. Repeated equality checking can be avoided by also adopting a persistent representation for interactive terms and environments (cf. *hash-consing* [7]).

In conclusion, it should be clear that our system fully supports the problem scenario. Dead branches are not explicitly represented, but their presentation can be derived easily. 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 our system.

An important practical consideration that we have ignored is incremental performance. As presented, synchronisation proceeds top-down and thus must traverse the entire reachable trace. An efficient implementation would proceed bottom-up, ignoring unaffected parts of the trace, exploiting the fact that the trace represents precisely the dependencies between sub-computations.

4 Related work

Reified evaluation arises in Acar's *self-adjusting computation* [8] (SAC), as well as in program visualisation and debugging tools. Reactivity is central to visual programming, spreadsheet languages, functional reactive programming (FRP) [9], 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 notions. A

```
equal = v12:False
+- val v7 = v7:Cons(v13:Zero,v14:Nil)
+- val v10 = v10:Cons(v8:Zero,v9:Cons(v23:Zero,v24:Nil))
+- [x: v7, y: v10] case_List = v12:False
   +- x = v7:Cons(v13:Zero,v14:Nil)
   +- [x': v13, y': v14] case_List = v20:False
      +- y = v10:Cons(v8:Zero,v9:Cons(v23:Zero,v24:Nil))
      +- [a: v8, b: v9] case_Bool = v19:False
         +- equal_nat = v16:False
         |  +- a = v8:Zero
         |  +- x' = v13:Zero
         |  +- [x: v8, y: v13] case_Nat = v16:False
         |     +- x = v8:Zero
         |     +- [x': v21] case_Nat = v22:False
         |        +- y = v13:Zero
         |        +- [] val v4 = v4:False
         +- [] val v0 = v0:False
[] case_List = v11:False
+- y = v10:Cons(v8:Zero,v9:Cons(v23:Zero,v24:Nil))
+- [a: v8, b: v9] val v2 = v2:False
[] case_Nat = v15:True
+- y = v13:Zero
+- [] val v6 = v6:True
[] equal = v18:True
+- b = v9:Cons(v23:Zero,v24:Nil)
+- y' = v14:Nil
+- [x: v9, y: v14] case_List = v18:True
   +- x = v9:Cons(v23:Zero,v24:Nil)
   +- [] case_List = v17:True
      +- y = v14:Nil
      +- [] val v3 = v3:True
```

(a) State 6′ (unstable)

```
equal = v12:False
+- val v7 = v7:Cons(v13:Zero,v14:Nil)
+- val v10 = v10:Cons(v8:Zero,v9:Cons(v23:Zero,v24:Nil))
+- [x: v7, y: v10] case_List = v12:False
   +- x = v7:Cons(v13:Zero,v14:Nil)
   +- [x': v13, y': v14] case_List = v20:False
      +- y = v10:Cons(v8:Zero,v9:Cons(v23:Zero,v24: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:Cons(v23:Zero,v24:Nil)
            +- y' = v14:Nil
            +- [x: v9, y: v14] case_List = v18:False
               +- x = v9:Cons(v23:Zero,v24:Nil)
               +- [x': v23, y': v24] case_List = v25:False
                  +- y = v14:Nil
                  +- [] val v1 = v1:False
[] case_List = v11:False
+- y = v10:Cons(v8:Zero,v9:Cons(v23:Zero,v24:Nil))
+- [a: v8, b: v9] val v2 = v2:False
[x': v21] case_Nat = v22:False
+- y = v13:Zero
+- [] val v4 = v4:False
[] val v0 = v0:False
[] case_List = v17:True
+- y = v14:Nil
+- [] val v3 = v3:True
```

(b) State 7 (stable)

Fig. 8: Evaluation store, states 6′ & 7

detailed analysis of prior work on spreadsheet languages remains to be done. Subtext [10] 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 *re-execute* of code fragments to synchronise the state of adaptive computations when the modifiables they read have changed. This 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.

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) [11].

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 [12], 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 Sam Davis, Jeff Foster, Kevlin Henney, Paul Levy, Robin Message, Tom Stuart, John Zabroski and three anonymous reviewers all provided helpful comments on earlier drafts.

References

1. Perera, R.: A first-order interactive programming language. Technical Report CSR-09-09, University of Birmingham, School of Computer Science (November 2009)
2. Shapiro, E.Y.: Algorithmic program debugging. ACM Distinguished Dissertations. MIT Press, Cambridge, MA, USA (1983)
3. Kishon, A., Hudak, P.: Semantics directed program execution monitoring. Journal of Functional Programming **5**(04) (1995) 501–547
4. Tanimoto, S.L.: VIVA: A visual language for image processing. Journal of Visual Languages and Computing **1**(2) (1990)
5. 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
6. Michie, D.: Memo functions and machine learning. Nature **218** (1968) 19–22
7. Filliâtre, J.C., Conchon, S.: Type-safe modular hash-consing. In: ML '06: Proceedings of the 2006 workshop on ML, New York, NY, USA, ACM Press (2006) 12–19
8. Acar, U.A.: Self-Adjusting Computation. Phd thesis, Department of Computing Science, Carnegie Mellon University (2005)
9. 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
10. 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
11. 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)
12. 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)