A First-Order Interactive Meta-Language

Roly Perera
School of Computer Science, University of Birmingham, United Kingdom
roly.perera@dynamicaspects.org

Abstract

TODO

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments – interactive environments; D.3.2 [*Programming Languages*]: Language Classifications – data-flow languages; D.3.4 [*Programming Languages*]: Processors - code generation, incremental compilers, run-time environments

General Terms Languages

Keywords interactive programming, incremental computation, intensional programming, data-flow languages

1. Introduction

My research concerns a declarative programming model I call *interactive programming*. Rather than a new kind of programming language, interactive programming is a new way of implementing declarative languages. The basic idea is to treat the evaluation of a closed term not as a *transient activity* that yields a single value, but as a *persistent structure* which responds to changes like a spreadsheet. External agents can interact with the structure, causing the system to adjust into a different state embodying a new computation. "Writing a program" in this paradigm is to start with a null computation and manipulate it into the desired computation, blurring the familiar distinctions between compile-time and run-time, between programmer and end-user, and between development tool and application.

Interactive programming enjoys several advantages over traditional systems. First, like spreadsheets, the synchronisation of state in response to interaction is an automatic feature, rather than something which has to be specially coded by the programmer. A related advantage is that applications need not resort to imperative I/O in order to interact. Instead, changes in the values of program constants serve as inputs, and changes in computed values as outputs. In addition, the persistence of the computation means that prior computations can be *reused*, allowing the system to respond efficiently. Finally, the entire execution history is available for inspection, making debugging straightforward.

This report is organised as follows. §2 presents a condensed version of the motivation offered elsewhere (Perera 2008), which reviewed some of the shortcomings of existing functional and procedural programming paradigms. §3 outlines the interactive programming approach, where support for interactivity comes "built in". I mention some connections to existing work, and some of the main challenges. §4 summarises progress to date, and §5 discusses future plans.

2. Motivation

> Besides, it has all the charm of inventing the science of navigation while already onboard ship.
>
> Robin Milner, on developing a "logical theory of interaction" (Milner 2006)

Almost all software is interactive. Yet it remains extremely difficult to build robust interactive systems using modern languages. Somehow we seem to be lacking a programming paradigm that supports interaction in any kind of straightforward way.

Arguably the aspect of interaction that gives us the most trouble is arranging for systems to exhibit some kind of persistent state. Instead of executing all at once and then disappearing, they must remain idle, waiting for the next interaction from the environment. How they respond to this interaction will typically depend on what has happened to them so far. We can implement this sort of stateful behaviour with imperative languages, but only at an unacceptable complexity cost. Functional languages fare even worse, in that they struggle to accommodate interaction at all.

Before we turn to an alternative in §3, I shall briefly expand on these points. These observations are necessarily over-simplified, but hopefully they get to the key issues.

2.1 Imperative languages

With imperative languages, we can achieve interactivity, but only via a rather indirect implementation technique. I shall call this the *state patching* approach. A state patching system consists of a store, used to encode the persistent state of the system, plus a family of imperative routines, called *event handlers*, that determine how the store evolves in response to various kinds of interaction. A notification scheme is employed to dispatch the event handlers as interactions occur; the job of each event handler is to apply a suitable "patch" to the state, so that the system maintains a consistent interpretation of its environment. State patching is the principle that underlies the Model-View-Controller (MVC) architectural pattern (Goldberg and Robson 1983).

For very simple programs, relying on side-effects to manage state is relatively harmless. But in general the complexity of the state-patching approach is entirely disproportionate to the complexity of the desired behaviour. Programming becomes dominated by the task of maintaining data integrity, with control flow and concurrent access to shared state the central concerns (Moseley and Marks 2006). The essential program logic is lost amongst all the state-management machinery. Unfortunately this approach is so ubiquitous that we rarely notice how problematic it is.

2.2 Functional languages

Pure functional programming (FP) of course avoids all these problems. But it does so by restricting itself to an algorithmic view of programs in which interaction is not really possible. In a pure language, we cannot simply extend a program with a global store and event handlers, as per the state patching approach. Instead the program must be provided with all its inputs at the outset, and then execute as a black box, spitting out a single result at the end. Interaction can only be simulated, restricting programmers to "closed world" scenarios where all interactions are known up front.

Since this limitation renders FP largely useless, real-world functional languages make various compromises. The Haskell solution is to require every program to have an outer layer of imperative I/O code. This wrapper layer explicitly reads from an input stream, passes the input to pure code, and explicitly writes the computed value to an output stream. Since Haskell is lazy, output can interleave with input, allowing the program to exhibit the persistent state characteristic of interactive systems.

In effect, Haskell programs communicate by pretending to be C programs. Functional programs are relegated to the status of unwelcome guests in a procedural environment, and lose the most important benefit of functional programming, compositionality. Like (Holyer et al. 1995), the approach I am exploring inverts this state of affairs, making composition and interaction trivial, and demoting most cases of the imperative to mere *observations of the interaction* between parts of a purely functional program.

3. Interactive programming

To introduce interactive programming, we begin with the following observation. *What systems mostly do is compute functions*. Consider how an air-traffic control system calculates an interpretation of radar data, augmented with additional information about flight numbers and other details. Or how a computer game renders a view of a virtual world from some hypothetical vantage point. In both cases, the application's job is essentially to compute a function.[1] If this is a fair characterisation of real-world systems, then it is natural to wonder whether we can somehow adapt FP to accommodate interaction more directly.

As it happens a declarative take on interaction is already familiar to us from the world of spreadsheets. At every idle state, a spreadsheet embodies the computation of a pure function of its argument cells. An interaction (changing a cell) mutates either a formula or an argument, and causes the spreadsheet to adjust into a new idle state embodying a different computation. The spreadsheet achieves automatically what the imperative state-patching approach sets out to achieve manually: the transformation of discrete changes in model state into discrete changes in view state, to reflect the function being computed by the view.

But of course spreadsheets are not fully-fledged programming languages. Some work has already been done to push spreadsheet languages in that direction, through the addition of features like recursion (Burnett et al. 2001) and higher-order functions (de Hoon et al. 1995). My research approaches the topic from the other direction, the aim being to make programs written in a pure higher-order language like lambda calculus behave like spreadsheets.

The basic idea is to identify the *state* of an interactive system with the *computation* of a (possibly higher-order) function at some particular argument. An interaction with the system that changes its state can then be only construed in one of two ways: either as a mutation of the argument, or a mutation of the function itself. An example of the first situation would be when the spatial location of the "point of view" changes in a computer game: the argument to the visualisation function has changed, although the function itself has not. For the second scenario, consider what happens when a player switches from a "perspectival" visualisation to an "orthographic" one showing the locations of all nearby players on a map. In this case the interaction has resulted in the selection of a different function, leaving the argument (the location of the player) unchanged.

In effect the language runtime provides external agents with the ability to explore nearby variants of the program without re-executing the whole thing from scratch. Programs no longer run once and then disappear, but hang around so that they can be interacted with. For convenience, I call a language implemented in this way *interactive*, although it is probably more accurate to say that it is the *implementation*

[1] Higher-order functions also make an appearance in these informal accounts of how systems behave. With a word processor, for example, it is quite natural to think of the system as *mapping* a rendering function to a sequence of characters in a document to generate their glyphs, a function which changes when the user selects a different font or point size.

of the language which is "interaction-enabled", rather than the language itself.

A central benefit of interactive languages is that the synchronisation of state in response to interaction is an automatic feature of how the language works, rather than something which has to be explicitly coded by the programmer. In the visual programming literature, this property is called *liveness* (Tanimoto 1990). Equally important is that systems written in such languages need not resort to imperative I/O in order to interact. Any change in the value of a constant in the program counts as an *input*; conversely, any change in a computed value is an *output*. The program need have no explicit notion of I/O at all. The simple power of this declarative approach to I/O is nicely demonstrated by a feature called "dynamic interactivity" which appeared in version 6 of *Mathematica*.[2]

Interactive languages are "spreadsheets all the way down". This contrasts with a traditional spreadsheet, where the execution of a formula has no observable internal structure. The computation becomes a persistent *data structure*, rather than an *activity*, with the process of building this structure analogous to reduction in a traditional non-interactive language. The persistence of the structure has two important consequences. First, there is the potential to reuse parts of it when computing the new state. Indeed, this sharing of structure between adjacent states is essential to an efficient implementation. This connects my research to the incremental computation literature, in particular the "self-adjusting computation" of Acar (Acar 2005).

The second consequence of persistence is that the full history of a computation is available for debugging. One can simply inspect the structure to see why the computation produced the value it did. This idea has been explored in the experimental interactive language Subtext (Edwards 2005).

4. Progress report

The rest of this report describes what I have achieved so far and how I hope to move forward. My overall goal is to develop some kind of formal account of interactive languages, and of course, implement a prototype system to demonstrate that the approach works in practice.

4.1 An interactive meta-language

The main technical decision I have taken so far is to treat interactivity at the meta-level, via a meta-language called IML (for *interactive meta-language*). By doing so we obtain a crucial simplification, namely the ability to treat interaction in a first-order setting. In fact we can restrict ourselves further: we need only consider interaction in the form of the mutation of *data values*, rather than mutation of code. IML thus need not not be fully interactive in the sense described in the last section, where I also envisaged interaction with functions.

This is because, at the meta-level, object-language terms are mere data values. By virtue of being implemented in IML, the object-language interpreter will react to changes in its arguments, producing the corresponding changes in its computed output. But because these arguments are arbitrary object-language terms, we effectively "inherit" interactivity from the first-order setting of IML into the higher-order setting of the object language. We can thus exploit a clean separation of program and data for our formal treatment of interaction, but still support interaction with code at the object-language level.

Another advantage of the tiered approach is that we can restrict ourselves to call-by-value (CBV) in IML's operational semantics, without similarly constraining individual object languages. CBV interacts better with memoisation, a crucial component of incrementality. However I concede that my envisaged "bootstrapping" of interactivity into a higher-order setting is unlikely to be straightforward when it comes to complex operational characteristics like incremental performance. But I hope to validate it at least on a conceptual level soon by exhibiting interactive IML implementations of both a pure functional language and a pure object-oriented language.

The rest of this section describes IML in more detail. The language is pure (non-imperative), with general recursion and named algebraic data types. The operational semantics for interaction makes essential use of imperative notions. This in some sense reflects our use of imperative languages to implement interactivity today, but significantly the imperative elements appear another meta-level up, in the meta-*meta*-language, rather than in IML code. In the implementation, this imperative meta-meta-language is actually embedded in Haskell, and so we have something like the rather complicated picture of Figure 1.

We shall consider the various aspects of IML via a running example. Where there are formal details inessential to the discussion, I shall refer the reader to the relevant section (indicated by a section number beginning A-) of the Appendix.

4.2 Running example

Our example program is given in Figure 2. It is a toy problem, but faithful to the intention of using IML as a meta-language. Our chosen object language is untyped lambda calculus, and the purpose of the example program is simply to compare two lambda terms for syntactic equality. Scaling this example to a full interpreter will be the next validation step, but this is what we have for now.

IML needs a convenient representation for object-language terms. To this end, we parameterise the language by a *type environment* Θ, providing a fixed set of named *algebraic data types*. We write $B : \{\overrightarrow{A_c}\}_{c \in C}$ to indicate that, in the

[2] http://www.wolfram.com/products/mathematica/newin6 /content/DynamicInteractivity/

the location in which a constant is stored. The syntactic form of a location is given by the loc rule in Figure 3. Note that these loc-terms appears only in pre-allocated code, not in user code.

There is a clear connection here to functional reactive programming (FRP) (Elliott 2008) and other "intensional" models of programming, as the effect of pre-allocation is to lift constants to streams of values, and functions to stream functions. Applied to our example program, pre-allocation produces the following store:

v:	$A.c(\overrightarrow{u})$
Loc	*Value*
0:	`Nat.Zero`
1:	`Nat.Succ(0)`
2:	`Nat.Zero`
3:	`Lam.Var(2)`
4:	`Lam.Abs(1, 3)`
5:	`Nat.Zero`
6:	`Nat.Succ(5)`
7:	`Nat.Zero`
8:	`Lam.Var(7)`
9:	`Lam.Abs(6, 8)`

We use the notation $v : A.c(\overrightarrow{u})$ to mean that the location v is assigned constructor c of data type A, with arguments given by the values held at locations $\overrightarrow{u}$ in the same store. The effect of pre-allocation on our example program is to transform the body of the program from

```
equal(Abs(Succ(Zero),Var(Zero)),
      Abs(Succ(Zero),Var(Zero)))
```

to

```
equal(loc(4),loc(9))
```

replacing the two arguments of `equals` by the terms $\mathsf{loc}_{4,\mathtt{Lam}}$ and $\mathsf{loc}_{9,\mathtt{Lam}}$. These locations identify two different structures in the store, which initially happen to encode the same value of type `Lam`. We also note that locations 4 and 9 are the *root locations* of this store, because there are no locations which mention them as children. The root locations of the pre-allocation store correspond exactly to the loc-terms that appear in the pre-allocated program.

A final observation about pre-allocation is that we only pre-allocate constants contained in the program body, not in the function definitions provided in the function environment. This is consistent with our separation of code and data at the meta-level, as we avoid having to treat IML function definitions as having any parts which are computed. There is no loss of expressiveness, since any constant appearing in a function definition can always be turned into an explicit parameter and provided in the program body instead.

4.4 Initial evaluation

Having pre-allocated our program, we now compute its initial state, through a process called *evaluation*. This differs from evaluation in a batch-mode functional language in that it yields a persistent structure, rather than a transient result.

Evaluation takes place in the context of the pre-allocation store, and so we can think of the root locations of this store as the "inputs" to the program provided by the environment. In fact, we can evaluate the program in *any* store that provides suitably typed values for these root locations. To make this slightly more precise, we informally define

Definition 4.1 (Initial store) *For any closed term M, define an* initial store for M *to be any store τ satisfying*

$$\mathsf{loc}_{v,A} \textit{ is a sub-term of } M \Rightarrow v : A \in \tau$$

omitting any mention of M if it is clear from the context. Allowing IML programs to contain constants which are then pre-allocated is really just a convenient way of allowing the programmer to write a program dependent on a store context and simultaneously give an initial store for it.

The evaluation relation is call-by-value (CBV); it is defined in Figure 4. It makes use of a global allocation operator **new**(), given in Definition A.15. The judgement $M \Downarrow^{\tau} o, \tau_1$ asserts that the evaluation of M in initial store τ produces a *store delta* τ_1, plus a result stored in location $o \in \tau_1$. A store delta (Definition A.13) is simply a store which can be "added" to another store because their domains do not overlap.

An important aspect of evaluation is that the interim values produced during evaluation are also kept in the store. (CBV ensures that these intermediate results are always fully evaluated values, and thus suitable for storing in locations.) This allows the *identity* of an evaluation to persist over changes to the value it computes. For this reason the substitution operation $\overrightarrow{v}^* M$, defined in §A.21, substitutes loc-terms holding the relevant values, rather than the values themselves, for free identifiers. Otherwise, the evaluation rules for IML are similar to those for any other pure CBV language.

4.4.1 Evaluations as persistent data structures

IML programs exhibit persistent state: rather than running through to termination and then exiting, they move between stable configurations representing their interpretation of particular inputs. Van Roy and Haridi call these interim states *partial terminations* (Van Roy and Haridi 2004).

This persistence requirement suggests a particular view of evaluation. (I alluded to this in the last section when I mentioned the "identity" of an evaluation.) In a batch-mode language, terms are data structures, and we usually think of evaluation as an activity which operates on terms. In IML, we treat evaluations as structures too, and we consider *interaction*, in the form of a modification to the initial store, to initiate an activity which operates on evaluations. The activity is something like the re-calculation that occurs in a spreadsheet when a cell changes. Since the IML program merely *adjusts* its state so that the computation agrees with its inputs, rather than re-computing it from scratch, we call this activity *synchronisation* (§4.5).

To formalise this notion, we take the inductive definition of our evaluation relation to characterise a *data type* in the meta-meta-language, roughly the data type of (converging) *evaluations*, where an "evaluation" is understood in the following technical sense.

Definition 4.2 (Evaluation step) *Define*

$$\mathit{EvalStep} \overset{\mathit{def}}{=} \sum_{A \in \Theta} \left(\mathit{Term}_{\vdash A} \times \sum_{\tau \in \mathit{Store}} (\mathrm{dom}(\tau) \times \mathit{Store}_{\tau+}) \right)$$

where $\mathit{Store}_{\tau+}$ denotes the set of τ-deltas for any store τ.

Define an *evaluation step* to be any element of EvalStep. Considering $M \Downarrow^{\tau} o, \tau_1$, for any $\vdash M : A$, to denote an evaluation step (A, M, τ, o, τ_1), we now interpret Figure 4 as an EvalStep-sorted signature Σ.

Definition 4.3 (Evaluation) *Define the family of sets*

$$\{\mathit{Eval}_{M \Downarrow^{\tau} o, \tau_1}\}_{M \Downarrow^{\tau} o, \tau_1 \in \mathit{EvalStep}}$$

to be an initial Σ-algebra.

Then define an *evaluation* to be an element of $\mathrm{Eval}_{M \Downarrow^{\tau} o, \tau_1}$ for some $M \Downarrow^{\tau} o, \tau_1$. Meta-variables e and variants range over evaluations; we write $e\langle M \Downarrow^{\tau} o, \tau_1 \rangle$ for any $e \in \mathrm{Eval}_{M \Downarrow^{\tau} o, \tau_1}$.

We should think of an evaluation $e\langle M \Downarrow^{\tau} o, \tau_1 \rangle$ as a derivation tree for, or equivalently proof of, the convergent CBV evaluation of M in τ. Such evaluations constitute the persistent states of running IML programs. Figure 5 shows the initial state produced by evaluating our example program in its pre-allocation store.

The main purpose of retaining the entire tree of dependent sub-computations involved in calculating one state is to permit the reuse of shared sub-computations when transitioning to the next state. In reality, some kind of trade-off between persistence and re-computation is likely to be required, which connects this work to Acar's.

4.5 Synchronisation

An interaction in effect poses a *subjunctive* ("what if?") question. "What would have the evaluation have looked like had the computation begun in a different initial store?" The answer to such a question is provided by *synchronisation*, which adjusts (edits) the "current" evaluation until it is consistent with what evaluation would have produced in that store.

Figure 6 defines the synchronisation judgement $e \overset{\tau'}{\curvearrowright} e'$, defined mutually inductively with a similar judgement $\overrightarrow{e} \overset{\tau'}{\curvearrowright} \overrightarrow{e'}$ for sequences of evaluations, omitted in the interests of brevity. (In Figure 6, an operation symbol of Σ should be taken to mean its interpretation in the initial Σ-algebra.) The responsibility of the synchronisation rules is to maintain as

```
equal(...)->44 []
|
+- loc(4)->10 [10=Abs(1,3)]
|
+- loc(9)->11 [11=Abs(6,8)]
|
`- case->44 [44=True]
   |
   +- loc(10)->12 [12=Abs(1,3)]
   |
   `- case->43 [43=True]
      |
      +- loc(11)->13 [13=Abs(6,8)]
      |
      `- and(...)->42 []
         |
         +- equal_nat(...)->26 []
         |  |
         |  +- loc(6)->14 [14=Succ(5)]
         |  |
         |  +- loc(1)->15 [15=Succ(0)]
         |  |
         |  `- case->26 [26=True]
         |     |
         |     +- loc(14)->16 [16=Succ(5)]
         |     |
         |     `- case->25 [25=True]
         |        |
         |        +- loc(15)->17 [17=Succ(0)]
         |        |
         |        `- equal_nat(...)->24 []
         |           |
         |           +- loc(0)->18 [18=Zero]
         |           |
         |           +- loc(5)->19 [19=Zero]
         |           |
         |           `- case->24 [24=True]
         |              |
         |              +- loc(18)->20 [20=Zero]
         |              |
         |              `- case->23 [23=True]
         |                 |
         |                 +- loc(19)->21 [21=Zero]
         |                 |
         |                 `- True->22 [22=True]
         |
         +- equal(...)->39 []
         |  |
         |  +- loc(8)->27 [27=Var(7)]
         |  |
         |  +- loc(3)->28 [28=Var(2)]
         |  |
         |  `- case->39 [39=True]
         |     |
         |     +- loc(27)->29 [29=Var(7)]
         |     |
         |     `- case->38 [38=True]
         |        |
         |        +- loc(28)->30 [30=Var(2)]
         |        |
         |        `- equal_nat(...)->37 []
         |           |
         |           +- loc(2)->31 [31=Zero]
         |           |
         |           +- loc(7)->32 [32=Zero]
         |           |
         |           `- case->37 [37=True]
         |              |
         |              +- loc(31)->33 [33=Zero]
         |              |
         |              `- case->36 [36=True]
         |                 |
         |                 +- loc(32)->34 [34=Zero]
         |                 |
         |                 `- True->35 [35=True]
         |
         `- case->42 [42=True]
            |
            +- loc(26)->40 [40=True]
            |
            `- loc(39)->41 [41=True]
```

Figure 5. Initial state for example program

invariants the store conditions established by evaluation, in particular the fact that the store delta associated with an evaluation always contains an entry for its output location. This “correctness” intuition is expressed by Theorem 4.4 below, which still needs to be proven. The leaf rule for e_loc ensures that if a location being read has changed, that its new value is stored in the output location in the modified store. The rules for composite evaluations synchronise their immediate sub-evaluations, with additional behaviour only for e_case, in the event that the value being case-analysed has changed, either so that it has a different constructor, or so that it has the same constructor but new child locations. In either case, synchronisation removes the existing clause evaluation and inserts a new one in its place, although more fine-grained behaviour is in principle possible.

Figure 7 shows the results of synchronisation in the context of our example. If after producing the initial evaluation shown in Figure 5, we changed one of the arguments of `equal` so that the terms were no longer syntactically identical, then synchronising the evaluation in the new initial store would give the evaluation shown, where the output location is now set to `False` in the adjusted store delta.

On the other hand, if we edited the terms being compared so that they were larger, but still syntactically equal, we would observe more recursion unfolding in the persistent structure, but the output location would still be set to `True`. This shows how computation is “transparent” in this model: we can observe not only the result, but how it was computed.

Theorem 4.4 (Correctness) *For any closed term $M : A$, any initial stores τ, τ', any delta τ_1 of τ, any deltas ${\tau_1}'$ and ${\tau_1}''$ of τ', and any evaluations e, e' where $e\langle M \Downarrow^{\tau} o, \tau_1\rangle$ and $e'\langle M \Downarrow^{\tau'} o, {\tau_1}''\rangle$,*

$$(M \Downarrow^{\tau} o, \tau_1) \wedge (M \Downarrow^{\tau'} o', {\tau_1}') \wedge (e \overset{\tau'}{\curvearrowright} e') \quad \Rightarrow$$
$$o' \uparrow (\tau' + {\tau_1}') = o \uparrow (\tau' + {\tau_1}'')$$

The *lifting* $o \uparrow \tau$ of o in τ is given in Definition A.16.

5. Work plan

Re-implement synchronisation imperatively The synchronisation relation defined in Figure 6 is currently implemented functionally, but needs to be rewritten in an imperative style. An imperative implementation is essential if sub-computations of the new state which are shared with the previous state are to be reused, since we can then create the new state by *editing* the existing state. This will involve determining which components of an “evaluation” constitute its persistent identity (and which are therefore invariant under mutation) vs. those which constitute its value (and so may vary with mutation). The most relevant work here is Acar’s *self-adjusting computation*, which combines memoisation with change propagation (synchronisation).

Evaluate trace distance and trace stability I also need to study Acar’s incremental analysis technique, trace stability, and the related notion of an algorithm being “monotone”

```
equal(...)->44 []
|
+- loc(4)->10 [10=Abs(1,3)]
|
+- loc(9)->11 [11=Abs(6,8)]
|
‘- case->44 [44=False]
   |
   +- loc(10)->12 [12=Abs(1,3)]
   |
   ‘- case->43 [43=False]
      |
      +- loc(11)->13 [13=Abs(6,8)]
      |
      ‘- and(...)->42 []
         |
         +- equal_nat(...)->26 [1.26939(()1..27429(o)1.26939(c)1.27(()1.26939(1)1.274T6939(])1.27429]T 6 TL T[(
         +6110)->12 [.=.>2(1,3)]
         |- case->43 [43]Fals
         T  0 0 0 1  0 0 0 1  q 10 0 0 1070 0 cm T 52 4.98132 T4.9817891156oo1
```

with respect to some class of input change. A trace is a representation of a computation which identifies all the function calls performed by that computation, and so is subsumed by our notion of an evaluation (Definition 4.3). It should be possible to adapt Acar's definition of the distance between two traces to work with evaluations. Acar uses trace distance to show that (in the monotone case) his change propagation algorithm transforms one computation into another in time bounded by the distance between the traces.

Demonstrate declarative I/O and compositionality Interactive computations should naturally compose, so that changes in the computed values ("outputs") of the former constitute changes in constants ("inputs") of the latter. This seems like a key advantage of my approach, and therefore it should be clear how this comes about. Dually, it should be possible to *decompose* a non-interactive system into interacting parts. Pre-allocation may turn out to be a special case of decomposition, where one of the parts is trivial.

Non-trivial example: interpreter for higher-order language Extend my example program to a non-trivial example. An important assumption I have made is that object languages will "inherit" interactivity from the IML meta-language, and this needs to be validated as soon as possible. So, it should be possible to write an interpreter in IML for a higher-order language like lambda calculus, edit the program provided as "input" to the interpreter, and have the interpreter's evaluation of that program update automatically.

Interactive demo An interactive demo would be very useful for conveying the basic idea. This would mean rewriting the prototype in something like F# to gain access to graphics libraries; the current Haskell prototype only simulates interaction.

Relationship to FRP and other intensional languages FRP and other dataflow languages are *intensional*, meaning that expressions take on different values in different *contexts* or environments. This seems to be essentially how I am treating state and interaction; initial stores serve as contexts and interactions as context switches. FRP therefore needs careful study. Uustalu and Vene's comonadic analysis of dataflow programming may also be relevant (Uustalu and Vene 2006).

A. Appendix

A.1 Syntax

Definition A.1 (Type context) *Fix a* type context Θ *a finite set, and define a* type *to be any element of* Θ. *Meta-variables* A, B *and variants range over types.*

Definition A.2 (Type environment) *Fix a set of constructor names* I *containing the distinguished element* ***null***. *Define*

$$\mathit{TypeDef} \overset{\mathit{def}}{=} \sum_{\{\mathbf{null}\} \subseteq C \subseteq_{\mathrm{fin}} I} (C \to \Theta^*)$$

and then define a type environment *to be any function* $f : \Theta \to \mathit{TypeDef}$ *with* $(\mathbf{null}, \epsilon) \in f(A)$ *for every type* A.

Write $\overrightarrow{A}$ for an element of Θ^*, viz. a finite sequence of types, and assume a similar convention for other meta-variables. From now on assume an ambient type environment typeDef, and write $B : \{\overrightarrow{A_c}\}_{c \in C}$ to denote $B \in \Theta$ with typeDef$(B) = \{\overrightarrow{A_c}\}_{c \in C}$.

Note that in the concrete syntax used for our example, the distinguished constructor name **null** never occurs, and a `case` expression with an argument of type A and clauses of type B is translated into the abstract syntax as a case-expression with an extra clause mapping $A.$**null** to $B.$**null**.

Definition A.3 (Function context) *Fix a* function context $(\Delta, \mathit{sig}_\Delta)$, *a pair of a finite set* Δ *and a function* $\mathit{sig}_\Delta : \Delta \to \Theta^* \times \Theta$. *Write* $f(\overrightarrow{A}) : B$ *to denote* $f \in \Delta$ *with* $\mathit{sig}_\Delta(f) = (\overrightarrow{A}, B)$.

We use de Bruijn indices to represent terms; thus syntactic equivalence is also α-equivalence.

Definition A.4 (Context) *Define a* context $(\Gamma, \mathit{type}_\Gamma)$ *to be a pair of a finite set* Γ *of natural numbers, called* identifiers, *and a function* $\mathit{type}_\Gamma : \Gamma \to \Theta$.

For convenience we will often identify Γ with $(\Gamma, \mathrm{type}_\Gamma)$. Meta-variable x and variants range over identifiers. Write $x : A \in \Gamma$ to denote $x \in \Gamma$ with $\mathrm{type}_\Gamma(x) = A$, and write $\varnothing$ for the empty context. Define Cxt to be the set of all contexts.

Definition A.5 (Context extension) *For any context* Γ *and any type* A, *define* Γ, A *to be the context*

$$\{\mathbf{zero}\} \cup \{\mathbf{succ}(x) \mid x \in \Gamma\}$$

with ***zero*** $: A \in \Gamma, A$ *and* ***succ***$(x) : \mathit{type}_\Gamma(x) \in \Gamma, A$. *Then for any context* Γ *and any sequence of types* $\overrightarrow{A}$, *define* $\Gamma, \overrightarrow{A}$ *in the obvious way.*

Definition A.6 (Location) *Define Loc to be the natural numbers, and a* location *to be any element of Loc. Meta-variables* v, u *and variants range over locations.*

The terms of the IML language are now defined by the grammar given in Figure 3. Meta-variables M, N and variants range over terms. We write $\Gamma \vdash M : A$ to assert that M has type A in context Γ; this notation generalises to sequences of terms $\overrightarrow{M}$ in the obvious way.

Definition A.7 (Function environment) *Define a* function environment *to be any element of*

$$\prod_{f(\overrightarrow{A_f}) : B_f \in \Delta} \mathit{Term}_{\varnothing, \overrightarrow{A_f} \vdash B_f}$$

Definition A.8 (Program) *Define a* program (def, M) *of type* A *to be a pair of a function environment def and a closed term* $\vdash M : A$.

A.2 Evaluation

A.2.1 Stores

Definition A.9 (Store entry) *Define the set*

$$\mathit{StoreEntry} \overset{\mathit{def}}{=} \sum_{A:\{\overrightarrow{B_{A,c}}\}_{c\in C_A}, c\in C_A} \left(\mathit{Loc}^{\mathrm{dom}(\overrightarrow{B_{A,c}})}\right)$$

and define a store entry $A.c(\overrightarrow{u})$ *to be any* $(A, c, \overrightarrow{u}) \in$ *StoreEntry. Informally, a store entry is a type* A*, a constructor* c *of* A*, and a sequence of* child locations $\overrightarrow{u}$ *whose length equals the arity of* c*.*

For any location v, write $v : A.c(\overrightarrow{u})$ for the pair $v \mapsto A.c(\overrightarrow{u})$. Also write $v : A$ to mean $v : A.c(\overrightarrow{u})$ for some c and $\overrightarrow{u}$; we say that v is of type A. Generalise this latter notation to sequences of location bindings $\overrightarrow{v} : \overrightarrow{A}$ in the obvious way.

Definition A.10 (Store) *Define a* store σ *to be a finite partial map from locations to store entries that satisfies*

1. Closure: *for any* $A : \{\overrightarrow{B_c}\}_{c\in C}$*,* $v : A.c(\overrightarrow{u}) \in \sigma \Rightarrow \overrightarrow{u} : \overrightarrow{B_c} \in \sigma$
2. Well-foundedness: *there are no infinite chains in* σ *formed by the child relation.*

Meta-variables σ and variants range over stores.

Definition A.11 (Roots) *For a store* σ*, define roots*(σ) *to be the set of locations* $v \in \sigma$ *such that there is no* $u \in \sigma$ *with* v *a child of* u*.*

Definition A.12 (Store ordering) *Define the following partial order* $\leq$ *on stores:*

$$\sigma_0 \leq \sigma_1 \Leftrightarrow \forall v.\forall c.\forall \overrightarrow{u}.(v : A.c(\overrightarrow{u}) \in \sigma_0 \Rightarrow v : A.c(\overrightarrow{u}) \in \sigma_1)$$

Stores can also describe *deltas* of other stores. Meta-variables τ and variants range over stores construed as deltas of some other store; we omit any mention of the other store if it is clear from the context.

Definition A.13 (Store delta) *For any store* τ*, define a* delta of τ *to be any store* τ_1 *with* $\mathrm{dom}(\tau) \cap \mathrm{dom}(\tau_1) = \varnothing$*.*

Observe that any store is a delta of the empty store.

Definition A.14 (Store addition) *For any store* σ *and any delta* τ*, define* $\sigma + \tau$ *to be the store* $\sigma \oplus \tau$*.*

Definition A.15 (New location) *For any type* A*, define* ***new***$(A.c(\overrightarrow{u}))$ *to be an atomic operation that, in the context of an ambient global store* σ*, non-deterministically chooses any location* $v \notin \mathrm{dom}(\sigma)$*, sets* σ *to be* $\sigma \cup \{v : A.c(\overrightarrow{u})\}$*, and then returns* v*.*

We observe that although the evaluation relation is non-deterministic (due to the non-determinism of allocation), it is deterministic up to *lifting*.

Definition A.16 (Lifting) *For any term* M *and any initial store* τ*, the* lifting $M \uparrow \tau$ of M in τ *to be the term that results from recursively replacing every loc-term in* M *by the equivalent ctr-term. [TODO: make precise.]*

We also write $o \uparrow \tau$ to mean $\mathsf{loc}_{o,A} \uparrow \tau$, for any $o : A \in \tau$.

Theorem A.17 (Determinism of evaluation) *For any store* τ*, any* τ*-deltas* $\tau_1, \tau_1{}'$*, any closed term* M *and any location* o*,*

$$M \Downarrow^{\tau} o, \tau_1 \wedge M \Downarrow^{\tau} o, \tau_1{}' \quad \Rightarrow \quad o \uparrow \tau_1 = o \uparrow \tau_1{}'$$

Proof TODO □

A.2.2 Substitution

Definition A.18 (Substitution) *For any contexts* Γ *and* Γ'*, define*

$$\mathit{Subst}_{\Gamma;\Gamma'} \overset{\mathit{def}}{=} \Gamma \to 1 + \mathit{Loc}$$

and define a substitution $k : \Gamma \to \Gamma'$ *to be any* $k \in \mathit{Subst}_{\Gamma;\Gamma'}$*.*

Definition A.19 (Substitution extension) *For any substitution* $k : \Gamma \to \Gamma'$ *and any type* A*, define* $k, A : \Gamma, A \to \Gamma', A$ *to be the substitution*

$$\{\textbf{\textit{zero}} \mapsto \textbf{\textit{nothing}}\} \cup \{\textbf{\textit{succ}}(x) \mapsto k(x) \mid x \in \Gamma\}$$

Then define $k, \overrightarrow{A} : \Gamma, \overrightarrow{A} \to \Gamma', \overrightarrow{A}$ *in the obvious way.*

Definition A.20 (Substitution into terms) *Define TermSubst to be the set*

$$\prod_{\Gamma,\Gamma'\in \mathit{Cxt}} \left(\mathit{Subst}_{\Gamma;\Gamma'} \to \prod_{A\in\Theta} (\mathit{Term}_{\Gamma\vdash A} \to \mathit{Term}_{\Gamma'\vdash A})\right)$$

For any substitution $k : \Gamma \to \Gamma'$, any term $\Gamma \vdash M : B$ and any function $* \in$ TermSubst, write k^*M for $*(\Gamma)(\Gamma')(k)(B)(M)$. Now define $* \in$ TermSubst to be the unique function satisfying the equations in Figure 8, for any term $\Gamma \vdash M : A$.

Definition A.21 (Substitution for a sequence of locations) *For any store* σ *and any sequence of locations* $\overrightarrow{v} : \overrightarrow{A} \in \sigma$*, define the substitution subst*$(\sigma, \overrightarrow{v}) : \varnothing, \overrightarrow{A} \to \varnothing$ *with*

$$\mathit{subst}(\sigma, \overrightarrow{v}) \overset{\mathit{def}}{=} \{i \mapsto \textbf{\textit{just}}(u_i) \mid i \in \mathrm{dom}(\overrightarrow{v})\}$$

where $\overrightarrow{u} = \mathit{reverse}(\overrightarrow{v})$*.*

Then when there is an implicit "current store" σ, write $\overrightarrow{v}^*M$ for the closed term $\mathrm{subst}(\sigma, \overrightarrow{v})^*M$, for any term $\varnothing, \overrightarrow{A} \vdash M : B$.

A.3 Interaction

Figure 9 defines a judgement $M \Downarrow^{\tau}_{\mathsf{v}} \tau_1, M'$, for any initial store τ, which, along with an analogous judgement for sequences, transforms a term M into a term M' where every maximal sub-term which consists purely of constructor calls has been replaced by a loc term denoting a location representing that sub-term.[3]

We say that a term M is *pre-allocated* iff $M \Downarrow_{\mathsf{v}}^{\tau} \varnothing, M$, for any store τ. We note that unless M is pre-allocated, τ_1 is an initial store for M'.

$x \in \Gamma$:

$$\frac{}{\mathsf{idf}_x \Downarrow_{\mathsf{v}}^{\tau} \varnothing, \mathsf{idf}_x}$$

${M'}_i = \mathsf{loc}_{u_i, B_i}$ for all $i \in \mathrm{dom}(\overrightarrow{M'})$ and $(v, \tau_2) =$ **new**$(A.c(\overrightarrow{u}))$:

$$\frac{\overrightarrow{M} \Downarrow_{\mathsf{v}}^{\tau} \tau_1, \overrightarrow{M'}}{\mathsf{ctr}_c(\overrightarrow{M}) \Downarrow_{\mathsf{v}}^{\tau} \tau_1 + \tau_2, \mathsf{loc}_{v,A}}$$

otherwise:

$$\frac{\overrightarrow{M} \Downarrow_{\mathsf{v}}^{\tau} \tau_1, \overrightarrow{M'}}{\mathsf{ctr}_c(\overrightarrow{M}) \Downarrow_{\mathsf{v}}^{\tau} \tau_1, \mathsf{ctr}_c(\overrightarrow{M'})}$$

$$\frac{M \Downarrow_{\mathsf{v}}^{\tau} \tau_1, M' \qquad \overrightarrow{\{N_c\}_{c \in C}} \Downarrow_{\mathsf{v}}^{\tau + \tau_1} \tau_2, \overrightarrow{\{{N'}_c\}_{c \in C}}}{\mathsf{case}(M, \{N_c\}_{c \in C}) \Downarrow_{\mathsf{v}}^{\tau} \tau_1 + \tau_2, \mathsf{case}(M', \{{N'}_c\}_{c \in C})}$$

$$\frac{\overrightarrow{M} \Downarrow_{\mathsf{v}}^{\tau} \tau_1, \overrightarrow{M'}}{\mathsf{app}_f(\overrightarrow{M}) \Downarrow_{\mathsf{v}}^{\tau} \tau_1, \mathsf{app}_f(\overrightarrow{M'})}$$

Figure 9. Pre-allocation judgement

Theorem A.22 (Idempotence of pre-allocation) *For any store τ, any τ-delta τ_1 and any terms M, M',*

$$M \Downarrow_{\mathsf{v}}^{\tau} \tau_1, M' \quad \Rightarrow \quad M' \textit{ pre-allocated}$$

Proof By induction over the derivation of $M \Downarrow_{\mathsf{v}}^{\tau} \tau_1, M'$. □

Theorem A.23 (Lifting inverts pre-allocation) *For any store τ, any τ-delta τ_1, and any terms M, M',*

$$M \Downarrow_{\mathsf{v}}^{\varnothing} \tau, M' \quad \Rightarrow \quad M' \uparrow \tau = M$$

Proof By induction over the derivation of $M \Downarrow_{\mathsf{v}}^{\tau} \tau_1, M'$. □

A.3.1 Top-level interaction loop

We now define the top-level "interaction loop", ignoring any possibility of concurrent interactions. Let $\vdash M : A$ be a closed term.

[3] So that the rule for case is deterministic, we fix a total order $\leq_I$ on constructor names, and define $\overrightarrow{\{N_c\}_{c \in C}}$ to be the elements of $\{N_c\}_{c \in C}$ sorted by $\leq_I$.

Definition A.24 (Configuration) *Define a* configuration *(τ, σ, e) for M to be a triple of an initial store τ, a global store $\sigma \geq \tau$, and an evaluation $e\langle M \Downarrow^{\tau} o, \tau_1 \rangle$.*

Definition A.25 (Initial configuration) *Define an* initial configuration *for M to be any configuration (τ, τ, e) for M satisfying $M \Downarrow_{\mathsf{v}}^{\varnothing} \tau$.*

Suppose (τ, σ, e) the *current configuration* for M. Then repeat forever:

1. Wait for an *interaction* with M in the form of a new initial store τ'.
2. Synchronise $e \overset{\tau'}{\curvearrowright} e'$.
3. Set the current configuration to (τ', σ', e').

By an *interaction with M*, we simply mean a new initial store τ' satisfying certain well-formedness constraints with respect to the global store σ.

Acknowledgments

TODO

References

Umut A. Acar. *Self-Adjusting Computation*. Phd thesis, Department of Computing Science, Carnegie Mellon University, 2005.

Margaret Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11 (02):155–206, 2001.

N.G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5): 381–392, 1972.

Walter A. C. A. J. de Hoon, Luc M. W. J. Rutten, and Marko C. J. D. van Eekelen. Implementing a functional spreadsheet in Clean. *Journal of Functional Programming*, 5:383–414, 1995.

Jonathan Edwards. 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*, pages 505–518, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094851.

Conal Elliott. Simply efficient functional reactivity. http://conal.net/papers/simply-reactive, 2008.

Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

Ian Holyer, Neil Davies, and Chris Dornan. The Brisk project: Concurrent and distributed functional systems. Technical Report CSTR-95-015, Department of Computer Science, University of Bristol, 1995. URL `http://www.cs.bris.ac.uk/Publications/Papers/1000069.pdf`.

Robin Milner. Turing, computing and communication. In Dina Goldin, Scott A. Smolka, and Peter Wegner, editors, *Interactive*

Computation: The New Paradigm. Springer, New York, NY, USA, 2006.

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

Roly Perera. Programming languages for interactive computing. In *Proceedings of the 2nd Workshop on the Foundations of Interactive Computation*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 33–52, New York, NY, USA, 2008. Elsevier.

Steven L. Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages and Computing*, 1(2), 1990.

Tarmo Uustalu and Varmo Vene. The essence of dataflow programming. In Z. Horváth, editor, *Central-European Functional Programming School*, volume 4164 of *Lecture Notes in Computer Science*, pages 135–167. Springer-Verlag Berlin Heidelberg, 2006.

Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, MA, USA, 2004. ISBN 0-262-22069-5. URL `http://www.info.ucl.ac.be/people/PVR/book.html`.

```
define {
    and(x,y).case x of {
        True.y
        False.False
    }
    equal(x,y).case x of {
        Var(x').case y of {
            Var(a).equal_nat(a,x')
            Abs(a,b).False
            App(a,b).False
        }
        Abs(x',y').case y of {
            Var(a).False
            Abs(a,b).and(equal_nat(a,x'),equal(b,y'))
            App(a,b).False
        }
        App(x',y').case y of {
            Var(a).False
            Abs(a,b).False
            App(a,b).and(equal(a,x'),equal(b,y'))
        }
    }
    equal_nat(x,y).case x of {
        Zero.case y of {
            Zero.True
            Succ(a).False
        }
        Succ(x').case y of {
            Zero.False
            Succ(a).equal_nat(x',a)
        }
    }
}
in
    equal(Abs(Succ(Zero),Var(Zero)),
          Abs(Succ(Zero),Var(Zero)))
```

Figure 2. Example IML program

$v : A \in \tau$ and $(o, \tau_1) = \mathbf{new}(\tau(v))$:

$$\frac{}{\mathsf{loc}_{v,A} \Downarrow^{\tau} o, \tau_1}\ \mathsf{e\_loc}$$

$A : \{\overrightarrow{B_c}\}_{c \in C}$ and $(o, \tau_2) = \mathbf{new}(A.c(\overrightarrow{o}'))$:

$$\frac{\overrightarrow{M} \Downarrow^{\tau} \overrightarrow{o}', \tau_1}{\mathsf{ctr}_c(\overrightarrow{M}) \Downarrow^{\tau} o, \tau_1 + \tau_2}\ \mathsf{e\_ctr}$$

$M : B$ and $B : \{\overrightarrow{A'_c}\}_{c \in C}$, with $o' : B.c(\overrightarrow{v}) \in \tau_1$ and $(o, \tau_3) = \mathbf{new}(\tau_2(o''))$:

$$\frac{M \Downarrow^{\tau} o', \tau_1 \quad \overrightarrow{v}^{*} N_c \Downarrow^{\tau + \tau_1} o'', \tau_2}{\mathsf{case}(M, \{N_c\}_{c \in C}) \Downarrow^{\tau} o, \tau_1 + \tau_2 + \tau_3}\ \mathsf{e\_case}$$

$f(\overrightarrow{B}) : A$:

$$\frac{\overrightarrow{M} \Downarrow^{\tau} \overrightarrow{o}', \tau_1 \quad \overrightarrow{o}'^{*} \mathrm{def}(f) \Downarrow^{\tau + \tau_1} o, \tau_2}{\mathsf{app}_f(\overrightarrow{M}) \Downarrow^{\tau} o, \tau_1 + \tau_2}\ \mathsf{e\_app}$$

Figure 4. Evaluation judgement for closed term in an initial store

$M = \mathsf{loc}_{v,A}$:

$$\frac{}{\mathsf{e\_loc}_{M \Downarrow\text{-} o, \_} \overset{\tau'}{\curvearrowright} \mathsf{e\_loc}_{M \Downarrow^{\tau'} o, \{o : \tau'(v)\}}}$$

$$\frac{\overrightarrow{e} \overset{\tau'}{\curvearrowright} \overrightarrow{e}' \langle \overrightarrow{M} \Downarrow^{\tau'} \overrightarrow{o}', {\tau_1}' \rangle}{\mathsf{e\_ctr}_{M' \Downarrow\text{-} o, \tau}(\overrightarrow{e}) \overset{\tau'}{\curvearrowright} \mathsf{e\_ctr}_{M' \Downarrow^{\tau'} o, {\tau_1}') + \{o : \tau(o)\}}(\overrightarrow{e}')}$$

$$\frac{\overrightarrow{e_1} \overset{\tau'}{\curvearrowright} \overrightarrow{e_1}' \langle \overrightarrow{M} \Downarrow^{\tau'} \overrightarrow{o}', {\tau_1}' \rangle \quad e_2 \overset{\tau' + \tau_1'}{\curvearrowright} {e_2}' \langle \overrightarrow{N} \Downarrow^{\tau' + {\tau_1}'} \overrightarrow{o}, {\tau_2}' \rangle}{\mathsf{e\_app}_{M \Downarrow\text{-} o, \_}(\overrightarrow{e_1}, e_2) \overset{\tau'}{\curvearrowright} \mathsf{e\_app}_{M \Downarrow^{\tau'} o, {\tau_1}' + {\tau_2}')}(\overrightarrow{e_1}', {e_2}')}$$

$\tau_1(o') = {\tau_1}'(o')$:

$$\frac{e_1 \langle M \Downarrow^{\tau} o', \tau_1 \rangle \overset{\tau'}{\curvearrowright} {e_1}' \langle M \Downarrow^{\tau'} o', {\tau_1}' \rangle \qquad e_2 \overset{\tau' + {\tau_1}'}{\curvearrowright} {e_2}' \langle N \Downarrow^{\tau' + {\tau_1}'} o'', {\tau_2}' \rangle}{\mathsf{e\_case}_{M' \Downarrow\text{-} o, \_}(e_1, e_2) \overset{\tau'}{\curvearrowright} \mathsf{e\_case}_{M' \Downarrow^{\tau'} o, {\tau_1}' + {\tau_2}' + \{o : {\tau_2}'(o'')\}}({e_1}', {e_2}')}$$

$\tau_1(o') \neq {\tau_1}'(o')$, $o' : B.c(\overrightarrow{v}) \in {\tau_1}'$, and $M = \mathsf{case}(M', \{N_c\}_{c \in C})$:

$$\frac{e_1 \langle M \Downarrow^{\tau} o', \tau_1 \rangle \overset{\tau'}{\curvearrowright} {e_1}' \langle M \Downarrow^{\tau'} o', {\tau_1}' \rangle \qquad e'_2 \langle \overrightarrow{v}^{*} N_c \Downarrow^{\tau' + {\tau_1}')} o'', {\tau_2}' \rangle}{\mathsf{e\_case}_{M \Downarrow\text{-} o, \_}(e_1, e_2) \overset{\tau'}{\curvearrowright} \mathsf{e\_case}_{M \Downarrow^{\tau'} o, {\tau_1}' + {\tau_2}' + \{o : {\tau_2}'(o'')\}}({e_1}', {e_2}')}$$

Figure 6. Synchronisation judgement

$$
\begin{array}{rcll}
k^*(\mathsf{loc}_{v,A}) & = & \Gamma' \vdash \mathsf{loc}_{v,A} & \\
k^*(\mathsf{idf}_x) & = & \Gamma' \vdash \mathsf{loc}_{v,A}, & \text{if } k(x) = \textbf{just}(v); \\
 & & \Gamma' \vdash \mathsf{idf}_x, & \text{otherwise} \\
k^*(\mathsf{ctr}_c(\overrightarrow{M'})) & = & \Gamma' \vdash \mathsf{ctr}_c(k^*\overrightarrow{M'}) & \\
k^*(\mathsf{app}_f(\overrightarrow{M'})) & = & \Gamma' \vdash \mathsf{app}_f(k^*\overrightarrow{M'}) & \\
k^*(\mathsf{case}(M', \{N_c\}_{c\in C})) & = & \Gamma' \vdash \mathsf{case}(k^*M', \{(k, \overrightarrow{B_c})^* N_c\}_{c\in C}), & \text{with } \Gamma \vdash M' : A', \\
 & & & \text{and } A' : \{\overrightarrow{B_c}\}_{c\in C}
\end{array}
$$

Figure 8. Substitution into terms