

INTERACTIVE FUNCTIONAL PROGRAMMING

Roland Perera

A thesis submitted to University of Birmingham
for the degree of doctor of philosophy



UNIVERSITY OF
BIRMINGHAM

School of Computer Science
Edgbaston
Birmingham
B15 2TT

UNIVERSITY OF
BIRMINGHAM

University of Birmingham Research Archive

e-theses repository

This unpublished thesis/dissertation is copyright of the author and/or third parties. The intellectual property rights of the author or third parties in respect of this work are as defined by The Copyright Designs and Patents Act 1988 or as modified by any successor legislation.

Any use made of information contained in this thesis/dissertation must be in accordance with that legislation and must be properly acknowledged. Further distribution or reproduction in any format is prohibited without the permission of the copyright holder.



Submitted	31 October, 2012
Examined	18 January, 2013
Corrected	24 April, 2013
Finalised	21 May, 2013

For Luca

(in case you ever wonder what I was doing)

Abstract

We outline a vision for a new kind of execution environment where applications can be debugged and re-programmed while they are being used. The overall concept we call *interactive programming*. In contrast to most other systems for live programming, interactive programming presents execution to the user as a live, explorable document. In contrast to the edit-and-continue features found in many debuggers, and to systems for patching software dynamically, we utilise a notion of *retroactive update*, where the computation transitions to a new consistent state when the program changes, rather than a hybrid of old and new. What changed in the execution is always explicit and visible to the user. Retroactive update relates our work to incremental computation.

We develop some key components of interactive programming in the setting of a pure, call-by-value functional language. We illustrate our ideas via a proof-of-concept implementation called LambdaCalc. Several important components of the overall vision, including efficient incremental update, scaling to realistic programs, supporting effectful programs, and dealing with non-termination, are left for future work. We implemented a comprehensive visualisation subsystem in LambdaCalc itself, but further performance work is required for this to be the basis of a working user interface.

Our specific achievements are as follows. First, we show how to *reify* the execution of a program into a live document which can be interactively decomposed into both sequential steps and parallel slices. We give a novel characterisation of forward and backward dynamic slicing and show that for a fixed computation, the two problems are described by a Galois connection. We extend the notion of slicing to reified computations, and formalise what it is for a slice of a computation to “explain” some part of a value. We show how being able to slice a computation interactively can help debugging.

Second, we introduce a novel execution indexing scheme which derives execution differences from program differences. Our scheme supports the wholesale reorganisation of a computation via operations such as moves and splices. The programmer is able to see the consequences of edits on the intensional structure of the execution. Where possible, node identity is preserved, allowing an edit to be made whilst an execution is being explored and the changes to be reflected in the user’s current view of the execution. This allows the user to see the *impact of code changes* while debugging. We illustrate this using figures generated by our implementation. Our self-hosted visualisation code is able to compute differences in visualisations, which we use to visualise differences in computations.

We conclude with a discussion of some of the challenges facing the proposed paradigm: space requirements, visualising large computations and data structures, computational effects, and integrating with environments that lack support for retroactive update.

Acknowledgements

Many years of fascinating conversation with my friend Russ Freeman started me off in this direction in the first place. James Cheney and Deepak Garg have been great mentors and collaborators, and I thank them both for giving me nice things (jobs). James came up with the phrase “self-explaining computation”. My examiners Dan Ghica and Henrik Nilsson provided high-quality feedback that made the slog of actually writing a thesis worthwhile.

Three people require a special mention. First, I’m indebted to Umut Acar for hosting me as an intern and later as a research visitor at MPI-SWS, Kaiserslautern. Without those eighteen months of focus I don’t think I would have made it. Second, I thank my *Doktorvater*, Paul Levy, whose focus and clarity are truly impressive and who has always been willing to have lengthy Skype chats at unreasonable times. (My only regret is that I still don’t understand call-by-push-value.) I also owe Paul for hooking me up with Neel who hooked me up with Umut. But the biggest debt I owe to my dad Siri, for lots of love and support, financial and otherwise.

Saarbrücken, Germany

April 2013

Contents

1	Overview	1
1.1	Plan of thesis	1
1.2	Scope	3
1.3	Contributions	4
2	Programming as Interaction	7
2.1	Self-explaining computation	8
2.2	Differential execution	12
2.3	Visualising structured values	20
2.4	Editing structured values	23
2.5	Interactive slicing	29
2.6	Interweaving testing, diagnosis and bug-fixing	34
2.7	Summary	37
3	Related Work	39
3.1	Recording and replaying execution	40
3.2	Algorithmic debugging	42
3.3	Debugging lazy programs	44
3.4	Program slicing	45
3.5	Provenance	46
3.6	Dynamic program visualisation & visual programming	47
3.7	Live programming & live coding	52
3.8	Spreadsheet languages	54
3.9	Functional reactive programming	55
3.10	Self-adjusting computation	56
3.11	Dynamic software updating	59
3.12	Execution indexing	61
3.13	End-user programming	62
4	Reifying Computation	65
4.1	Reference language	66
4.2	Syntax all the way down	66
4.3	Tracing semantics	69
5	Slicing Computation	73
5.1	Partial programs and partial values	74

5.2	Characterising dynamic slicing	76
5.2.1	Forward dynamic slicing	76
5.2.2	Backward dynamic slicing	78
5.2.3	Differential slices	80
5.3	Program slicing as backwards execution	81
5.3.1	Unevaluation	82
5.3.2	Correctness of tracing evaluation	84
5.3.3	Unevaluation computes least program slices	85
5.4	Trace slicing	87
5.4.1	Computation of least explanations	90
6	Differencing Computation	91
6.1	Indexed syntax	93
6.2	Indexing evaluation	96
6.3	Delta computations	101
7	Conclusion	105
7.1	Appraisal	105
7.2	Future work	106
7.2.1	Scaling to realistic programs	106
7.2.2	Efficient incremental update	107
7.2.3	Distributed systems	108
7.2.4	State and I/O	109
7.2.5	Demand-indexed computation	110
7.2.6	Primitive operations	111
	Appendices	123
A	Additional Proofs	123
A.1	Proof of Theorem 3	123
A.2	Proof of Theorem 4	124
A.3	Proof of Lemma 6	125
A.4	Proof of Theorem 5	125
A.5	Proof of Theorem 8	127
A.6	Proof of Theorem 9	129
B	Delta Visualisation in LambdaCalc	131
B.1	Visualising differences by differencing visualisations	131
B.2	Coding for stability	132

1 Overview

To understand a program you must become both the machine and the program.

Alan Perlis, *Epigrams on Programming* [Per82]

Programming is an ongoing dialogue between programmer and programming environment. Development activities, such as tweaking code, writing new test cases, stepping through a computation in a debugger, and so on, are *questions* that we ask of our programming environment. The programming environment *answers* by providing feedback or transitioning into a new state. The questions and answers are woven into a complex web of interaction, as we switch between testing and understanding, changing and fixing.

In this thesis, we propose a model of programming that supports these interwoven, interactive Q&A sessions directly. The key idea is to treat the execution of a program as a persistent, explorable, spreadsheet-like document. Because the user can interact with and modify a running program, our proposal is a form of *live programming* (Related Work, §3.7). We call our approach *interactive programming*. Compared with other approaches to live programming, the unique feature of interactive programming is that an edit results in a *delta* to the program execution and its visualisation, allowing our working context to *update* into a new consistent state, rather than forcing us to reconstruct it from scratch each time, or risk continuing in an inconsistent state.

Interactive programming, as sketched here, is an ambitious goal. As we clarify in §1.2 below, our achievements in this thesis are modest compared to this ambitious vision. In §1.3 we list our specific achievements.

Supplementary material. The following additional material is available online:

http://dynamicspects.org/papers/thesis/LambdaCalc.tar.gz	Haskell source code, plus examples
https://vimeo.com/45867320	Slide presentation (part 1)
https://vimeo.com/43760460	Slide presentation (part 2)

1.1 Plan of thesis

The thesis is organised into seven chapters and two appendices.

Chapter 1. Overview. We introduce the key idea of interactive programming, outline the plan of the thesis, discuss the scope and limitations of the work, and then summarise our contributions, including the relationship to previous publications of the author.

Chapter 2. Programming as Interaction. We outline our vision of interactive programming, which motivates the technical development in the rest of thesis. Although our prototype LambdaCalc only implements some components of the vision, we are able to use it to illustrate the central ideas. To set the scene, we describe programming is an ongoing Q&A session between programmer and programming environment. We divide the questions into two kinds. “How” questions concern a single computation, in particular the relationship between program and output. Normally “how” questions are difficult to answer because programs are black boxes. In LambdaCalc such questions are enabled via two forms of interactive decomposition of a running program: into sequential steps and into parallel slices.

“What if” questions concern the relationship between one computation and another. They subsume the familiar notion of an edit, which we construe as a question of the form, “How would my program’s behaviour change if it were modified in the following way?” We allow “what if” questions to be asked while the programmer is pursuing the answer to a “how” question. Using figures obtained from LambdaCalc, we argue that the programmer should not have to restart a computation in order to understand it or change it but should be able to move seamlessly between using, understanding and editing to suit the task at hand.

Chapter 3. Related Work. We relate our work to previous work on tracing, debuggers, program visualisation, live programming, spreadsheets, functional reactive programming, provenance, program slicing, incremental computation, execution indexing and end-user programming.

Chapter 4. Reifying Computation. We allow the user to look inside a computation. We enable this by *reifying* execution: turning the process of execution into a description of the process. Our techniques are standard here, but this is needed for what follows. We first introduce the baseline language used for the rest of the thesis, and then show how reified computations take the form of an “unrolling” of the program that grows as the program runs. We call these reified computations *traces*. Traces are built by a tracing interpreter that transcribes the big-step evaluation of a term into a data structure.

Chapter 5. Slicing Computation. We allow the user to query the relationship between *parts* of the output and *parts* of the program, for given computation. In the literature this is called *dynamic slicing*. Slicing queries can be asked in two directions, depending on whether they seek to relate partial programs to partial outputs, or partial outputs to partial programs. We formalise the notion of a *slice* (prefix) of a program and extend the notion of execution to program slices. This analysis gives rise to a *Galois connection*, implying that for any slice of the output of the program, there is a least slice of the program able to compute it. *Differential* program slices, differences between one slice of a program and another, enable more fine-grained Q&A about the relationship between input and output.

We then show how to efficiently calculate least program slices for a given output slice by running the

program backwards for that portion of the output. The traces we built in Chapter 4 guide the backward execution. The notions of slicing and differential slicing are extended to traces as well as programs, revealing information that a program slice cannot. We give a trace-slicing algorithm and show that it calculates the least trace slice that “explains”, in a technical sense, a given portion of the output.

Chapter 6. Differencing Computation. We allow the user to edit a program and observe the resulting delta in its execution. This is possible via *indexed traces* where nodes are identified by injectively assigned indices. Indexed traces are built by a deterministic indexing interpreter which derives an index for each node in the computation from indices provided on the input program. We show how two indexed traces can be subtracted to obtain a trace delta. Trace deltas are more general than the differential slices introduced in Chapter 5, which can describe only increasing or decreasing changes. Differences between indexes traces can describe complex reorganisations of a computation such as moves and splices.

Chapter 7. Conclusion. We assess the strengths and weaknesses of our work, and discuss future directions: scaling to realistic programs, efficient differential execution, distributed systems, computational effects, and demand-indexed computation. We also sketch a more realistic treatment of primitive operations.

Appendix A. Additional Proofs. We give longer proofs omitted from the main body of the thesis.

Appendix B. Delta Visualisation in LambdaCalc. By adding a reflection facility to LambdaCalc, we were able to use differential execution to implement much of the GUI code which visualises the deltas that differential execution itself produces. We use this as a case study to discuss the viability of interactive programming as a form of incremental computation, with several examples from our code base.

1.2 Scope

To make our problem more tractable, we adopt several simplifying (and unrealistic) assumptions, which we now set out. We also describe a significant limitation of our implementation, and give a sense of how the gap between our current implementation and the long-term vision might be narrowed. We say more about this in Future Work, §7.2.

Linguistic simplifications. We restrict ourselves to a pure, call-by-value functional language with recursion and data types. Despite this simplified setting, we are able to put our language to practical use, using it to generate all the visualisations of executions that appear in the thesis. In Future Work, §7.2, we discuss more realistic features, in particular concurrency, distribution, and side-effects. We also ignore non-terminating programs; our implementation hangs in the presence of divergence and our main theorems only consider programs which terminate. Ideally the user would be able to interrupt a divergent or long-running computation, interact with it, and resume it. While this is a plausible extension of the ideas presented here, it is beyond the scope of the thesis.

Toy examples. To avoid having to address non-trivial visualisation and performance issues, we restrict our examples to toy programs. These suffice to illustrate the philosophy of interactive programming in Chapter 2, but fall well short of demonstrating that the approach can scale even to medium-sized programs. The *space overhead* associated with treating computations as explorable data structures is one concern; thankfully existing work on functional debuggers has explored various techniques that could be applicable here. We discuss these in Future Work, §7.2.1. The *visualisation* challenges for more realistic programs are another concern; “zoomable” user interfaces and custom visualisations for particular data types may help here. We discuss these in §7.2.1 too and also explain why it makes sense to build such visualisations in LambdaCalc itself.

Implementation limitations. We validated several of our key ideas via a proof-of-concept implementation. We call this system LambdaCalc, after the early spreadsheet systems VisiCalc and SuperCalc, since it has much of the flavour of a “spreadsheet for functional programming”. The visualisation subsystem of LambdaCalc is mostly self-hosted. This allows us to *visualise differences by differencing visualisations*, something explained in more detail in Appendix B. This was a powerful validation of our system’s ability to generate useful execution deltas, but unfortunately our self-hosted visualisation layer is too slow to form the basis of a point-and-click GUI, even for the toy programs considered here. (The irony of presenting “interactive programming: the non-interactive version” has not escaped us.)

Instead, we provide a set of browsing and editing combinators which allow an interaction with a program and its execution to be scripted in Haskell. A script can ask LambdaCalc to render the UI state associated with any configuration in a window, or as a PDF or Postscript file. Having experimented with a much faster implementation technique which still supports self-hosting, we believe that LambdaCalc’s performance shortcomings are related to naïve implementation decisions, rather than inherent to our goals. Nevertheless, a live demo is unfortunately beyond the scope of this thesis. In Future Work, §7.2.2, we discuss efficiency issues that *are* related to our longer-term goals, in particular *incremental update*, which means updating a computation in time proportional to the size of the execution delta.

1.3 Contributions

The specific contributions of the thesis are as follows. We summarise the relationship to previously published work at the end of this section.

Technical. In Chapters 5 and 6, we make several technical contributions in the area of dynamic program slicing (Related Work, §3.4) and execution indexing (Related Work, §3.12):

- *Slicing problem definition.* We give a novel characterisation of the problem of backward dynamic slicing with respect to criteria on the output specified in terms of partial values. We show that extending evaluation to partial programs (which characterises *forward slicing*) gives rise to a family of Galois connections, thereby uniquely determining the backward-slicing problem.

- *Program slicing algorithm.* We give an algorithm for efficiently computing backward dynamic slices of programs by “unevaluating” a trace back into a program slice. Our algorithm improves on earlier work on slicing for functional programs by computing slices which are *minimal* with respect to fine-grained criteria on the output.
- *Trace slicing.* We extend slices to traces (explanations), formalise the notion of a slice of a trace being sufficient to explain the selected part of the output, and show that our algorithm computes the least such slice for the selected output.
- *Execution indexing scheme.* We introduce a novel indexing scheme which deterministically assigns indices to trace nodes based on indices provided on program nodes. Our scheme can be used to derive execution differences that preserve program differences. In particular, our scheme supports wholesale structural reorganisation of a computation.

Implementation. We implemented our system predominantly in Haskell but much of our visualisation code in LambdaCalc itself. The key contribution here is a demonstration of the potential of interactive programming as a form of differential computation:

- *Visualising differences by differencing visualisations.* Our self-hosted visualisation code demonstrates the practical utility of our execution indexing scheme. We were able to visualise execution differences by reflecting LambdaCalc traces back into LambdaCalc and then computing visualisation differences by comparing the output of a visualisation function applied to different input traces.

Conceptual. Many of the ideas that distinguish our view of programming from traditional batch-mode IDEs and compilers have been explored in other settings dating back as far as the 1960s. The ideas and concepts that appear to be unique to our work are:

- *Change is always explicit.* In our approach, change detection is automatic and pervasive. Just as there is no hidden computation, there are no “hidden changes”. We argue that tracking structural deltas is not just about making a programming medium *responsive*, but also about letting the user see what is happening.
- *Interweaving of computations and values.* We introduce a UI design principle that allows the user to move smoothly from an extensional view, where the computation maps a monolithic value to a monolithic value, to increasingly intensional views which expose more of the value assembly and disassembly.

Relationship to previously published work. The technical parts of Chapters 4 and 5 were presented at ICFP 2012 [PACL12]; the majority of the work was mine. Some of my earlier work [Per04, Per08] informed the discussion in Chapters 2 and 3. A workshop paper [Per10] presents a substantially less mature version of the ideas in Chapter 6. The rest of the thesis was not published previously; in particular the other jointly-authored work mentioned in Chapter 3 [AAP12] does not form part of this thesis.

2 Programming as Interaction

At the beginning of Chapter 1 we described programming as an *interactive dialogue* between programmer and tool. Here we divide the questions “asked” by the programmer into two main flavours. *How* (or *provenance*) questions, which we consider in Chapters 4 and 5, concern a particular program and what it computes. These are the questions that commonly arise during debugging. How did this result get to be zero? Why was that true rather than false? *What if* questions, on the other hand, which are the focus of Chapter 6, concern variants of a program and how they relate to each other. “What if” questions are framed by making changes to either code or data, and typically arise during coding, testing and bug-fixing. What value would the program produce for a different input? What value would a different program produce for the same input? The emphasis of the ongoing dialogue often shifts between constructive, diagnostic and remedial.

This interactive construal of programming may have some intuitive appeal, but is poorly supported by traditional programming environments. For example, we often want to see the impact of a fix in the middle of a complex debugging activity. In particular, we want to see the *impact of the fix* on our *current view of the execution* of the program. That carefully constructed view isolates a problem, and we want to know if the fix makes the problem go away. To use the terminology just introduced, we would like the answer to the “what if” question to take the form of a *change* to the answer to the “how” question which forms our current debugging context. But to ask a “what if” question usually jeopardises any active “how” question in one of two ways. Either we must exit the program, apply the change, and restart, discarding the original context entirely. Or, if our development environment supports some kind of edit-and-continue feature, we can avoid restarting, but with only subsequent execution incorporating the fix. This second scenario is more convenient but now the integrity of our all-important debugging context is compromised.

The importance of the debugging context should be apparent if we consider that we rarely ask a single provenance question in isolation. Instead, the system’s answer to our first question is only a partial answer, which invites another question, and so on. This is roughly 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 outcome of our interactive Q&A session is a complex tree of provenance-related questions and answers that “explains” the result of interest. Once we have obtained this intensional view of a computation tailored to a specific comprehension task, what we are typically most interested in is *what would happen to that chain of explanations* if something were different. Would this function still behave correctly *for the right reasons* if I were to remove an element from the list? Would this change to a base case of a recursive function fix it *in the way I expect*? Our carefully constructed view is not an ephemeral concern, but remains important for as long as we are interested not only in what our program does, but in how it does it.

Interactive programming addresses this need by allowing the programmer to make changes to a program

whilst in the middle of a complex debugging or comprehension activity and have the consequences of those edits reflected immediately in the view of the execution, without having to lose more of their working context than necessary. There is no separate debugging mode: the user programs in a debugger, or alternatively, debugs from within an editor. But unlike edit-and-continue debuggers, the view of the computation is updated, after each change, so that things are as though the program had been written differently in the first place. We build this approach on a foundation which treats execution as something concrete and persistent, instead of hidden and ephemeral. Computations are “self-explaining”: not black-box processes that compute values, but interactive, spreadsheet-like documents describing how values are computed.

In our long-term picture of interactive programming, “online debuggability” becomes an intrinsic feature of the execution environments in which applications live, allowing usage to blend seamlessly into understanding or fixing and then back to using. Although we are still some way from a practical realisation of the approach, we have implemented several important components of the approach in a system called LambdaCalc, a spreadsheet-like execution environment for functional programs. Unlike normal spreadsheet languages, LambdaCalc supports familiar functional programming features like recursion, higher-order functions and data types. Cells contain nested spreadsheets, so that all intermediate computations can be explored. A LambdaCalc computation is “spreadsheets all the way down”. Jonathan Edwards explored a similar nested-spreadsheet concept in the programming language Subtext [Edw05], but without considering many of the features we present here. In its current form, LambdaCalc lacks an interactive UI, and can only be controlled programmatically via an editing and browsing API. For the purposes of explaining the overall vision of interactive programming, it is convenient to gloss this shortcoming for the remainder of this chapter, and talk as though the UI were able to accept and respond to user input efficiently.

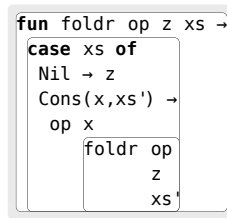
2.1 Self-explaining computation

To debug or understand a program, we often need to know how a certain part of the output was computed. Yet by the time the need arises, the information in question has usually been discarded. Our only hope is to reconstruct it, perhaps by manually instrumenting the code with print statements and restarting, re-launching in a debugger, or running the program in “tracing” mode and browsing the resulting execution log in an offline tool.

This is unfortunate considering that what we want to access has just taken place in the interpreter. Moreover, in a modern distributed environment, it is not always feasible to rely on re-execution as a way of recovering computational history. The sub-computation of interest may be external to our system, such as a calculation associated with an online transaction. We may lack the authority or the means to instrument it or to re-execute it in a debugger. If the computation was effectful, is not clear that “re-running” it even makes sense. For these scenarios, a more realistic goal is an execution environment which makes it possible to discover what happened after the fact.

In this thesis, we propose a kind of runtime environment in which computational history is always available. We explore this paradigm, which we call *interactive programming*, for pure, sequential languages, deferring distribution and effectful computations to future work (§7.2.3 and §7.2.4). In our approach, a pro-

gram run is not a black-box activity that yields a value, but rather an *explorable document* describing how a value was calculated. The user is able to peek inside a computation, and understand the steps taken to obtain a result, without having to restart the program in a separate debugger. We illustrate this by showing how a user would run and interact with a program in an interactive UI based on our LambdaCalc prototype, with figures obtained directly from our implementation.



```

fun foldr op z xs →
  case xs of
    Nil → z
  | Cons(x, xs') →
    op x
      foldr op
        z
        xs'

```

Figure 2.1 Definition of `foldr`

Figure 2.1 defines the familiar functional programming operation `foldr` which right-folds a binary operation over a list. The thin borders make it easier to visually parse the expression into sub-expressions. Figure 2.2 then shows an application of `foldr` to three arguments. The three arguments are the function `sumSquares`, the seed value `0`, and a list of three integers. When the user first loads the program into LambdaCalc, it executes and produces the visualisation shown in (a). The input expression has been visually paired with its result in a spreadsheet-like cell with two components. The grey panel in the top right-hand corner is a *value pane*, and displays the integer `416`, which is the value of the *computation* in the white panel that encloses it. The small rectangular tab on the left-hand side of the value pane tells the user that this value is *explained by* the attached computation. The reader will notice that the bindings for the arguments to `foldr` are also shown in grey, because they too are values. But these values have no rectangular tab indicating an associated explanation: this is because they were not computed, but were instead provided as they are.

Thus far, the interaction is like a graphical version of a *read-eval-print* loop (REPL), the interactive top-level prompt found in most functional language implementations, where the user can type in expressions and have them evaluated. But in LambdaCalc the user can do much more than just ask for an expression to be evaluated. Note the ellipsis in the bottom left-hand corner of (a). This conveys to the user that the explanation of how `416` was computed is *incomplete*, and that they can see more of that explanation by clicking on the ellipsis. Doing so causes two things to happen, as shown in Figure 2.2(b). First, the ellipsis has disappeared and been replaced by a small double-headed arrow \rightarrow , followed by a view of some more of the steps taken by the interpreter to calculate the result. The arrow indicates that there was some control flow, in this case entering the body of the function `foldr`; to the right of the arrow we see the execution of the body of `foldr`. The second thing is that the *bindings* of the arguments to the formal parameters of `foldr` are made explicit via a visual convention reminiscent of call-by-keyword (named argument) syntax [FB09]. The first argument has been prefixed by `op:`, indicating that the parameter `op` of `foldr` has been bound to `sumSquares`. Bindings for the other two parameters `z` and `xs` are also indicated.

Technically, expanding the ellipsis reveals some steps of the operational semantics. The effect is similar

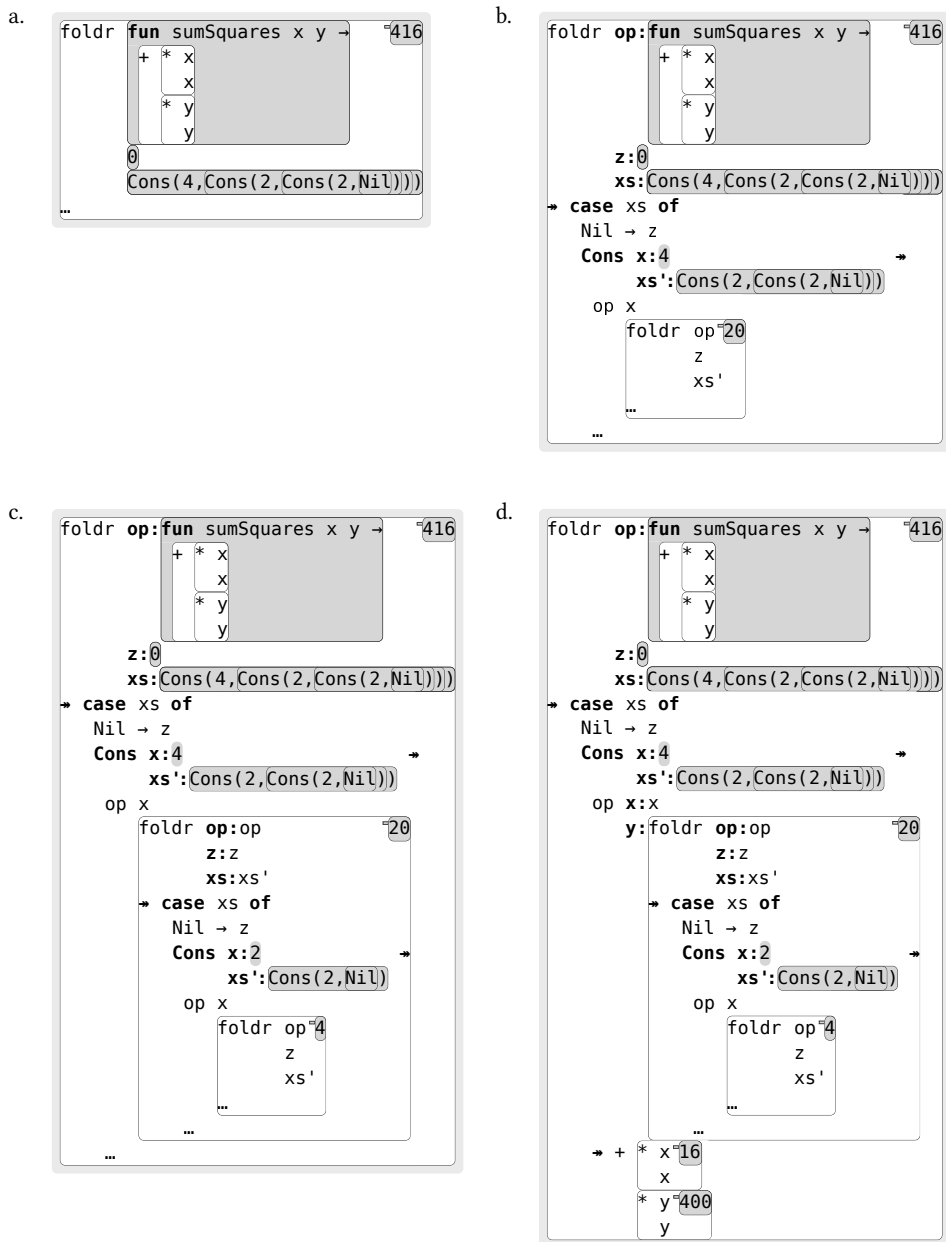


Figure 2.2 Interactive drill-down into computation

to stepping into a function call in a debugger, but rather than having to restart in a different “mode”, the user instead inspects the computation directly from the editor, by simply unfolding the source code *in situ*. Modulo a few minor syntactic extensions to aid comprehension, such as the argument-binding convention just described, the “language of computation” is the language of programs. The programmer need only learn one notation and can understand the execution of a program in terms that are already familiar.

The first thing we see inside the executed body of `foldr` is a case analysis of the list argument `xs`. The pattern `Cons(x, xs')` was matched, indicated by the bold highlighting of the constructor name and pattern variables, and the fact that `x` and `xs'` are bound to the components of `xs`, which is shown using the same `' : '` notation used for function arguments.¹ (Binding occurrences of variables are always in this format.) Inside the `Cons` branch is an application of `op` to two arguments, namely `x` and a recursive call to `foldr`. The recursive call appears as a *nested cell*, with its own value pane. The ellipsis underneath the recursive call indicates that there is another executed function body here to explore, which is currently collapsed. The body of the call to `op` is also collapsed.

A simple rule determines whether a sub-computation is visualised in its own cell, or rendered as part of the parent cell. The rule is really just the visual analogue of the so-called “tail-call” compiler optimisation [Ste77]. When the parent computation returns the value computed by the child directly, the value pane of the child is “reused” by the parent and the child cell coalesced into the parent. This happens with function applications and `let` expressions, which return the result of evaluating the body, and also with case expressions, which return the result of evaluating the selected branch. On the other hand, when the child computation yields a value which is not just passed upward, its value pane cannot be reused by the parent computation, and the child is then visualised in its own cell so that the intermediate result is still available for inspection. Thus the (white-backgrounded) “computation” component of any cell can always be read as a linear chain of steps that locally terminates in the result shown in the value pane.

The tail-call convention explains why the recursive call in Figure 2.2(b) appears as a nested call: the integer 20 returned by the recursive call is “intercepted” by the application of `op`, rather than just passed upward. Visual conventions such as these, while an important part of any practical implementation, are not essential to the idea of interactive programming. Later we will see other conventions, such as hiding the explanation associated with a computed value completely, or hiding the dead branches of a conditional. One advantage of working with reified computation is that decisions regarding presentation can be taken after the program has run and applied to the computation afterwards, as discussed in Related Work, §3.6.

In Figure 2.2(c) the user continues to explore the behaviour of `foldr` by expanding the recursive invocation. A second recursive call is nested inside, producing the intermediate result 4 which is again passed to `op`, resulting in 20. In (d) the user takes a different path, choosing to expand the application of `op` instead, and thereby browse into the body of `sumSquares`. Because the user chooses interactively whether to expand a particular sub-computation into an intensional view, or to keep it collapsed and see only the computed result, the specific view of the execution they have at any point in time reflects the information they currently need to understand it. These needs vary depending on what they are trying to do and how well they already understand different parts of the program. The partial views of the running program thereby obtained are *partial explanations* of how some aspect of the program works. It is these potentially complex views which comprise the user’s “comprehension context” and whose structure we are concerned with preserving, where possible, as the program changes.

To summarise the story so far, a LambdaCalc execution is a structured document which the user is able

¹ The attentive reader will notice that, unlike the `0` bound to `z`, the `4` bound to `x` has no border. The convention is that a primitive value which is merely a sub-component of another value has no border. Here, `4` is part of the list bound to `xs`, whereas `0` is the value of a computation. The particular details of the UI are not essential to our message, however.

to explore interactively in order to understand. We treat execution not as a process that unfolds in time and computes a value, but as a structure which unfolds in space and describes how a value can be computed. We call this *reification*: the explicit representation of something formerly implicit or tacit.² The precise form that the reification takes is informed by the observation that big-step evaluation proceeds by a top-down decomposition of the program. This allows us to treat execution as an “unrolling” or *in situ* inflation of the program, with variables taking on actual values, sub-expressions being associated with the values they compute, and nested structure arising from the execution of function bodies. Executed function bodies are the only source of additional structure beyond what was present in the original program.

Sometimes computations take a very long time to produce a result, possibly forever. In a more realistic implementation, a user would be able to interrupt a long-running computation and observe a *partial* trace reifying the computation which has taken place so far. For the purposes of this thesis, we consider only terminating computations which cannot be interrupted.

2.2 Differential execution

We often run the same program or program fragment with modified inputs, when testing, debugging, or simply exploring our intuitions about how a program works. In a traditional compiled or interpreted language this involves running the program again from scratch with the new input. Equally, we often run a *modified* program on the *same* input, when testing new code or trying to fix a bug. So-called “live coding” or live programming systems (Related Work, §3.7) allow code to be edited on the fly while the program is running, but the execution itself usually remains a black box, and the semantics of update are typically ill-defined.

In LambdaCalc, when the user makes a change to the program their view of its execution is updated automatically. The result is a new computation with some highlighting indicating the parts that changed. The system uses a memoisation-like scheme to determine when a particular part of the new computation notionally has the same “identity” as, although possibly distinct contents from, some part of the old computation. The difference between (syntactic) equality and “identity” in this sense is similar to the `equal` vs. `eq` distinction in Lisp: “identity” corresponds to `eq`-style pointer equality.³ When a node of the computation persists into the new state, the information about how the user was browsing it can also be preserved, even though the contents of that node may have changed. This allows the user to see the impact of the change expressed *in terms of* the aspect of the program they were viewing previously.

A simple edit is shown in Figure 2.3, which continues with the `foldr` example. In (a), the user moves the edit focus to the first element of the list argument and changes 4 to 6. Changed values are highlighted in blue; this brings the user’s attention to the fact that the value of the overall computation has updated to 436, but also that the intermediate value representing the square of `x` updated, to 36. This seems like a reasonable outcome, so in (b), the user now changes the second element of the input list to see what happens in that case. This time the square of `y` changes, but the user can see by the binding information indicated by `y`: that `y` is actually computed by a recursive fold over the tail of the list. Changes to different parts of the input

² The Oxford English Dictionary defines *reify* to mean “to make (something abstract) more concrete or real; to regard or treat as if having material existence” [OED09].

³ Philosophers often use the term “numerically identical” [Noo11], but in computing this would be easy to misinterpret.

The diagram illustrates the evaluation of the `sumSquares` function on the list `[6, 3, 2]`. It is divided into two main sections: the initial state (left) and the recursive step (right).

Initial State (Left):

- `foldr op: fun sumSquares x y →` (Line 661)
- `z: 0`
- `xs: Cons(6, Cons(3, Cons(2, Nil)))`
- `case xs of`
 - `Nil → z`
 - `Cons x: 6`
 - `xs': Cons(3, Cons(2, Nil))`
 - `op x: x`
 - `y: foldr op: op` (Line 25)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 3`
 - `xs': Cons(2, Nil)`
 - `op x: x`
 - `y: foldr op: op` (Line 4)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 2`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 9)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 1`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 16)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 25)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 36)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 42)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 48)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 54)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 60)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 66)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 72)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 78)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 84)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 90)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 96)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 102)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 108)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 114)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 120)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 126)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 132)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 138)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 144)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 150)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 156)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 162)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 168)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 174)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 180)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - `xs': Nil`
 - `op x: x`
 - `y: foldr op: op` (Line 186)
 - `z: z`
 - `xs: xs'`
 - `case xs of`
 - `Nil → z`
 - `Cons x: 0`
 - <

Figure 2.4 Structural editing with delta-highlighting

14

information and recovers the original tree.

The significance of the sharing of common substructure will become apparent in §2.3. Here what matters is that we can also use term graphs to support retroactive update. If we allow the term-graph representation of two programs to have nodes in common, we can *identify* parts of the two programs whenever they are represented by the same node (or if you prefer, stored at the same location). It then becomes meaningful to talk of the “same” program part existing in two programs with very different shapes. To the programmer, the program is mutable: they obtain an updated program by modifying an existing one. In particular they are able to express structural reorganisations of the program that arise commonly during programming, such as moves and splices, which are not so straightforward to capture as deltas of pure tree-structured syntax.

Retroactive update is then enabled by taking a similar term-graph approach to the representation of reified computations. But whereas the user obtains a modified program directly, by editing an existing one, they obtain a modified computation only indirectly, by executing a modified program. The new computation will usually overlap with the previous computation at certain locations, allowing it to be expressed as a *delta* to the previous one. In the LambdaCalc GUI, we present the updated computation graph in an unravelled, tree-like view, and use colouring to highlight the (asymmetric) difference between the new computation and the previous one. Green indicates that a node is *new*, i.e. did not exist in the previous computation. Otherwise, the node existed in the previous computation, although possibly with different contents, which are then shown in blue.

An example of how a structural modification of the program can translate into a structural modification of its execution is given in Figure 2.4(b). Here, the user has edited the definition of `sumSquares` so that it calculates the average of two squares. The edit happened as follows. First, they turned the `+` into `/`, which is now highlighted in blue. Second, they spliced in a new application of `+` (shown in green) around the expressions `* x x` and `* y y`, leaving `/` missing a second argument. For the missing argument they supplied the new value 2, which is also shown in green. The effect of the program change on the computation to apply a similar transformation to each application of `sumSquares`. The additional intermediate values produced by the new `+` node are visible as new value panes shown in green, and the implied changes in the values computed by the various calls to `sumSquares` and recursive calls to `foldr` are shown in blue. A more mature GUI than the one we have implemented in LambdaCalc would smoothly animate the transition between (a) and (b) to provide a more robust visual realisation of node identity.

Allowing a live computation to be re-programmed on the fly poses non-trivial user-interface and usability challenges. For example, the editing scenario just described involved an intermediate state in which the operation `/` was missing one of its arguments. In such situations, there are choices to make about whether to update the computation and how to handle the potentially ill-formed state. Our prototype implementation only permits programmatic editing, so these concerns are mostly out of scope. For a real-world implementation, we envisage some kind of *structure-aware editor* [DGH⁺80], cognisant of the document’s underlying syntax. The editor would be responsible for error recovery, timing and frequency of updates, and the interplay between textual and structural editing. Although structure editors remain unpopular, a recent study of Java programmers showed that, in practice, code changes rarely utilise the full flexibility of free text editing [KAM05]. That structural changes are the norm gives us some confidence that structure-aware editors do

indeed have a plausible role to play in making our approach practical.

The delta shown in Figure 2.4(b) will make more sense if we say something about how nodes in the computation graph are built from nodes in the program graph. The key principle is that the evaluation function, which builds the reified computation from the program, is *homomorphic* with respect to the labelled DAG structure. In other words, if two program nodes e_1 and e_2 are related in the program graph by an edge labelled with n , indicating that e_2 is the n th child of e_1 in the program syntax, then in any context where e_1 and e_2 are evaluated to produce reified computation nodes T_1 and T_2 , the nodes T_1 and T_2 will be related by an edge labelled with n in the computation graph, so that T_2 is the n th child of T_1 . If evaluation relates program graphs and computation graphs in this way then structural reorganisations of the program will be preserved into the structure of its execution. This enables the computation delta to reflect the edit that the user made, at any rate in those parts of the new computation that reuse nodes from the old computation.

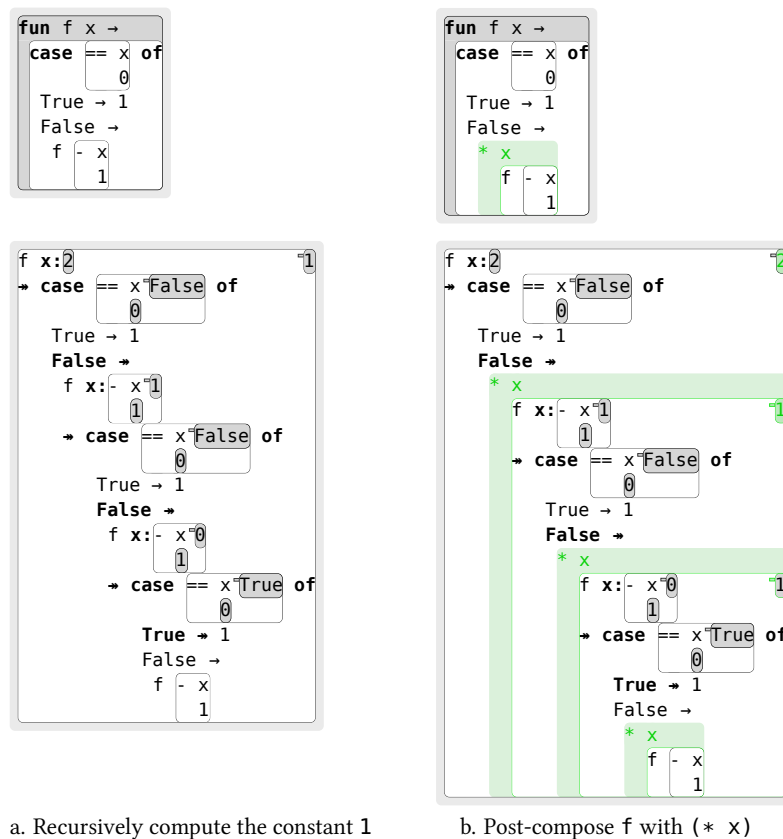
The homomorphism principle means that each distinct *edit path* between the same two programs results in a distinct computation delta. Suppose for example the user had spliced in `/` around the existing `+`, rather than changing `+` to `/` and then splicing in `+`. Then the corresponding nodes in the computation would be modified in an analogous way, with a new application of `/` being spliced into every invocation of `sumSquares`. This preservation of edits is precisely what we want, because then the computation delta is *explained by*, or attributable to, the program delta, even though the final program (and therefore by determinism the computation) is the same as with the first edit. We discuss this important topic in more detail in §2.4, in relation to Figure 2.9, where we consider an example based on `map`.

Our working hypothesis is that directly observing the impact of edits on execution in this way can be helpful even when the programmer already has good intuitions about how the program works. Those intuitions can be reaffirmed (or perhaps disconfirmed) interactively as they work, allowing them to proceed more confidently. When the programmer lacks such intuitions, directly observing the impact of changes has the potential to be even more useful. Novice programmers, for example, can see directly what is going on and can interactively explore the consequences of simple algorithmic changes. Indeed, given the likely effort involved in scaling our approach to real-world systems, a more realistic initial target might be a programming environment for teaching and learning-by-exploration.

What is more, modern software engineering practices such as test-driven development [Bec02] and refactoring [Fow99] eschew building complex behaviours all at once. Instead complex behaviours are built by modifying existing, simpler ones. In test-driven development, one introduces a new function by starting with a simpler function of the right type and gradually editing it until it has the desired behaviour, as confirmed by a unit test. Some practitioners have argued for a *bottom-up* programming style where functions are developed independently of their intended calling context [Gra94]. When the desired function has a complex type, this permits starting with a function of a simpler type and gradually transforming not only its behaviour but also its type through a series of edits. With refactoring, the idea is to preserve the extensional behaviour but reorganise the code to make subsequent edits and maintenance easier. Implicit in all these practices is a view of programming as the systematic application of code changes that transform *intensional structures*, and sometimes extensional behaviours, in well-defined ways. We argued explicitly for such a perspective ourselves in earlier work on “micro-refactoring” [Per04]. Interactive programming is a step towards

supporting these incremental programming methodologies more directly.

The significance of execution deltas to this view of programming is that they are more focused than entire executions. The deltas isolate and reveal individual features of an algorithm, saving the programmer from having to tease these separate strands apart mentally. Instead, they can formulate simple hypotheses, test them interactively, and receive informative visual feedback. The interactions leading to Figure 2.4(a) for example might have been an empirical exploration of the hypothesis that changing a numerical quantity in the input list has no bearing on the structure of the fold, but only on the value it computes.



a. Recursively compute the constant 1

b. Post-compose f with (`* x`)

Figure 2.5 Creating the factorial function

Here is another example which shows how execution deltas complement our view of programming as the transformation of intensional structure. Figure 2.5(a) shows a simple function which takes an integer argument `x`, counts down from `x` to `0`, and then just returns the constant `1`. This is a plausible intermediate definition en route to the definition of a more complex function such as factorial. Underneath we see its fully-expanded execution when applied to `2`. In (b), the user edits the definition of `f` to multiply by `x` the value returned by the recursive call, so that the function now indeed computes the factorial of its argument. The overall result of the computation is now `2` instead of `1` but more importantly the way the result is computed has a quite different structure and this is reflected visually to the user. Previously, `f` was tail-

recursive; a 1 was computed and passed all the way to the top, as was visually apparent in (a) by the entire computation having a single value pane. In (b), we see that *very same* 1 being “intercepted” part of the way up and multiplied by the current value of x at the point, producing a “different” value node (shown in green) which happens to also have the value 1. This result is in turn intercepted at the enclosing recursive call and again multiplied by the local value of x , producing the final result 2. The visual tail-call “optimisation” that we described in §2.1 above has been *unapplied*, and the change in the visualisation made explicit to the user via delta highlighting. The result is a concrete presentation of how the edit they just made transformed the intensional behaviour of the function.

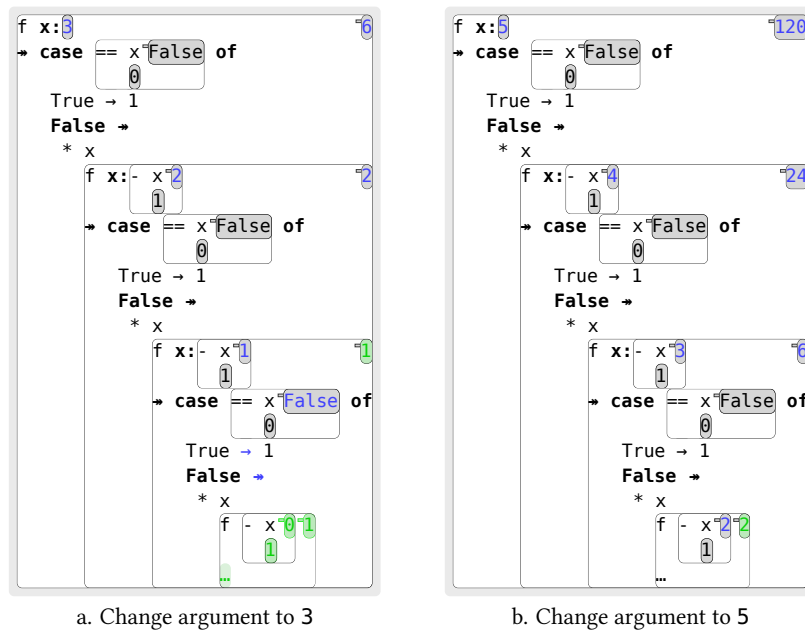


Figure 2.6 Testing the factorial function

In Figure 2.6(a), the user goes on to test their new definition at other arguments. First they edit the argument from the value 2 that it had in Figure 2.5(b) to 3. This causes the scrutinee in the final conditional test of the execution to change from `True` to `False` and therefore control to switch branches. This is indicated by the \Rightarrow arrow following the pattern `True` becoming \rightarrow , and the \rightarrow arrow following the pattern `False` becoming \Rightarrow , both changes being highlighted in blue. This confirms the user’s expectation that when the argument is incremented the recursion will go one level deeper. Indeed, the recursive call in the `False` branch, which was inactive in the previous state, is now live, as is its argument, which can be seen to evaluate to 0. And there is a new executed function body (a green ellipsis), indicating that there is now more structure to explore. The overall effect of the edit has been to “relocate” the user’s view to a different execution of factorial, whilst preserving the fact that they are viewing the computation only to a certain depth. Then in (b), the user increases the argument from 3 to 5. This time there are *two* new calls to `f` compared to the previous state, because the argument grew by 2; however because the view is still restricted

to the same depth, both the new calls are hidden away under the ellipsis. Nevertheless, the user can see that the last four values of factorial which were calculated are now 2, 6, 24 and 120.

So again, the potential benefit to the user in this case is that they can see – rather than merely imagine – exactly *how* changing the value of an argument controls the depth of the recursion. Since *f* does more work for larger arguments, incrementing the argument adds new computation at the end of the execution, by deferring the termination of the recursion.

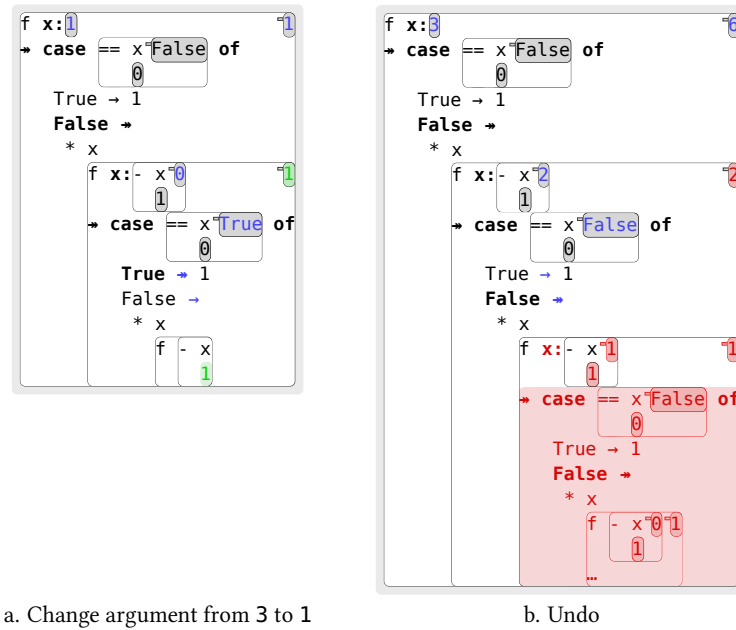


Figure 2.7 Using undo to obtain a deletion delta

Symmetrically, making the argument smaller would delete computation from the end, by making the recursion terminate earlier. In Figure 2.7(a), the user has undone the edit of 3 into 5 and changed it to 1 instead. The computation has indeed become smaller. However, the deltas that we visualise in LambdaCalc are asymmetric and cannot simultaneously show creation and deletion. This is so that the content of the UI at a given state can be read as a bona fide execution by simply disregarding the delta highlighting. The deltas we have seen up till now, which are the result of edits, we call *creation* deltas. A creation delta shows the present state relative to a prior one. A node unique to the present has therefore just been created; such nodes, as we have seen, are shown in green. But if the edit also causes nodes to be deleted, they will not be visible in the resulting delta since they are no longer part of the present state.

The user can however obtain a delta showing any deleted nodes by simply *undoing* the edit, as shown in Figure 2.7(b). Now the UI shows what we call a *deletion* delta. A deletion delta shows the present state relative to a hypothetical future state, in this case the state in which *f* was applied to 1, from which the user just returned via undo. A node unique to the present is now one which is scheduled for deletion in that future state; such nodes are shown in red. Here the user can see the part of the execution which will no

longer take place. Deltas therefore come in two flavours which differ only in the colour they use to highlight “fresh” nodes, i.e. nodes which occur in the present state but not in the state with which the present state is being compared. Changed nodes appear in blue in both kinds of delta. Deletion deltas have a subjunctive flavour; whereas creation deltas compare things to how they *were*, deletion deltas compare things to how they *would be*. The point is that there are no hidden changes: the user can always access the full content of the symmetric delta by undoing and redoing the edit.

The idea of making execution live and tangible is similar to the philosophy of programming environments like VisiProg [HW85] and Flogo II [Han03]. The key novelty that we introduced in this section is to extend this idea with explicit execution deltas, which have two important benefits. First, the user can directly observe the effects of program edits on the behaviour of their program, as the transformation of intensional structure. Second, their view of the computation can often be retained in the new state, preserving their working context, and allowing editing and comprehension to be interwoven. We illustrated this interactive approach to programming with some simple examples intended not only to convey the potential benefits, but also to bring home the absurdity of having to “play computer” in order to understand what is happening inside a computer carrying out that very task. We neglected an important question, one which is unfortunately mostly out of scope for this thesis, namely how to scale this approach effectively to large programs. We consider this topic in Future Work, §7.2.1.

2.3 Visualising structured values

In the previous section, the notion of term graph helped us make sense of the idea of the “same” sub-computation or value existing in different computations. To “change” a node is to retain it in the new state, but label it with a different constructor, or change the outgoing edges in which it participates. We now show that reuse of nodes also arises naturally *within* a state if computations and values are represented as term graphs.

Consider functional computations that operate on structured values such as lists and trees. These programs take values apart, via pattern-matching and projection, and assemble new ones, via construction. When these assembly and disassembly actions take place, the sub-graphs representing values are automatically reused. For example when projecting a component out of a structured value, we return it “by reference”, and similarly when we construct such a value, we include its components “by reference”. Sharing also takes place when a computation simply returns a value constructed elsewhere: when evaluating a variable, we return “by reference” the value to which it is bound, and for a conditional expression, we return “by reference” the value computed by the selected branch. This kind of natural sharing is a consequence of *dataflow*, the paths that values take through the computation.

Implementations of mature functional languages often use term graphs, because they tend to be based on imperative languages with pointers in which graphs are easy to implement. The kind of reuse described above is then the default behaviour, in that when mentioning a node, one must explicitly copy its sub-graph to avoid sharing its representation. With lazy languages, sharing has an additional role to play in avoiding the duplication of delayed computations [Wad71]. In these cases sharing is technically an optimisation

because it is unobservable at the level of the computation – one cannot write a program whose abstract behaviour differs depending on whether or not values are shared.

In LambdaCalc, the situation is slightly different, because the paths taken by values through the computation are made explicit to the user. (We will see this more comprehensively in §2.5 below.) These paths are determined by the semantics of the language, and establish a relation of *synonymy* between values. This synonymy in turn serves as the basis of a visualisation technique with a dual role: enabling more compact views, and helping the user understand a computation by revealing how it decomposes and composes values. This is not an “optimisation” particular to the implementation but rather an aspect of the semantics exposed by the tool.

Let us consider an example. In Figure 2.8(a), the user initially sees the value `Some("claire")`. As before, the rectangular tab to the left of the value pane indicates that the value has an explanation, but in this example the explanation is completely hidden. (It would be more consistent to display a single ellipsis for the empty partial explanation, but for compactness we omit it.) This illustrates a presentation option which might for example be enabled by default in a distributed setting whenever a value is computed remotely.

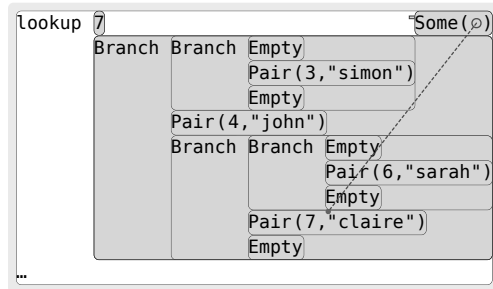
Now suppose the user wants to understand the provenance of that value, for example why the string `"claire"` appears here. They would start by revealing the partial explanation shown in (b). This partial explanation indicates that the result was computed by applying a function called `lookup` to two arguments: an integer 7, and a binary tree containing some data. Suppose that here the user knows that the nodes of the tree store (k, s) pairs and are sorted by integer keys k , and that `lookup` returns either `Some(s)`, where s is the string associated with k in the tree, or `None` if k is not found. What the partial explanation tells them is that the key 7 was found in the tree, but also that the value `"claire"` is not just equal to, but is *identical to* (synonymous with) the occurrence of the same string in the input tree. This is indicated by the dotted line pointing back from the output to the occurrence of that string in the input. Since this notion of synonymy coincides with the sharing that arises naturally as a consequence of dataflow, we call these *sharing links*.

However, although these links do represent a kind of sharing, an implementation of our approach may internally choose to store data quite differently from the arrangement implied by the sharing links. In a distributed setup, values might be “shared” in this abstract sense but duplicated in the implementation; and conversely, values might be “distinct” in this abstract sense but equal and therefore able to share a representation. Such implementation choices are *not* part of the abstract operational model, and therefore would be invisible to a user of the system. (It may occasionally be useful to see details of the underlying language implementation too, but that is not our goal here; throughout this thesis we will only be concerned with execution *with respect to a reference semantics*.)

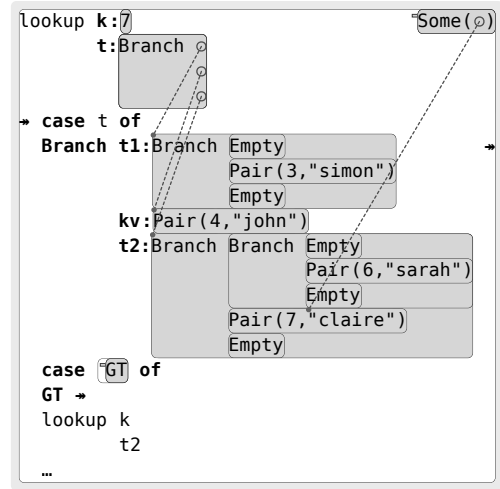
The sharing links become more informative as the user exposes more of the computation. In (c), they expand the ellipsis to reveal the body of `lookup`. We see that the tree passed to `lookup` is bound to the parameter `t` and then immediately pattern-matched as a `Branch` node. (For this example, we have suppressed the presentation of dead branches.) Pattern-matching binds the variables `t1`, `kv` and `t2` to the components of `t`, establishing further sharing links. But note how the visualisation shows the components of `t` pointing to the values of the three variables bound to them, rather than the other way around. The rule is that the *first* occurrence of a value in the visualisation, with respect to a postorder traversal of the view structure, is

Some("claire")

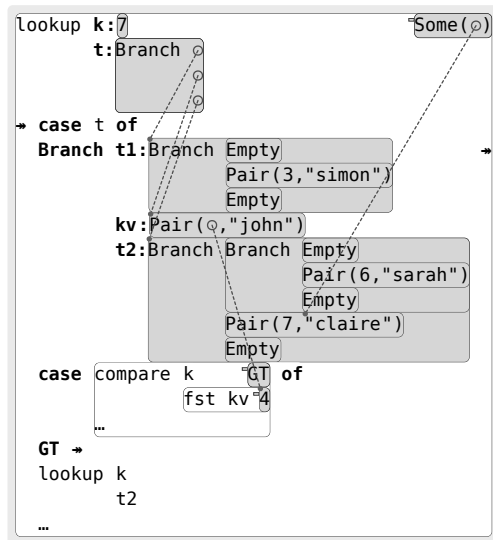
(a) Explanation completely hidden



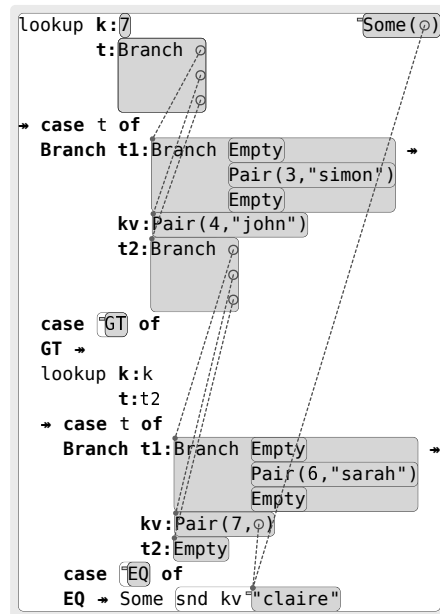
(b) Partly expanded



(c) Browsing into body of lookup



(d) Partial explanation of GT



(e) Browsing into recursive call

Figure 2.8 Exploring a computation to reveal value assembly and disassembly

the one that is actually rendered, with any other occurrences then being rendered as sharing links. This (admittedly simplistic) convention means that the user can observe sub-values propagate into the computation

via pattern-matching and projection, and propagate out of the computation via construction.

Moreover, by controlling how much they see of the execution, the user can control how much decomposition of the input tree they see. After the initial pattern-match for t , we see that some kind of comparison was made, yielding the value `GT` (“greater than”), whose explanation is also hidden. On the basis of that result, `lookup` was called recursively on the subtree bound to $t2$. In (d), the user reveals the explanation behind `GT`: the key k being searched for was compared with the first component of kv , the key-value pair currently being considered. An additional sharing link indicates the consumption of the first component of kv by the projection `fst kv`. In (e), the user expands the recursive call, and sees something analogous to what happened in (c): namely t being bound to a tree and then pattern-matched as a branch, causing more binding, and as a consequence more sharing. The user has moved smoothly from an extensional view of the computation where the function monolithically mapped input to output, to a more intensional view where the input has been broken into sub-values distributed through the computation. It is quite visible now to the user how `lookup` recursively consumes its tree argument.

To recap, the operational semantics of a functional language can be interpreted in a way that exposes the fine-grained paths that values take through the computation. This information can be exploited to make visualisations both more compact and more informative. Indeed, to neglect this aspect of the computation, as debuggers generally do, is to hide from the user an important aspect of what their program actually does, namely assemble and disassemble values. In Chapter 5 we will show that, with a modest extension to the interpreter, this fine-grained structure is inherent in the semantics, not an internal detail of an implementation. This may explain why it arises naturally in term-graph implementations, albeit as an optimisation.

Benefits aside, the visualisation scheme shown here based on sharing links is rather naïve, and quickly degrades in usefulness in the presence of non-linear sharing and as the amount of computation being visualised increases. A better approach, that would require additional implementation effort, would be to visualise the transitive reduction of the dataflow graph directly: in other words, the paths taken by values through the computation, rather than the relation of synonymy which those paths entail. We would expect this to scale better because the edges of this graph are more “local” than sharing links. It would also make more explicit the connection to the slicing features we discuss in §2.5, which work by back-propagating demand along exactly these edges. Unfortunately this is beyond the scope of the present work.

2.4 Editing structured values

As we have seen, value nodes are constructed exactly once and then reused as needed. The construction of a structured value is not monolithic but distributed through the computation, with different sub-computations responsible for creating the various parts of the value. For example a `Cons` value will have been built by a unique `Cons` computation, which deferred to sub-computations to calculate the head and tail and then included those results by reference. The “local” information about the value – the `Cons` constructor itself and the pointers to the head and tail components – needs to be stored somewhere. Any kind of non-deterministic allocation would not sit well with our goal of reusing nodes across computations in a systematic way. But since the information we wish to store is associated with a unique computation node, it can safely be stored

at a value node injectively determined by the computation node.

This now invites the question of how we allocate nodes to computations themselves. This would be trivial if there were no functions, since the computation graph would be isomorphic to the program graph. As it is, a particular expression node of the source program can be evaluated multiple times within the same overall computation. On the one hand, naïvely opting to use a “fresh” computation node for each such evaluation would prevent any sharing of executed function bodies between different computations, precluding computation deltas of the form shown earlier in §2.2. On the other hand, using the same computation node to represent each of these evaluations is no good because the expression may behave differently depending on the value of the arguments supplied to containing functions.

This latter observation suggests a simple policy for allocating locations to computations. Since our language is deterministic, fixing the values of all arguments to containing functions suffices to fix the behaviour of an expression in any context that arises within a given top-level computation. Therefore, it suffices to have the location where a computation is stored be injectively determined by the location of its source expression plus the locations of the arguments to all containing functions. This is a “safe” policy because in a given state (top-level computation), we can unravel any of these argument sub-graphs into a unique value, and so there is no possibility of attempting to assign differently-behaving computations to the same location. Or to put things contrapositively, if we were to evaluate that same expression in a context where one of the argument values were different, then it would also be the case that that argument were stored at a different location and therefore that we would be storing that particular computation at a different location. On the other hand, by keying on locations only, this scheme does allow the “same” computation to exist in different states with different contents. And broadly speaking this is what we want: to be able to *reuse* nodes from the previous computation, even when they take on different contents.

The node assignment policy just described is not merely a matter of implementation detail, because its consequences are quite apparent to the user. It can result in useful output deltas when we splice into or rearrange the elements of structured values such as lists and trees. Moreover the associated computation delta can help “explain” the output delta. Suppose the user evaluates the expression `map incr Cons(2,Cons(4,Nil))`, where `incr` is defined elsewhere and simply increments its argument. The expression evaluates to the list `Cons(3,Cons(5,Nil))`. Now suppose the user inserts a new `Cons` node, also containing 4, at the second position of the input list. As shown in Figure 2.9(a), this causes a new `Cons` node to appear in the output, also at the second position, containing the value of `incr 4`, namely 5. This seems intuitive enough. But if the user browses into the execution of `map`, as shown underneath in (b), they can also see how this output delta came about. The first thing to notice is that some child pointers become visible as *sharing links* (§2.3 above). Following our usual highlighting conventions, changed links are shown in blue, and “new” links – links whose source nodes are new – appear in green. The two sharing links shown in blue indicate where tail-pointers were modified to accommodate the insertions into the input and output.

Inside the outer execution of `map`, the argument list is pattern-matched as a `Cons` with head and tail components bound to `x` and `xs'` respectively. The tail component, which is bound to `xs'`, is the *new* input node, and is subsequently passed to a recursive call to `map`, where it becomes the value of the `xs` argument. The copy `xs`: of the formal parameter used to indicate this is highlighted in green. This is because the

copy of a formal parameter used to indicate an argument binding is considered part of the *execution of the function body*, despite syntactically appearing to be part of the application. So here, because the executed body of the recursive call is new, the formal parameter `xs` is also new. And in turn, the executed body is new because it is running in the context of a new argument, namely the new `Cons` cell to which `xs` is bound. On the other hand, the formal parameter `f` is not highlighted. This is because this occurrence of `f` can be attributed to the execution of the body of the partial application `map f`, which has no visual presence beyond providing this additional parameter. The body of this partial application is *not* new, because `f` is bound to the same closure node that it was in the previous state.

Inside the first recursive call, the situation is then reversed. The list argument, which now starts with the new `Cons` node, is pattern-matched as another `Cons` with head and tail components again bound to `x` and `xs'`. But this time `xs'` is bound to an existing `Cons` node (shown in grey) from the previous state. When this value is in turn passed as an argument to the *second* recursive call to `map`, the `xs` binding this time has no highlighting, indicating that the execution of `map` with that argument node existed in the previous state.

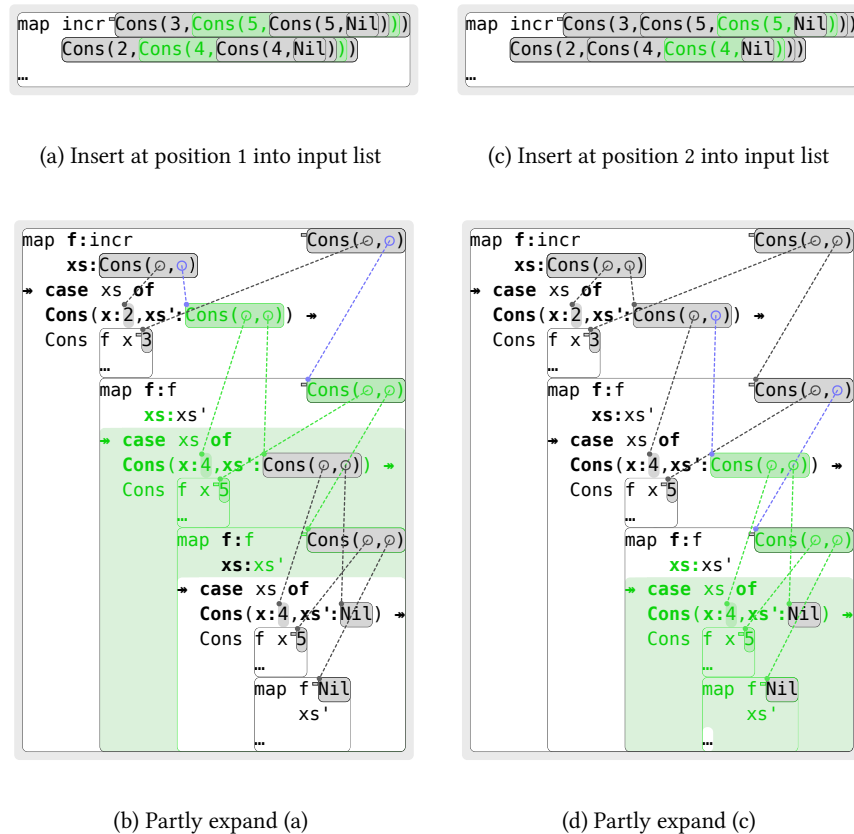


Figure 2.9 Exploring different insert positions for `map`

The user certainly need not understand all these subtleties of the delta at once. They need only note that new “work” is spliced into the computation at the position of the new node in the input list. Moreover, they

can see that the newly executed fragment of map code includes the construction of a Cons node, which is therefore itself new; the new node ends up being spliced into the output list at the same position that the new node appeared in the input. This splicing behaviour is made possible by the node allocation scheme, which injectively assigns locations to computations (based on program and argument locations), and to values (based on the locations of the computations which construct them). Thus the computation delta “explains” the output delta.

This is in contrast with the factorial example earlier, where increasing the value of the argument always caused new computation to appear at the end. The reason for this was that an atomic value such as a primitive integer can only represent a *quantity*. A *delta* in such a value can therefore only carry information about how its magnitude changed. With structured values there can be multiple edit paths between any two such values. Suppose the user were to undo the edit in Figure 2.9(b) and instead insert the new Cons node at the third position instead of the second, as shown in (c). Now the new output cell occurs at the third position in the output list. And if the user again examines the execution, as shown beneath in (d), they can see that the new computational work now appears at the end of the computation, rather than in the middle. Once again, the computation delta “explains” the output delta.

So changes to structured values can result in more interesting deltas than changes to primitive values, which can only express changes in magnitude. Here, the executions in (b) and (d) have the same unravelling: if we discard sharing information, they represent exactly the same pure computation. The difference between these two edits is therefore unobservable to the program itself. So why should the difference matter to the programmer? Well, given two structured values there are inevitably multiple edit paths between them. The particular edit path taken by the user is a “question” that asks how the computation would differ if the program were to differ in some particular way. The answer is a *delta* representing the edit that must be made to the computation in order to accommodate the user’s edit. The answer is therefore tailored to the question. In this example it will soon become apparent to the user that *wherever* they insert the new node into the input, the new node in the output will appear at the corresponding position. This is a reflection of the parallel nature of map. It is clearly beyond the capability of our tool to reveal the parallel nature of map to the user directly. But by permitting the user to ask arbitrary questions and, where feasible, obtain reasonable answers, what our approach does support is *inductive* (in the philosophical sense) and *abductive* inference⁴. The user can formulate a general, or explanatory, hypothesis consistent with a particular behaviour they have observed, and can then empirically test that hypothesis by making further observations. This route alone cannot lead to deductive certainty, but it can be useful in two ways. First, a counterexample immediately contradicts a false hypothesis. Second, results consistent with a hypothesis can increase confidence and deepen intuitions.

Suppose the user were now to change the definition of the `incr` function being mapped over the list, as shown in Figure 2.10(a). They change it from a function which increments its argument into a predicate which determines whether its argument is greater than 3, thus changing its type as well. The overall structure of the computation is unaffected by this change. The changes all lie within the invocations of `incr` itself,

⁴ The term *abduction* was first introduced by Charles Peirce, for whom to “abduce” Q from P was to “surmise from an observation of P that Q may be true because then P would be a matter of course” [Pei98]. According to Peirce, good abductive reasoning involves not simply a determination that Q is *sufficient* for P , but also that Q is among the *most economical* explanations for P .

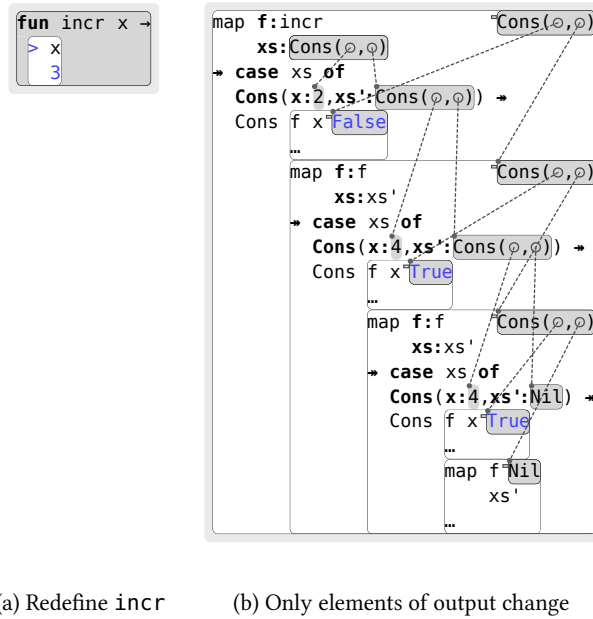
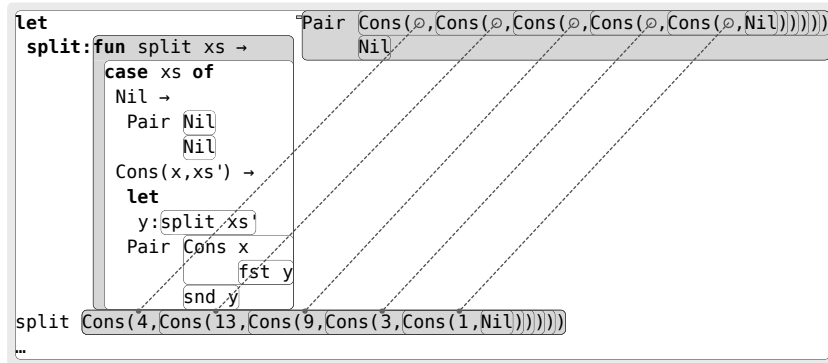


Figure 2.10 Exploring other intuitions about `map`

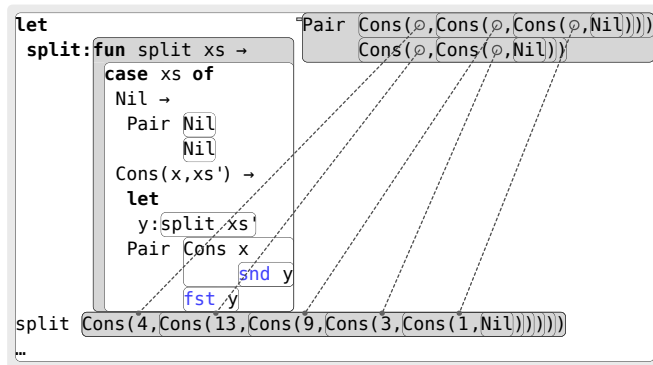
which are currently hidden, and in the values they return, which are highlighted here in blue. There is a general property at play here too: namely that `map` is parametric in the behaviour of the function being mapped, at least if that function is defined at every element of the input list. In other words the part of the computation that relates to the execution of `map` itself is not sensitive to what the mapped function does. As before, there are many (in this case, infinitely many) similar questions which have answers which are essentially a consequence of this fact, and the user is free to ask these as they see fit.

Because node assignments are deterministic, we can think of the reuse behaviour shown in these examples as a kind of *memoisation* operating at the level of locations, rather than at the level of values. The memo keys are the locations where arguments are stored, rather than the arguments themselves, and the output of a memo lookup is the location where the result is stored, rather than the result itself. Moreover every expression is independently “memoised”, not just function bodies. And whereas memoisation is normally considered an optimisation that applies within a computation, here it is observable as the reuse of locations across computations. This is similar to the use of memoisation in self-adjusting computation (Related Work, §3.10). The identities of computations and of values are interdependent: distinct expression-nodes or distinct value-nodes for the arguments to containing functions give rise to distinct computation-nodes, and in turn value-nodes constructed by distinct computation-nodes are distinct.

Because of this interdependence, certain kinds of algorithm exhibit *instabilities* with respect to certain kinds of change. An instability is when an innocuous-seeming change in the program causes a large delta in the execution, because the differential execution mechanism is unable to reuse nodes from the previous computation. An example of this is when a new node is inserted into the input of a recursive function which uses an accumulator. When the new input element is encountered, it causes a fresh execution of the function.



(a) Incorrect version of `split` (computes $\text{id} \times \text{const Nil}$)



(b) Fix preserves all node identities but “reorganises” the output

Figure 2.11 Fixing broken implementation of `split`

This new call in turn constructs an accumulator, which is then itself new, to pass to the next call. The same thing happens again, and the result is a cascade of fresh node identities for all computations and values constructed downstream of the point where the new node was encountered. Similar stability issues arise in self-adjusting computation.

While this is a general issue with our approach, there are sometimes idiomatic solutions to stability problems which involve a small amount of redesign to data structures or algorithms. We discuss some of these idioms in Appendix B. This places some burden on the user, but in our experience the effort is not much worse than organising programs to be tail-recursive. This is a common practice in the programming language Scheme [KCR98], where it is used to ensure that recursive functions execute in constant space. In the worst case, we are unable to provide useful delta information or to maintain the user’s view state across the destabilising edits, forcing the user to fall back to a more traditional debugging/programming style.

In any event, in many cases our node assignment scheme produces informative and useful deltas. As a final example, which also illustrates a limitation of our current GUI, Figure 2.11 shows a fix being applied

to an incomplete implementation of `split`, a component of mergesort which partitions a list into a pair of similarly-sized lists. A standard functional implementation of `split` uses a swap at each recursive call, in order to cons elements alternately onto one and then the other of the two lists being constructed. In (a), the programmer has almost completed the implementation, but is missing the swap step; the two lists returned by their current code are the original input list, and the empty list. In (b), they implement the swap step by replacing `fst` by `snd` and `snd` by `fst`. This edit is stable with respect to the identity of all computations and value nodes. In fact, the only changes are of some of the *pointers* in the output: between the Cons cells and their tails, and between the pair itself and its second component. The delta is therefore very informative: it tells us precisely how individual Cons nodes are shuffled around in the output as a consequence of this program change. Unfortunately these pointers have no visual presentation in the LambdaCalc GUI, and so there is nothing to highlight in blue. Treating the border around each node as a visualisation of the pointer to that node from its parent would address this. But even then, indicating merely *that* the pointers changed does not do justice to the very precise information contained in the delta. To usefully present informative deltas like this that consist mainly of moves, it would be important to animate these changes or in some other way make the continuity of identity across the states explicit.

2.5 Interactive slicing

Interactive programming allows the black-box computation of a value to be interactively disassembled into an explanation of how the value was computed. The decomposition we have considered so far is *stepwise*, or sequential: we break a computation down into child computations that have to complete before the parent computation can proceed. This sequential decomposition is nicely complemented by decomposition along another axis, namely of a computation into parallel execution flows. This kind of parallel decomposition is usually called *slicing* because it cuts vertically through the sequential structure of the computation. Slicing allows the user to interactively ask questions about how *parts* of the program relate to *parts* of the output. Slicing naturally fits into interactive programming because having the history of a computation available makes slices easy to calculate. In this thesis we extend existing slicing techniques for functional languages (Related Work, §3.4 below) with more fine-grained questions and answers and allow not just programs but also explanations to be sliced.

The user can ask a slicing question in one of two directions, forward and backward. These questions can be fine-grained, focusing on specific parts of the program or its output. A “forward slicing” question has the form: which parts of the output is this part of the program needed for? A “backward slicing” question has the form: which parts of the program are only needed for this part of the output? Forward and backward slicing questions are “dual”. Forward questions are so-called because their answers depend on how *lack of availability* of some part of the program propagates forward through the execution to the output. Backward questions are so-called because their answers depend on how *lack of demand* on some part of the output propagates backward through the execution to the program.

Figure 2.12 shows some interactions corresponding to forward questions. In each sub-figure, there are two top-level views. On the left is a function called `zipW` (“zip with”) which generalises the usual `zip` operation

on lists to take an additional binary combining function as its first argument. On the right is an application of `zipW` to three arguments: an anonymous binary function of type $\text{int} \rightarrow \text{int} \rightarrow \text{int} \times \text{int}$, and two lists of integers. The binary function simply returns the pair $(x + 1, y + 1)$ given arguments x and y . The value pane in the top-right corner shows the output, which is a list of pairs of integers.

As is usual with `zip`-like functions, `zipW` discards the extra elements if one list is longer than the other. This behaviour is implemented by the two case clauses that detect when either `xs` or `ys` is empty. The body of each clause is a `Nil`; which of these two `Nil`s is evaluated depends on which condition is detected first. In the process of understanding this code, the user might notice that although here the two input lists have the same length, the end of `xs` will be recognised before the end of `ys`, and that therefore the first `Nil` will be the only one that is evaluated. Moreover they would probably intuit that this `Nil` is “needed” in the sense that it was used to construct the `Nil` at the end of the output list.

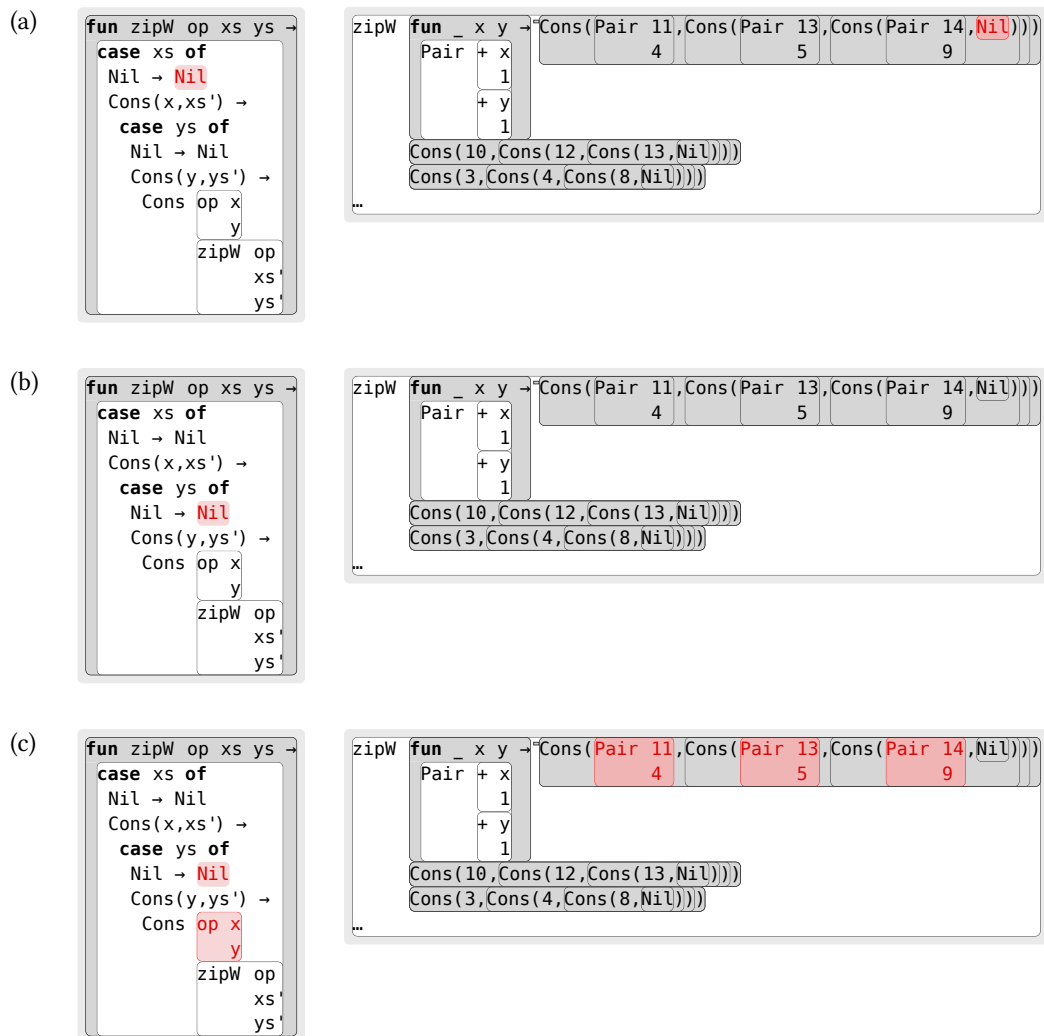


Figure 2.12 Forward-slicing to investigate contribution of parts of `zipW` to output

In Figure 2.12(a), the user explores this intuition directly by simply *selecting* the first of these `Nil` expressions. Unlike an edit, which results in a creation delta where new nodes are shown in green, a selection is interpreted as a *hypothetical deletion*, and so is shown as a deletion delta, where nodes scheduled for deletion are highlighted in red. (See §2.2 above.) The selection indicates to the system that the user is interested in what would happen if this `Nil` were *suspended*, or made to block, so that it were unable to produce a value. The system responds by highlighting the `Nil` at the end of the output list in red. This tells the user that the highlighted part of the output would as a consequence also block. To put it another way, the `Nil` highlighted in the output depends, in this execution, on the `Nil` highlighted in the program. We call the partially highlighted output an *output slice*; the output slice is a response to the question that the user posed in the form of a partially highlighted program, which we call a *program slice*. The notion of dependency at work here can be understood by thinking of the interactive program as a *parallel data-driven* computation, in which all nodes in the output are computed in parallel by the forward flow of operators and operands, possibly with some shared dependencies. In selecting an expression, the user is asking which output nodes depend in the present execution on the availability of that expression.

In (b), they select the second `Nil` which is evaluated only when `ys` is empty. Nothing is highlighted as a consequence in the output, meaning that suspending this expression would have no effect at all on the present execution. Given what they confirmed in (a), this is unsurprising. But our claim is not that these are difficult algorithmic properties to understand. Rather, our point is that to leave it to the user to simulate the execution of the program in their head in order to understand their program in this sort of way is unreasonable – not only because it is unreliable and labour-intensive, but also because it flatly ignores the fact that this calculation has already taken place in the interpreter. Discovering, or perhaps just double-checking, how parts of the program contribute to the output should be trivial, because all the required work has already been done.

In (c) they leave the second `Nil` selected (although it makes no difference whether they do so or not) and then select another part of the definition of `zipw`: the application of the binary combining operation `op`. Suspending this part of the program would have a more drastic effect on the output, namely that all of the output pairs would also block. The visualisation exposes parallel structure in the execution directly, meaning that the user no longer has to infer it. Instead, they can now see that `op x y` is responsible for populating the elements of the output list. Moreover, it is clear that the rest of the computation could proceed even with that code suspended, and would compute the “skeleton” of the output list, leaving placeholders for the three applications of `op` which would be filled when they became available. This is a more modular and fine-grained perspective on execution than afforded by the usual view of a program as a monolithic block of text producing monolithic output. The sequential decomposition we showed in earlier sections also breaks down monolithic programs; here the decomposition is parallel, into slices.

The use of the subjunctive mood to explain the red highlighting of a deletion delta is intentional. Red parts of the program are not *actually* suspended, as evidenced (precisely) by the fact that the red parts of the output which depend on them are still being computed. Rather, and as we explained earlier in §2.2, red highlighting should be read as a *hypothetical deletion*. With editing, the user formulates a “what if” question by modifying the current state – turning the hypothetical state of affairs under consideration into the actual

state of affairs. With slicing, the user formulates the question as a hypothetical action which may never be actualised.

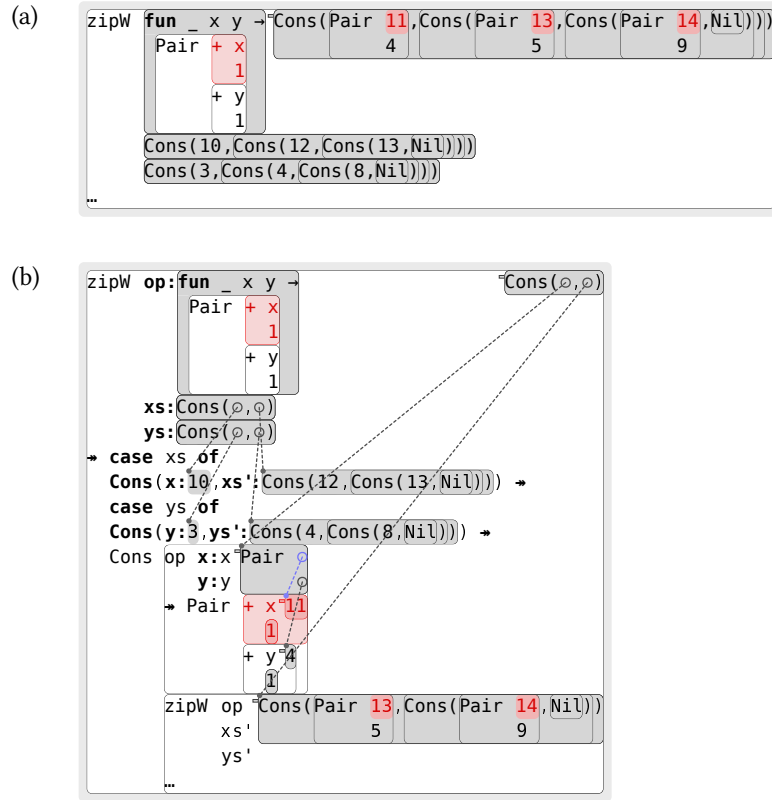


Figure 2.13 (a) Forward-slice functional argument to `zipW`; (b) expand into sliced partial explanation

So far we have considered forward slicing extensionally, looking at how it can relate program to output. It is also possible to interweave slicing with sequential decomposition, unfolding the computation of an output slice into a sliced explanation. In Figure 2.13 the user has changed focus and decided to investigate the contribution of the combining function itself to the output of `zipW`. In (a) they select the first sub-expression `x + 1` of the pair constructor inside the anonymous function. In response, the first component of each output pair is selected. This reveals that this part of the function is needed to compute those parts of the output, but does not reveal *how*. In (b) they start to disassemble the computation into a partial explanation, which reveals how the output slice was computed. They expand two function calls: first the outermost call to `zipW`, and then the invocation of its `op` argument nested within the `zipW` call.

This is enough to reveal that the output slice, which is a slice of a list of pairs, was constructed by consing a slice of a pair onto a smaller list slice produced by a recursive call. The pair slice was itself constructed by an application of `op`, inside whose execution the evaluation of `x + 1` is highlighted, being a copy of the corresponding (and similarly highlighted) part of the sliced function to which `op` is bound. The blue sharing link pointing to the result 11 of this computation is so coloured because in the hypothetical future state with

which the present state is being compared, the $x + 1$ computation does not exist and the child pointer is null.

The `zipW` example will also help illustrate the difference between backward and forward slicing. With forward slicing, the user hypothetically suspends some part of the program and sees which parts of the output would block as a consequence. “Parallel, data-driven” is the computational intuition required to understand the slicing behaviour. With backward slicing, the execution flow is reversed: the user hypothetically *relinquishes demand* on some part of the output, and the system responds by showing which parts of the program would no longer be needed. “Parallel, demand-driven” is the computational intuition required for backward slicing. Whereas forward slicing questions take the form of “availability absences” propagating forward, backward slicing questions take the form of “demand absences” propagating backwards.

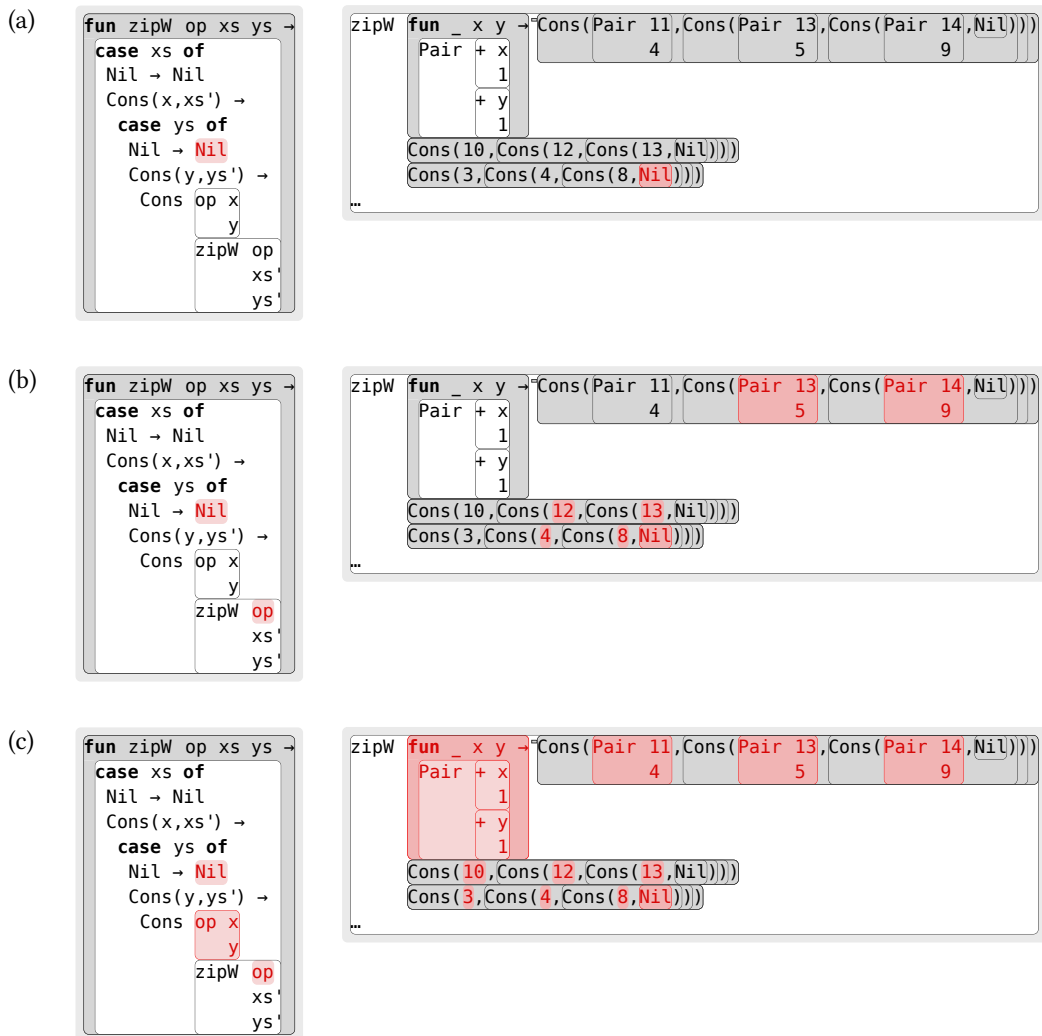


Figure 2.14 Backward-slicing to investigate neededness of parts of `zipW` for parts of output

A degenerate but important sub-case of backward slicing is to backward-slice with respect to the entire output: to ask which parts of the program would not be needed supposing the entire output were needed. The user asks such a question for the `zipW` example in Figure 2.14(a). Two `Nil`s are highlighted as a consequence. The first is the `Nil` highlighted in the body of `zipW` which we already knew to be unneeded via the *forward-slicing* question we asked in Figure 2.12(b). But we also see that the `Nil` that terminates the second input list is also not needed. These two facts happen to be related: it is because we never reach the end of the second list argument that we never exercise the case branch corresponding to `ys` being empty.

In (b), the user asks a more fine-grained backward-slicing question by selecting the last two elements of the output list, the values `Pair(13,5)` and `Pair(14,9)`. This asks what would happen if these elements of the list were not needed. The system responds by additionally highlighting the last two elements of the two input lists. This seems obvious enough: it is only the construction of those particular output pairs which consumes those particular input elements. What is slightly less obvious is that the `op` argument in the recursive call to `zipW` would also not be needed. Once we are inside the recursive call, we make no more uses of `op`. Thus slicing can reinforce or engender operational intuitions in the user before they even inspect the execution.

A backward slice is initiated by selecting some part of the *output*, whereas a forward slice is initiated by selecting some part of the *program*. This must be borne in mind when comparing Figures 2.14 and 2.13 since the direction of the “flow” is not apparent from the figures themselves. Now suppose the user goes on in Figure 2.14(c) to select the first element `Pair(11,4)` of the output list as well. Then clearly none of the input list elements would be needed. Moreover, the application `op x y` would not be needed either, since we would never apply the operation. In fact we would not need the functional argument to `zipW` at all. If the user were to experiment, they would soon discover that unlike availability absences, demand absences combine in a discontinuous way: the effect of combining them can be strictly greater than the combination of their individual effects.

2.6 Interweaving testing, diagnosis and bug-fixing

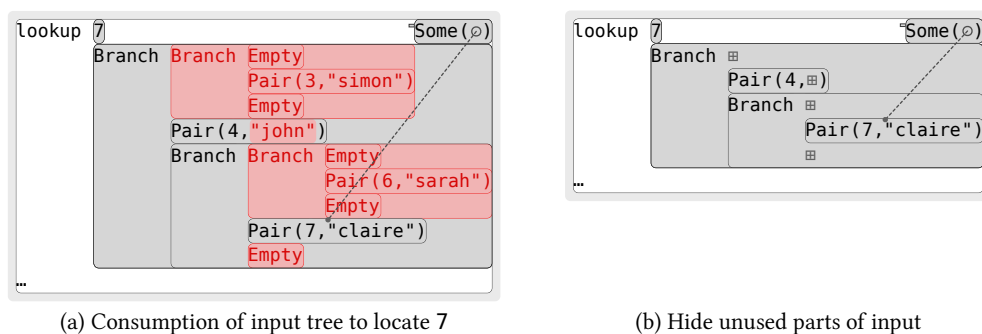


Figure 2.15 Slicing input to `lookup`

Programming normally forces a “mode change” and a restart when switching between testing, diagnosis

and bug-fixing. The point of interactive programming is to allow execution to be retroactively explored and updated so that these activities can be interwoven. Figure 2.15 returns to the lookup example we introduced in §2.3. In (a), we see a run of lookup which finds the key 7. The user backward-slices with respect to the entire output, so that the unused parts of the input tree are highlighted in red. What they see is consistent with their understanding of binary search: for a successful lookup, only the path to the located node will be consumed.

In (b), we introduce a LambdaCalc visualisation option which allows nodes which are unneeded in a particular execution to be collapsed down to a \boxplus . This can be performed automatically after every update, or applied manually. Collapsed nodes can be re-expanded; the \boxplus symbol is intended to be suggestive of this. Collapsing to a \boxplus is different in intention from hiding the execution of a function with an ellipsis. An ellipsis indicates that we are not interested in the “internals” of a function application, but only in the value it computes. A computation collapsed to a \boxplus is disregarded entirely, result included. For differencing purposes, collapsed nodes are treated as *deleted*. In Figure 2.15(b), collapsing the unused parts of the tree to \boxplus reduces the amount of information that the user has to digest: it is more obvious now that only one string in the tree is consumed, namely “claire”, and that only two keys are consumed, namely 4 and 7.

In Figure 2.16 the user does some more testing and discovers a problem. In a real-world interactive programming system, one would create collections of “exemplar” function applications explicitly tagged as test cases or assertions. This would allow the user to “freeze” the extensional behaviour of functions at particular arguments and to automatically be notified of any test failures whenever a code change caused assertions to be violated. This is a significant improvement over unit testing, as there is no need to write separate test code. It is a relatively easy feature to layer on top of what we already provide, but we do not consider it further here. Similar ideas are discussed by Edwards [Edw04]. Instead the user tests interactively, obtaining Figure 2.16(a) from Figure 2.15(b) by editing the search key from 7 to 3. The “automatic” backward-slicing setting mentioned above is enabled, so unused nodes are automatically collapsed after every update, before the delta is calculated. The overall computation returns None, meaning 3 was not found. However, suppose the user is certain that the data is in the tree and that the tree is sorted properly. What is more, they can see that an Empty is being consumed in the new state, meaning that the search reached the end of a terminal path in the tree. This Empty is shown in green because it was unneeded and therefore *collapsed* in the previous state. For differencing purposes, collapsed is the same as absent. Now that it is needed it is no longer collapsed and therefore appears new.

Unsure whether the problem is with the code or the input tree itself, the user decides to start debugging the test case by expanding the ellipsis. This partially reveals a backward slice of the execution: they can see the tree being taken apart, along the lines of what we showed earlier in §2.3, but here the unused parts of the tree remain collapsed, providing a more focused view of what is happening. The $t1:\boxplus$ binding in the pattern-match of t indicates that the first subtree was discarded, and the fact the new Empty node is inside the subtree bound to $t2$ tells them visually that the search proceeded into $t2$.

The greenness of the $LT \rightarrow$ branch indicates that the case expression took a different branch this time compared to when we looked up 7. Although this by no means pinpoints the problem it at least identifies some code that was run in the incorrect execution but not in the previous (correct) execution. The fact that

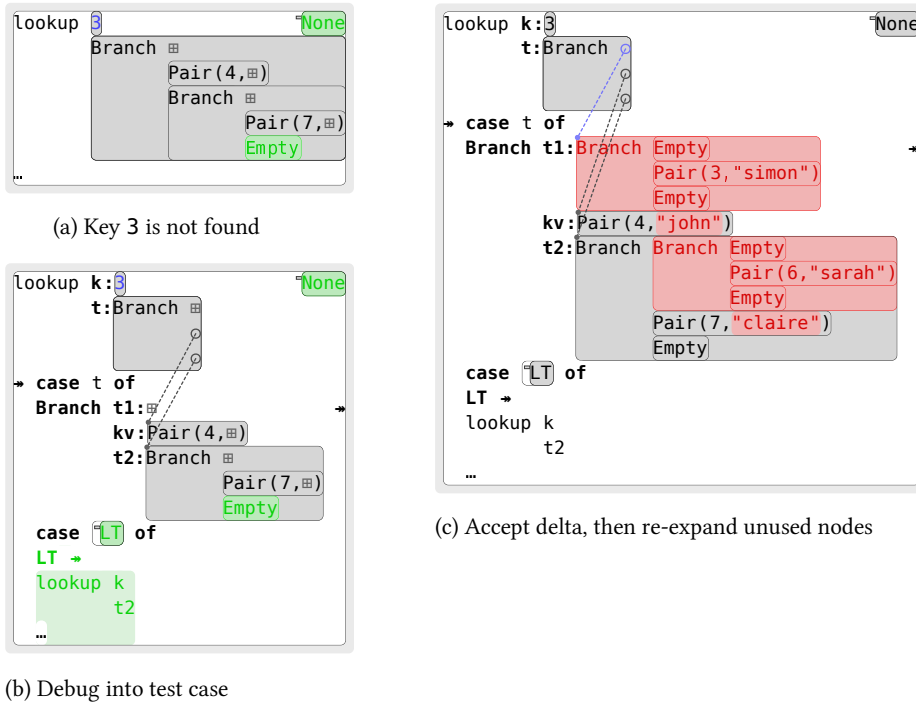


Figure 2.16 Finding and analysing bug in `lookup`

the branch appears green is a simple consequence of our decision to suppress dead conditional branches. When a conditional flow changes, the newly-live branch will appear out of nowhere (and thus seem “new”) and the previously live branch will simply disappear. However the user would still be directed towards this bit of code if we were visualising dead branches as well: the arrow glyphs \Rightarrow and \rightarrow would change and the recursive application of `lookup` would transition from dead to live. We saw an example of this in Figure 2.6(a), with `factorial`.⁵

On the other hand, although the application `lookup k t2` is green, the ellipsis underneath is not. This means that `lookup` was applied to the subtree to which `t2` is bound in the previous state, when we looked up 7. This should ring alarm bells, but again is not in itself conclusive. So the user decides to inspect the data more carefully. They can do so while retaining the particular view of the execution they currently have. In Figure 2.16(c) they expand the collapsed \boxplus nodes. This has the effect of dismissing the delta visible in (b) and then showing the unused nodes again in red. The node being searched for can be seen within `t1`. The recursive call in the `LT` branch now looks culpable, as it recurses into `t2`. The user then changes `t2` into `t1` to obtain Figure 2.17(a), which causes the overall result to switch from `None` to `Some("simon")`. The ellipsis under the recursive call turns green, indicating that a *new function body* has been executed. This is because

⁵ A better design for compact visualisation of conditionals would be to retain dead branches in the execution trace, but only allow the user to view one branch at a time. The live branch would be selected by default, but the user would be able to “flip” through all the branches (perhaps using the *z*-axis to animate). This would provide the desired compactness of presentation, but without conditional-flow changes causing nodes to appear and disappear.

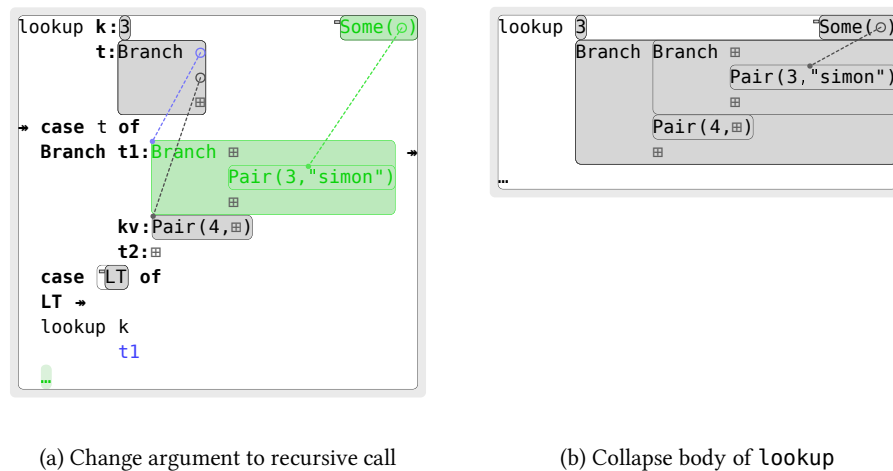


Figure 2.17 Fixing the error and returning to testing

lookup is now being called with an argument distinct from any argument with which it was called in the previous state, corresponding to a new region of the tree being searched. Finally, the “demand profile” of the computation changes. First, `t1` which was not needed at all before, is now partly needed: the path to the node with key 3 is needed, to be precise. This prefix of the subtree is shown in green, because it was collapsed in the previous state. Second, `t2` is no longer needed and so is collapsed. (There is no highlighting to indicate this because we are looking at a creation delta. If the user wanted to see this aspect of the change in neededness, they could undo to the previous state, where `t2` would then be highlighted in red.) Most important, this update to the execution preserves how the user was viewing this part of the computation. They are still debugging the program, and indeed are at the same “point” in the execution that they were before, only now with a corrected version of the program. In (b) they have accepted the current delta, “put away” the body of lookup, and returned to testing.

Admittedly, this is a very simple example and showed a bug that any experience programmer could have fixed without any kind of interactive debugging feature. Our goal here is not to convince the reader that interactive programming is indispensable for simple problems like this. Instead we would like the reader to consider the wider ramifications of being able to transition smoothly from testing to debugging to bug-fixing and back to testing, with no artificial “mode” changes and minimal loss of working context. We make our point with small examples so that we can defer tackling some of the scalability issues discussed in Future Work, §7.2.

2.7 Summary

Our belief is that programming is an inherently interactive activity that would benefit from more explicit support for interaction from a programming tool. We introduced the idea of a “how” question which is an interactive exploration of the current execution to understand how it works. We also introduced the idea of

“what if” question which we characterised as a question about a hypothetical state of affairs – how *would* the computation behave if things were different? We showed that a natural way to ask such a question is to make that state of affairs come about and observe what happens. Very often the answer to a what-if or a how-question is not the expected one, inviting further questions and the exploration of other possibilities. So the process continues interactively.

A significant shortcoming of traditional debuggers, editors and compilers is that they 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 means either resorting to an unsound edit-and-continue feature, or restarting the debugging session and effectively forgetting the carefully constructed chain of provenance questions. This is the key problem we set out to solve.

The conceptual model just presented, via our LambdaCalc prototype, reifies computation into an interactive, spreadsheet-like document, in which the formulae are themselves nested spreadsheets, and in which all changes are inherently explicit. To frame a provenance question is to browse into an execution and explore intermediate computations; to pose a “what if” question is to modify something and observe how the structure changes. Like a spreadsheet, one *navigates* between nearby computations by editing. The unique feature of our system is that the programmer can navigate whilst in a complex view state, allowing coding, program comprehension, testing, bug diagnosis and bug-fixing to be interleaved to suit the task at hand. Some changes destabilise node identities to the extent that the best the user can obtain is essentially an entirely new computation. In these scenarios, the user’s context cannot be preserved across computations, and interactive programming degenerates into something closer to the usual programming experience where task interleaving is not possible.

3 Related Work

In this thesis we show how three concepts fit together to form the basis of an interactive programming system. The three concepts are: the *reification* of computations into trace-like data structures (Chapter 4); the *slicing* of these reified computations to expose the fine-grained relationship between program and output (Chapter 5); and *differencing* computations in order to make the consequences of program changes explicit (Chapter 6). The reification, slicing and differencing of executions arise in many previous areas of research, albeit often in slightly altered forms. We expand briefly on the three concepts before considering these existing applications and techniques in detail.

Reifying computation means treating the runtime behaviour of a program as *data*, so that it can be subject to further analysis. To reify is to transition from “use” to “mention”; here, it entails specifying a data type of executions and giving a method for transcribing the temporal process of execution into the spatial structure of the data type.¹ In the literature, reified computations are often called *traces*, and the process of building one is called *tracing*. Tracing is an established technique used widely in applications that require a “whole execution” perspective on a computation. Options for building traces include instrumenting the interpreter, and instrumenting the program (§3.1); applications of traces include algorithmic debugging (§3.2), dynamic program visualisation (§3.6), debugging lazy functional programs (§3.3), provenance (§3.5), program slicing (§3.4), self-adjusting computation (§3.10) and execution indexing (§3.12). In interactive programming, reified computations play a central role: we expose them directly to the user for interactive inspection, and they also enable the other features of our system, namely slicing and differencing of computations.

Slicing is a substantial research field in its own right. The goal is usually to decompose a program into independently executable parts, for example for algorithmic debugging (§3.2), debugging lazy functional programs (§3.3), or parallelisation. Trace-like structures, such as dynamic dependence graphs, are sometimes used to calculate slices, although often these traces are not intended for human consumption. In LambdaCalc, we use slicing to allow the user to decompose a computation interactively into execution flows which can be understood independently.

Differencing computations is the problem of describing how two executions differ, which can be useful for locating bugs and understanding the impact of changes. It presupposes that the computations exist in a concrete, or reified, form; the difference or delta between two (reified) computations then takes the form of an *edit* that transforms one computation into the other. Execution indexing (§3.12) explores differencing techniques; applications of executing differencing are found in provenance (§3.5) and program slicing (§3.4). The notion of computation delta is also important in incremental computation, in particular self-adjusting

¹ Reification is also an instance of the famous extra “level of indirection”, attributed to David Wheeler [Lam07], which can supposedly solve any problem in computer science.

computation (§3.10), because it plays a role in problem characterisation. Given two computations T_1 and T_2 , which reify the execution of programs e_1 and e_2 , then *incremental computation* can be characterised as the problem of deriving T_2 from T_1 , given the difference between e_1 and e_2 , in time asymptotically equal to the size of the difference between T_1 and T_2 . It is the opportunity of reusing parts of T_1 that offers the prospect of building T_2 asymptotically faster than simply executing e_2 from scratch. The output of each incremental update is a trace that serves as the input to the next update. In this thesis we consider techniques for calculating computation deltas that reflect program edits made by the user, but we leave the issue of efficient incremental update for future work (§7.2.2).

3.1 Recording and replaying execution

Bernstein and Stark [BS95] suggest that a debugger should be seen as a general-purpose tool for observing the behaviour of a program according to some well-defined operational model. We take a similar view of traces: they should record the behaviour of the program according to a specific operational model, which we call the *reference semantics*. In particular, as we mention in §2.3, traces should not distinguish executions which the reference semantics regards as equivalent. Thus low-level tracing techniques that expose implementation details, such as those used some in time-travel virtual machines [Lew03, KDC05], are not relevant here.

The structure of traces depends on the chosen reference semantics, then, and can be considered independently of methods for building them. For a big-step semantics, the traces can to a first approximation simply be the finite derivation trees for the evaluation judgement. In other words, interpreting the evaluation judgement as a signature Σ , a suitable data type of traces is the initial Σ -algebra. A small-step (transition) semantics does not immediately give rise to an inductive data type; a trace format widely used in this setting has been the *redex trail* of Sparud and Runciman [SR97], later enhanced into the *augmented redex trail* (ART) of Wallace et al. [WCBR01]. The formal correctness of redex trails with respect to the original transition semantics was neglected until Brassel’s work on redex trails for lazy logic languages [BHHV04] and Chitil and Luo’s formalisation of ART traces [CL07].

The main difference between ART traces and the traces we define in Chapter 4 is that an ART records a small-step term-rewriting derivation, rather than a big-step derivation. ARTs have generally been used for lazy languages, where graph-rewriting approaches are common [Wad71], but the trace format itself is flexible enough to support other reduction strategies. The contraction of a redex is recorded by the addition of a *new node* for the reduct, with an edge linking it to the redex, rather than contracting the redex in situ as would normally happen in graph reduction. The fact that the ART is only ever added to is similar to how our traces are an inflation or “unrolling” of the source program. Moreover Silva and Chitil use an ART for a system which combines algorithmic debugging with fine-grained program slicing [SC06]; we contrast this with our slicing approach in §3.4 below.

Execution under a small-step semantics can also be recorded into a trace with a big-step shape; a well-known example is the *evaluation dependence tree* (EDT) of Nilsson and Sparud [NS96], which we discuss in more detail in §3.3 below. The authors informally compare an EDT to a big-step call-by-value derivation where unneeded arguments remain unevaluated. As our system does, the EDT employs an environment-

based semantics to record source names of variables.

Now we turn to methods for building traces. The general idea is to *observe* the execution in some way, and then transcribe the observations into a trace. One approach is a source-to-source technique that transforms the program to be traced into an instrumented, “self-tracing” version that builds the trace as it executes. The trace describes the execution of the uninstrumented program. This approach has been used widely in debuggers for both lazy languages [CRC⁺03, SN95, BJ97] and strict languages [TA95]. Since the tracing code is part of the user program, there is no possibility of its inadvertently depending on implementation details of the interpreter. Moreover the instrumented program can be executed on any interpreter for the language.

An alternative approach to tracing is to instrument the interpreter instead of the program. While evaluating the program, the instrumented interpreter builds a description of the evaluation that would have taken place under the reference interpreter. Chitil [Chi01] and Brassel [ibid.] both use this technique to record redex trails.

Kishon et al. [KHC91] describe a more abstract approach, which they present in a denotational setting. Their technique can be used for applications other than tracing, and works for any language for which a continuation semantics can be given. From the continuation semantics, they derive a *monitoring semantics*, which is parameterised by a *monitor state* and a set of *monitoring functions*. The monitoring semantics threads the monitoring state through the computation; the monitoring functions are used to transform the monitoring state at each point where control is transferred to the continuation. Thus monitoring semantics are a general, denotational approach to observing execution. The approach can be used to construct a trace by using a “tracing monitor” whose job it is to record the observations. Kishon et al. also show how a monitoring interpreter can be optimised into an instrumented interpreter by partially evaluating with respect to a particular monitor, and into an instrumented program by partially evaluating with respect to a particular program. Thus their approach is powerful enough to subsume both instrumentation approaches mentioned above.

We do not need the generality of Kishon et al., and so the approach we take to tracing in this thesis is to use an instrumented interpreter. The traces we build are, roughly speaking, derivation trees for a big-step reference semantics (Chapter 4). Derivation trees “proper” carry environments, contexts, and other meta-values as indices; we omit many of these details from our traces. We also do not formally establish a correspondence between traces and big-step derivations. Instead, the correctness property that our traces satisfy is that they contain enough information to be able to *run the computation backwards*, recovering enough of the original program to compute the value whose computation was traced (§5.3.2). How this relates to other notions of trace correctness is a topic for future study.

Given a trace, we often want to recover a “stepping” or “focused” view of the execution, allowing the user to recover the individual steps recorded in the trace. We call this *replaying*. If the trace is a small-step trace, such as an ART, then it is possible to replay small-step reduction steps directly from the trace, as for example described by Chitil and Luo [CL07]. When the trace is a big-step derivation tree, there is no explicit notion of evaluation “step” to recover. Da Silva shows how to derive a stepping transition system from a big-step semantics [dS91]; this can be useful for showing how debuggers correspond to a big-step reference semantics. For example da Silva goes on to prove the correctness of a stepping debugger by exhibiting a

bisimulation between his derived transition system and the debugger.

A trace affords a spatial, “all at once” view of a computation. Replaying it recovers the temporal, stepping view of the computation from which the trace was recorded. Indeed, record and replay relate these two perspectives to each other: a trace can be “replayed” as a stream of observations of the computation; and any finite prefix of such a stream can be “recorded” into a trace. Victor argues convincingly for the importance of being able to switch freely between these perspectives [Vic11]. For systems that provide the temporal, stepping view, basing that on trace replay means that execution can be explored after the program has run and the focus moved freely backwards and forwards in time. On the other hand, traces can be expensive to create and store; we consider efficient ways of representing traces in Future Work, §7.2.1. It is possible to provide a temporal, stepping view without an underlying trace, by simply allowing execution to be suspended and resumed by the user. However it is not then possible to step backwards, because at each transition information is lost about prior states. We discuss ways of presenting execution based on all of these approaches in §3.6 below.

3.2 Algorithmic debugging

In LambdaCalc, the user can peek behind a value to obtain an explanation of how it was computed. Typically, that explanation will mention other values, which in turn have their own explanations. These nested explanations can be hidden by default. Keeping the explanations of intermediate values hidden unless the user explicitly requests to see them allows the user to backtrack efficiently through a large computation, only inspecting the computational history of values which are of interest.

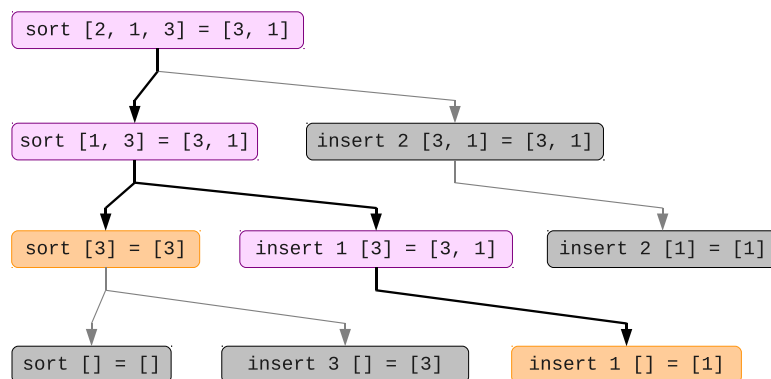


Figure 3.1 Evaluation tree for erroneous insertion sort

This mode of exploration is closely related to *algorithmic debugging*, a method for semi-automatic bug localisation originally due to Shapiro [Sha83]. Algorithmic debugging assumes an oracle, perhaps the programmer or a database of expected test results. For any step in the computation, the oracle must be able to say whether that step yielded the expected result. If the oracle can provide correct answers, then algorithmic debugging will reliably locate the step in the computation where the bug first manifested, and thus be able

to determine the faulty source expression.

Since Shapiro's original formulation, algorithmic debugging has been extended in several directions, for example by Nilsson and Fritzson to a lazy functional setting [NF92]. Cheda and Silva provide an up-to-date overview of the state of the art [CS09]. We illustrate the idea with an example adapted from Fritzson [FAS94]. It shows an algorithmic debugging session which locates a bug in an implementation of insertion sort, with the user playing the role of oracle. We omit the source code, as the example is easy enough to understand from the evaluation tree shown in Figure 3.1. For the moment ignore the colour scheme. The dialogue in Figure 3.2 shows the questions asked of the user by the automatic procedure, along with the user's responses. The algorithm starts at the root of the evaluation tree, at each node visited asking the user whether the result at that node is correct (yes) or incorrect (no), and proceeding depth-first into incorrect children. The algorithm has located the bug when it reaches a node which is incorrect, but whose children are all correct. (We avoid red and green to prevent confusion with the colour scheme introduced in Chapter 2.) Referring back to the evaluation tree in Figure 3.1, where the meaning of the colours should now be apparent, it should be intuitively clear that the number of questions that have to be answered is often logarithmic in the size of the computation. Grey indicates nodes that are not visited at all.

```
sort [2, 1, 3] = [3, 1] ?
> no
sort [1, 3] = [3, 1] ?
> no
sort [3] = [3] ?
> yes
insert 1 [3] = [3, 1] ?
> no
insert 1 [] = [1] ?
> yes
An error has been located in the function 'insert'.
```

Figure 3.2 Algorithmic debugging session for erroneous insertion sort

As suggested, there is a strong affinity between the interactive nature of algorithmic debugging, and our approach to execution browsing. To make the connection clearer, we consider a list of desiderata for UIs for algorithmic debugging due to Westman and Fritzson [WF93]. They begin by identifying a number of shortcomings of the kind of algorithmic debugging experience suggested by Figure 3.2. They note the lack of easy navigability to the *source code* of the relevant functions, obscuring the relationship between the questions and the original program. They also note that the *execution context* is absent, making it hard for the programmer to understand why a particular question is being asked. Finally, they argue for flexible starting points for debugging sessions, in preference to always starting at the beginning of the execution, plus the ability to skip trusted or irrelevant parts of the execution.

Westman and Fritzson then sketch an alternative design for an algorithmic debugger which addresses these deficiencies. They forgo the purely textual Q&A session of Figure 3.2 in favour of an experience oriented

around an interactive exploration of the evaluation tree itself. This locates each question at the step of the computation to which it pertains, making the context immediately apparent. Combined with the ability to easily access the source code for any node, as well as to expand and collapse nodes, their design overcomes most of the problems identified with the original interface. Although restricted to the exploration of a single execution, the system they describe is quite close to ours. We more tightly integrate source code with the evaluation tree, in particular showing live variable bindings and thereby making the full execution context apparent to the user. On the other hand LambdaCalc provides no debugging automation, and instead the user would have to drive the process manually.

A final feature Westman and Fritzson consider important for algorithmic debugging is the ability to save a debugging context and resume it later, restoring the potentially complex view of the execution from the prior session. Although we do not implement this feature yet, it is worth briefly considering how it could be added. In Chapter 2, we showed how LambdaCalc allows complex debugging views to persist across certain kinds of edit, through the assignment of a unique identity to each node in the computation. Since these identities are derived deterministically from the identity of nodes in the source program, to make a computation persistent across user sessions we need only store the source code in a way that preserves the identity of those nodes. Persistent views of executions are then readily implementable.

There is also a close connection between algorithmic debugging and slicing, since in general only part of the computation contributes to a given erroneous result. This connection was first studied by Pereira in the context of Prolog [Per86], and then extended to functional languages. Chitil combined algorithmic debugging with dynamic slicing [Chi05], but only allowed slicing with respect to the entire output, rather than specific parts of the output. Silva and Chitil, using a more formal approach, developed a fine-grained slicing technique based on Augmented Redex Trails [SC06]; as with our approach, the user is able to slice a sub-computation with respect to some *part* of the output which is deemed incorrect. We return to this in §3.4 below.

3.3 Debugging lazy programs

Lazy (call-by-need) execution can be difficult to follow because function arguments are not evaluated until they are needed. Control flow jumps around the program in deterministic, but potentially confusing ways. Taking a slightly more abstract view of call-by-need evaluation can help. One approach is the *evaluation dependence tree* (EDT) of Nilsson and Sparud [NS96]. The EDT represents the sharing that arises in call-by-need, but omits details of when particular redexes are demanded. Instead, an argument that is eventually needed is recorded in the EDT as evaluated at the point at which it is passed to the function, as in call-by-value. Unneeded arguments remain unevaluated, preserving the termination behaviour of call-by-name.² The advantage for debugging is that the EDT execution flow more closely mirrors the structure of the original program.

An EDT thus resembles a call-by-value trace where some expressions are suspended. In fact an EDT is very similar to the *backward-sliced* call-by-value computations introduced in Chapter 2 and discussed in

² For this reason we avoid an earlier name for this, *strictification*, introduced by Nilsson and Fritzson [NF94].

detail in Chapter 5. The connection becomes apparent if we interpret a *deleted* node in a sliced computation as indicating the location of a thunk that would never be forced in a lazy computation. Consider a lazy computation which produces a non-empty list. The output will be of the form $\text{Cons}(e_1, e_2)$ where e_1 and e_2 are thunks. We call a value of this shape, which has been evaluated only to the top-level constructor, a *lazy tuple*.³ The lazy computation will have done only enough evaluation to compute the output to that depth. On the other hand, a call-by-value evaluation of the same program will fully evaluate all expressions reached during execution, producing an eager value such as $\text{Cons}(3, \text{Cons}(4, \text{Nil}))$. But if we were to relinquish demand on the head and tail of the output list and then *backward slice*, we would obtain a call-by-value computation from which the parts no longer needed have been deleted. The dynamic backward slice of the computation thus seems to capture exactly what happened in the lazy evaluation. Moreover, the evaluation of each argument expression, to the extent that it took place, is recorded in the trace as happening *before* the evaluation of the function body, as in an EDT.

Biswas was the first to make this connection between backward dynamic slicing and laziness. However, his theorem only relates backward slices and lazy computations for programs whose output is a single nullary constructor such as `Nil` [Bis97]. This restriction is a consequence of his approach to slicing, which is always with respect to the entire output of the program. Lazy evaluation corresponds more generally to backward slices where the demand on the output is specified by a lazy tuple. Our slices are more general still, in allowing output demand to range all the way from no demand at all to fully strict output. This suggests an interesting direction for future work: generalising lazy evaluation to compute with arbitrary output demand, specified by an output slice, rather than with the fixed demand implied by the requirement to compute a lazy tuple. We discuss this in Future Work, §7.2.5.

3.4 Program slicing

A key feature of LambdaCalc is the ability to interactively explore the fine-grained relationship between input and output. This can be done in two directions: the user can highlight part of the program, and have the part of the output that it contributes to automatically highlighted; or they can highlight part of the output, and have those parts of the program that only contribute to that part of the output automatically highlighted. Both of these exploratory capabilities are a form of dynamic slicing: the former is *forward* dynamic slicing and the latter *backward* dynamic slicing. The term “slicing” is originally due to Weiser [Wei81].

The program slicing literature is large, covering both static and dynamic techniques with a wide variety of properties. See Xu *et al.* [XQZ⁺05] for a comprehensive survey. Slicing techniques were initially developed for imperative languages [Tip95, FT98]. Here we consider work on dynamic slicing for functional languages. The work of Biswas [Bis97], Ochoa *et al.* [OSV08] and Silva and Chitil [SC06] is closest to ours. Biswas computes slices of strict, higher-order programs, but only considers slicing with respect to the entire output. Our slicing criteria are more flexible, allowing specific portions of the output to be selected. Ochoa *et al.* also allow more flexible slicing criteria, but only for first-order lazy logic programs.

³ Sometimes such values are said to be in *weak head-normal form* [Jon87], but we avoid that term as it sometimes has a more specific technical meaning.

Silva and Chitil’s work, which we also mentioned above in §3.2 in relation to algorithmic debugging, is probably the most similar to ours. Their approach is based on the Augmented Redex Trail (ART), a trace format described earlier in §3.1. They too permit parts of values and expressions to serve as slicing criteria, but again in a lazy setting. Although our techniques appear to be the first where fine-grained output criteria can be used in a strict, higher-order setting, this difference is arguably minor. More significant is that Silva and Chitil are only able to show that their slices are minimal with respect to a notion of *weak influence*, a form of dynamic dependency between nodes in the ART. We give an order-theoretic account of dynamic slicing which characterises the problem in terms of programs, allowing us to state purely extensionally what it is for a dynamic slice to be minimal, independently of any notion of trace. In Silva and Chitil’s work, the notion of minimality is tied to the ART.

While Silva and Chitil’s approach is based on the ART, Biswas and Ochoa et al. rely on labelling parts of the program and propagating the labels through the execution to determine which parts of the program contribute to the output. Although we use something similar for execution differencing in Chapter 6, our approach to slicing does not require label propagation. Our traces also reflect closely the syntax of expressions, allowing us to “unevaluate” trace slices back to program slices, as described in §5.3.

In our earlier collaborative work on security and provenance mentioned in §3.5 below [AACP12], we also investigated slicing techniques based on big-step traces. The “disclosure slicing” algorithm presented there is similar to the trace slicing algorithm given here in §5.4; it ensures that a sliced trace retains enough information to show how a particular output was produced. But in this thesis we move beyond the slicing techniques described there, in also considering program slices, giving a novel abstract characterisation of the problem of dynamic program slicing, and showing how to do fine-grained forward and backward differential slicing. We also presented these developments in subsequent work [PACL12].

3.5 Provenance

Provenance is a relatively recent field concerned with the auditing and analysis of the origins and computational history of data. It has applications in databases [BKT01, BCV08, GKT07, FGT08], security [CJPR08, SCH08] and scientific workflow systems [BF05, SPG05, DF08]. Reified computations are already a general form of provenance; for any value they provide a complete computational account of how it came to be. However, existing provenance techniques have focused on more specific kinds of query, such as “what particular tuples in the database contributed to this result?” – so-called “where”, “why” and “how” forms of provenance. Answering such queries may require knowing certain things about the execution, but the answer may not itself take the form of a computation.

Most provenance work has focused on languages of limited expressiveness, such as database query languages, rather than general-purpose languages, and has often lacked any formal semantic foundation. The techniques either rely on some form of trace, or work by propagating annotations through the computation. Some efforts in a more formal direction include Hidders et al. [HKS⁺07], who model workflows using a core database query language extended with non-deterministic, external function calls, and partially formalize a semantics of *runs* which are used to label the operational derivation tree for the computation.

Although the goals of provenance overlap broadly with ours – allowing users to understand where values come from and perhaps how they were computed – specific forms of provenance are not particularly relevant to our work. The significance is more that the kind of execution environment that we propose allow provenance queries to be answered *ex post facto*, i.e. after the computation has already taken place. (This is sometimes called *offline analysis*.) The problem of provenance is essentially one of dynamic program analysis, namely being able to obtain a suitable abstract view of an execution. If the history of a computation is readily available, offline provenance queries can be supported without having to anticipate their need, or even the kind of queries that might be asked. In earlier collaborative work on security and provenance with Acar, Ahmed and Cheney [AAP12], we showed how offline analyses corresponding to many previously-studied forms of provenance can be performed on traces and in particular on trace slices. This work does not form part of this thesis, but is a useful demonstration of the utility of reified computation as ubiquitous infrastructure. And reified computation can of course be useful for dynamic program analyses other than provenance; for example Salkeld et al. use execution traces to carry out offline analyses using aspect-oriented techniques [SXC⁺11], which are normally only applicable at runtime.

3.6 Dynamic program visualisation & visual programming

In interactive programming the user is always located at a particular execution. In our implementation, we present that execution to the user visually as a structure that they can explore. Textual or graphical presentation of a program’s execution, usually to aid comprehension, is called *dynamic program visualisation*. Debuggers are an important special case of dynamic program visualisation in which the comprehension task is to diagnose an anomalous behaviour. When the user can also change the program via the same system, it is a form of *visual programming*. Since the latter can include the former, we consider both topics together; at the end of the section we consider two visual programming systems where there is no visualisation of execution, but only of the results of a computation. The “live update” aspect of visual programming we discuss separately in §3.7 below. In this section we concentrate on visualisation.

Visual programming has its origins in early work in graphical user interfaces (GUIs), human-computer interaction (HCI) and the development of the personal computer. Between the early 1960s and the late 1980s there were three particularly influential systems: Sutherland’s Sketchpad [Sut63], Borning’s ThingLab [Bor79], and Smith’s Alternate Reality Kit [Smi87]. However these systems were based on constraint programming, which does not concern us. Spreadsheet languages are another important example of visual programming system which we treat separately (§3.8, below).

There are two main ways of visualising execution, corresponding to the two perspectives on a computation discussed in §3.1 above. The first is the animated, “stepping” approach familiar from integrated development environments (IDEs), which locates the programmer at a point in time and allows them to step forwards through the execution and possibly backwards. We call this the *temporal* approach as it represents computational history in time. Stepping backwards is only possible in the temporal approach if the visualisation is based on an underlying trace. The other approach is to present computational history *spatially*, as a data structure. With the first approach, one challenge is to make apparent to the user the connection between

the dynamic visualisation and the source program, which is usually presented separately (as a spatially extended structure, such as a text buffer). We shall see that when both code and execution are presented spatially, there are opportunities for a more unified presentation. We consider text-centric systems to fall within the domain of dynamic visualisation and visual programming whenever the text is used not just to present a static program but also to visualise its execution.

The level of detail in dynamic program visualisation techniques varies from presenting every execution detail to focusing only on certain features, such as system-level components, interactions, runtime data structures, threads, or erroneous behaviours such as deadlocks. (For a recent survey, see Reiss [Rei07].) Ignoring some aspects of the computation and focusing on others allows the visualisation to be tailored to specific problems and can also make the techniques more applicable to large programs. Trace-based approaches allow decisions about level of detail to be made after the program has run, without having to re-run the program, since visualisations can be defined as functions over the reified computation. This is important for other forms of dynamic analysis than visualisation; for example provenance (§3.5, above). Moreover as we argued in Chapter 2, *ex post facto* analysis may be the only option in a distributed setting if the computation we want to debug or visualise is irrevocably in the “past”. In LambdaCalc, we take a relatively naive approach to visualisation and simply allow the entire execution to be explored.

LambdaCalc’s approach to visualisation is spatial, but we will first consider some systems based on the temporal approach, including the stepping debuggers commonly found in IDEs, by way of comparison. With these systems, a standard visualisation scheme involves maintaining separate views of the source code and the execution context, and then *highlighting* the region of the source code corresponding to the current point in the execution. The dynamic context is usually the call stack with active variable bindings. The user can single-step the highlighted position in the source code with the option to “step over” or “step into” procedure calls. These debugger implementations tend to be rather ad hoc, exposing compiler implementation details, rather than being tied to a particular abstract machine. Moreover they are not usually based on replay of a trace, and so going backwards is not usually supported.

Chitil’s Hat Explore stepping debugger for Haskell [Chi05] has a more explicit connection to an operational semantics than typical IDE debuggers. It is also trace-based, taking a lazy computation represented as an evaluation dependence tree (EDT), a trace format described in §3.3 above. Figure 3.3, adapted from Chitil, shows Hat Explore in action, with a buggy Haskell version of insertion sort similar to the example used in §3.2 above. (Recall that in Haskell, strings are lists of characters, and that the top-level program returns a value in the IO monad.) On the left-hand side, the lower half of the frame contains the source code of the program; the expression highlighted in blue corresponds to the current location in the EDT. The EDT itself is not fully visualised; instead, as with most stepping debuggers, the user only sees the “call stack”, plus the current location. This context appears in the upper half of the frame. The current location is also highlighted in blue, to emphasise the connection to the highlighted source expression underneath. The call stack is essentially the path to the root of the EDT. The right-hand side shows the state that results from stepping *over*, rather than *into*, the call to `insert` highlighted on the left. This has the effect of returning from the enclosing call to `sort` and popping item 4 from the context.

Another temporal approach used in stepping debuggers is exemplified by Chang et al.’s debugger for

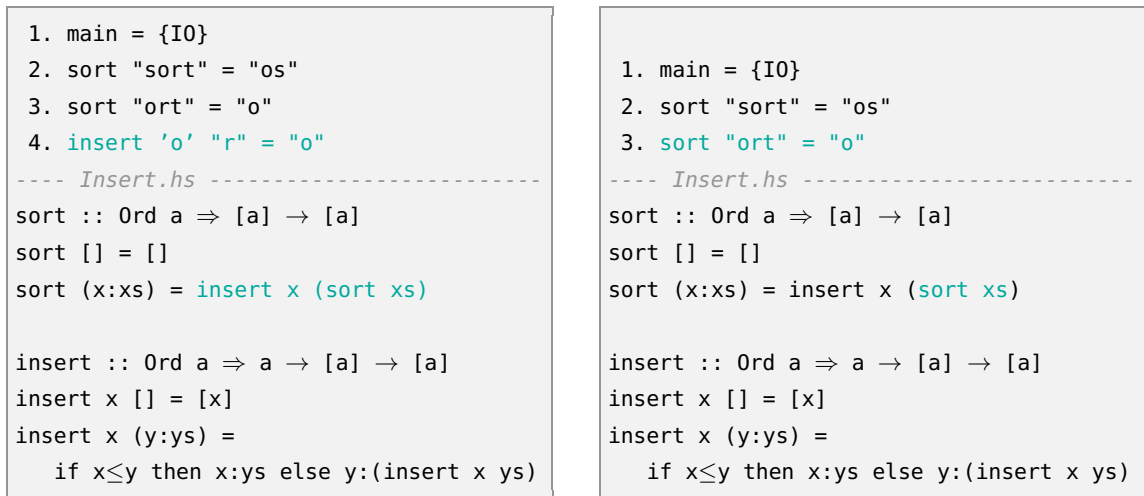


Figure 3.3 Stepping through insertion sort in Hat Explore

Lazy Racket, a lazy Scheme-like language [CCBF11]. The idea is to animate the execution as a rewriting of the program text, emulating the reduction semantics rather than a big-step semantics. Figure 3.4 shows an example adapted from Clements et al. [CFF01]. On the left-hand side, at top of the window, we see the Scheme source code for factorial. Underneath is what remains of a partly-executed call to `fact 14`, which has now been reduced to `(* 14 (fact 13))`. The blue highlighting shows the current redex; the right-hand side shows the state that results when the user performs the reduction step, replacing the redex by its contractum, which is shown in purple. (In Clements et al.’s implementation, the user sees these two views side-by-side.)

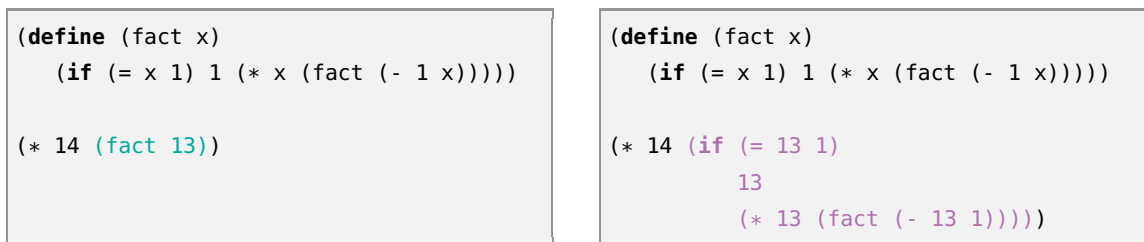


Figure 3.4 Stepping through factorial in Lazy Racket

With a temporal approach, the programmer steps backwards and forwards in time in order to see the details of the execution. In the spatial approach, the user browses into a trace, or reified computation, to reveal internal structure. This is approach taken in LambdaCalc, and also in George’s pedagogical system EROSI [Geo00], Hancock’s pedagogical system Flogo II [Han03], and Edwards’ language Subtext [Edw05, Edw07]. A spatial view of computation lends itself more to *direct manipulation*, a term due to Shneiderman [Shn83]. In a direct manipulation interface, the GUI is “transparent” and merely reveals, and permits interaction with, structure already present. When the trace is a big-step trace, one spatial approach is to present the execution

as an *unfolding* of the program source.

Figure 3.5 shows the execution in Subtext of a function called `Factorial` applied to an argument 3. The underlying language is a pure, untyped object calculus based on prototypes, rather than classes. The example code is still incomplete: the programmer has not yet added the multiplication step required to implement factorial. Here we only explain enough of the UI to give the reader an impression of the similarity to our approach. The application `Factorial 3` is represented as an *object* with a method called `1st`, whose body is the constant 3, representing the argument binding, and a method called `=`, whose body computes the result. The methods of this object, and of the object representing the nested call `Factorial 2`, are shown vertically aligned. The body of `=` returns the value of the `=` method of a local `Choice` object, with three methods `if`, `then` and `else`, encoding a conditional. The methods of the `Choice` object, and of the remaining objects in this example, are shown in an alternative layout style which aligns the methods horizontally within parentheses, with the exception of the `=` method, which is shown to the right of the closing parenthesis. The user can toggle between the horizontal and vertical layouts as required. The `if` method of the `Choice` in turn returns the value of the `=` method of a primitive `Equality` object, and so on.

The recursive call is a *copy* of the containing `Factorial` object, as indicated by the *compass* at the top of the object, pointing to the source of the copy. Here, the copy locally overrides the `1st` method to bind it to the value 2 computed by the `Difference` object local to the containing `Factorial` call. Copying is ubiquitous in Subtext. For example, in the inner call to `Factorial`, the `1st` argument has a compass indicating that its value 2 is a *copy* of the 2 computed by the application of `Difference` in the enclosing application. The key things to note are that recursion is unwound, like it is in a big-step derivation tree, and also that execution structure is live and modifiable. If the user were to add the required invocation of `Multiply` into the outer `Factorial` object, the inner copy would automatically inherit it and the computation would update correctly. There are some differences between our approach and Subtext. Subtext makes dataflow explicit via copy links; while dataflow explains the sharing of values that arises in `LambdaCalc`, the dataflow edges themselves are not explicitly represented. On the other hand, interactive dynamic slicing allows the user to indirectly observe the structure of the dataflow graph, and in particular its *parallel* structure, which is not explicit in Subtext. Moreover, in `LambdaCalc` all changes are described by explicit deltas, allowing the user to see the precise consequences of changes.

Hancock’s `Flogo II` also represents (sequential) execution as an unrolling of the original program [Han03]. A `Flogo II` program is a “machine made of text” (p. 73). There are other similarities to `LambdaCalc` and Subtext: for example intermediate values are shown associated with the step in the computation which produced them, and their display can be toggled on and off. Unlike our system, `Flogo II` supports concurrent processes as well as sequential execution. There is no provision for sharing, and the UI approach they use is not suitable for structured values. Executions are directly editable; the edits are reflected back to the original program and propagated to all other executions of the same function.

The earliest visual programming system to embrace this philosophy however was probably Weiser and Henderson’s 1985 `VisiProg` concept [HW85], implemented for Basic and Pascal in a system called XED. Inspired by `VisiCalc`, the first spreadsheet system, `VisiProg` provided a live programming environment, immediately reflecting the user’s changes in the visualised computation. The *z*-axis was used to visualise dynamics:

```

Factorial  $\Delta$  Identity function  $\odot$ 
  1st: 3
  Difference(Factorial.1st  $\ominus$  3, 1) = 2
  Factorial :: Factorial  $\odot$ 
    1st: Difference. =  $\odot$  2
    Difference(Factorial.1st  $\ominus$  2, 1) = 1
    Factorial(Difference. =  $\odot$  1) = 1
    Equality(Factorial.1st  $\ominus$  2, 0) = False
    Choice(if: Equality. =  $\odot$  False, then: 1, else: Factorial. =  $\odot$  1) = 1
    =: Choice. =  $\odot$  1
  Equality(Factorial.1st  $\ominus$  3, 0) = False
  Choice(if: Equality. =  $\odot$  False, then: 1, else: Factorial. =  $\odot$  1) = 1
  =: Choice. =  $\odot$  1

```

Figure 3.5 Recursive function call in Subtext, partly expanded

for example each iteration of a loop was displayed in its own window, using a slightly skewed overlapping. The intention was to add other interactive programming features to VisiProg, in particular incremental update (discussed further by Karinthe and Weiser [KW87]) and interactive dynamic slicing. However, it appears that neither of these aspects of VisiProg went beyond the discussion stage, and the significance of VisiProg lies more in the vision than in the details that were actually worked out.

Finally, we consider two visual programming systems which are related to LambdaCalc, but which expose only the extensional behaviour of computations to the user, keeping the internal structure hidden. Elliott’s *tangible functional programming* is based on the notion of a “tangible value”, which is a composable, visual presentation of a pure (and possibly higher-order) value [Ell07]. Elliott implemented his concept in a Haskell-based system called Eros. In Eros, one composes values by composing their visualisations: for example, one obtains the visualisation of a pair by pairing the visualisations of its components. The main novelty of Elliott’s work is what he calls *deep application*, a set of combinators that allow functions to be transformed to operate on parts of structured values and a method for translating mouse gestures into sequences of such combinators. This is useful for *gestural* composition of tangible values. As mentioned above, functions are treated purely extensionally; a visualisation of a function is an interactive view that allows the user to sample the function dynamically, i.e. manipulate its input interactively and see the corresponding output change, but not to “see inside” the execution of the function.

The other system is Hanna’s Vital, a live, visual extension of Haskell [Han02]. The user interface provides a number of built-in spreadsheet-like interactive visualisations tailored to particular data types. These can be combined to build new visualisations for other data types. For example, a triangular array with elements of type `Maybe Int` might be rendered as rows of cells, with `Nothing` rendered as an empty cell, and `Just n` rendered as a cell containing `n`. When the user types in a Haskell expression which creates or manipulates a value of one of these data types, the value of that expression is automatically rendered using the relevant visual presentation. Moreover, the user can edit the sub-values that appear in the visualisation and have those changes reflected back to the Haskell expression. Vital extends Haskell, and so is based on Haskell’s lazy evaluation strategy; by exploring more of the visual presentation, the user can force more evaluation of

the data structure being visualised. But again, Vital only provides an extensional view of a computation.

3.7 Live programming & live coding

Live programming environments combine editing and execution.⁴ The programmer can change a running program and immediately see the effect on the execution. Visual programming languages (§3.6 above) are often live. Live programming has many applications: providing useful feedback to the user; upgrading running systems; and creative activities that benefit from interjecting the user’s activities into the ongoing computation. The term *live* is sometimes attributed to Tanimoto [Tan90], although examples of such systems date from considerably earlier. Liveness is perhaps best thought of as a property of a programming environment rather than of a language; many kinds of language (functional, imperative, logic) have been implemented in a live style. Liveness is to be contrasted with functional reactive programming (FRP, §3.9 below) and self-adjusting computation (SAC, §3.10 below) which assume a fixed program syntax during execution.

What a “running program” means varies. At one extreme, the program is a closed expression and “running” in fact means *finished running*: the states of the live programming environment are idle, representing the execution of that program to termination. When the user changes the program, the system transitions into a new idle state representing the execution of a different program. This form of live programming embodies the *retroactive update* notion that we introduced in Chapter 2. Retroactive update has a subjunctive flavour: the environment allows the user to step *sidewise* in time, into an alternate execution where it is “as though” the program had been written differently from the outset. *Spreadsheets* are the most familiar live systems based on the retroactive paradigm. They are usually based on pure, first-order functional languages without recursion, although there are more general-purpose spreadsheet languages which bear a closer relationship to our work (§3.8, below). Version 6 of the commercial product *Mathematica* implemented a retroactive spreadsheet-like feature called “dynamic interactivity” [Wol91].

Retroactivity also arises in SAC (§3.10 below). Although SAC is technically not a form of live programming, because it only permits changes to *inputs*, it explores an important aspect of live programming, namely efficient update, via an algorithm called *change propagation*. The retroactivity appears in the correctness of change propagation, which is defined as consistency with a “from scratch” run of the program with the modified inputs. Finally, interactive programming, as proposed here, is a form of live programming with retroactive update. We explored interactive programming for a first-order functional language in earlier work, restricting the edits to changes to the values of program constants [Per10]. In this thesis, we extend this facility to arbitrary program edits and a higher-order language.

At the other end of the spectrum from retroactive update, we find live programming systems where we can “hot swap” new code into the running system and *continue execution*, with the new code only influencing subsequent behaviour. For example, game developers often make changes to game code whilst in an active game. It is usually impractical to then try to obtain an adjusted version of their present game state that

⁴ At the time of writing, Wikipedia prefers the term “interactive programming” for live programming, but to the best of our knowledge this term is not widely used either in the programming community, or in the research literature. To avoid confusion with our notion of “interactive programming”, we stick with conventional usage.

corresponds to a from-scratch run of the game under the new code. It might not even be possible if the change in behaviour they introduced would cause a large deviation in game state. But even when feasible, for rapid development and testing it is often more convenient to treat the part of the game being updated as an *open system* and the “computational past” of the game as part of its environment that it cannot change. In professional games development environments, this ability to “hot swap” is considered essential; we also see it in many general-purpose runtime environments. The HotSpot Java Virtual Machine allows definitions to be changed while execution is suspended at a breakpoint and then execution to be resumed under the new definitions [Dmi01]; the concurrent functional language Erlang has “hot code swapping” [Nau10]; many web development frameworks allow code to be modified and reloaded into the browser without the application having to be restarted.

This kind of live programming is also popular for interactive audio synthesis, music performance, and interactive graphics, using systems like Impromptu [SG10], based on Scheme, and the domain-specific audio/-music language SuperCollider [McC02]. In such settings, especially when the emphasis is on performance, the term *live coding* [CMRW03] or *on-the-fly programming* [WC04] is often preferred. (Used like this, “live” has the connotation of live performance, as well as the editing of a live program.) In live coding systems, a number of processes run concurrently, implementing music synthesisers, controllers, or animated graphics. The code for these processes is edited by the programmer, new definitions hot-swapped into the interpreter, and the ongoing behaviour of the system modified on the fly. The user’s interactions and the mutating program together form a hybrid computation that interweaves changes to the program with execution of the program in an informal and ad hoc way. This is fine – desirable, even – for creative applications and whenever the emphasis is on the process rather than the end product.

Moreover, in the presence of effects, from-scratch retroactive update may be downright impossible. The problem is when the running application we wish to update is a proper part of a larger computational system. In order for it to be “as though” the previous computation never happened, any effects emitted by that application must be *reversible*. For example in SAC, retroactivity in the presence of mutable state is supported by requiring that all mutating operations be *revocable* [ABW⁺10]. But almost by definition, the effectful interactions of an open system with other agents are typically irreversible. Writing output to a remote console, committing a financial transaction, and emitting a sound through a speaker are all examples of actions that cannot usually be revoked. One option is simply not to provide retroactivity, as per live coding. The risk is of reaching states or error conditions which would be unreachable under any from-scratch execution consistent with either version of the program.

On the other hand, many interactive applications can make use of something in between “whole program” retroactive update, which is feasible only under a closed-world assumption, and continuous/immediate update. They have a natural transactional unit of update, e.g. a user session, and can benefit from retroactive update at that granularity, before any effects are committed to the wider world. We discuss this possibility along with dynamic software updating (DSU, §3.11 below), where the goal is to perform a software upgrade without having to stop servicing user requests.

Finally, there has been a recent surge of interest in live programming environments for popular languages like Javascript, Clojure and Python, in response to an influential presentation by Bret Victor [Vic12a]. Al-

though to date there have only been prototypical implementations of these systems, their intended use cases relate them to the approach proposed in this thesis. In these systems, the traditional read-eval-print loop (REPL), where the user repeatedly types expressions at a prompt and the REPL answers with their values, is replaced by an interactive interface where the user *edits* an expression and sees its value update. As with LambdaCalc, the program and its output behave like a reactive, spreadsheet-like document. In a follow-up essay, Victor argues for the importance of being able to explore the internals of a computation, in addition to observing the updated output [Vic12b]. And indeed an important goal of interactive programming is to augment the conceptual model of live programming with explorable computations and explicit deltas.

A significant problem that these proposed live programming systems face is that update is achieved by *re-execution*. The problem is not performance, but again the interaction with effects. If the user changes their code so that an effect no longer happens, there is no way to “revoke” the effect that was emitted during the previous update, since it has already been committed to the outside world. Equally, any effects which are carried forward unchanged into the new state are re-executed during the update, which unless the effect happens to be idempotent, will generally be incorrect. In LambdaCalc, we do not treat computational effects, but in Future Work, §7.2.4, we sketch how our model might be extended with *reified effects* in order to avoid these problems.

3.8 Spreadsheet languages

Spreadsheets were early examples of live, visual programming environments. When a cell’s formula is modified by the user, the spreadsheet engine automatically updates the value of that cell and any dependent cells; a runtime error occurs if there are cycles in the dependency graph. Typical spreadsheet systems are not fully-fledged programming languages, lacking basic features like procedural abstraction and recursion. On the other hand, spreadsheets provide interactivity features that normal programming language implementations do not, such as liveness and easier access to intermediate values. In this thesis we argue that a spreadsheet-like model is a compelling implementation paradigm for pure functional languages, and show how to derive a spreadsheet-like programming system for a simple functional language, ignoring the question of efficient implementation. Here we review work that studies the opposite problem, namely adding functional programming capabilities to a spreadsheet system.

Burnett et al.’s Forms/3 spreadsheet language supports recursion, abstract data types and procedural abstraction [BAD⁺01]. In addition to these programming language features, Forms/3 also departs from the usual grid-like arrangement of cells found in spreadsheets and instead allows the user to arrange cells in a flexible visual configuration called a *form*. Because it also provides various built-in graphical data types such as geometric shapes and layout components, the user can visually arrange their code so that it also serves as a simple GUI for the user. Spreadsheets have always blurred the distinction between “programming” environment and “end-user” environment, but with this feature Forms/3 takes things a step further. (We discuss the potential of the spreadsheet paradigm for so-called *end-user programming* in §3.13 below.) Spreadsheet-based interactive graphics were also explored in Wilde and Lewis’s NoPumpG II and its predecessor NoPumpG [WL90].

De Hoon et al.’s FunSheet system retains the grid-like layout of traditional spreadsheets, but adds lazy, higher-order functions to the formula language [dHRvE95]. However, user-defined functions remain external to the spreadsheet and are treated as black boxes. Columns in the spreadsheet however are modelled as first-class functions which map indices to values; for example the formula `foldr (+) 0 (map D [5..7])`, where `D` names a column, computes the sum of cells D5 to D7. Clack and Braine also study the addition of lazy, higher-order functions to spreadsheets, among other features [CB97]. In their system, functions are parameterised spreadsheets, which improves on FunSheet in reusing the syntax of formulae and cells for definitions, and in providing easier access to intermediate values. Peyton Jones et al. add user-defined functions to Microsoft Excel, which also take the form of parameterised spreadsheets [Jon03].

One important difference between interactive programming and these spreadsheet systems is the level of transparency provided. In a spreadsheet, an intermediate value is visible only if the user has explicitly chosen to visualise it by associating a cell with that value. Otherwise, the user must manually extract the relevant sub-formula into a new cell and then mention the new cell in the formula of the original cell. In FunSheet, still more effort is involved because of the distinction between spreadsheet code and external functions. Either way, the user must refactor to debug, not dissimilar to having to add print statements to observe intermediate results in a language with imperative I/O.

In interactive programming, every intermediate value of the computation lives in its own “cell”, and cells and formulae are interwoven “all the way down”. Each invocation of a function instantiates its own live copy of the function body. The program can be debugged simply by revealing hidden structure, avoiding the need for intrusive refactoring. Moreover, the contents of a cell can include the contents of another cell by reference, an important step towards supporting structured data types, which are not considered by any spreadsheet systems we are aware of, except for Vital [Han02], a spreadsheet-like extension to Haskell discussed in §3.6 above. On the other hand, one feature of the spreadsheet paradigm not supported by our approach is the ability to use any cell as an input to another part of the computation, as long as no cycles are thereby introduced. This is possible in a spreadsheet because every cell has a unique name. In LambdaCalc, the normal naming and scoping rules of the reference language determine what can be mentioned in a given context.

We are not aware of any work which considers execution differencing in spreadsheets; efficient update has of course been studied, but is not relevant to our work. A method for debugging and slicing spreadsheets is presented informally by Reichwein et al. [RRB99], but not in a setting where cells can contain structured values.

3.9 Functional reactive programming

Functional reactive programming (FRP) is a dataflow programming model that uses the building blocks of functional programming. FRP grew out of Elliott and Hudak’s animation language Fran [EH97], and is suited to interactive applications involving time-indexed computation such as animation and robotics. There are now many variants of FRP, but most include notions of (continuous-time) *signals*, also called *behaviours*, which are values varying over continuous time, and *event signals*, values varying over discrete time. One

advantage of treating time continuously is temporal resolution-independence, allowing animations to be smoothly stretched or compressed. The inclusion of event signals allows FRP to model *hybrid systems*: systems that can both “flow” (exhibit continuity) and “jump” (exhibit discontinuity).

An FRP program is divided into two levels: a “functional” level consisting of pure functions, and a “reactive” level consisting of *signal functions*, which may be causal or pure. Signal functions map signals to signals; a signal function is *causal* if its output signal at time t may depend on the value of its input signal at times up to and including t . A signal function is *pure* if its output signal at t only depends on the value of its input signal at t . At the top level, an FRP program is reactive, consisting of signal functions written explicitly at the reactive level, composed with others obtained by lifting functions from the functional layer into pure signal functions.

There is some sort of connection between FRP and interactive programming. If we ignore the explicit delta information, then an interactive programming system can be thought of an FRP system where there is a single signal function, namely the interpreter itself, whose input and output signals carry values representing programs and their reified executions. Explicit notions of delta have not to our knowledge been considered formally in the context of FRP. Cooper and Krishnamurthi describe a change propagation algorithm for Scheme-based FRP [CK06], but do not show that their algorithm is correct or state what it would be for their algorithm to be correct. Sculthorpe and Nilsson also briefly mention change propagation in FRP, but do not provide any algorithms [SN10]. The notion of delta matters if one wants to represent the kind of structured changes required for interactive programming using FRP signals. A signal carrying a structured data type such as an abstract syntax tree (AST) does not describe *how* the AST is changing but only supplies the values it takes on at different times. We discuss incrementalising interactive programming in Future Work, §7.2.2.

3.10 Self-adjusting computation

The ability to make changes to a program and observe the resulting delta in its execution conceptually ties interactive programming to incremental computation. To be clear about what we mean by “incremental computation”, it is helpful to distinguish two related problems, given two executions T_1 and T_2 . The first problem is how to obtain a “useful” delta between T_1 and T_2 . Without further qualification, this problem is rather open-ended, and it is usually made more tractable by making additional assumptions which help constrain the notion of delta. Execution indexing (§3.12) investigates a number of these. Interactive programming also falls into this camp; in our case, the additional information we exploit is the *program delta* provided to the system by the user. How we utilise the program delta to obtain an execution delta is the topic of Chapter 6.

The second problem is the one we mentioned at the beginning of the chapter, and usually presupposes some kind of answer to the first question, namely some method for comparing T_1 and T_2 . It is the problem of obtaining T_2 from T_1 in time asymptotically equal to the size of the difference between T_2 and T_1 . We call this the problem of *incremental computation*. The challenge is to exploit the information in T_1 to efficiently calculate T_2 . This is a tough problem, and solving it has usually involved compromising on generality. Solving it in the general setting of interactive programming is non-trivial and we defer this entirely to future

work (§7.2.2).

Self-adjusting computation [Aca05, ABD07] is the most mature approach to incremental computation to date. The first time a SAC program is executed, the runtime records a trace identifying how parts of the computation depend on other parts. Subsequently, an input can be modified, and the output updated by a *change propagation* algorithm, which exploits the information in the trace to perform the update efficiently. Early formulations of SAC required the programmer to deal explicitly with much of the technical machinery involved in making a computation self-adjusting. Although the resulting programs were much easier to write than hand-coded dynamic algorithms, the approach was still difficult and error-prone. Later versions such as Δ ML [LWFA08] also require the programmer to encode the dependency structure of the program, but provide a type system which ensures that the programmer uses the encoding method correctly.

Both SAC and our system reify a computation into a data structure capturing the dependencies between parts of the computation. The main differences are in the extent and nature of the reification. In SAC, the programmer must explicitly identify the “changeable” aspects of the computation. The type constructor `mod` distinguishes *changeable data*, which may change after the initial evaluation, from *stable* data, which cannot. The special function type \rightarrow_C is the type of functions which produce and consume changeable data. SAC traces do not record the entire computation, but only the dependencies between “changeable” parts of the computation, and the code fragments associated with them. These code fragments are *re-executed* to synchronise the state of changeable computations when the modifiabiles they read have changed. For example, the following, taken from [CDHA11], shows a changeable function which computes $x^2 + y$:

```
squareplus: int * int mod  $\rightarrow_C$  int
squareplus (x, y) =
  let x2 = x * x in
    read y as y' in
      let r = x2 + y' in
        write(r)
```

This program is self-adjusting with respect to changes in y but not x , reflected by the fact that y has type `int mod`, but x just has type `int`. The `read` operation contains the code which depends on y . If we change the value of y , change propagation will re-execute the body of the `read`, but *reuse* the computation of x^2 . On the other hand, the following program is self-adjusting with respect to changes in x but not y :

```
squareplus: int mod * int  $\rightarrow_C$  int
squareplus (x, y) =
  let x2 = mod (read x as x' in write(x' * x')) in
    read x2 as x2' in
      let r = x2' + y in
        write(r)
```

We will not explain the details of the second example, other than to note that it is possible to avoid making $x2$ into a `mod` only by further restructuring. As these examples show, incrementalising even a trivial program

can involve significant effort and much rewriting if the programmer changes their mind about the parts that are to be considerable changeable.

An important recent development, *implicit SAC*, eliminates the need for explicit dependency encoding entirely [CDHA11]. Instead, the programmer writes general-purpose ML-like code, providing annotations on the main function specifying which inputs to the program are to be considered modifiable. The compiler then infers a self-adjusting version. In this form SAC is closest in concept to the work presented in this thesis, because the programmer writes what is essentially a normal functional program.

Nevertheless, there remain several important differences between our work and SAC in any of its forms. Most obvious is that we do not offer any form of efficient update, which is the primary concern of SAC. Another key difference is that partial reification ties self-adjusting computation to a hybrid execution model which interweaves change propagation with re-execution. The re-execution aspect interacts poorly with effects such as I/O and memory allocation, since effects may be re-executed during change propagation. Without some care, effects may be duplicated, and non-deterministic effects such as memory allocation may vary. *Memoising allocators* can be used to determinise allocation across re-executions, and *revocable* effects used to avoid duplication and to permit transitions to executions in which an effect no longer takes place [ABLW⁺10]. (The interplay between effects and update based on re-execution is also discussed in the context of live programming in §3.7 above.)

By contrast, in interactive programming, computations are fully reified into a persistent [DSST86] data structure. Reified computations are data structures “all the way down”, recording individual steps of the big-step semantics. The normal notion of execution has been replaced by *descriptions of executions*. In a language with effects, this would also be true of the effects: the computation would include *descriptions of effects*, but no actual effects will have taken place. This avoids any problems associated with re-execution of effects, but also invites the question of how our approach can accommodate “actual” I/O, such as controlling actuators or writing data to a file. We discuss this in Future Work, §7.2.4.

As well as being only “partial” in this sense, SAC traces are usually the result of executing a program that has been subjected to one or more intermediate translation phases, for example into continuation-passing style. Thus SAC traces are not particularly useful for human consumption, as they do not correspond directly to the evaluation of the source program. This is not a problem for SAC, where the goal is only efficient update.

Until recently SAC only supported “monotonic” trace reuse, which is when change propagation is only able to re-use chunks of trace if they occur in the new computation in the same order in which they occurred in the original computation. With monotonic change propagation, if nodes in the computation move around arbitrarily as a result of an update, reuse opportunities may be missed. In the worst case, no reuse at all may be possible and the update may take as long as a from-scratch run. Recent formal work on *non-monotonic* SAC [LWAB12] has lifted this restriction, although the approach has not yet been implemented. As discussed in §3.12 below, in our work, we calculate execution deltas that handle moves and other restructurings correctly, but we leave the development of efficient change propagation algorithms for Future Work, §7.2.2.

A final but crucial difference is that SAC only supports modification of data. Although this includes higher-order values, in other words functions, functional values are opaque, rather than represented as changeable data structures. With interactive programming, any part of the program can be modified and an

execution delta calculated corresponding to that modification. This makes our approach a suitable basis for an interactive programming environment, in which the user can make a new program by editing an existing running one, but also means that existing change propagation algorithms cannot be directly transferred to our setting.

3.11 Dynamic software updating

As we saw in §3.7 above, retroactive update relies on a closed-world assumption: that every past interaction falls within the scope of the update. When this assumption cannot be justified, we must pursue update solutions which are not so committed to reinventing the entire past. Dynamic software updating (DSU) seeks a compromise between the unrestricted chaos of live coding, and the impractical rigidity of global retroactive update. The goal is to balance flexibility – permitting as broad a class of updates as possible – with safety, which in early DSU work simply meant that subsequent execution would not fail with a runtime type error as a consequence of the update. We refer the reader to Hicks and Nettles for a survey of techniques [HN05]. Here we describe the Proteus system of Stoyle et al. [SHB⁺07] and its successor Proteus-tx by Neamtiu et al. [NHFP08], which we take to be roughly representative of the state of the art. One challenge facing DSU is that, while consistency with a from-scratch run is evidently too strong a property, merely guaranteeing that an update will not introduce a type error is too weak.

First, we review the version of DSU which only guarantees type safety. Then we discuss its shortcomings, with an example, and consider an improved approach and its connection to retroactive update. In the Proteus approach to DSU, the user first specifies a collection of modifications called an *update* which they intend to apply to a running system. An update is a set of new bindings for selected global symbols, with no fine-grained information about how they changed. (This is in contrast with interactive programming, where changes are syntactic deltas.) DSU also permits data type definitions to change, which we do not support. The update must then include a “migration” function which will be used, when the update is applied, to transform values created under the old data type definition to the new one.

An *update point* is a static program location specified by the user. At runtime, if an update is pending and an update point is reached for which the update is deemed type-safe, the update is performed and then execution resumed. Otherwise execution continues as normal. One thing which distinguishes DSU from retroactive update is that invocations of functions which are active when the update is applied will finish executing with the old version of the code, and only subsequent invocations will use the new code; and of course none of the effects of prior invocations will be retroactively amended to reflect the changes. For this mixture of old and new behaviour to be type-safe, DSU relies on a property called “con-freeness”. Intuitively, the idea behind con-freeness (for an update to a data type definition, for example) is that old code that will resume after the update will not *concretely* manipulate values of that type. Accessing a field or pattern-matching are examples of concrete usage; simply mentioning a value of that type, without relying on its representation, does not compromise con-freeness. Notions of con-freeness are also defined for function and variable update, and then the authors give an algorithm which statically approximates con-freeness for a given update site.

This form of DSU ensures that updating is sound from a typing point of view. However it does not rule out other runtime errors which would not arise under LambdaCalc-style retroactive update, but only under a hybrid execution of old and new code. Figure 3.6 shows an example from Neamtiu et al. [NHFP08]. Suppose the program on the left is edited into the program on the right by moving the call of `g` in `h` into the body of `f`. For the sake of simplicity also suppose that this is the only call of `f` in the program. Now, if we dynamically update the program during an invocation of `h`, just before `h` makes the call to `f` (at the point indicated in the figure), then that call to `f` will invoke the new version of `f`, which will call `g`. Then, the old definition of `h` will finish executing, resulting in `g` being called twice, even though this is not possible under either version of the program alone. If `g` has side-effects, this could be catastrophic, and moreover such hybrid executions are very hard to reason about. DSU in this form is nothing more than a type-safe version of live coding.

| | |
|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| <pre> proc f () = ... proc h () = ... // update point f(); g(); </pre> | <pre> proc f () = ... g(); proc h () = ... // update point f(); </pre> |
|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|

Figure 3.6 Type-safe but potentially incorrect update via DSU

This observation prompted Neamtiu et al. to develop the *transactional version consistency* approach to DSU [ibid.]. The idea of transactional version consistency is to allow the user to designate blocks of code as *transactions* whose execution will always be attributable to a single version of the program. They use a *contextual effect system* which, for any expression, statically computes approximations of the effect of the computation that has already taken place (the *prior effect*), and of the effect that has yet to take place (the *future effect*). An update is permitted during the execution of a transaction if it will be “as though” that transaction had run under the new code from the outset, *or* under the old code from the outset, but not under a hybrid of the two, using the prior and future effects of the update point to decide this conservatively. If neither situation obtains, then either the update or the transaction must be rejected.

This transactional approach is quite effective for so-called “long-running” applications, such as online retail websites, which commit many transactions to a database. In practice, these systems are actually structured around relatively short-lived, concurrently executing user sessions. The long-living parts of the application are the *external effects* of code – changes to data, and so on. Each session either aborts or executes to completion, committing changes to a shared persistent store. For these systems, an update semantically equivalent to re-running all past transactions with the new code against an empty database might not be feasible to execute for performance reasons or more likely because the business model does not permit “changing the past”. For such applications, transactional version consistency is at least the basis of a more

structured approach to DSU, although the possibility remains of things going wrong for business logic that extends over multiple transactions.

Stoyle et al. attribute the difficulties with DSU which transactional version consistency addresses to the *flexibility of being able to change the program in the future* [SHB⁺07]. But from the vantage point of a system like LambdaCalc, they can equally be attributed to the *inflexibility of being unable to change the program in the past*. Indeed, it seems that transactional version consistency is a tacit acknowledgment of the need, even in long-running systems, for some of the semantic consistency of retroactive update. In Future Work, §7.2.2, we discuss an approach to DSU based on transaction-level retroactive update, rather than transactional version consistency. The idea is to use a more “local” version of retroactive update, for which the semantic correctness criterion is consistency with a from-scratch run of an individual transaction or session, not of an all-encompassing closed program.

3.12 Execution indexing

There are many reasons for wanting to compare executions. Typical applications include regression testing, debugging, and program comprehension. For example, Zeller’s iterative method for computing so-called *cause transitions* involves comparing two executions, a failing one and a successful one that closely resembles it [Zel02]. To compare executions, we are normally faced with a crude mixture of mental simulation, print statements, unit tests, and single-step or breakpoint debugging to identify the points at which the executions diverge. With interactive programming, the consequences of changes on the execution are always explicitly visible. Our emphasis is not on dynamic analyses or procedures for bug diagnosis, but on making the mechanism underlying a computation more immediate and tangible, in part through explicit deltas.

As mentioned in §3.10 above, for comparing executions to be a tractable problem, we require an *indexing scheme* which identifies, or “aligns”, steps in the executions being compared. When two steps align, they are considered, for differencing purposes, to be each other’s counterpart in their respective executions. An indexing scheme indicates that two steps from different computations align by assigning them the same index. A indexing scheme must be injective within a single execution, as noted by Xin et al.: distinct sub-traces of that execution must be assigned distinct indices [XSZ08]. The *structural indexing* approach Xin et al. propose assigns indices by building a trace of the execution and then, for every point in the execution, setting its index to be the path of that point in the trace. Sumner and Zhang [SZ10] extend Xin et al.’s scheme with an indexing scheme for heap-allocated memory locations, allowing pointer-based data structures to be compared in the presence of aliasing. Both papers take an approach to tracing which is a somewhat ad hoc, rather than semantics-directed, but a more serious problem is that steps can only be aligned if their ancestors in the trace are also aligned. This rules out deltas that splice new structure around existing sub-computations, including many of the examples we considered in Chapter 2.

The execution indexing scheme that we present in Chapter 6 is simple but to the best of our knowledge novel. We start by assuming that an indexing scheme has already been provided for the input program. This is easily justified by thinking of the program as represented in a store assigning a unique address to every sub-expression, and then using the addresses as the indices. We then use these program indices to

compute indices for each step in the execution of that program. We ensure that the graph structure implied by the indexing is preserved into the execution: so that, for example, splicing code into a function body at a certain point in the expression will cause the execution of that code to be spliced into all executions of that function body at the corresponding point in the trace. This means that arbitrary moves and other structural re-arrangements are translated homomorphically into rearrangements of the trace. The upshot for the user is that trace deltas are easily attributed to program deltas. (The same is not true of the *values* associated with each step of the computation; these are not homomorphically related to the program.)

Johnson et al. combine execution indexing and execution differencing with slicing, to reduce the size of the deltas [JCC⁺11]. They call their approach *differential slicing*; this is different from the technique that we call “differential slicing” in Chapter 5, in that their differential slices compare distinct executions. Our technique only compares two slices of the same execution.

3.13 End-user programming

We usually think of programming as an inherently complex activity entrusted to the care of skilled engineers. A goal of programming language research is to find ways to tame this complexity and make programming easier. The field of *end-user programming* comes at things from the other direction, aiming to bring to end-users and non-experts some of the power of programmers. Interactive programming puts “programming” and “using” on a continuum: programming is the manipulation of a running application in order to change its behaviour, and every user, in principle at least, has access to this capability. In practice, not every end-user will have the inclination, or authority perhaps, to make fundamental changes to the logic of an application, but no part of the application is fundamentally hidden or immutable. (We discuss the importance of access control in Future Work, §7.2.3.)

End-user programming is a broad and heterogeneous topic which we will only briefly summarise, because the connection to interactive programming is mostly conceptual. The first end-user programming systems were spreadsheets (§3.8, above), which brought some of the power of programming to business users with accounting rather than programming expertise. Spreadsheets and other office applications offer further end-user customisation and automation via *macros* which the user creates by “recording” an interaction they have with the GUI into a scripting language like Microsoft’s Visual Basic for Applications [Sep00].

More generally, if the system supports *programming by example*, the topic of Halbert’s PhD thesis [Hal84], then it may be able to extract a simple program from an exemplar performance of a task provided by the user. The influential 1984 Tinker system of Lieberman and Hewitt [LH80] had an interesting interactive approach to programming by example. The user built new procedures by working out individual steps of the procedure in concrete situations. Tinker displayed the value computed by an individual step as it was performed; when this value was then used in a subsequent step, the code associated with the computation of that value would automatically be incorporated into the new step. Edwards’ *example-centric programming* is a more recent incarnation of the idea [Edw04]. Our approach to interactive programming may suit this style of programming by example, because of the way we attribute the parts of a computed value to individual steps of the computation and always situate the user at a concrete execution of a function, rather than at a

static function definition.

Truchard and Kodosky’s 1987 LabVIEW system brought the convenience of spreadsheets to scientists and engineers [WT96]. Their system allowed a user with domain expertise, but not necessarily programming expertise, to build configurations of “virtual instruments”, using a graphical dataflow language extended with a looping construct. Both spreadsheets and LabVIEW are examples of the importance of *domain-specific languages* to end-user programming. Visual programming languages (§3.6, above) also commonly feature in end-user programming solutions because they can easily be customised into domain-specific programming interfaces. Repenning’s AgentSheets is a spreadsheet-like visual programming system which allows non-expert users to build interactive, web-based simulations [Rep93].

The most well-known form of web-based end-user programming is the *mashup*. A mashup is a web application, consisting mostly of plumbing, built out of existing web services, web forms and online data sources. Mashups can do things like retrieving data from various sources, processing it, and publishing the result as a feed; or adding and removing various widgets from a website to change its appearance and then performing some data entry on one of its forms. Some recent mashup development environments are surveyed by Grammel and Storey [GS10]. One problem they identify is that mashup tools often lack support for basic programming activities like testing and debugging.

Shortcomings like these are, we believe, an inevitable consequence of treating end-user programming as a second-class activity distinct from “real programming”. By contrast, interactive programming puts “programming” and “using” on a continuum. This has the potential to transform how we approach end-user programming. One half of the story is that programmers themselves would often welcome being able to move seamlessly between the roles of programmer and end-user. A programmer unfamiliar with a language or application domain should be able to learn about it by playing with an existing application, interactively disassembling it to understand how it works, eventually changing it and adding new features. *Programming often starts as using*. Equally, testing is a form of using. When something goes wrong, the programmer should be able to diagnose and fix the problem “in situ”, without having to restart the test case in a debugger. The other half of the story is that an end-user unfamiliar with programming should be able to progressively discover details about the inner workings of an application. As they gain confidence, they can experiment with simple customisations, and eventually move into task automation and full-blown development. *A user should be able to grow into a programmer*.

4 Reifying Computation

In interactive programming, computations are “self-explaining”, meaning they can be opened up and interactively explored. Execution is *reified*: the activity of execution has been transformed into a description of that activity. This process of taking a dynamic process and transcribing its behaviour into a static record is conventionally called *tracing*.

The term “tracing” does not convey quite the end-user intuition we have in mind. “Tracing” over-emphasises the dynamic process – a trace must be a trace *of* something. We would prefer the user to think of the spatially extended structure *as* the execution. But for the sake of clarity, we adopt the terminology of traces in this thesis, although we also sometimes treat “computation” and “reified computation” as synonymous.

As discussed in Related Work (§3.1 above), tracing is widely used for debugging and other offline dynamic analyses. There are several forms a trace might take depending on the semantics of the language in question. A trace can be built by instrumenting the program to be traced or by instrumenting the interpreter in which it runs. Once obtained, an execution trace can be explored as a static structure or *replayed* to recover the original dynamic behaviour. For LambdaCalc, we use a big-step reference semantics (§4.1); traces take the approximate form of big-step derivation trees (§4.2); and we build them using a tracing interpreter (§4.3). Our approach to tracing is not particularly novel but is the foundation for Chapters 5 and 6.

| | |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Types | $\tau ::= 1 \mid b \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \mu\alpha.\tau \mid \alpha$ |
| Variable contexts | $\Gamma ::= \bullet \mid \Gamma, x : \tau$ |
| Expressions | $e ::= x \mid () \mid c \mid e_1 \oplus e_2 \mid \mathbf{fun} \ f(x).e \mid e_1 \ e_2$
$\mid (e_1, e_2) \mid \mathbf{fst} \ e \mid \mathbf{snd} \ e \mid \mathbf{inl} \ e \mid \mathbf{inr} \ e$
$\mid \mathbf{case} \ e \ \mathbf{of} \ \{\mathbf{inl}(x_1).e_1; \mathbf{inr}(x_2).e_2\} \mid$
$\mid \mathbf{roll} \ e \mid \mathbf{unroll} \ e$ |
| Values | $v ::= c \mid (v_1, v_2) \mid \mathbf{inl} \ v \mid \mathbf{inr} \ v \mid \mathbf{roll} \ v$
$\mid \langle \rho, \mathbf{fun} \ f(x).e \rangle$ |
| Environments | $\rho ::= \bullet \mid \rho[x \mapsto v]$ |

Figure 4.1 Reference language: abstract syntax

4.1 Reference language

First we introduce a typed, call-by-value language with familiar functional programming constructs such as recursive types and higher-order functions. We refer to this as the *reference language*, since it forms the setting for the rest of the thesis. The reference language is idealised compared to the concrete language with data types implemented in LambdaCalc, but the latter easily desugars into it. We defer as future work extending our approach to other language features, e.g. state (§7.2.4) and concurrency (§7.2.3).

The syntax of the reference language is given in Figure 4.1. Types include the usual unit, sum, product and function types, plus iso-recursive types $\mu\alpha.\tau$, type variables α , and primitive types b . Variable contexts are defined inductively as either the empty context \bullet or the extension $\Gamma, x : \tau$ of a context Γ with a binding from x to τ , hiding any existing bindings for x in Γ . Expressions include the unit value $()$, standard introduction and elimination forms for projection products, sums and recursive functions, `roll` and `unroll` forms for recursive types, primitive constants c , and applications $e_1 \oplus e_2$ of primitive operations. The typing judgements $\Gamma \vdash e : \tau$ for expressions and $\Gamma \vdash \rho$ for environments are given in Figure 4.2; the latter means that ρ is a well-formed environment for Γ . The signature Σ assigns to every primitive constant c the primitive type $c : b \in \Sigma$, and to every primitive operation \oplus the argument types and return type $\oplus : b_1 \times b_2 \rightarrow b \in \Sigma$.

Evaluation for the reference language is given by a conventional call-by-value big-step semantics, shown in Figure 4.3. The judgement $\rho, e \Downarrow_{\text{ref}} v$ states that expression e in closing environment ρ evaluates to value v . Values include the usual forms, plus closures $\langle \rho, \text{fun } f(x).e \rangle$. The choice of an environment-based semantics is deliberate: environments will be helpful later when we want to record an execution as an unrolling of the program syntax, by allowing us to retain variable names in traces. As usual $\hat{\oplus}$ means \oplus suitably interpreted in the meta-language.

Evaluation is deterministic and type-preserving. We omit the proofs, which are straightforward inductions.

Lemma 1 (Type preservation for \Downarrow_{ref}). *If $\Gamma \vdash e : \tau$ and $\Gamma \vdash \rho$ and $\rho, e \Downarrow_{\text{ref}} v$ then $\vdash v : \tau$.*

Lemma 2 (Determinism of \Downarrow_{ref}). *If $\rho, e \Downarrow_{\text{ref}} v$ and $\rho, e \Downarrow_{\text{ref}} v'$ then $v = v'$.*

As mentioned, the language used for our examples in Chapter 2 can be easily desugared into the reference language. Named data types desugar into anonymous recursive sums-of-products, and case expressions and constructor calls desugar into nested elimination and introduction forms for the corresponding desugared types. Although the desugaring is straightforward, an implementation which relied on it would produce traces in the reference language, which would then have to be “resugared” for presentation to the user. In the setting of the differential evaluation introduced in Chapter 2, §2.2 and described in more detail in Chapter 6, the desugaring and resugaring would also have to operate differentially. This is beyond the scope of the present work, and so for LambdaCalc we implement the sugared language directly.

4.2 Syntax all the way down

One of the key benefits of interactive programming is that the programmer can move smoothly from an extensional view of a computation, as a program paired with a value, to progressively more intensional

$\boxed{\Gamma \vdash e : \tau}$

$$\begin{array}{c}
\frac{}{\Gamma \vdash () : 1} \quad \frac{}{\Gamma \vdash x : \tau} \quad x : \tau \in \Gamma \quad \frac{}{\Gamma \vdash c : b} \quad c : b \in \Sigma \quad \frac{\Gamma \vdash e_1 : b_1 \quad \Gamma \vdash e_2 : b_2}{\Gamma \vdash e_1 \oplus e_2 : b} \quad \oplus : b_1 \times b_2 \rightarrow b \in \Sigma \\
\\
\frac{\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathbf{fun} f(x).e : \tau_1 \rightarrow \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \\
\\
\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{fst} e : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{snd} e : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathbf{inl} e : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathbf{inr} e : \tau_1 + \tau_2} \\
\\
\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathbf{case} e \text{ of } \{\mathbf{inl}(x_1).e_1; \mathbf{inr}(x_2).e_2\} : \tau} \quad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \mathbf{unroll} e : \tau[\mu\alpha.\tau/\alpha]} \\
\\
\frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \mathbf{roll} e : \mu\alpha.\tau}
\end{array}$$

$\boxed{\Gamma \vdash \rho}$

$$\frac{}{\bullet \vdash \bullet} \quad \frac{\Gamma \vdash \rho \quad \vdash v : \tau}{\Gamma, x : \tau \vdash \rho[x \mapsto v]}$$

$\boxed{\vdash v : \tau}$

$$\begin{array}{c}
\frac{}{\vdash () : 1} \quad \frac{}{\vdash c : b} \quad c : b \in \Sigma \quad \frac{\Gamma \vdash \rho \quad \Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e : \tau_2}{\vdash \langle \rho, \mathbf{fun} f(x).e \rangle : \tau_1 \rightarrow \tau_2} \quad \frac{\vdash v_1 : \tau_1 \quad \vdash v_2 : \tau_2}{\vdash (v_1, v_2) : \tau_1 \times \tau_2} \\
\\
\frac{\vdash v : \tau_1}{\vdash \mathbf{inl} v : \tau_1 + \tau_2} \quad \frac{\vdash v : \tau_2}{\vdash \mathbf{inr} v : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash v : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \mathbf{roll} v : \mu\alpha.\tau}
\end{array}$$

Figure 4.2 Reference language: typing judgements

$$\boxed{\rho, e \Downarrow_{\text{ref}} v}$$

$$\begin{array}{c}
\frac{}{\rho, x \Downarrow_{\text{ref}} \rho(x)} \quad \frac{}{\rho, c \Downarrow_{\text{ref}} c} \quad \frac{\rho, e_1 \Downarrow_{\text{ref}} c_1 \quad \rho, e_2 \Downarrow_{\text{ref}} c_2}{\rho, e_1 \oplus e_2 \Downarrow_{\text{ref}} c_1 \hat{\oplus} c_2} \quad \frac{}{\rho, \text{fun } f(x).e \Downarrow_{\text{ref}} \langle \rho, \text{fun } f(x).e \rangle} \\
\\
\frac{\rho, e_1 \Downarrow_{\text{ref}} v_1 \quad \rho, e_2 \Downarrow_{\text{ref}} v_2 \quad \rho'[f \mapsto v_1][x \mapsto v_2], e \Downarrow_{\text{ref}} v}{\rho, e_1 e_2 \Downarrow_{\text{ref}} v} v_1 = \langle \rho', \text{fun } f(x).e \rangle \\
\\
\frac{\rho, e_1 \Downarrow_{\text{ref}} v_1 \quad \rho, e_2 \Downarrow_{\text{ref}} v_2}{\rho, (e_1, e_2) \Downarrow_{\text{ref}} (v_1, v_2)} \quad \frac{\rho, e \Downarrow_{\text{ref}} (v_1, v_2)}{\rho, \text{fst } e \Downarrow_{\text{ref}} v_1} \quad \frac{\rho, e \Downarrow_{\text{ref}} (v_1, v_2)}{\rho, \text{snd } e \Downarrow_{\text{ref}} v_2} \quad \frac{\rho, e \Downarrow_{\text{ref}} v}{\rho, \text{inl } e \Downarrow_{\text{ref}} \text{inl } v} \\
\\
\frac{\rho, e \Downarrow_{\text{ref}} v}{\rho, \text{inr } e \Downarrow_{\text{ref}} \text{inr } v} \quad \frac{\rho, e \Downarrow_{\text{ref}} \text{inl } v_1 \quad \rho[x_1 \mapsto v_1], e_1 \Downarrow_{\text{ref}} v}{\rho, \text{case } e \text{ of } \{\text{inl}(x_1).e_1; \text{inr}(x_2).e_2\} \Downarrow_{\text{ref}} v} \\
\\
\frac{\rho, e \Downarrow_{\text{ref}} \text{inr } v_2 \quad \rho[x_2 \mapsto v_2], e_2 \Downarrow_{\text{ref}} v}{\rho, \text{case } e \text{ of } \{\text{inl}(x_1).e_1; \text{inr}(x_2).e_2\} \Downarrow_{\text{ref}} v} \quad \frac{\rho, e \Downarrow_{\text{ref}} v}{\rho, \text{roll } e \Downarrow_{\text{ref}} \text{roll } v} \quad \frac{\rho, e \Downarrow_{\text{ref}} \text{roll } v}{\rho, \text{unroll } e \Downarrow_{\text{ref}} v}
\end{array}$$

Figure 4.3 Reference language: call-by-value evaluation

$$\begin{array}{ll}
\text{Closure variables} & \gamma ::= \bullet \mid \gamma, x \\
\text{Traces} & T ::= x \mid c \mid T_1 \oplus_{c_1, c_2} T_2 \mid (T_1, T_2) \\
& \mid \text{fst } T \mid \text{snd } T \mid \text{inl } T \mid \text{inr } T \\
& \mid \text{case } T \text{ of } \{\text{inl}(x_1).T_1; \text{inr}(x_2).e_2\} \\
& \mid \text{case } T \text{ of } \{\text{inl}(x_1).e_1; \text{inr}(x_2).T_2\} \\
& \mid \text{fun } f(x).e \mid T_1 T_2 \triangleright \langle \gamma, \text{fun } f(x).T \rangle \\
& \mid \text{roll } T \mid \text{unroll } T
\end{array}$$

Figure 4.4 Syntax of traces

views which expose the innards of the computation. The user manipulates the view to suit a particular comprehension task. The computation is presented to the user as an unrolled expression: execution is “syntax all the way down”. The user can therefore understand an execution by relying mainly on the concepts they already use to think about programs.

Figure 4.4 gives the abstract syntax of traces; the typing rules are given in Figure 4.5. The judgement $\Gamma \vdash T : \tau$ states that T can be assigned type τ in Γ . Traces closely mirror expressions, which allows us to represent traces as unrolled expressions. A reasonable intuition is that the trace recording the execution of an expression is that same expression, but with the traces of any function calls inlined recursively into their call sites. Although well-typedness alone is insufficient to ensure this, if $\Gamma \vdash T : \tau$ then T describes the computation of a value of type τ .

The typing rules are mostly self-explanatory, but the primitive operation trace $T_1 \oplus_{c_1, c_2} T_2$ and the application trace $T_1 T_2 \triangleright \langle \gamma, \text{fun } f(x).T \rangle$, where γ is a list of identifiers, probably look somewhat mysterious. The trace of a primitive operation $T_1 \oplus_{c_1, c_2} T_2$ records not only the evaluation of the operands, but also their values c_1 and c_2 ; these will be required later for slicing. To type such a trace, we type T_1 and T_2 as normal, and then require that the type of each value matches the type of the corresponding trace.

The trace of a function application $T_1 T_2 \triangleright \langle \gamma, \text{fun } f(x).T \rangle$ records not only the evaluation of the closure and argument T_1 and T_2 , but also the evaluation T of the closure body. To type such a trace in Γ , we type T_1 and T_2 in Γ , but type T in Γ' extended with bindings for f and x . The role of the Γ' is to enumerate the variables which may be used by the closure body from the lexical context in which it was defined. The reason that Γ' is unrelated to Γ is that the closure is computed dynamically. So that the overall application trace is closed by Γ , we use an auxiliary function $\text{vars}(-)$ to strip the type information from Γ' , leaving just its variables γ , which we bundle with T into a closure-like trace form $\langle \gamma, \text{fun } f(x).T \rangle$. Storing $\text{vars}(\Gamma')$ rather than Γ' avoids carrying type information in the trace and, in the tracing evaluation rules, having to derive typing contexts at run-time.

A minor detail is that our LambdaCalc implementation associates traces with the values they compute, so that they can be displayed to the user. It is safe to omit these value annotations from the formalism. In particular, they are required neither for slicing (Chapter 5) nor for differencing (Chapter 6).

4.3 Tracing semantics

We now define a *tracing semantics* for the reference language just presented. The rules, given in Figure 4.6, are identical to those for \Downarrow_{ref} , except that they construct a trace as well as a value. A terminating tracing evaluation for an expression $\Gamma \vdash e : \tau$ in environment ρ for Γ , written $\rho, e \Downarrow v, T$, yields both a value $v : \tau$ and a trace $\Gamma \vdash T : \tau$ describing how v was computed. Before explaining the tracing semantics, we dispense with a few preliminary properties. Where the proofs are straightforward inductions or closely analogous to those for the reference language they are omitted. First, tracing evaluation is deterministic.

Lemma 3 (Determinism of \Downarrow). *If $\rho, e \Downarrow v, T$ and $\rho, e \Downarrow v', T'$ then $v = v'$ and $T = T'$.*

The tracing semantics and the reference semantics agree on values.

$$\boxed{\Gamma \vdash T : \tau}$$

$$\begin{array}{c}
\frac{}{\Gamma \vdash () : 1} \quad \frac{}{\Gamma \vdash x : \tau} \quad x : \tau \in \Gamma \quad \frac{}{\Gamma \vdash c : b} \quad c : b \in \Sigma \\
\\
\frac{\Gamma \vdash T_1 : b_1 \quad \vdash c_1 : b_1 \quad \Gamma \vdash T_2 : b_2 \quad \vdash c_2 : b_2}{\Gamma \vdash T_1 \oplus_{c_1, c_2} T_2 : b} \quad \oplus : b_1 \times b_2 \rightarrow b \in \Sigma \quad \frac{\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun } f(x).e : \tau_1 \rightarrow \tau_2} \\
\\
\frac{\Gamma \vdash T_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash T_2 : \tau_1 \quad \Gamma', f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash T : \tau_2}{\Gamma \vdash T_1 T_2 \triangleright \langle \text{vars}(\Gamma'), \text{fun } f(x).T \rangle : \tau_2} \quad \frac{\Gamma \vdash T_1 : \tau_1 \quad \Gamma \vdash T_2 : \tau_2}{\Gamma \vdash (T_1, T_2) : \tau_1 \times \tau_2} \\
\\
\frac{\Gamma \vdash T : \tau_1 \times \tau_2}{\Gamma \vdash \text{fst } T : \tau_1} \quad \frac{\Gamma \vdash T : \tau_1 \times \tau_2}{\Gamma \vdash \text{snd } T : \tau_2} \quad \frac{\Gamma \vdash T : \tau_1}{\Gamma \vdash \text{inl } T : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash T : \tau_2}{\Gamma \vdash \text{inr } T : \tau_1 + \tau_2} \\
\\
\frac{\Gamma \vdash T : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash T_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{case } T \text{ of } \{\text{inl}(x_1).T_1; \text{inr}(x_2).e_2\} : \tau} \\
\\
\frac{\Gamma \vdash T : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash T_2 : \tau}{\Gamma \vdash \text{case } T \text{ of } \{\text{inl}(x_1).e_1; \text{inr}(x_2).T_2\} : \tau} \quad \frac{\Gamma \vdash T : \mu\alpha.\tau}{\Gamma \vdash \text{unroll } T : \tau[\mu\alpha.\tau/\alpha]} \\
\\
\frac{\Gamma \vdash T : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \text{roll } T : \mu\alpha.\tau}
\end{array}$$

Figure 4.5 Typing rules for traces

Theorem 1. $\rho, e \Downarrow_{\text{ref}} v \iff \exists T. \rho, e \Downarrow v, T$

Tracing evaluation is type-preserving.

Lemma 4 (Type preservation for \Downarrow). *If $\Gamma \vdash e : \tau$ and $\Gamma \vdash \rho$ with $\rho, e \Downarrow v, T$, then $\vdash v : \tau$ and $\Gamma \vdash T : \tau$.*

We can now explain the tracing evaluation judgement. The idea is that tracing evaluation equips every value with a trace which “explains” it. The trace of a variable x is just the corresponding trace form x ; as stated in Lemma 4, the trace of $\Gamma \vdash e : \tau$ is not closed but is instead also typed in Γ . The traces of other nullary expressions are just the corresponding nullary trace form. For non-nullary forms, such as projections, pairs, and case expressions, the general pattern is to produce a trace which looks like the original expression except that any executed sub-expressions have been inflated into their traces. For example a trace of the form $\text{case } T \text{ of } \{\text{inl}(x_1).T_1; \text{inr}(x_2).e_2\}$ records the scrutinee and the taken branch unrolled into their respective executions T and T_1 . The non-taken branch e_2 is kept in the trace to be consistent with our notion of a trace as an unrolled expression.

Tracing a primitive operation $e_1 \oplus e_2$ records not only the traces T_1 and T_2 of the operands, but also their values c_1 and c_2 . As mentioned earlier, this anticipates the backward-slicing technique presented in the next chapter (§5.3), which relies on being able to back-propagate neededness information through each step in the

$$\boxed{\rho, e \Downarrow v, T}$$

$$\begin{array}{c}
\frac{}{\rho, x \Downarrow \rho(x), x} \qquad \frac{}{\rho, c \Downarrow c, c} \qquad \frac{\rho, e_1 \Downarrow c_1, T_1 \quad \rho, e_2 \Downarrow c_2, T_2}{\rho, e_1 \oplus e_2 \Downarrow c_1 \hat{\oplus} c_2, T_1 \oplus_{c_1, c_2} T_2} \\
\\
\frac{}{\rho, \mathbf{fun} f(x).e \Downarrow \langle \rho, \mathbf{fun} f(x).e \rangle, \mathbf{fun} f(x).e} \\
\\
\frac{\rho, e_1 \Downarrow v_1, T_1 \quad \rho, e_2 \Downarrow v_2, T_2 \quad \rho'[f \mapsto v_1][x \mapsto v_2], e \Downarrow v, T}{\rho, e_1 \ e_2 \Downarrow v, T_1 \ T_2 \triangleright \langle \mathbf{vars}(\rho'), \mathbf{fun} f(x).T \rangle} \ v_1 = \langle \rho', \mathbf{fun} f(x).e \rangle \\
\\
\frac{\rho, e_1 \Downarrow v_1, T_1 \quad \rho, e_2 \Downarrow v_2, T_2}{\rho, (e_1, e_2) \Downarrow (v_1, v_2), (T_1, T_2)} \qquad \frac{\rho, e \Downarrow (v_1, v_2), T}{\rho, \mathbf{fst} e \Downarrow v_1, \mathbf{fst} T} \qquad \frac{\rho, e \Downarrow (v_1, v_2), T}{\rho, \mathbf{snd} e \Downarrow v_2, \mathbf{snd} T} \\
\\
\frac{\rho, e \Downarrow v, T}{\rho, \mathbf{inl} e \Downarrow \mathbf{inl} v, \mathbf{inl} T} \qquad \frac{\rho, e \Downarrow v, T}{\rho, \mathbf{inr} e \Downarrow \mathbf{inr} v, \mathbf{inr} T} \\
\\
\frac{\rho, e \Downarrow \mathbf{inl} v_1, T \quad \rho[x_1 \mapsto v_1], e_1 \Downarrow v, T_1}{\rho, \mathbf{case} e \mathbf{of} \{ \mathbf{inl}(x_1).e_1; \mathbf{inr}(x_2).e_2 \} \Downarrow \quad v, \mathbf{case} T \mathbf{of} \{ \mathbf{inl}(x_1).T_1; \mathbf{inr}(x_2).T_2 \}} \\
\\
\frac{\rho, e \Downarrow \mathbf{inr} v_2, T \quad \rho[x_2 \mapsto v_2], e_2 \Downarrow v, T_2}{\rho, \mathbf{case} e \mathbf{of} \{ \mathbf{inl}(x_1).e_1; \mathbf{inr}(x_2).e_2 \} \Downarrow \quad v, \mathbf{case} T \mathbf{of} \{ \mathbf{inl}(x_1).e_1; \mathbf{inr}(x_2).T_2 \}} \\
\\
\frac{\rho, e \Downarrow v, T}{\rho, \mathbf{roll} e \Downarrow \mathbf{roll} v, \mathbf{roll} T} \qquad \frac{\rho, e \Downarrow \mathbf{roll} v, T}{\rho, \mathbf{unroll} e \Downarrow v, \mathbf{unroll} T}
\end{array}$$

Figure 4.6 Tracing semantics: call-by-value evaluation

computation. The value annotations are used in the slicing of primitive operations, which are not amenable to this propagation technique.

Aside from the values of primitive operands, a trace has more content than its original expression only where functions are called. Tracing an application $e_1 \ e_2$ yields the application trace $T_1 \ T_2 \triangleright \langle \text{vars}(\rho'), \text{fun } f(x).T \rangle$, where T_1 and T_2 record the evaluation of e_1 and e_2 , and T records the evaluation of the body of the closure $v_1 = \langle \rho', \text{fun } f(x).e \rangle$. Here, $\text{vars}(-)$ is overloaded to mean the function which discards the value bindings from ρ' , leaving only its variables. This ensures that the overall application trace is well-typed. A practical implementation might retain the values for visualisation purposes; at present LambdaCalc does not visualise closure environments.

In Related Work (§3.1), we proposed that traces should record the behaviour of a program according to a specific operational model. This suggests that a correctness criterion for our tracing semantics would relate traces to derivation trees in the reference semantics. However, we take a different approach which involves formalising our notion of a trace *explaining* a value. Our tracing semantics is correct in that it yields values equipped with traces which do indeed explain them in this technical sense (Theorem 4, §5.3.2). The formal notion of “explanation” is fundamentally tied to slicing, and so we defer the statement and proof of this theorem to the next chapter.

5 Slicing Computation

Computations can usually be broken down into parallel execution flows that are at least somewhat independent, regardless of whether implementations actually take advantage of this. These parallel strands of execution are called “slices” because they cut vertically through the dependency structure of the computation. Being able to view slices interactively allows a programmer to explore the relationship between parts of the output and parts of the program, as we saw in Chapter 2, §§2.5 and 2.6. In this thesis we consider only *dynamic* slicing, i.e. queries of this nature which pertain to a specific execution.

Dynamic slicing questions can be asked in two directions. Forward slicing questions concern the parts of the output which *must be deleted* as consequence of deleting some part of the program. Backward slicing questions concern the parts of the program which *may be deleted* as a consequence of deleting some part of the output. In this chapter we show how such questions can be supported in a way that both complements and utilises the reified computations introduced in the previous chapter.

In §5.1 we make precise the notion of a “part” of a program or value. We start by extending the syntax with *holes*, written \square , in the style of Biswas [Bis97]. A hole is a no-op expression, inhabiting every type, which evaluates to itself. Expressions and values generalise to *partial expressions* and *partial values* which may contain holes, and which are partially ordered. The order has $e \sqsubseteq e'$ whenever we can obtain e from e' by replacing some sub-expressions by holes; we say that e is a *prefix* of e' .

In §5.2 we model forward dynamic slicing, for a fixed program, as the execution of some prefix of the program to obtain a prefix of its output, where holes represent lack of availability. Our key contribution here is to show that this formulation of forward slicing uniquely determines the backward dynamic slicing problem for the same computation. What we show is that backward slicing, where holes represent lack of demand, is the problem of calculating the lower adjoint of the function which computes forward slices. To account for our construal of slicing questions as *changes* in availability (forward slicing) or demand (backward slicing), we introduce the idea of a *differential* program slice, the pair of a partial program and a smaller one. Differential slices enable more fine-grained Q&A about the relationship between input and output.

In §5.3 we give an algorithm called *unevaluation* which efficiently calculates backward slices, making use of the computational history recorded in a trace. Whereas evaluation unrolls a program, unevaluation rolls it back up again, recovering enough of the original program to be able to compute the required prefix of its output. Unevaluation only computes program slices; in §5.4, we give an additional algorithm which slices a trace into a partial trace retaining just enough information to “explain” the partial output.

5.1 Partial programs and partial values

With a statement-based language, a slice can be represented simply as a program from which some statements have been deleted. This approach is unsuitable for expression-based languages with fixed-arity constructors, as noted by Biswas [Bis97]. Following Biswas, we therefore introduce a new expression form \square , called *hole*, which inhabits every type, and use holes to represent deleted sub-terms of an expression or value. For example we can “slice” the expression $\text{inl } (3 \oplus 4)$ by replacing its left sub-expression by \square , obtaining $\text{inl } (\square \oplus 4)$. The additional syntax rules are given at the top of Figure 5.1. From now on, talk of expressions and values should be understood more precisely to mean *partial* expressions and *partial* values, which may contain holes. We also take the terms “expression slice” and “value slice” to be synonymous with partial expression and partial value.

Let us recall the standard initial-algebra construction of expressions as sets. (What we say here applies equally to values.) An expression is a set of odd-length paths, satisfying two properties characteristic of “tree-hood”, prefix-closure, and deterministic extension, plus well-foundedness. A *path* is an alternating sequence $\langle k_0, n_0, \dots, k_{i-1}, n_{i-1}, k_i \rangle$ of constructors k and child indices n . Prefix-closure means that if a path is in the set, then each of its prefixes is in the set; deterministic extension means that all paths agree about the value of k_i for a given position in the tree. Writing the child indices in bold to distinguish them from constructors, the following set of such paths comprises the partial expression $\text{inl } (3 \oplus \square)$:

$$\underbrace{\begin{array}{l} \langle \text{inl} \rangle, \\ \langle \text{inl}, \mathbf{0}, \oplus \rangle, \\ \langle \text{inl}, \mathbf{0}, \oplus, \mathbf{0}, 3 \rangle \end{array}}_{\text{inl } (3 \oplus \square)}$$

The \square form then has a natural interpretation as the empty set. To see why introducing \square into the syntax gives rise to a partial order \sqsubseteq on expressions and values, we need only interpret the relation \sqsubseteq as the inclusion order \subseteq on these sets. For example, the fact that $\text{inl } (3 \oplus \square)$ and $\text{inl } (3 \oplus 4)$ are related by \sqsubseteq is because the sets of paths that comprise the two expressions are related by \subseteq :

$$\underbrace{\begin{array}{l} \langle \text{inl} \rangle, \\ \langle \text{inl}, \mathbf{0}, \oplus \rangle, \\ \langle \text{inl}, \mathbf{0}, \oplus, \mathbf{0}, 3 \rangle \end{array}}_{\text{inl } (3 \oplus \square)} \subseteq \underbrace{\begin{array}{l} \langle \text{inl} \rangle, \\ \langle \text{inl}, \mathbf{0}, \oplus \rangle, \\ \langle \text{inl}, \mathbf{0}, \oplus, \mathbf{0}, 3 \rangle, \\ \langle \text{inl}, \mathbf{0}, \oplus, \mathbf{1}, 4 \rangle \end{array}}_{\text{inl } (3 \oplus 4)}$$

An expression smaller than a given expression e is thus a variant of e where some paths have been truncated in a way which preserves prefix-closure; it is e with some sub-expressions “deleted”. The absence of those sub-expressions is indicated in the conventional syntax by the presence of a \square . We call such a truncated version of e a *prefix* of e , or a *slice* of e , and denote the set of such expressions by $\text{Prefix}(e)$.

Definition 1 (Prefixes of e). $\text{Prefix}(e) = \{e' \mid e' \sqsubseteq e\}$

$$\begin{array}{c}
\text{Expressions } e ::= \dots \mid \square \\
\text{Values } v ::= \dots \mid \square \\
\\
\boxed{\Gamma \vdash e : \tau} \\
\\
\dots \qquad \qquad \qquad \overline{\Gamma \vdash \square : \tau} \\
\\
\boxed{\vdash v : \tau} \\
\\
\dots \qquad \qquad \qquad \overline{\vdash \square : \tau} \\
\\
\boxed{\rho, e \Downarrow_{\text{ref}} v} \\
\\
\begin{array}{c}
\dots \qquad \overline{\rho, \square \Downarrow_{\text{ref}} \square} \qquad \overline{\rho, e_1 \Downarrow_{\text{ref}} \square} \qquad \overline{\rho, e_1 \Downarrow_{\text{ref}} c_1 \quad \rho, e_2 \Downarrow_{\text{ref}} \square} \qquad \overline{\rho, e_1 \Downarrow_{\text{ref}} \square} \\
\rho, e_1 \oplus e_2 \Downarrow_{\text{ref}} \square \qquad \rho, e_1 \oplus e_2 \Downarrow_{\text{ref}} \square \qquad \rho, e_1 \Downarrow_{\text{ref}} c_1 \quad \rho, e_2 \Downarrow_{\text{ref}} \square \qquad \rho, e_1 \Downarrow_{\text{ref}} \square \\
\rho, \text{fst } e \Downarrow_{\text{ref}} \square \qquad \rho, \text{snd } e \Downarrow_{\text{ref}} \square \qquad \rho, \text{case } e \text{ of } \{\text{inl}(x_1).e_1; \text{inr}(x_2).e_2\} \Downarrow_{\text{ref}} \square \qquad \rho, \text{unroll } e \Downarrow_{\text{ref}} \square
\end{array}
\end{array}$$

Figure 5.1 Additional rules for partial expressions

What is more, the set $\text{Prefix}(e)$ forms a finite, distributive, bounded lattice with greatest element e , least element \square , and meet \sqcap and join \sqcup corresponding to intersection and union on the underlying sets of paths.¹ In fact we can take the intersection of any two expressions e_1 and e_2 even when they are not prefixes of a common expression, because intersection preserves prefix-closure and deterministic extension. Thus $e_1 \sqcap e_2$ is always defined and yields the greatest lower bound of e_1 and e_2 .

However, taking the union of e_1 and e_2 only yields a well-formed expression when e_1 and e_2 have compatible structure. This is because set union does not in general preserve deterministic extension: consider taking the union of $\text{inl } 3 \oplus 4$ and $\text{inl } 3 \oplus 5$, for example. But if e_1 and e_2 are *upper-bounded*, that is to say if there is an e such that $e_1 \sqsubseteq e$ and $e_2 \sqsubseteq e$, then they do indeed have compatible structure. Here we can see that $\text{inl } (3 \oplus \square) \sqcup \text{inl } (\square \oplus 4)$ is defined:

$$\begin{array}{ccc}
\overbrace{\langle \text{inl}, \rangle} & \overbrace{\langle \text{inl}, \rangle} & \overbrace{\langle \text{inl}, \rangle} \\
\overbrace{\langle \text{inl}, 0, \oplus, \rangle} & \cup & \overbrace{\langle \text{inl}, 0, \oplus, \rangle} \\
\overbrace{\langle \text{inl}, 0, \oplus, 0, 3 \rangle} & & \overbrace{\langle \text{inl}, 0, \oplus, 0, 3 \rangle} \\
\overbrace{\langle \text{inl}, 0, \oplus, 1, 4 \rangle} & = & \overbrace{\langle \text{inl}, 0, \oplus, 1, 4 \rangle} \\
\text{inl } (3 \oplus \square) & & \text{inl } (\square \oplus 4) & & \text{inl } (3 \oplus 4)
\end{array}$$

Thus when e_1 and e_2 are both elements of $\text{Prefix}(e)$, the join $e_1 \sqcup e_2$ is defined and yields the least upper bound of e_1 and e_2 .

¹From now on, by the unqualified term “lattice” we will mean a finite, distributive, bounded lattice.

5.2 Characterising dynamic slicing

So we have a way of representing programs or values with missing parts: we simply replace the parts we want to delete with appropriately typed holes. To provide the raw components of the differential slices just described, what we now need is a way of determining how much of the output we can compute given only some prefix of the program (forward slicing), and how little of the program is needed if we need only some prefix of the output (backward slicing). It turns out that these problems are so closely related that in fact each determines the other.

5.2.1 Forward dynamic slicing

The intuition we proposed for forward slicing in Chapter 2 was that if a step in the computation consumes some program part which is unavailable, the output of that step must also be unavailable. In other words, unavailability propagates forward through the computation. This is straightforward to capture by extending the reference semantics \Downarrow_{ref} with the additional rules for propagating holes given in Figure 5.1. Hole itself evaluates to \square , and moreover for every type constructor, there are variants of the elimination rule which produce a hole whenever the sub-computation in the elimination position produces a hole. The new rules do not affect type preservation (Lemma 1) or determinism (Lemma 2), and so from now on by \Downarrow_{ref} we shall mean the extended version of the rules, with these lemmas taken to apply to the new definition.

There are two things to note about the hole-propagation rules. First, no special treatment is required when the *argument* to a function evaluates to a hole; the behaviour we want in this case is precisely that the unavailability of the argument should only matter if that argument is actually consumed by the execution of the function. Second, the rules for primitive operations take them to be strict in both operands. Even when this is true of an actual implementation, it may not accurately reflect the dependency of the operation on its arguments: for example \times need not consult the second argument if the first argument is \emptyset . We discuss a more realistic treatment of primitives in Future Work, §7.2.6.

First we note that, since evaluation with hole-propagation can produce partial values, environments must also be partial, in other words map variables to partial values. This gives rise to a partial order on environments; specifically, we overload \sqsubseteq to mean the relation that has $\rho \sqsubseteq \rho'$ iff $\text{dom}(\rho) = \text{dom}(\rho')$ and $\forall x \in \text{dom}(\rho). \rho(x) \sqsubseteq \rho'(x)$. For any Γ , we will write \square_Γ for the smallest partial environment for Γ , namely the ρ such that $\rho(x) = \square$ for every $x \in \text{dom}(\Gamma)$. Again, the set $\text{Prefix}(\rho)$ forms a lattice where the join $\rho' \sqcup \rho''$ is the partial environment $\{x \mapsto \rho'(x) \sqcup \rho''(x) \mid x \in \text{dom}(\rho)\}$, and similarly for meet. Since environments are defined inductively, environment extension with respect to a variable x is a lattice isomorphism in the following sense. Suppose $\Gamma \vdash \rho$ and $\vdash v : \tau$. Then for any x , the bijection $-[x \mapsto -]$ from $\text{Prefix}(\rho) \times \text{Prefix}(v)$ to $\text{Prefix}(\rho[x \mapsto v])$ satisfies:

$$(\rho' \sqcup \rho'')[x \mapsto u \sqcup u'] = \rho'[x \mapsto u'] \sqcup \rho''[x \mapsto u'] \quad (5.1)$$

$$(\rho' \sqcap \rho'')[x \mapsto u \sqcap u'] = \rho'[x \mapsto u'] \sqcap \rho''[x \mapsto u'] \quad (5.2)$$

Extending evaluation with hole-propagation rules has some important consequences which are summarised in Theorem 2 below. First we define the following family of partial functions indexed by terminating

programs.

Definition 2 ($\text{eval}_{\rho,e}$). Suppose $\rho, e \Downarrow_{\text{ref}} v$. Define $\text{eval}_{\rho,e}$ to be \Downarrow_{ref} domain-restricted to $\text{Prefix}(\rho, e)$.

For readability, we will drop the ρ, e subscript from $\text{eval}_{\rho,e}$ whenever it is applied to a prefix of (ρ, e) and the (ρ, e) is clear from the context. Now we make three observations. Collectively, they assert that $\text{eval}_{\rho,e}$ is *meet-semilattice homomorphism* from $\text{Prefix}(\rho, e)$ to $\text{Prefix}(v)$. First, $\text{eval}_{\rho,e}$ is a total function. Because unavailability propagates, introducing a hole into a terminating program cannot yield a non-terminating program but only one which produces less output. Second, least and greatest elements are preserved, which is just immediate from the definitions. Finally, $\text{eval}_{\rho,e}$ preserves meets, or intersection of slices. This third property implies the monotonicity already alluded to.

Theorem 2 ($\text{eval}_{\rho,e}$ is a meet-semilattice homomorphism). Suppose $\rho, e \Downarrow_{\text{ref}} v$. Then:

1. If $(\rho', e') \sqsubseteq (\rho, e)$ then $\text{eval}(\rho', e')$ is defined: there exists u such that $\rho', e' \Downarrow_{\text{ref}} u$.
2. $\text{eval}(\square) = \square$ and $\text{eval}(\rho, e) = v$.
3. If $(\rho', e') \sqsubseteq (\rho, e)$ and $(\rho'', e'') \sqsubseteq (\rho, e)$ then $\text{eval}(\rho' \sqcap \rho'', e' \sqcap e'') = \text{eval}(\rho', e') \sqcap \text{eval}(\rho'', e'')$.

Proof. Part (2) is immediate from the definition of \Downarrow_{ref} and $\text{eval}_{\rho,e}$. For parts (1) and (3), we proceed by induction on the derivation of $\rho, e \Downarrow_{\text{ref}} v$, using the hole propagation rules from Figure 5.1 whenever the evaluation would otherwise get stuck, and Equation 5.2 for the binder cases. \square

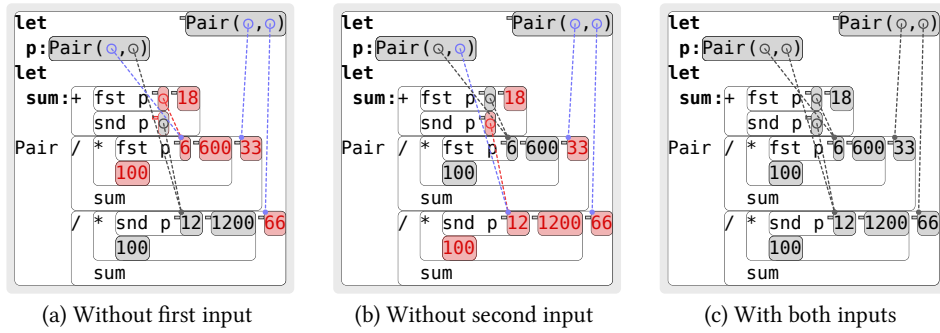


Figure 5.2 $\text{eval}_{\rho,e}$ does not preserve joins

Although $\text{eval}_{\rho,e}$ preserves meets, it does not preserve joins, i.e. union of slices, as illustrated in Figure 5.2. The program normalises two integers 6 and 12 by computing the proportion that each is of their sum and returning the result as a pair of percentages. In (a), we “damage” the program by (hypothetically) deleting the first input 6. If the sum of the two numbers cannot be calculated, neither can either of the outputs. The situation is reversed in (b), where we hypothetically delete the second input 12. If we *combine* the two program slices – by taking their join, in (c) – we repair the ability of the program to compute the sum. But in so doing we also repair the ability of the program to calculate both components of the output. In general,

because removing part of a program can dramatically impair its ability to function, combining program parts can dramatically restore that capability.

If $\text{eval}_{\rho,e}$ is to provide canonical answers to forward-slicing questions then it must compute *as much output as possible* for the prefix of (ρ, e) it is given. This is indeed the case, but it will be easier to make sense of this once we have introduced backward slicing.

5.2.2 Backward dynamic slicing

We will now see how forward slicing as just construed uniquely defines the problem of dynamic backward slicing informally sketched in Chapter 2. Our intuition there was that backward slicing tells us how much of the program is still needed if we only need some prefix of the output. To state this formally, we need to make precise the notion of there being “enough” program to compute that part of the output; we can do so by appealing to our monotonic forward-slicing function $\text{eval}_{\rho,e}$. Suppose $\rho, e \Downarrow_{\text{ref}} v$ and some partial output $u \sqsubseteq v$ specifying how much of the output is needed. If (ρ', e') is capable of computing at least u , we say that (ρ', e') is a *slice of* (ρ, e) for u .

Definition 3 (Slice of (ρ, e) for u). Suppose $\rho, e \Downarrow_{\text{ref}} v$ and $u \sqsubseteq v$. Then any $(\rho', e') \sqsubseteq (\rho, e)$ is a *slice of* (ρ, e) for u if $\text{eval}(\rho', e') \sqsupseteq u$.

Operationally, the intuition is that it is fine to consume a hole during evaluation as long as we are computing a part of the output that is not needed.

Now, a canonical answer to a backward-slicing question is the *smallest* program slice for the prefix of v in question. At it happens, the fact that $\text{eval}_{\rho,e}$ preserves meets guarantees the existence of such a slice. This stems from a basic property of meet-semilattice homomorphisms. If A and B are lattices, then every meet-preserving function $f^* : A \rightarrow B$ is the *upper adjoint* of a unique Galois connection. The *lower adjoint* of f^* , written $f_* : B \rightarrow A$, which preserves joins, inverts f^* in the following minimising way: for any output b of f^* , the lower adjoint yields the smallest input a such that $f^*(a) \sqsupseteq b$. In fact each adjoint determines the other:

$$f^*(a) \sqsupseteq b \iff a \sqsupseteq f_*(b)$$

This is easier to understand if we plug in $\text{eval}_{\rho,e}$ and its lower adjoint, which we shall call $\text{uneval}_{\rho,e}$ because it maps values to programs. Analogously with $\text{eval}_{\rho,e}$ we drop the ρ, e subscript from $\text{uneval}_{\rho,e}$ when the argument is a prefix of v and the (ρ, e) is clear from the context.

Corollary 1 (Existence of least program slices). Suppose $\rho, e \Downarrow_{\text{ref}} v$. Then there exists a unique function $\text{uneval}_{\rho,e}$ from $\text{Prefix}(v)$ to $\text{Prefix}(\rho, e)$ such that for any $(\rho', e') \sqsubseteq (\rho, e)$ and any $u \sqsubseteq v$ we have:

$$\text{eval}(\rho', e') \sqsupseteq u \iff (\rho', e') \sqsupseteq \text{uneval}(u)$$

Proof. Immediate from Theorem 2. □

For every prefix of the program there is a largest output slice which consumes at most that much of the program; and for every prefix of the output there is a smallest program slice which produces at least that much output.

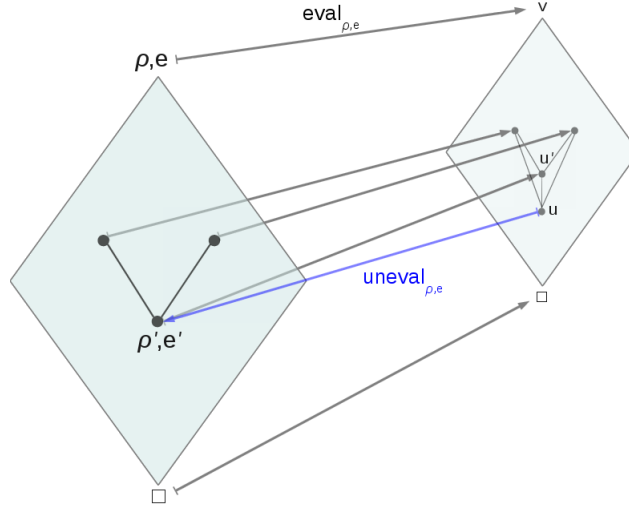


Figure 5.3 Closure under meet of slices of (ρ, e) for u

It is instructive to consider why the meet-preservation of $\text{eval}_{\rho, e}$ ensures that $\text{uneval}_{\rho, e}$ exists. Let S be the set of all slices of (ρ, e) for some fixed u and let (ρ', e') be their meet. Because evaluation preserves meets, (ρ', e') evaluates to the meet u' of the values that the elements of S evaluate to. But all these values are larger than u , and therefore so is u' . Thus (ρ', e') is itself an element of S , namely the smallest one, so we can set this to be the value of $\text{uneval}(u)$. This is depicted informally in Figure 5.3. The larger diamond on the left is the lattice $\text{Prefix}(\rho, e)$; the smaller diamond on the right is the lattice $\text{Prefix}(v)$. For this particular u , there are exactly three elements of S , indicated by the three points in the left-hand lattice. Thus $\text{uneval}_{\rho, e}$ satisfies the following:

$$\text{uneval}_{\rho, e}(u) = \bigwedge \{(\rho', e') \in \text{Prefix}(\rho, e) \mid \text{eval}_{\rho, e}(\rho', e') \sqsupseteq u\} \quad (5.3)$$

and principle $\text{uneval}(u)$ could be calculated by enumerating all the program slices for u and taking their meet. In the next section we will see a better approach.

Before we move on we contrast the behaviour of backward slicing with forward slicing. Whereas $\text{eval}_{\rho, e}$ preserves meets and not joins, $\text{uneval}_{\rho, e}$ preserves joins and not meets. Figure 5.4 revisits the normalisation example from Figure 5.2 to give an example of how meets are not preserved. Since we are backward-slicing, we manipulate the demand on the output of the computation. In (a), we relinquish demand on the first output 33. Although we then do not need the entire $(\text{fst } p) * 6 / \text{sum}$ calculation associated with it, we still need to calculate the sum because we need it for the second output, and this in turn means we still need both inputs. The situation is reversed in (b), where we relinquish demand on the second output 66. But when we combine these demand *absences* – by taking the meet of the output slices, as in (c) – we no longer need the sum, nor indeed either input. Thus backward slicing exhibits a kind of conservativity: part of the program can be relinquished only if it is not needed anywhere.

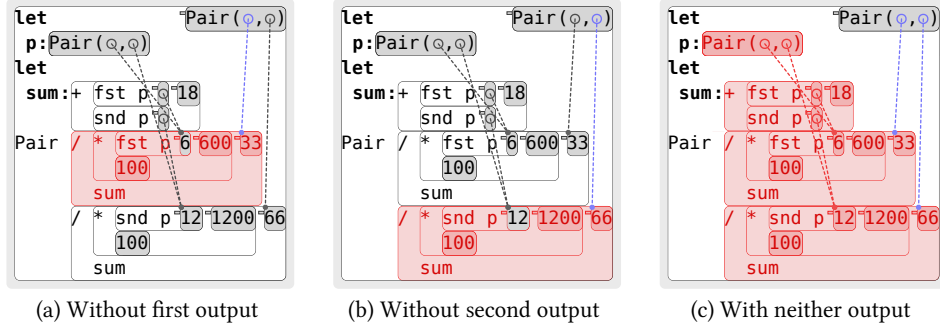


Figure 5.4 $\text{uneval}_{\rho, e}$ does not preserve meets

5.2.3 Differential slices

In Chapter 2, §§2.5 and 2.6, we showed several examples where the user *selected* some part of the program or output in order to initiate a forward or backward dynamic slice. The red highlighting on the selected node was explained as a *deletion delta*: a comparison between the present state, and a hypothetical future state in which that node was deleted. The red parts pick out the *complement of a prefix* of the expression, a kind of negated slice capturing the difference between two expressions related by \sqsubseteq . The complement of a partial expression is not itself a partial expression, since the underlying set of paths is not prefix-closed.

In our implementation, these deltas that arise during slicing are just a special case of the more general form of delta illustrated in §2.2, which can describe complex structural reorganisations. We will be looking at these in detail in Chapter 6. But it is possible to explain differential slices without recourse to the more involved techniques presented there.

A deletion delta can be expressed in the formalism presented so far as a *pair* of partial expressions (e, e') where $e \sqsubseteq e'$. We call such a pair a *differential slice*. More precisely, for an expression e , we define $\text{Diff}(e)$ to be the following sub-lattice of $\text{Prefix}(e) \times \text{Prefix}(e)$:

Definition 4 (Differential slice). Define $\text{Diff}(e)$ to be the lattice with carrier set $\{(e', e'') \mid e' \sqsubseteq e'' \sqsubseteq e\}$ and meet and join defined component-wise.

Moreover, the Galois connection for slices of a terminating computation lifts to differential slices in the natural way, yielding differential slices that are minimal because their components are, and because pairing preserves \sqcap and \sqcup .

Definition 5 (Differential slicing). Suppose $\rho, e \Downarrow_{\text{ref}} v$. Then define the Galois connection $\langle \text{eval}, \text{uneval} \rangle_{\text{Diff}(\rho, e)}$ from $\text{Diff}(\rho, e)$ to $\text{Diff}(v)$, where

$$\begin{aligned} \text{eval} &\stackrel{\text{def}}{=} \text{eval}_{\rho, e} \times \text{eval}_{\rho, e} \text{ domain-restricted to } \text{Diff}(\rho, e) \\ \text{uneval} &\stackrel{\text{def}}{=} \text{uneval}_{\rho, e} \times \text{uneval}_{\rho, e} \text{ domain-restricted to } \text{Diff}(v) \end{aligned}$$

The larger component e' of a differential slice (e, e') is the present state; the smaller component e is the future state relative to which e' is being compared. Because e and e' are related by \sqsubseteq , calculating the difference

between them is easy. We simply traverse e and e' simultaneously, identifying sub-expressions present in e' but absent in e . Nodes unique to the present are scheduled for deletion in the future and highlighted in red in the user interface.

5.3 Program slicing as backwards execution

In the previous section, we showed that, for any $\rho, e \Downarrow_{\text{ref}} v$, the forward-slicing function $\text{eval}_{\rho,e}$ has a unique backward-slicing adjoint $\text{uneval}_{\rho,e}$ which yields the smallest program slice for any partial output $u \sqsubseteq v$. We also saw that to calculate $\text{uneval}_{\rho,e}(v)$, we could in principle consider every prefix of the program, and take the meet of those large enough to compute v . Such an approach would not lead to a practical algorithm. Instead what we would like to do is somehow infer backwards from the unneeded parts of the output to the unneeded parts of the input.

The computational history stored in the form of the *traces* introduced in Chapter 4 allows us to do precisely that. Having the history available means the computation can be “rewound” and a partial program reconstructed sufficient to compute that portion of the output. We call this procedure *unevaluation*, since it is a form of backwards execution. In this section, we give a definition of unevaluation, and show that for any $\text{eval}_{\rho,e}$, the unevaluation algorithm, supplied with a suitable trace, implements $\text{uneval}_{\rho,e}$.

$$\begin{array}{c}
 \text{Traces } T ::= \dots \mid \square \\
 \\
 \boxed{\Gamma \vdash T : \tau} \\
 \\
 \dots \qquad \overline{\Gamma \vdash \square : \tau} \\
 \\
 \boxed{\rho, e \Downarrow v, T} \\
 \\
 \frac{}{\rho, \square \Downarrow \square, \square} \quad \frac{\rho, e_1 \Downarrow \square, T_1}{\rho, e_1 \oplus e_2 \Downarrow \square, \square} \quad \frac{\rho, e_1 \Downarrow c_1, T_1 \quad \rho, e_2 \Downarrow \square, T_2}{\rho, e_1 \oplus e_2 \Downarrow \square, \square} \quad \frac{\rho, e_1 \Downarrow \square, T_1}{\rho, e_1 e_2 \Downarrow \square, \square} \quad \frac{\rho, e \Downarrow \square, T}{\rho, \text{fst } e \Downarrow \square, \square} \\
 \\
 \frac{\rho, e \Downarrow \square, T}{\rho, \text{snd } e \Downarrow \square, \square} \quad \frac{\rho, e \Downarrow \square, T}{\rho, \text{case } e \text{ of } \{\text{inl}(x_1).e_1; \text{inr}(x_2).e_2\} \Downarrow \square, \square} \quad \frac{\rho, e \Downarrow \square, T}{\rho, \text{unroll } e \Downarrow \square, \square}
 \end{array}$$

Figure 5.5 Additional rules for partial traces

First we need to extend the syntax and semantics of traces to accommodate \square as given in Figure 5.5. As with expressions, \square induces a partial order on traces, which we again denote by \sqsubseteq , and for any trace T there is a lattice $\text{Prefix}(T)$ of prefixes of T . Note that if $\Gamma \vdash T : \tau$ and $S \sqsubseteq T$, then $\Gamma \vdash S : \tau$. The hole propagation rules for the tracing semantics are the same as for the reference semantics except that they supply a hole trace alongside a hole value. These additional rules do not affect determinism (Lemma 3), type preservation (Lemma 4), or agreement with the reference semantics on values (Theorem 1). So from

now on, by \Downarrow we shall mean the tracing semantics extended with these additional rules, with these lemmas again taken to apply to the new definition. Moreover, if we domain-restrict \Downarrow to the prefixes of a terminating program $\rho, e \Downarrow v, T$ then we obtain a meet-semilattice homomorphism from $\text{Prefix}(\rho, e)$ to $\text{Prefix}(v, T)$. However we do not state this formally but instead use the meet-preservation of the reference semantics where necessary.

LambdaCalc actually implements different tracing rules for the hole-propagation cases from those given in Figure 5.5. In each case, rather than just returning a hole trace alongside the hole value, instead it returns a trace which explains how the hole was computed. New trace forms are required to record these explanations, since they contain expressions which did not unroll into their executions because an earlier sub-computation attempted to consume a hole. For example, the implemented hole-propagation rule for functions and sums look like this:

$$\frac{\rho, e_1 \Downarrow \square, T_1}{\rho, e_1 e_2 \Downarrow \square, T_1 e_2} \quad \frac{\rho, e \Downarrow \square, T}{\rho, \text{case } e \text{ of } \{\text{inl}(x_1).e_1; \text{inr}(x_2).e_2\} \Downarrow \square, \text{case } T \text{ of } \{\text{inl}(x_1).e_1; \text{inr}(x_2).e_2\}}$$

A trace $T_1 e_2$ records an application where the computation of the function produced a hole and so execution did not proceed into e_2 . A trace $\text{case } T \text{ of } \{\text{inl}(x_1).e_1; \text{inr}(x_2).e_2\}$ records a case expression where the scrutinee evaluated to a hole and so execution did not proceed into a branch. The “data-driven” intuition we used to explain forward slicing in Chapter 2 is useful for understanding these trace forms: one can think of $T_1 e_2$ as an application that is blocking pending the availability of a function, and $\text{case } T \text{ of } \{\text{inl}(x_1).e_1; \text{inr}(x_2).e_2\}$ as a conditional that is blocking pending a value for the scrutinee.

We do not model this behaviour in the formalism for purely technical reasons. For properties like monotonicity to make sense with such trace forms, we need to formally embrace our informal notion of traces as unrolled expressions. In particular we need expressions to be part of the same partial order as traces, with expressions as just the “least unrolled” trace forms. Suppose we have a program that evaluates to $T_1 T_2 \triangleright \langle \gamma, \text{fun } f(x).T \rangle$. If we make the program smaller in some way so that it can only evaluate as far as $S_1 e_2$, then monotonicity requires that $S_1 e_2 \sqsubseteq T_1 T_2 \triangleright \langle \gamma, \text{fun } f(x).T \rangle$. But this only makes sense if we can treat an application as “smaller than” its unrolling into an application trace and more generally an expression as smaller than any of its unrolled trace forms. Moreover, for simplicity, we would like to retain the interpretation of the \sqsubseteq order on traces as the inclusion order on the underlying sets of paths. A simple way to satisfy both requirements is to treat an application $e_1 e_2$ as “shorthand” for a trace of the form $e_1 e_2 \triangleright \square$; indeed this is precisely what our implementation does. However, we leave modelling this aspect of the implementation to future work.

5.3.1 Unevaluation

We now define our program slicing algorithm, *unevaluation*, in Figure 5.6. For a value $\vdash v : \tau$ and trace $\Gamma \vdash T : \tau$, the judgement $v, T \Downarrow^{-1} \rho, e$ states that T can be used to unevaluate v to partial environment $\Gamma \vdash \rho$ and partial expression $\Gamma \vdash e : \tau$. A key part of the definition which for convenience is omitted from Figure 5.6 is the following. Every time a rule takes the join of two values or environments, there an implicit side-condition stating that the joins exist. For example, the application rule has an implicit side-condition

$$\boxed{v, T \Downarrow^{-1} \rho, e \text{ where } \Gamma \vdash T : \tau}$$

$$\begin{array}{c}
\frac{}{\square, T \Downarrow^{-1} \square_{\Gamma}, \square} \qquad \frac{}{v, x \Downarrow^{-1} \square_{\Gamma, x \mapsto v}, x} v \neq \square \qquad \frac{}{c, c \Downarrow^{-1} \square_{\Gamma}, c} \\
\\
\frac{c_2, T_2 \Downarrow^{-1} \rho_2, e_2 \quad c_1, T_1 \Downarrow^{-1} \rho_1, e_1}{v, T_1 \oplus_{c_1, c_2} T_2 \Downarrow^{-1} \rho_1 \sqcup \rho_2, e_1 \oplus e_2} v \neq \square \qquad \frac{}{\langle \rho, \mathbf{fun} f(x).e \rangle, \mathbf{fun} f(x).e' \Downarrow^{-1} \rho, \mathbf{fun} f(x).e} e \sqsubseteq e' \\
\\
\frac{v, T \Downarrow^{-1} \rho[f \mapsto v_1][x \mapsto v_2], e \quad v_2, T_2 \Downarrow^{-1} \rho_2, e_2 \quad v_1 \sqcup \langle \rho, \mathbf{fun} f(x).e \rangle, T_1 \Downarrow^{-1} \rho_1, e_1}{v, T_1 T_2 \triangleright \langle \gamma, \mathbf{fun} f(x).T \rangle \Downarrow^{-1} \rho_1 \sqcup \rho_2, e_1 e_2} v \neq \square \\
\\
\frac{v_2, T_2 \Downarrow^{-1} \rho_2, e_2 \quad v_1, T_1 \Downarrow^{-1} \rho_1, e_1}{(v_1, v_2), (T_1, T_2) \Downarrow^{-1} \rho_1 \sqcup \rho_2, (e_1, e_2)} \quad \frac{(v_1, \square), T \Downarrow^{-1} \rho, e}{v_1, \mathbf{fst} T \Downarrow^{-1} \rho, \mathbf{fst} e} v_1 \neq \square \quad \frac{(\square, v_2), T \Downarrow^{-1} \rho, e}{v_2, \mathbf{snd} T \Downarrow^{-1} \rho, \mathbf{snd} e} v_2 \neq \square \\
\\
\frac{v, T \Downarrow^{-1} \rho, e}{\mathbf{inl} v, \mathbf{inl} T \Downarrow^{-1} \rho, \mathbf{inl} e} \qquad \frac{v, T \Downarrow^{-1} \rho, e}{\mathbf{inr} v, \mathbf{inr} T \Downarrow^{-1} \rho, \mathbf{inr} e} \\
\\
\frac{v, T_1 \Downarrow^{-1} \rho_1[x_1 \mapsto v_1], e_1 \quad \mathbf{inl} v_1, T \Downarrow^{-1} \rho, e}{v, \mathbf{case} T \text{ of } \{\mathbf{inl}(x_1).T_1; \mathbf{inr}(x_2).e_2\} \Downarrow^{-1} \quad \rho_1 \sqcup \rho, \mathbf{case} e \text{ of } \{\mathbf{inl}(x_1).e_1; \mathbf{inr}(x_2).\square\}} v \neq \square \\
\\
\frac{v, T_2 \Downarrow^{-1} \rho_2[x_2 \mapsto v_2], e_2 \quad \mathbf{inr} v_2, T \Downarrow^{-1} \rho, e}{v, \mathbf{case} T \text{ of } \{\mathbf{inl}(x_1).e_1; \mathbf{inr}(x_2).T_2\} \Downarrow^{-1} \quad \rho_2 \sqcup \rho, \mathbf{case} e \text{ of } \{\mathbf{inl}(x_1).\square; \mathbf{inr}(x_2).e_2\}} v \neq \square \\
\\
\frac{v, T \Downarrow^{-1} \rho, e}{\mathbf{roll} v, \mathbf{roll} T \Downarrow^{-1} \rho, \mathbf{roll} e} \qquad \frac{\mathbf{roll} v, T \Downarrow^{-1} \rho, e}{v, \mathbf{unroll} T \Downarrow^{-1} \rho, \mathbf{unroll} e} v \neq \square
\end{array}$$

Figure 5.6 Slicing rules: unevaluation

stating that v_1 and $\langle \rho, \text{fun } f(x).e \rangle$ have an upper bound and also that ρ_1 and ρ_2 have an upper bound. Later we will show that these implicit side-conditions are satisfied whenever T is produced by a tracing evaluation which yields v (Theorem 4 below). Unevaluation is deterministic, which is a straightforward induction, relying on the $v \neq \square$ side-conditions in Figure 5.6.

Lemma 5 (Determinism of unevaluation). *If $v, T \Downarrow^{-1} \rho, e$ and $v, T \Downarrow^{-1} \rho', e'$ then $(\rho, e) = (\rho', e')$.*

Unevaluation traverses the trace and folds it back into an expression from which the unneeded bits have been discarded. As with evaluation, least elements are preserved, which simply means that holes map to holes: unevaluating the value \square produces the expression \square and \square_Γ , the smallest environment for Γ . Unevaluating the trace of a variable x with v yields x as an expression and the smallest environment for Γ mapping x to v , which we write as $\square_{\Gamma.x \mapsto v}$. The general pattern for non-nullary trace constructors is that the traces of the sub-computations are unevaluated and the resulting partial environments joined. For example we unevaluate a pair trace with a pair (v_1, v_2) by unevaluating the two sub-traces with v_1 and v_2 respectively and joining the partial environments thereby obtained. When binders are involved, well-typedness allows us to safely extract partial values for the bound variable by pattern-matching the relevant partial environment. For example with a `case` trace for `in1`, the selected branch is unevaluated, producing a partial environment of the form $\rho_1[x \mapsto v_1]$, where v_1 is a partial value which is then injected into the sum type and used to slice the scrutinee.

Unevaluating the application of a primitive operation retrieves the values c_1 and c_2 previously cached in the trace and uses these to unevaluate the arguments. Without the cached values it would not be clear how to proceed, because the demand placed on the operands of a primitive operation is internal to that operation. As mentioned earlier, the present approach treats all primitive operations as strict in both operands, although we revisit this in Future Work, §7.2.6.

The application rule is the most interesting. For a trace $T_1 \ T_2 \triangleright \langle \gamma, \text{fun } f(x).T \rangle$, we unevaluate T to obtain a slice e of the original function body, plus an environment $\rho[f \mapsto v_1][x \mapsto v_2]$ where ρ is a slice of the environment in which the closure was captured, and v_1 and v_2 are slices describing the usage of f and x respectively inside T . The closure variables γ , which are equal to $\text{vars}(\rho)$, have no role to play; recall that they serve only to make the trace well-typed. Since T contains all recursive uses of the function, v_1 (which is a slice of a closure) captures how much of f was used below this step of the computation. We then join v_1 with $\langle \rho, \text{fun } f(x).e \rangle$ to merge in information about the usage of the function at the present step, and use it to unevaluate T_1 .

5.3.2 Correctness of tracing evaluation

We can now return to the correctness property for tracing evaluation mentioned at the end of Chapter 4. It hinges on making precise the notion of a trace *explaining* a value. We start with the observation that not every well-typed trace T can be used to unevaluate a value v of the same type. On the one hand, T might have some strange (but well-typed) structure that could never be produced by evaluation, so that the required joins do not exist. On the other hand, T might have the right structure, but be too small, in the sense of containing insufficient information to unevaluate v . So a key property of T with respect to v is whether it

is able to guide the unevaluation of v . When T has this property, we say that it *explains* v . Clearly there is no unique explanation of a given v .

Definition 6 (T explains v). For any value v and any trace T , we say that T *explains* v iff there exist ρ, e such that $v, T \Downarrow^{-1} \rho, e$.

The key correctness property of tracing evaluation is that it produce traces that explain the value computed. Before showing that this is indeed the case, we first show that for any trace T which explains v , where $v, T \Downarrow^{-1} \rho, e$, there is a monotonic function $\text{tr-uneval}_{v,T}$ from $\text{Prefix}(v)$ to $\text{Prefix}(\rho, e)$. In fact $\text{tr-uneval}_{v,T}$ also preserves joins, but monotonicity is sufficient for our purposes. First we define $\text{tr-uneval}_{v,T}$.

Definition 7 ($\text{tr-uneval}_{v,T}$). Suppose T explains v . Then define $\text{tr-uneval}_{v,T}$ to be the partial function $\{u \mapsto (\rho, e) \mid u \sqsubseteq v \text{ and } u, T \Downarrow^{-1} \rho, e\}$.

Now we show that $\text{tr-uneval}_{v,T}$ is in fact total and monotonic. We omit the v subscript from $\text{tr-uneval}_{v,T}$ when it is applied to a prefix of v and v is clear from the context.

Theorem 3 (Monotonic unevaluation function).

Suppose T explains v . Then:

1. For any $u \sqsubseteq v$, $\text{tr-uneval}_T(u)$ is defined.
2. If $u \sqsubseteq u' \sqsubseteq v$ then $\text{tr-uneval}_T(u) \sqsubseteq \text{tr-uneval}_T(u')$.

Proof. See Appendix, §A.1. □

Monotonicity means that the less output we demand, the less program we consume. Now we can show that tracing evaluation to v does indeed produce a trace able to explain v . For the sake of the proof, we show simultaneously that unevaluation after evaluation is deflationary: that the unevaluation of a value is smaller than the program which computed it.

Theorem 4 (Tracing evaluation produces explanations).

Suppose $\rho, e \Downarrow v, T$. Then T explains v . Moreover, for any $(\rho', e') \sqsubseteq (\rho, e)$:

$$\text{tr-uneval}_T(\text{eval}(\rho', e')) \sqsubseteq (\rho', e')$$

Proof. See Appendix, §A.2. □

5.3.3 Unevaluation computes least program slices

Finally we are in a position to show that, for any $\text{eval}_{\rho,e}$, our unevaluation algorithm computes $\text{uneval}_{\rho,e}$. First we make the following observation. If we are able to unevaluate v with T , in other words if T explains v , then for any sub-trace U of T which was used to unevaluate an intermediate value $u \neq \square$, we must also have had $U \neq \square$, since otherwise unevaluation would have got stuck. (Non-empty values need non-empty traces to guide their unevaluation.) But conversely, we can also observe that if U was used to unevaluate an intermediate value $u = \square$, then U was discarded in its entirety. (Empty values don't need traces at all.) In fact whenever T suffices to unevaluate v , any larger trace is equally good.

Lemma 6. Suppose S explains v . Then any $T \sqsupseteq S$ explains v . Moreover, for any $u \sqsubseteq v$, we have $\text{tr-uneval}_S(u) = \text{tr-uneval}_T(u)$.

Proof. See Appendix, §A.3. □

It is also useful to have a lemma which composes some of our previous observations.

Lemma 7. Suppose $\rho, e \Downarrow v, T$ and $(u, S) \sqsubseteq (v, T)$. If S explains u then $\text{tr-uneval}_S(u) \sqsubseteq (\rho, e)$.

Proof. Suppose $\rho, e \Downarrow v, T$ and $(u, S) \sqsubseteq (v, T)$ where S explains u . Then:

$$\begin{aligned} & \text{tr-uneval}_S(u) \\ = & \text{tr-uneval}_T(u) \quad (T \sqsupseteq S \text{ and Lemma 6}) \\ \sqsubseteq & \text{tr-uneval}_T(v) \quad (\text{Theorem 3}) \\ \sqsubseteq & (\rho, e) \quad (\text{Theorem 4}) \end{aligned}$$

□

Next, we show that unevaluating u produces a program slice for u .

Theorem 5 (Correctness of unevaluation). Suppose $\rho, e \Downarrow v, T$. If $(u, S) \sqsubseteq (v, T)$ and S explains u , then $\text{eval}(\text{tr-uneval}_S(u)) \sqsupseteq u$.

Proof. See Appendix, §A.4. □

It is now easy to see that $\text{eval}_{\rho, e}$ and $\text{tr-uneval}_{v, T}$ form a Galois connection for any $\rho, e \Downarrow_{\text{ref}} v, T$. And since each adjoint component of a Galois connection determines the other, it follows that $\text{tr-uneval}_T = \text{uneval}_{\rho, e}$ via Corollary 1.

Theorem 6 (Computation of least program slices). Suppose $\rho, e \Downarrow v, T$. For any $u \sqsubseteq v$ and any $(\rho', e') \sqsubseteq (\rho, e)$ we have:

$$\text{eval}(\rho', e') \sqsupseteq u \iff (\rho', e') \sqsupseteq \text{tr-uneval}_T(u)$$

Proof. For the \implies direction, suppose $\rho', e' \Downarrow u', S$ with $u' \sqsupseteq u$. Note that $S \sqsubseteq T$ and $u' \sqsubseteq v$ by monotonicity.

$$\begin{aligned} (\rho', e') & \sqsupseteq \text{tr-uneval}_S(u') && (\text{Theorem 4}) \\ & \sqsupseteq \text{tr-uneval}_S(u) && (\text{Theorem 3}) \\ & = \text{tr-uneval}_T(u) && (S \sqsubseteq T, \text{Lemma 6}) \end{aligned}$$

For the \impliedby direction, suppose $(\rho', e') \sqsupseteq \text{tr-uneval}_{v, T}(u)$. Then:

$$\begin{aligned} \text{eval}(\rho', e') & \sqsupseteq \text{eval}(\text{tr-uneval}_T(u)) && (\text{Theorem 2}) \\ & \sqsupseteq u && (\text{Theorem 5}) \end{aligned}$$

□

5.4 Trace slicing

Even at the level of programs, forward and backward slicing can reveal something of how a program works internally, by showing how different parts of the program are consumed by different parts of the output, and different parts of the output produced by different parts of the program. *Computation* slices, or trace slices, give a fuller picture, explaining in detail how these dependencies come about. A *forward* trace slice can be obtained simply by evaluating a program slice, using the tracing semantics; the resulting trace slice is a *forward explanation*, accounting for how that prefix of the program produces the prefix of the output that it does. Figure 2.13(b) in Chapter 2 gave an example of a forward trace-slice that the user obtained by forward-slicing an application of `zipW` and then evaluating it.

In this section we show how to calculate *backward* trace-slices, or backward explanations. A backward explanation accounts for how a prefix of the output consumes the prefix of the program that it does. A backward explanation is typically not the trace of any forward evaluation; in fact in Future Work, §7.2.5, we propose that these backward explanations be understood as traces of lazy computations. A backward trace-slice can be more informative than a plain program slice, because a given part of the program can be relinquished only when it is not used anywhere. (See the discussion at the end of §5.2.2 above.) A backward trace-slice records the individual uses of a given program part and thus can track how it is consumed differently at different points in the execution. In order to preserve the ability of the program to compute the required prefix of the output, the usage ultimately recorded in the program slice must be at least the join of these individual uses.

We illustrate backward trace-slices with the `zipW` example too. We ask the reader to consult Figure 2.14 from Chapter 2, which showed an application of `zipW` to a function and two lists; recall also the graphical notation for sharing we introduced in §2.3. In Figure 2.14(b), the programmer had selected the last two elements of the output list. We learned something from the resulting program slice, namely that the argument `op` in the recursive call to `zipW` was no longer needed. Figure 5.7 shows an exploration of the (backward) execution for that example. We can see `op` used in the outer call of `zipW` to construct the first element of the output, but also that inside the recursive call, where it is mentioned twice, it is no longer used, and therefore need not be passed as an argument. This now invites the question of why `op` was not passed to the second recursive call, and might lead the user to further exploration.

The other salient detail in the backward trace-slice is that the expression `op x y` which computed the value `Pair(13,5)` need not be executed. But this is not enough to relinquish `op x y` from the program slice, because it is still needed in the outermost `zipW` call. The general pattern is that program resources are consumed non-linearly (“forked”) during execution, and during backwards execution these various uses are “joined” via the join operation \sqcup . A function can be used differently each time it is called, but there is only one representation of the function in the program slice, and so the program slice can at best only contain the least upper bound of the individual uses. By contrast the trace slice shows each individual use.

The definition of backward trace-slicing is given in Figure 5.8. The judgement $v, T \searrow \rho, S$ states that backward-slicing the trace $\Gamma \vdash T : \tau$ with respect to a partial value $\vdash v : \tau$ yields partial environment $\Gamma \vdash \rho$ and partial trace S . As with unevaluation, there are side-conditions, which we omit from the figure for convenience, on the rules which take joins, asserting that the joins exist. Trace slicing is deterministic;

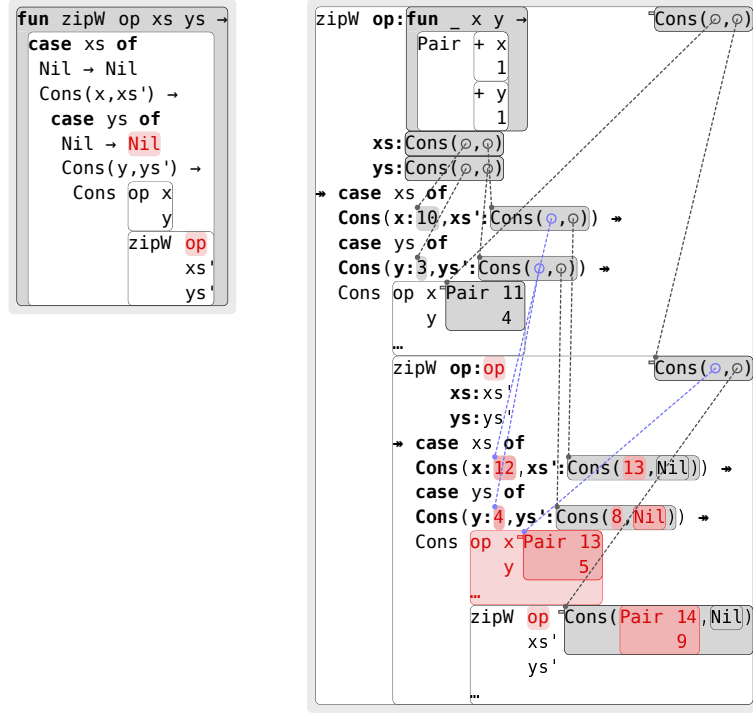


Figure 5.7 Backward trace-slice of `zipW` showing information that cannot be stored in the program slice

the proof is a straightforward induction, relying on the $v \neq \square$ side-conditions in Figure 5.8.

Lemma 8 (Determinism of trace slicing). *Suppose $v, T \searrow \rho, S$ and $v, T \searrow \rho', S'$. Then $\rho = \rho'$ and $S = S'$.*

One way to understand the algorithm is as a component of the unevaluation algorithm we gave in Figure 5.6, which can be factored into two phases. The first phase backward-slices the trace, discarding unneeded parts; the second phase then collapses the sliced trace into a program slice by discarding executed function bodies. The trace-slicing algorithm given in Figure 5.8 corresponds to the first phase.

Given a trace T which explains v , the algorithm calculates the smallest prefix S of T which preserves the “explanatory power” of the trace with respect to v , in that S retains sufficient information to unevaluate v . The trace-slicing rules are similar in flavour to those for unevaluation, but sub-computations are sliced rather than unevaluated back to expressions. The significant difference is the application case, where we both slice *and* unevaluate the executed function body T : we slice to obtain a *trace* slice S , and we unevaluate to obtain an *expression* slice e . The former is incorporated into the sliced application trace; the latter is merged into v_1 and used to slice T_1 . Unevaluation of the function body also yields an environment slice, but we disregard it; in Theorem 7 below we show that it is identical to the one obtained by slicing the function body.

We can slice T with v whenever T explains v . Moreover the partial environment we obtain is the one we would obtain via unevaluation. In the next section we will see that something like the converse is also true.

Theorem 7. $v, T \Downarrow^{-1} \rho, e \implies \exists S. v, T \searrow \rho, S.$

$$\boxed{v, T \searrow \rho, S \text{ where } \Gamma \vdash T : \tau}$$

$$\begin{array}{c}
\frac{}{\Box, T \searrow \Box_\Gamma, \Box} \quad \frac{}{v, x \searrow \Box_{\Gamma.x \mapsto v}, x} v \neq \Box \quad \frac{}{c, c \searrow \Box_\Gamma, c} \\
\\
\frac{c_2, T_2 \searrow \rho_2, S_2 \quad c_1, T_1 \searrow \rho_1, S_1}{v, T_1 \oplus_{c_1, c_2} T_2 \searrow \rho_1 \sqcup \rho_2, S_1 \oplus_{c_1, c_2} S_2} v \neq \Box \quad \frac{}{\langle \rho, \mathbf{fun} f(x).e \rangle, \mathbf{fun} f(x).e' \searrow \Box_\Gamma, \mathbf{fun} f(x).e} e \sqsubseteq e' \\
\\
\frac{v, T \searrow \rho[f \mapsto v_1][x \mapsto v_2], S \quad v, T \Downarrow^{-1} _, e \quad v_2, T_2 \searrow \rho_2, S_2 \quad v_1 \sqcup \langle \rho, \mathbf{fun} f(x).e \rangle, T_1 \searrow \rho_1, S_1}{v, T_1 T_2 \triangleright \langle \gamma, \mathbf{fun} f(x).T \rangle \searrow \rho_1 \sqcup \rho_2, S_1 S_2 \triangleright \langle \gamma, \mathbf{fun} f(x).S \rangle} v \neq \Box \\
\\
\frac{v_2, T_2 \searrow \rho_2, S_2 \quad v_1, T_1 \searrow \rho_1, S_1}{(v_1, v_2), (T_1, T_2) \searrow \rho_1 \sqcup \rho_2, (S_1, S_2)} \quad \frac{(v_1, \Box), T \searrow \rho, S}{v_1, \mathbf{fst} T \searrow \rho, \mathbf{fst} S} v_1 \neq \Box \quad \frac{(\Box, v_2), T \searrow \rho, S}{v_2, \mathbf{snd} T \searrow \rho, \mathbf{snd} S} v_2 \neq \Box \\
\\
\frac{v, T \searrow \rho, S}{\mathbf{inl} v, \mathbf{inl} T \searrow \rho, \mathbf{inl} S} \quad \frac{v, T \searrow \rho, S}{\mathbf{inr} v, \mathbf{inr} T \searrow \rho, \mathbf{inr} e} \\
\\
\frac{v, T_1 \searrow \rho_1[x_1 \mapsto v_1], S_1 \quad \mathbf{inl} v_1, T \searrow \rho, S}{v, \mathbf{case} T \text{ of } \{\mathbf{inl}(x_1).T_1; \mathbf{inr}(x_2).e_2\} \searrow \quad \rho_1 \sqcup \rho, \mathbf{case} S \text{ of } \{\mathbf{inl}(x_1).S_1; \mathbf{inr}(x_2).\Box\}} v \neq \Box \\
\\
\frac{v, T_2 \searrow \rho_2[x_2 \mapsto v_2], S_2 \quad \mathbf{inr} v_2, T \searrow \rho, S}{v, \mathbf{case} T \text{ of } \{\mathbf{inl}(x_1).e_1; \mathbf{inr}(x_2).T_2\} \searrow \quad \rho_2 \sqcup \rho, \mathbf{case} S \text{ of } \{\mathbf{inl}(x_1).\Box; \mathbf{inr}(x_2).S_2\}} v \neq \Box \\
\\
\frac{v, T \searrow \rho, S}{\mathbf{roll} v, \mathbf{roll} T \searrow \rho, \mathbf{roll} S} \quad \frac{\mathbf{roll} v, T \searrow \rho, S}{v, \mathbf{unroll} T \searrow \rho, \mathbf{unroll} S} v \neq \Box
\end{array}$$

Figure 5.8 Trace slicing

Proof. Straightforward induction on the derivation of $v, T \Downarrow^{-1} \rho, e$. The only non-trivial case is the application rule, because we invoke the \Downarrow^{-1} judgement from the \searrow judgement. Then we use that \Downarrow^{-1} is deterministic (Lemma 5). \square

5.4.1 Computation of least explanations

The key correctness property of trace slicing with v is that it preserves the ability of the trace to explain v . But it also produces an explanation *consistent with* the original explanation, in the sense of being smaller than it.

Theorem 8 (Correctness of trace slicing). *If $v, T \searrow \rho, S$ then S explains v and $S \sqsubseteq T$.*

Proof. See Appendix, §A.5. \square

Now we can make an observation about the application rule for \searrow where we both slice the function body T to obtain S and unevaluate T to obtain e , where v is the partial output used for both. By Theorem 8, we know that S explains v and moreover by Lemma 6 that unevaluation with S is the same unevaluation with T . Therefore, we could equally have used S instead of T to obtain e . Then the effect of unevaluation would be just to discard function bodies, because the trace guiding the unevaluation would have already been sliced.

By Lemma 6 we can also see that because slicing a trace with v always yields a smaller trace that explains v , the larger trace must also explain v . By determinism the judgements agree on environments.

Corollary 2. $v, T \searrow \rho, S \implies \exists e. v, T \Downarrow^{-1} \rho, e$.

Proof. Suppose $v, T \searrow \rho, S$. Then S explains v and $S \sqsubseteq T$ by Theorem 8. But if $S \sqsubseteq T$, then T also explains v by Lemma 6, and so there exist ρ' and e such that $v, T \Downarrow^{-1} \rho', e$. Then $\rho = \rho'$ by Theorem 7 and the determinism of \searrow (Lemma 8). \square

When we slice T with v we obtain the canonical explanation of v compatible with T , in that it is the smallest prefix of T which still explains v :

Theorem 9 (Trace slicing yields the smallest compatible explanation). *Suppose $v, T' \searrow \rho, S$, and any $T \sqsubseteq T'$ that explains v . Then $S \sqsubseteq T$.*

Proof. See Appendix, §A.6. \square

These “least explanations” are the slices the user sees in LambdaCalc when they backward-slice a computation, although they are typically presented in a differential form which highlights the difference between the present explanation, and a hypothetical future one in which less of the output is needed. In the next chapter, we consider a more general form of differential computation, where the changes are not merely increasing or decreasing but involve structural rearrangements of the computation.

6 Differencing Computation

One man’s constant is another man’s variable.

Alan Perlis, *Epigrams on Programming* [Per82]

In Chapter 2 we explained a “what if” question as an interactive query concerning the relationship between two computations. When we *edit* a program we usually have a “what if” of some sort in mind, perhaps only tacitly. Sometimes we lack a clear idea what will happen, so we make the change and *see* what happens. Other times, we have an idea of what should happen, so we make the change in order to *check* what happens. Test-driven development [Bec02] codifies the latter style of question into a programming methodology. Normally all we get to see is some new output, with no indication of what parts of the output are different, or what happened differently during the execution to account for the output difference. Moreover, if the question comes to mind in the middle of debugging or testing, then to ask the question we usually have to discard the very context that prompted it in the first place: we have to stop the program, apply the change, and restart. Live programming environments improve on this aspect of the problem (Related Work, §3.7), but provide no visibility on what changed or why.

We propose explicit change as a pervasive feature of an execution environment. The idea is that every change in a computed value should be *accounted for* by some explicit change in the execution. This is a *differential* notion of the kind of computational transparency that we have already advocated: no changes go unrecorded or unexplained. In this chapter, we define a key component of such a system, a notion of *differential evaluation* which derives execution differences from program differences. Differential evaluation implements the *retroactive update* scheme introduced in Chapter 2. It can be used to apply a change to a program whilst debugging or testing, without having to throw away the all-important context that prompted the change. To be practical, differential evaluation requires an efficient incremental implementation; we defer this to future work (§7.2.2). But in Appendix B we show that it is possible to put differential execution to use without an incremental implementation.

Before we explain our solution, let us briefly consider the problem of comparing executions. Differencing generally has two phases: an *alignment* phase which decides how parts of the two structures correspond to each other, and then a phase which constructs a delta transforming one structure into the other, assuming the alignments derived in the first phase. The computationally hard part of the problem is usually alignment. For tree-structured data, like computations, there is usually no unique optimal way, in the sense of

minimising the resulting delta, to align the two computations. And even when there is, it will generally be very hard to compute: even sequence-differencing is NP-complete if move operations are permitted [SS02], and tree differencing can clearly encode sequence differencing if the deltas allow siblings to be reordered. So for execution differencing to be tractable, we must forgo optimal alignment and instead pursue approaches which are simpler but either somewhat ad hoc or able to utilise additional information to guide the alignment. This general approach is sometimes called *execution indexing* (Related Work, §3.12), because the idea is to construct an assignment of indices to nodes with the intention of aligning nodes across computations when they have the same index. Prior work in execution indexing explores several such schemes, including counting the invocations of a particular statement, and matching the execution context against a context-free grammar.

The key novelty of our approach to this problem is that we derive our execution deltas from *program deltas*. The program deltas are created automatically as the programmer edits the program. Program deltas simplify the problem by providing additional information for our indexing scheme to exploit. Our indexing scheme is deterministic, connecting execution nodes back to the program nodes they come from, allowing us to derive computation deltas that reflects the deltas the user applies to the program. For example, our approach can interpret certain program changes as structural reorganisations of the execution, such as the splicing in of new operations or the re-ordering of computations, which no differencing algorithm for pure tree-structured values would be likely to infer.

The following example shows that, without this extra information, it is often unclear how to align nodes, even with trivial programs and executions which seem obviously related. Then we will see how taking the program delta into account simplifies the problem. The reader will recall the factorial example from Chapter 2. In Figure 2.5(a), we showed a precursor of factorial which implemented the constant function 1 by counting down from its argument x to 0 and then just returning 1. In Figure 6.1, we show the application of this function to 0 and to 1 side-by-side.

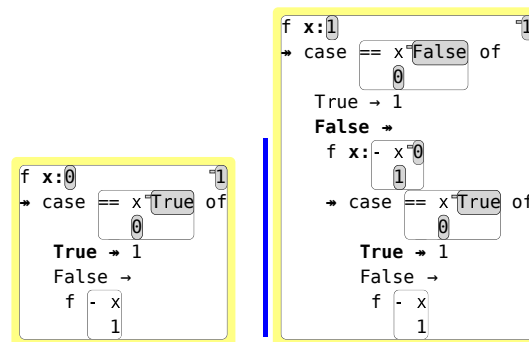


Figure 6.1 A plausible alignment of a pair of executions

Now consider how we might align nodes in these two very simple computations, for comparison purposes. We might note that, modulo one small difference, the entire left-hand side, from the execution of f 0 downwards, is a *suffix* (highlighted with the blue bar) of the right-hand side. The single difference is that

on the left x is bound to the constant 0 , whereas on the right it is bound to the expression $x - 1$, which also happens to have the value 0 . So purely syntactically, one plausible alignment would entail $f\ 0$ on the left becoming $f\ (x - 1)$ on the right.

In Figure 6.2(a) we see a different alignment decision. Modulo a few differences, the entire left-hand side also appears as a *prefix* (again highlighted with a blue bar) of the right-hand side. Under this alignment, the $f\ 0$ on the left-hand side becomes $f\ 1$ on the right. Additionally, the expression $x == 0$ which has the value `True` on the left has the value `False` on the right; and the recursive call on the left is dead, whereas the its counterpart on the right is live with an executed function body underneath. This alignment implies a bit more “work” to get from the left-hand side to the right-hand side than did the alignment in Figure 6.1. Nevertheless the choice here is quite plausible.

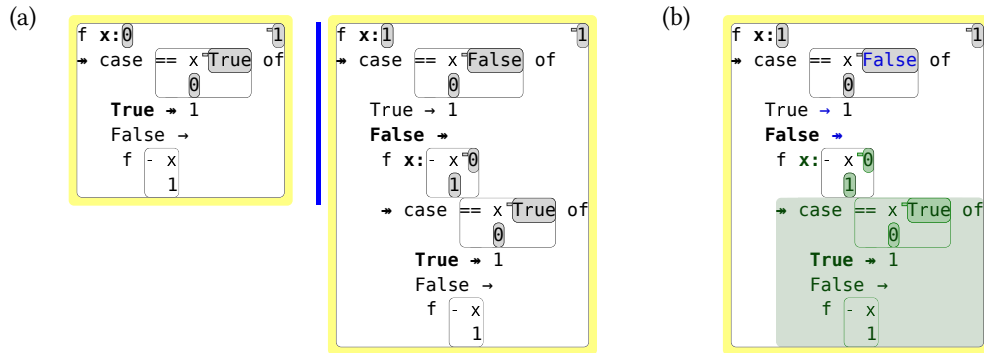


Figure 6.2 (a) Another plausible alignment of the executions from Figure 6.1; (b) execution delta derived in LambdaCalc

But now suppose we also know how the user edited the underlying program: they edited $f\ 0$ into $f\ 1$ by changing the 0 into a 1 . This is consistent with the second alignment but not with the first, where the execution of $f\ 0$ became the execution of $f\ (x - 1)$ rather than the execution of $f\ 1$. So in fact although the first alignment leads to a smaller delta, the resulting delta is not faithful to the edit made by the programmer. Figure 6.2(b) shows the final delta we compute in LambdaCalc based on this program edit.

In summary, utilising the program delta when deriving a computation delta not only simplifies the alignment or execution indexing problem, but also permits computation deltas to reflect the edits made by the programmer. We now describe how this works in more detail. §6.1 extends the syntax of the reference language and the syntax of traces from Chapter 4 with *indices* which serve to identify nodes. §6.2 extends our previous tracing semantics into an *indexing* tracing semantics which evaluates an indexed program to an indexed value and an indexed trace. Then in §6.3 we show how to compare two indexed expressions, value or traces to obtain a delta expression, value or trace.

6.1 Indexed syntax

We start by requiring that a program be annotated with *indices* identifying its individual parts. As noted in Related Work, §3.12, this assumption is easily justified by thinking of the program as represented in a store

| | |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Index | $\alpha ::= i \mid \alpha_1 : \alpha_2$ |
| Indexed expression | $e ::= r^\alpha$ |
| Raw expression | $r ::= x \mid () \mid c \mid e_1 \oplus e_2 \mid \text{fun } f(x).e \mid e_1 e_2$
$\mid (e_1, e_2) \mid \text{fst } e \mid \text{snd } e \mid \text{inl } e \mid \text{inr } e$
$\mid \text{case } e \text{ of } \{\text{inl}(x_1).e_1; \text{inr}(x_2).e_2\}$
$\mid \text{let } x = e_1 \text{ in } e_2 \mid \text{roll } e \mid \text{unroll } e$ |
| Indexed trace | $T ::= R^\alpha$ |
| Raw trace | $R ::= x \mid c \mid T_1 \oplus T_2 \mid (T_1, T_2)$
$\mid \text{fst } T \mid \text{snd } T \mid \text{inl } T \mid \text{inr } T$
$\mid \text{case } T \text{ of } \{\text{inl}(x_1).T_1; \text{inr}(x_2).T_2\}$
$\mid \text{case } T \text{ of } \{\text{inl}(x_1).e_1; \text{inr}(x_2).T_2\}$
$\mid \text{fun } f(x).e \mid T_1 T_2 \triangleright \langle \gamma, \text{fun } f(x).T \rangle$
$\mid \text{let } x = T_1 \text{ in } T_2 \mid \text{roll } T \mid \text{unroll } T$ |
| Indexed value | $v ::= u^\alpha$ |
| Raw value | $u ::= c \mid (v_1, v_2) \mid \text{inl } v \mid \text{inr } v \mid \text{roll } v$
$\mid \langle \rho, \text{fun } f(x).e \rangle$ |
| Raw environment | $\rho ::= \bullet \mid \rho[x \mapsto v]$ |

Figure 6.3 Indexed expressions, traces, values and environments

assigning a unique address to every sub-expression, as might be the case with a structure-aware editor. We can simply use the addresses as the indices.

Figure 6.3 gives the syntax of indices α , plus the syntax of indexed expressions, values, traces and environments. The indices on the source program are of the form i , where i is drawn from a countably infinite set of *source indices*. The index assignments that arise during execution, as we are going to see, also include indices of the form $\alpha_1 : \alpha_2$ where α_1 and α_2 are indices; that is to say binary trees with source indices at the leaves. Our intention is that two traces only be assigned the same index within a single closed computation if they represent sub-computations that behave identically.

The syntax of indexed expressions follows the reference language extended with traces as defined in Chapter 4. Indexed expressions, which carry indices, are defined mutually inductively with *raw* expressions, which do not. We use a similar pattern for indexed values and indexed traces. Environments only come in a “raw” form, mapping identifiers to indexed values; there are no indexed environments. Apart from the indices themselves, the indexed syntax differs from the reference language only in introducing an explicit `let` form. Under the indexing semantics, `let` does not so easily desugar into a local function application.

It is sometimes convenient to write the function `fun $f(x).e$` as `fun f,x (e)`, the application $e_1 e_2$ as `app(e_1, e_2)`, the variable x as `var x` , and so on, in order to make the syntactic constructor explicit. The general form is `c(e_1, \dots, e_n)`, where c is a constructor and e_1, \dots, e_n are the immediate sub-expressions. The notation is useful for defining operations like differencing that treat programs and their executions as

data. Here, we can use it to define the *erasure* of an indexed expression e , which recovers a normal expression from e by erasing indices.

Definition 8 (Erasure of an indexed expression).

$$|c(e_1, \dots, e_n)^\alpha| \stackrel{\text{def}}{=} c(|e_1|, \dots, |e_n|)$$

Indexed expressions are typed using a judgement $\Gamma \vdash_\circ e : \tau$ which is defined inductively. There are similar \vdash_\circ judgements for values, traces and environments. We omit the definitions, since they are almost identical to the \vdash typing judgements for the reference language given in Figure 4.1. The reference typing and indexed typing judgements agree if we ignore indices.

Lemma 9 (Agreement with referencing typing).

1. If $\Gamma \vdash_\circ e : \tau$ then $\Gamma \vdash |e| : \tau$.
2. If $\Gamma \vdash e : \tau$ and $|e'| = e$ then $\Gamma \vdash_\circ e' : \tau$.

and similarly for indexed values, traces and environments.

We now introduce some operations which more explicitly treat the indices as locations. For an indexed expression, value or trace to be suitable for differencing, it must satisfy the following *injective indexing* property: any two sub-expressions, sub-values or sub-traces which have the same index must be syntactically equal. The importance of this requirement for execution differencing is also noted by Xin et al. [XSZ08]. When differencing two computations, we need to be able to say how a node differs from its counterpart with the same index in the other computation. This only makes sense if the index identifies a unique (although possibly shared) structure in each computation.

To specify injective indexing more precisely, we define the symbol $\&$ (“index of”) to denote the function which takes any indexed expression e^α to α . Similar overloaded definitions hold for indexed values and indexed traces, and for the definitions which follow. We write $e_1 \preceq e_2$ to mean that e_1 is a sub-term of e_2 .

Definition 9 (Indexing of e). The *indexing* $\&|_e$ of an expression e is $\&$ domain-restricted to the indexed sub-expressions of e :

$$\&|_e \stackrel{\text{def}}{=} \{r^\alpha \mapsto \alpha \mid r^\alpha \preceq e\}$$

Then the requirement that e have an injective indexing can be stated as the requirement that $\&|_e$ be injective. If e has an injective indexing, then it can also be represented in a less redundant way: as a *store* mapping each index in e to the “local” information associated with that index, namely a constructor and some child indices which are also in e . We shall see later that treating indexed expressions as stores is useful for differencing.

Definition 10 (Store for an indexed expression). Suppose e is an indexed expression with $\&|_e$ injective. Then define:

$$\underline{e} \stackrel{\text{def}}{=} \{\alpha \mapsto c(\alpha_1, \dots, \alpha_n) \mid c(r_1^{\alpha_1}, \dots, r_n^{\alpha_n})^\alpha \preceq e\}$$

Figure 6.4 gives an example of these stores. In the upper half of the figure are two expressions e_1 and e_2 , each injectively indexed. (In fact each satisfies a stronger property, namely that each index only occurs

once.) The small red boxes give the indices assigned to each node. (For the rest of this chapter, we will adopt this graphical notation for expressions, in preference to LambdaCalc visualisations. It is less compact, but more explicit about child pointers, which suits the topic of the chapter.) In the lower half of the figure are the corresponding stores e_1 and e_2 . The reader may have noticed that e_2 and e_1 have some indices in common. This happens when the programmer creates a new expression by editing an existing one. Here, the programmer obtained e_2 from e_1 by a series of refactorings: first they inlined the variables x and y , obtaining the expression f (5, 6), an intermediate state which is not shown; then they extracted the pair (5, 6) as the value of a new local variable z . We will use this scenario as a running example.

We give one last definition before we present indexed evaluation. We define a function on indexed expressions called *instantiation* with α ; instantiating an indexed expression e with an index α “offsets” the index of each sub-expression of e by α .

Definition 11 (Instantiating an expression by an index).

$$[c(e_1, \dots, e_n)^{\alpha'}]_{\alpha} \stackrel{\text{def}}{=} c([e_1]_{\alpha}, \dots, [e_n]_{\alpha})^{\alpha':\alpha}$$

The role of instantiation will become clear in the next section. For now we just note that instantiation only transforms indices and so preserves erasure. Instantiation also preserves injectivity of indexing.

6.2 Indexing evaluation

To calculate an execution delta, we need two indexed traces. An indexed trace is built from an indexed program by *indexing evaluation*, which is defined in Figure 6.5. The judgement $\rho, e \Downarrow_{\circ} v, T$ states that indexed expression $\Gamma \vdash_{\circ} e : \tau$ evaluates in environment $\Gamma \vdash_{\circ} \rho$ to indexed value $\vdash_{\circ} v : \tau$ and indexed trace $\Gamma \vdash_{\circ} T : \tau$. Indexing evaluation uses the indices on the source program to deterministically assign indices to every trace and every computed value. As mentioned above, the indices on traces and values can be thought of as *locations* which allow both the *sharing* of equally-valued things within a computation, and the *comparison* of differently-valued things across different computations.

We now state two basic properties of indexing evaluation. First, indexing evaluation is deterministic, which is a straightforward induction given that $[-]_{\alpha}$ is a function.

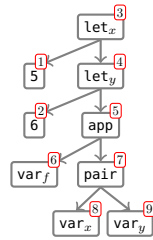
Lemma 10 (Determinism). *Suppose $\rho, e \Downarrow_{\circ} v, T$ and $\rho, e \Downarrow_{\circ} v', T'$. Then $(v, T) = (v', T')$.*

Second, indexing evaluation agrees with tracing evaluation extended with `let`, if we ignore the indices.

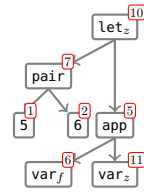
Lemma 11 (Agreement with tracing evaluation).

1. *If $\rho, e \Downarrow_{\circ} v, T$ then $|\rho|, |e| \Downarrow |v|, |T|$.*
2. *If $\rho, e \Downarrow_{\circ} v, T$ and $|\rho'| = \rho$ and $|e'| = e$ then $\rho', e' \Downarrow_{\circ} v', T'$ with $|v'| = v$ and $|T'| = T$.*

Proof. Straightforward induction in each direction, using the equalities $|\rho|(x) = |\rho(x)|$ for the variable case, $|\rho|[x \mapsto v] = |\rho[x \mapsto v]|$ for the binder cases, and $||e|_{\alpha}| = |e|$ for the application case. \square



$|e_1| \stackrel{\text{def}}{=} \text{let } x = 5 \text{ in let } y = 6 \text{ in } f(x, y)$



$|e_2| \stackrel{\text{def}}{=} \text{let } z = (5, 6) \text{ in } f z$

| | | |
|---|---|-------------------------|
| 1 | ↦ | 5 |
| 2 | ↦ | 6 |
| 3 | ↦ | let _x (1, 4) |
| 4 | ↦ | let _y (2, 5) |
| 5 | ↦ | app(6, 7) |
| 6 | ↦ | var _f |
| 7 | ↦ | pair(8, 9) |
| 8 | ↦ | var _x |
| 9 | ↦ | var _y |

e_1

| | | |
|----|---|-------------------------|
| 1 | ↦ | 5 |
| 2 | ↦ | 6 |
| 5 | ↦ | app(6, 11) |
| 6 | ↦ | var _f |
| 7 | ↦ | pair(1, 2) |
| 10 | ↦ | let _z (7, 5) |
| 11 | ↦ | var _z |

e_2

Figure 6.4 Injectively indexed expressions e_1 and e_2 as stores e_1 and e_2

$$\boxed{\rho, e \Downarrow_{\circ} v, T}$$

$$\begin{array}{c}
\frac{}{\rho, x^{\alpha} \Downarrow_{\circ} \rho(x), x^{\alpha}} \quad \frac{}{\rho, c^{\alpha} \Downarrow_{\circ} c^{\alpha}, c^{\alpha}} \quad \frac{\rho, e_1 \Downarrow_{\circ} c_1^{\alpha_1}, T_1 \quad \rho, e_2 \Downarrow_{\circ} c_2^{\alpha_2}, T_2}{\rho, (e_1 \oplus e_2)^{\alpha} \Downarrow_{\circ} (c_1 \hat{\oplus} c_2)^{\alpha}, (T_1 \oplus T_2)^{\alpha}} \\
\\
\frac{\rho, e_1 \Downarrow_{\circ} v_1, T_1 \quad \rho[x \mapsto v_1], e_2 \Downarrow_{\circ} v, T_2}{\rho, (\text{let } x = e_1 \text{ in } e_2)^{\alpha} \Downarrow_{\circ} v, (\text{let } x = T_1 \text{ in } T_2)^{\alpha}} \quad \frac{}{\rho, (\text{fun } f(x).e)^{\alpha} \Downarrow_{\circ} \langle \rho, \text{fun } f(x).e \rangle^{\alpha}, (\text{fun } f(x).e)^{\alpha}} \\
\\
\frac{\rho, e_1 \Downarrow_{\circ} v_1, T_1 \quad \rho, e_2 \Downarrow_{\circ} v_2, T_2 \quad \rho'[f \mapsto v_1][x \mapsto v_2], [e]_{\alpha_2} \Downarrow_{\circ} v, T}{\rho, (e_1 e_2)^{\alpha} \Downarrow_{\circ} v, (T_1 T_2 \triangleright \langle \text{vars}(\rho'), \text{fun } f(x).T \rangle)^{\alpha}} v_1 = \langle \rho', \text{fun } f(x).e \rangle^{\alpha_1}, v_2 = u_2^{\alpha_2} \\
\\
\frac{\rho, e_1 \Downarrow_{\circ} v_1, T_1 \quad \rho, e_2 \Downarrow_{\circ} v_2, T_2}{\rho, (e_1, e_2)^{\alpha} \Downarrow_{\circ} (v_1, v_2)^{\alpha}, (T_1, T_2)^{\alpha}} \quad \frac{\rho, e \Downarrow_{\circ} (v_1, v_2)^{\alpha'}, T}{\rho, (\text{fst } e)^{\alpha} \Downarrow_{\circ} v_1, (\text{fst } T)^{\alpha}} \quad \frac{\rho, e \Downarrow_{\circ} (v_1, v_2)^{\alpha'}, T}{\rho, (\text{snd } e)^{\alpha} \Downarrow_{\circ} v_2, (\text{snd } T)^{\alpha}} \\
\\
\frac{\rho, e \Downarrow_{\circ} v, T}{\rho, (\text{inl } e)^{\alpha} \Downarrow_{\circ} (\text{inl } v)^{\alpha}, (\text{inl } T)^{\alpha}} \quad \frac{\rho, e \Downarrow_{\circ} v, T}{\rho, (\text{inr } e)^{\alpha} \Downarrow_{\circ} (\text{inr } v)^{\alpha}, (\text{inr } T)^{\alpha}} \\
\\
\frac{\rho, e \Downarrow_{\circ} (\text{inl } v_1)^{\alpha'}, T \quad \rho[x_1 \mapsto v_1], e_1 \Downarrow_{\circ} v, T_1}{\rho, (\text{case } e \text{ of } \{\text{inl}(x_1).e_1; \text{inr}(x_2).e_2\})^{\alpha} \Downarrow_{\circ} v, (\text{case } T \text{ of } \{\text{inl}(x_1).T_1; \text{inr}(x_2).T_2\})^{\alpha}} \\
\\
\frac{\rho, e \Downarrow_{\circ} (\text{inr } v_2)^{\alpha'}, T \quad \rho[x_2 \mapsto v_2], e_2 \Downarrow_{\circ} v, T_2}{\rho, (\text{case } e \text{ of } \{\text{inl}(x_1).e_1; \text{inr}(x_2).e_2\})^{\alpha} \Downarrow_{\circ} v, (\text{case } T \text{ of } \{\text{inl}(x_1).T_1; \text{inr}(x_2).T_2\})^{\alpha}} \\
\\
\frac{\rho, e \Downarrow_{\circ} v, T}{\rho, (\text{roll } e)^{\alpha} \Downarrow_{\circ} (\text{roll } v)^{\alpha}, (\text{roll } T)^{\alpha}} \quad \frac{\rho, e \Downarrow_{\circ} (\text{roll } v)^{\alpha'}, T}{\rho, (\text{unroll } e)^{\alpha} \Downarrow_{\circ} v, (\text{unroll } T)^{\alpha}}
\end{array}$$

Figure 6.5 Tracing evaluation with indexing

Type preservation is then immediate from Lemmas 11 and 9. Given Lemma 11, the tracing aspect of the judgement is covered by §4.3, so here we need only explain how indices are assigned.

Our goal will be for there to be *only one way to evaluate a given indexed expression within a closed computation*. More precisely, if $\rho, e \Downarrow_{\circ} v, T$ and $\rho', e \Downarrow_{\circ} v', T'$ both occur in the evaluation of a closed computation, then $\rho = \rho'$. Then we can adopt the following simple policy for assigning indices to traces and values. The index of a trace is always the index of the expression of which it is the unrolling; and the index of a value is always the index of the expression which constructed it. The injectivity of indexing should then follow from determinism, although we leave this as a conjecture (Conjecture 1 below).

Not every indexing evaluation rule specifies the index on the value it returns. If the rule returns a value computed elsewhere, then that value already has an index and we simply return that indexed value unchanged. This increases the likelihood of reuse of indices both within and across computations.

There are two things we need to do to meet our requirement there be only “one way” to execute a given indexed expression within a single computation. First there must be no *sharing* in any expression that we evaluate. Consider the indexed program $(\text{let } x = 8 \text{ in } (f^2 x^3)^1, \text{let } x = 9 \text{ in } (f^2 x^3)^1)$. We have shown indices, as superscripts, only on the nodes which occur more than once. Here the indexed application $(f^2 x^3)^1$ is included twice, once in a context where x is bound to 8, and once in a context where x is bound to 9. When this code is run, each occurrence of the application will be inflated into an application trace containing an executed function body. But the two executed bodies will generally differ, breaking injectivity of trace indexing.

Definition 12 (Sharing-free). An indexed expression e is *sharing-free* if each index in e occurs only once.

If e is sharing-free, then clearly $\&|_e$ is injective. Moreover instantiation preserves sharing-freeness.

The other opportunity for an indexed expression e to run in different environments within the same computation is when e is in the body of a closure f , because then f can be invoked with different arguments. To eliminate this possibility, in the application rule, before running the body e' of f , we *instantiate* e' with the index of v to obtain $[e']_{\&v}$, which is a copy of e' unique to v . If we can assume that any argument passed to f with the same location as v will be equal to v , then instantiation, along with the sharing-freeness of e' (which is preserved by instantiation), is sufficient to ensure that any $e \preceq [e']_{\&v}$ will run in a unique environment. By the time we are running the body of any closure, the original expression has been instantiated with the indices of the arguments to all containing functions.

To illustrate indexing for values and closure bodies, we revisit the example indexed program e_1 from the top-left of Figure 6.4. In Figure 6.6(a) and (b), we run e_1 in both the tracing semantics and the indexing semantics, in an environment ρ binding f to the closure $\langle \bullet, \text{fun } f(w).(\text{fst } w^{14})^{13} \rangle^{12}$ returning the first component of its argument. The white nodes are traces; the grey nodes are values. In (a), we evaluate $|e|$ in $|\rho|$; in (b) we evaluate e in ρ . In (b), when a trace returns a value computed elsewhere in the part of the computation we can see, it is shown with a dotted border and no contents, rather than as a sharing link. For example, both lets, the app, and the executed function body all return the indexed value 5^1 . This is consistent with the reference semantics, if we erase the indices. Similarly, the variables x and y evaluate to 5^1 and 6^2 and these are then mentioned directly by the pair. The variable f also returns a value constructed elsewhere, the closure with index 12, but we do not use a dotted border for the closure because

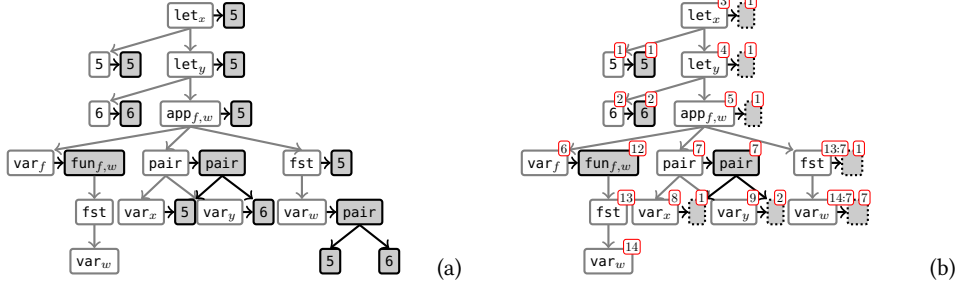


Figure 6.6 Traces resulting from evaluation in (a) tracing semantics and (b) indexing semantics

it was constructed by a part of the computation we cannot see. The remaining trace nodes build values, rather than reusing them, and assign them the index of the expression, which is the same as that of the trace. Instantiation occurs in (b) when we apply the closure f to the pair argument with index 7. We instantiate the body of f with the argument index before evaluating it, so that $(\text{fst } w^{14})^{13}$ becomes $(\text{fst } w^{14:7})^{13:7}$.

We now state our conjecture. We would like that, given a sharing-free closed program to start with, indexing evaluation always yields an output with an injective indexing.

Conjecture 1. *Suppose $\bullet, e \Downarrow_{\circ} v, T$ with e sharing-free. Then $\&_{|(v,T)}$ is injective.*

To summarise the intuition for the conjecture, there is a unique environment in which a given indexed expression within a closed computation is evaluated, for the following reasons. Within the execution of a closure body with a particular argument, an indexed expression cannot occur twice, because the original program was sharing-free and instantiation preserves sharing-freeness. The same expression cannot be executed in different environments as a result of applying the same closure to different arguments, because the body of the closure is always instantiated with the argument index before being evaluated. If every indexed expression runs in a unique environment, then by determinism its trace is unique and the value it constructs, if any, is unique.

What has proved tricky about the conjecture and likely to require further work to establish is how to generalise the statement to open programs, so that it can serve as a suitable induction hypothesis. Naïvely for example one might think that it would be sufficient for $\&_{|(\rho,e)}$ to be injective, and for e and any expressions contained within ρ to be sharing-free. However this is not the case. The following program shows what goes wrong without further constraints on the environment. Suppose we have the following closure and environment:

$$v_1 = \langle \bullet, \text{fun } f(x).(\text{inl } x^4)^1 \rangle^3 \quad \rho \stackrel{\text{def}}{=} \{y \mapsto 4^{1:2}, f \mapsto v_1\}$$

Suppose we then use ρ to evaluate the indexed expression $(y^5, (f^6 3^2))^7$. The result will be a pair containing the value $4^{1:2}$ from the environment, but also a different value with the same index. This violates injectivity on the output, not because any part of the input violates injectivity, or because there is any sharing within expressions, but because the execution is *inconsistent* with the information implied by the structure of the indices already in ρ . The derivation is shown below; we omit the trace component for clarity, since

the problem is apparent just with values:

$$\frac{\rho, y^5 \Downarrow_{\circ} 4^{1:2} \quad \frac{\rho, f^6 \Downarrow_{\circ} v_1 \quad \rho, 3^2 \Downarrow_{\circ} 3^2 \quad \frac{\bullet[f \mapsto v_1][x \mapsto 3^2], x^{4:2} \Downarrow_{\circ} 3^2}{\bullet[f \mapsto v_1][x \mapsto 3^2], ((\text{inl } x^4)^1)_2 \Downarrow_{\circ} (\text{inl } 3^2)^{1:2}}}{\rho, (f^6 3^2)^7 \Downarrow_{\circ} (\text{inl } 3^2)^{1:2}}}{\rho, (y^5, (f^6 3^2)^7)^8 \Downarrow_{\circ} (4^{1:2}, (\text{inl } 3^2)^{1:2})^8}$$

The problem here is that the environment ρ already contains the indexed value $4^{1:2}$. However, the `inl` expression which forms the closure body has index 1, and the argument passed to the closure has index 2. When we instantiate the closure body with the argument index before evaluating it, we append 2 to the index of every expression in the body, obtaining the expression $(\text{inl } x^{4:2})^{1:2}$. When this is then evaluated, it produces a `inl` value whose index is $1 : 2$. This is finally plugged into a pair, along with the value $4^{1:2}$ from the environment which already has that index. Since the two components of the pair have the same index but different raw contents, injectivity is violated.

The subtlety is that the indices in an environment say something about the computation which has already taken place. In this case, the indexed value $4^{1:2}$ is really claiming to have been built by evaluating 4^1 within the body of a function applied to an argument with index 2. This is inconsistent with the fact that the expression we then evaluated in the context of ρ involved evaluating a different constructor with the same index. Thus, if we are to prove the conjecture by induction on the \Downarrow_{\circ} derivation, there is work to do to better understand the invariants on open programs.

6.3 Delta computations

Given two indexed expressions, traces or values, we can *difference* them to obtain a delta expression, delta trace or a delta value. As usual, we define things for indexed expressions and assume that the definitions extend to traces and values.

A indexed delta-expression is an indexed expression with some additional colour indicating how it differs from another indexed expression. It has the same syntax as an indexed expression, except that every constructor and every index α has been coloured *green* to mean “new”, *blue* to mean “changed”, or has been left black to mean “unchanged”, following the colour scheme introduced in Chapter 2. If a constructor is marked as new, then the expression itself is new, and all the child indices of that delta expression are also new. This is because the indices on the children of a node represent *pointers* to those children, with the pointers being part of the parent; thus if the parent is new, its child pointers are necessarily new. The root index is always left black, because there is no parent that it is part of. Figure 6.7 defines the (asymmetric) differencing operation. For expressions e and e' with injective indexing, $e \ominus e'$ is the indexed delta-expression e'' which colours e to show what is new or changed relative to e' . Note that $\text{dom}(e \ominus e') = \text{dom}(\underline{e})$, because the delta is asymmetric.

The upper half of Figure 6.8 shows the two programs from Figure 6.4 being compared, with $e_1 \ominus e_2$ on the left and $e_2 \ominus e_1$ on the right. The refactoring that we described earlier is more explicit now: looking at the right-hand delta, for example, we can see that the `let` binding for z , and the use of z , are both new, but that

$$\boxed{e \ominus e'}$$

$$e \ominus e' \stackrel{\text{def}}{=} e''$$

$$\text{where } \underline{e}'' = \left\{ \alpha \mapsto \begin{cases} \underline{e}(\alpha) \ominus \underline{e}'(\alpha) & \text{if } \alpha \in \text{dom}(\underline{e}') \\ \mathbf{c}(\alpha_1, \dots, \alpha_n) & \text{if } \alpha \notin \text{dom}(\underline{e}') \text{ and } \underline{e}(\alpha) = c(\alpha_1, \dots, \alpha_n) \end{cases} \middle| \alpha \in \text{dom}(\underline{e}) \right\}$$

$$\boxed{\mathbf{c}(\alpha_1, \dots, \alpha_n) \ominus \mathbf{c}'(\alpha'_1, \dots, \alpha'_m)}$$

$$\begin{aligned} \mathbf{c}(\alpha_1, \dots, \alpha_n) \ominus \mathbf{c}'(\alpha'_1, \dots, \alpha'_m) &\stackrel{\text{def}}{=} \mathbf{c}(\alpha_1 \ominus \alpha'_1, \dots, \alpha_n \ominus \alpha'_n) \\ \mathbf{c}(\alpha_1, \dots, \alpha_n) \ominus \mathbf{c}'(\alpha'_1, \dots, \alpha'_m) &\stackrel{\text{def}}{=} \begin{cases} \mathbf{c}(\alpha_1 \ominus \alpha'_1, \dots, \alpha_n \ominus \alpha'_n) & \text{if } n \geq m \\ \mathbf{c}(\alpha_1 \ominus \alpha'_1, \dots, \alpha_n \ominus \alpha'_n, \alpha_{n+1}, \dots, \alpha_m) & \text{otherwise} \end{cases} \\ &\text{where } \mathbf{c} \neq \mathbf{c}' \end{aligned}$$

$$\boxed{\alpha \ominus \alpha'}$$

$$\begin{aligned} \alpha \ominus \alpha &\stackrel{\text{def}}{=} \alpha \\ \alpha \ominus \alpha' &\stackrel{\text{def}}{=} \alpha \\ \text{where } \alpha &\neq \alpha' \end{aligned}$$

Figure 6.7 Asymmetric difference of indexed expressions

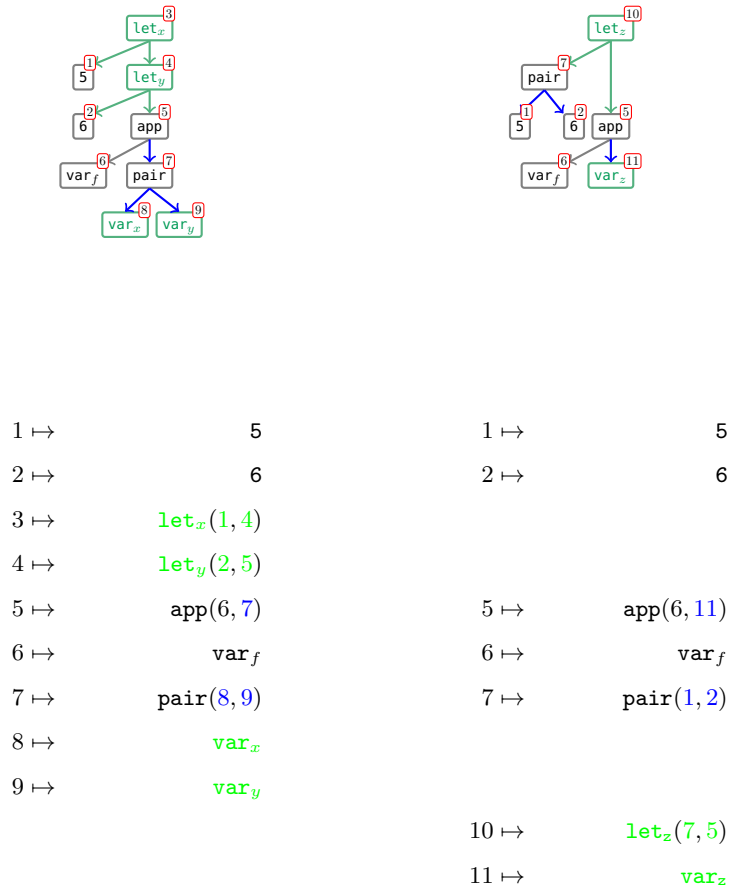


Figure 6.8 Delta terms via delta stores

the remaining nodes were re-used from the previous expression. The edges connecting the pair node to its children are blue because the pair has children with different indices in the other state.

In the lower half of Figure 6.8, we can see how the difference between two indexed expressions e_1 and e_2 is the indexed expression whose store is the difference between $\underline{e_1}$ and $\underline{e_2}$. An index in the domain of $\underline{e_1}$ which is not in the domain of $\underline{e_2}$ identifies a node which is *new* in e_1 . Both the constructor and each of the child indices are marked as new. An index in the domain of both stores indicates a node which may have changed. In the LambdaCalc UI, child indices are often not visualised, and so this aspect of the deltas is sometimes not visible. (The `split` example in Figure 2.11 is one such case in point.) The constructor associated with an index can also change, although this is not illustrated in this example. Moreover, if the arity of the constructor is strictly greater in e_1 than in e_2 , the child indices which are e_1 but not in e_2 will be marked as new in e_1 , indicating that the slot or field containing the child index did not exist in e_2 . There are more sophisticated approaches to dealing with differences in arity when a constructor changes, but we do not consider them here.

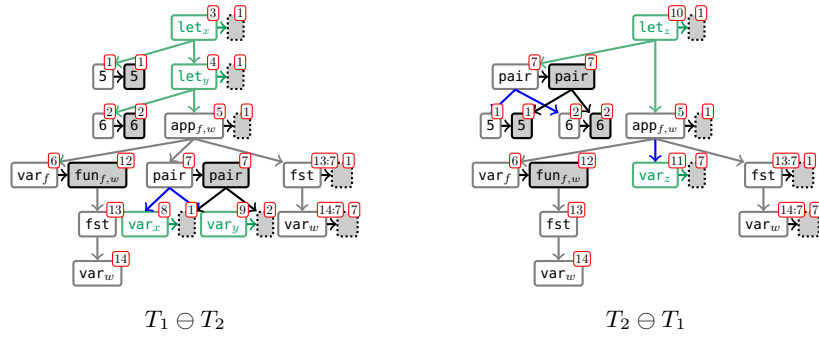


Figure 6.9 Before and after delta-traces associated with a particular refactoring

Finally, Figure 6.9 shows the delta traces that result from comparing the executions T_1 and T_2 of the program before and after the refactoring. We can see immediately that, because of sharing, the identity of the value computed by the program is invariant under this particular transformation. If this value were passed as an argument to a function, the identity of the function body would also be preserved into the new execution. It is “stability” properties like these that allow the consequences of certain changes to be localised and informative. Here, the consequences on the execution of inlining x and y and extracting the resulting pair to another local variable z are entirely local.

7 Conclusion

We conclude by considering some of the strong points and weak points of our research (§7.1). We also identify some directions for future work (§7.2). For future work we focus mainly on addressing shortcomings of our current proposal and challenges for a practical implementation, but also consider some opportunities.

7.1 Appraisal

The shortcomings and contributions of the thesis are already summarised in §§1.2 and 1.3. Our core proposal is a new way of thinking about *execution* in programming languages. We are not committed to particular language features; some version of interactive programming could probably be made to work with features and language paradigms other than the ones considered here. Although many of our concepts overlap with existing work, interactive programming remains a relatively unexplored direction in programming language research. It is also one of increasing practical significance thanks to growing interest in live programming environments for widely-used languages like JavaScript and Clojure. We hope to have convinced the reader that the many challenges that lie ahead for this paradigm are worth tackling.

There are several omissions and shortcomings of our proposal. Perhaps the biggest concern is scalability; all our examples were conspicuously simple. Although we intentionally deferred scalability issues to future work, dealing only with small examples admittedly weakens our case. We address several scalability concerns in §7.2.1 below. An efficient incremental implementation of differential execution (§7.2.2 below) will also be necessary if our idea is to be applicable to everyday programming, and so is another critical aspect of our vision which remains unproven. However, in this case, the substantial successes of self-adjusting computation (Related Work, §3.10) on similar problems are a reason to be optimistic. In terms of the formal treatment of differential execution, the contributions of Chapter 6 are diminished by having the key property stated only as a conjecture.

Finally, the decision to use LambdaCalc to partly implement its own GUI and forgo an implementation with reasonable performance was a risky strategy. Although this makes our proof-of-concept less convincing, we chose to take the opportunity to use our approach to solve a non-trivial programming problem. We refer the reader to Appendix B for more details.

7.2 Future work

7.2.1 Scaling to realistic programs

Equipping every value with a full account of how it was computed has the potential to incur substantial space overhead. Thankfully, this issue has already received some attention in the area of trace-based debuggers. One straightforward idea is to partition the operational semantics into deterministic and non-deterministic components. Only the non-deterministic choices need to be recorded; the deterministic parts of the trace can be discarded and recovered as needed by a re-execution which is “determinised” using the previously recorded choices. Implementations can use this technique to trade space for time. One example is the *checkpointing* technique used in time-travel debuggers for systems-level programming [Lew03, KDC05]; regular snapshots are taken of the heap, from which any intermediate state can then be recovered by determinised re-execution. Another example is described by Pothier and Tanter; their system retains summary information about bounded-size regions of the trace called *execution blocks*, which can then be discarded and recovered when needed, again by determinised re-execution [PT11].

Nilsson also explored trading space for time in his “piecemeal” tracing system for the lazy language Freja [Nil99]. His implementation is based on the evaluation dependence tree (EDT) trace format described in Related Work, §3.3, and materialises parts of the EDT on an as-needed basis. His system allows tracing to be started at particular functions of interest, akin to setting a breakpoint in a traditional debugger. Unlike the checkpointing systems mentioned above, however, Nilsson’s implementation re-executes from scratch each time a new part of the trace is required, so that the specific *demand* associated with that part of the computation can be correctly determined. In §7.2.5 below we discuss a generalisation of lazy computation we call *demand-indexed computation* which might support a more efficient implementation of piecemeal tracing. Claessen et al.’s work on Hat, based on redex trails, also explores efficient trace storage for large, multi-module programs [CRC⁺03]. Moreover, both systems support the declaration of “trusted” components for which no internal trace is recorded. A challenge for a system with untraced components however is how to support precise backward-slicing; one possibility is to require untraced components to export a slicing operation. This is similar to how we would like to treat primitive operations in the future (§7.2.6 below).

Large traces also present a significant visualisation challenge. In Chapter 2 we showed some simple techniques for taming the complexity of traces, including the ability to slice away unneeded parts of computations, and also to hide away their internals. Beyond these, there are many dynamic program visualisation methods applicable to large computations. Related Work, §3.6, contains some pointers into the literature. Here we discuss three important challenges. The first concerns extending the reference language with a store and store operations (§7.2.4 below). The resulting traces would be threaded with store states, with significant redundancy, in that store states would often differ in only small ways. The challenge would be to design a visualisation paradigm that exploited this redundancy for compactness, similar to how we currently exploit the redundancy in values. Recall from §2.3 that when sharing occurs *within* a computation, we use a rather simplistic approach based on *sharing links* to obtain more compact views. The trick of eliminating visual redundancy by showing how values are included into other values is really another kind of *differencing*, only occurring within a computation rather than between computations. Whereas cross-computation

deltas are shown using red and green colouring, within-computation deltas are shown using sharing links, indicating how values at one point in the computation are derived from values computed earlier.

An efficient visualisation scheme for stores would probably have to be based on similar principles, representing stores differentially. Each store state would be visualised as a modification of an earlier one, although not necessarily the immediately prior one, depending on how much of the computation the user had expanded. Something more sophisticated than mere sharing links would be needed, since stores deltas would involve overriding as well as addition and deletion. The visualisation challenges here may therefore be related to the *representational* challenges that arise for *persistent data structures*, which must retain prior versions in an efficient way [DSST86]. Functional techniques for persistent data structures, such as Okasaki’s purely functional data structures [Oka99], may suggest designs.

A different challenge is presented by medium-sized structured values, such as small database tables or lists with thousands of elements. (We consider large data structures, such as those that arise in scientific computing or large-scale cloud computing, to be out of scope for the foreseeable future.) Medium-sized data structures however could feasibly be handled in a system like LambdaCalc, using custom visualisations that provide summary information or other approximations of the information in the data structure, combined with a facility that allows the user to interactively drill-down to obtain greater detail as necessary. *Zoomable user interfaces* [PF93] would be one plausible approach that might fit well with our order-theoretic approach to slicing. So that these visualisations can themselves take advantage of the within-computation and cross-computation sharing which are integral to our approach, these visualisations would need to be coded in LambdaCalc itself. Then the interactive programming implementation would provide support for establishing the node identities required to represent values as term graphs with sharing, allowing the author of the custom visualisation to concentrate on visualisation logic, which would be expressed as a pure function. Indeed this is precisely the approach we took to visualising executions themselves, as described in Appendix B.

The final topic to discuss here is visualising code written in a higher-order or point-free style, such as a parser written using a combinator library. We should point out first that the environment captured as part of a closure is currently not visualised; this is of course important generally, but becomes especially so in the presence of code that makes extensive use of higher-order functions. This is relatively easy to fix by introducing some concrete syntax for the captured bindings, reusing the $x:$ notation employed in Chapter 2. These would then be shown, in addition to the argument bindings, on entry to the closure body. The broader question of how to visualise programs written in a heavily point-free style is a tricky one, and remains a subject for future study. We have yet to experiment with code structured around continuation-passing style or monads, for example. The “pointful” visualisation technique of LambdaCalc will probably still be useful, but it may need to be complemented with other techniques more aligned with treating programs as dataflow networks, where composition rather than application is the primary building block.

7.2.2 Efficient incremental update

For our approach to be viable for real-world programming, an incremental implementation of differential execution (retroactive update) is essential. In Related Work, §3.10, we defined this as the problem of obtaining

a new computation from an existing one in time asymptotically equal to the size of the trace delta. The basic idea is to exploit the fact that the trace represents precisely the dependencies between sub-computations. One can propagate changes bottom-up through the trace, terminating the propagation when the change has no more consequences. A bottom-up update algorithm of this kind is usually called *change propagation*. The algorithm must avoid so-called “glitches”, observable behaviours which would not be possible in any consistent computation. Glitches can occur when a node is updated before all of its dependent computations have finished updating. A standard technique is to use a priority queue to ensure that updates are correctly ordered. One challenge is efficiently supporting move operations that restructure parts of the trace; techniques from non-monotonic self-adjusting computation may apply [LWAB12].

Given an efficient implementation, it may be possible to apply a retroactive update at a finer grain than an entire closed computation, for example at the level of a transaction. This may provide a way to avoid some of the shortcomings of the *transactional version consistency* approach to dynamic software updating. We refer the reader to Related Work, §3.11, for the background. Suppose we want to push a change to pricing, purchasing terms or other business logic while a transaction is active. With dynamic software updating, even with transactional version consistency, there are several potentially undesirable outcomes. The transaction may not see the update, because it was applied too late in its execution. This means it will complete under the old code, which is consistent, but may not be what we wanted. Under some circumstances, it may not be possible to apply the update at all, because no consistent update point was available. With retroactive update, it would be possible to reflect changes to the transaction at any point during its execution, but with the semantics being equivalent to having applied the changes at the beginning of the transaction.

Retroactive update presents challenges of its own, in particular efficient implementation as already discussed, but also the opportunity to do more reliable online update of deployed applications. We also considered the applicability of retroactive update to transactional concurrency in earlier work [Per08]. Finally, we should note that applying retroactive update at a finer grain than an entire closed computation will not just be useful, but essential, if we wish to combine interactive programming with effects, in environments in which effects are not revocable. We discuss this in §7.2.4 below.

7.2.3 Distributed systems

Every program is a part of some other program and rarely fits.

Alan Perlis, *Epigrams on Programming* [Per82]

Throughout most of this thesis, we assumed that the programmer was the author as well as end-user of the program being inspected. But an equally important possibility is that the computation of interest is a remote application, perhaps a live data feed provided by another website or a financial transaction executed against our bank account. Our locally running program is merely a client of that other system. Relative to our vantage point inside the larger computation, these external parts of the computation have already happened. We do not have control over the larger computation and cannot ask it to “restart”. We cannot therefore

expect to re-run any part of it in a debugger. This is a very compelling motivation for “self-explaining” computation, i.e. runtime environments that provide online debuggability as a *service*.

To support distributed applications, we are extending our slicing technique to concurrent processes, in joint work with Deepak Garg. In the approach we are exploring, we use a process calculus to model the concurrent communication aspects of the computation and a functional language to model local computation at the processes. The local computations are reified into expression traces like the ones defined in Chapter 4; the evolution of the process configuration is recorded into a *process trace* which takes the form of a directed acyclic graph recording the non-deterministic allocation of channels and the forking and joining of processes. The resulting distributed process traces, where each agent is responsible for reifying its own part of the computation, have applications ranging from distributed debugging and provenance tracking to security. We are also working to extend the order-theoretic characterisation of dynamic slicing that we gave in Chapter 5 to the non-deterministic setting. The idea is to isolate *individual transitions* of the non-deterministic process semantics, and then show that a Galois connection captures the input-output dependencies for that transition. We hope to use composition of Galois connections to extend this analysis to any finite sequence of transitions.

Reified distributed computations also introduce challenges of their own. In particular, exposing the innards of a computation with multiple participants has privacy, data abstraction and intellectual property ramifications, and may also expose new security vulnerabilities. Cheney discusses some of these issues in relation to provenance, and introduces a formal model for provenance security, based on *disclosure* and *obfuscation* [Che11]. Disclosure and obfuscation are related to forward and backward slicing; disclosure states that some property is always revealed by a trace, whereas obfuscation states that some property is never revealed.

Finally, a critical question is how to integrate a system based on retroactive update, such as LambdaCalc or for that matter a self-adjusting program (Related Work, §3.10), into a larger system which may be incompatible with retroactive update. The question is most pertinent when the boundary between the system and its environment is effectful, meaning that the system with retroactive update can commit effects to the external system. If the external system lacks the capability to revoke effects, or the user lacks the authority to, then committed computations must become read-only. We discuss this in §7.2.4 below.

7.2.4 State and I/O

Interactive programming can be extended with effects by extending reified computations with *reified effects*. Reified effects are persistent *descriptions* of effects, rather than actual effects modifying the external world. A reified effectful computation has a natural subjunctive interpretation: it describes effects that it *would* emit to the world *if* it were “committed”. As mentioned above, to efficiently reify computations threaded with mutable data structures like stores, purely functional persistence techniques such as those described by Okasaki [Oka99] would be appropriate.

A subjunctive treatment of effects has two important advantages. First, there is no such thing as “duplicating” a reified effect – a reified effect, as a mere description of an effect, is a pure value. A reified effectful computation can therefore be repeatedly updated in an interactive programming environment without inviting the problems associated with both live programming (Related Work, §3.7) and self-adjusting computation

(§3.10) in the presence of effectful code. Equally, an effect can be revoked by simply unsplicing it from the reified computation.

Second, reification provides the *isolation* properties usually associated with transactions [HR83]: the programmer can experiment with the effect on the world they want before they commit to it. This is useful for many kinds of interaction – collaborative document editing or programming, financial transactions like online purchasing, and patching of online systems with bug-fixes (see the discussion on dynamic software updating in §7.2.2 above). When the transaction is finally committed, the hypothetical effects become actual effects applied to the outside world.

This is all very well, but when an actual effect is committed to the outside world, our ability to perform retroactive update with respect to that effect goes away – unless the computational environment into which we are plugged permits the revocation of that effect. If the outside world is the *real* world, and the effect was to emit a sound or control an actuator, this is unlikely to be the case. Moreover the user may lack the authority to revoke the effect, say if it were an online purchase. (Cancelling an order is not the same as making it so that the order never happened.) The upshot of this is that programming systems that combine retroactive update with external effects must be prepared to do so with smaller computational units than entire closed programs, such as transactions. Once an effect has been committed, the transaction or computation associated with it may become read-only. It will still be possible to browse the execution to understand the provenance of the effect, but the effect itself, and the account of how it came to be, will have become fixed.

7.2.5 Demand-indexed computation

In Related Work, §3.3, we discussed a connection between backward dynamic slicing and lazy evaluation. This connection was first proposed by Biswas [Bis97] but only formally explored for slices calculated with respect to outputs consisting of a single nullary constructor such as `Nil`. This was due to a limitation of Biswas’ slicing approach. In our more general slicing setting, it appears that this connection extends not only to partial outputs of the form `Cons(□, □)`, or *lazy tuples*, but more generally to arbitrary partial values, suggesting a generalisation of lazy evaluation. The idea is to treat an output slice as a *demand pattern* which determines how much computation to do. For “self-explaining” computation, this is a considerable advantage over building a full call-by-value trace and then discarding the parts no longer needed. To make this work, holes need to be treated as suspended computations.

The demand pattern cannot in general be known up front but must be specified interactively, at least initially. The computation is first evaluated with a generic demand which we write as \Box meaning “evaluate to lazy tuple”. One might also think of \Box at the list type as being shorthand for a “disjunctive” pattern `Nil|Cons(□, □)`. (A disjunctive pattern is a partial expression which does not necessarily satisfy deterministic extension.) Say this evaluation resulted in a partial value `Cons(□, □)` where the holes should be taken to represent thunks. Then the user could replace a hole by \Box so that the partial output is now `Cons(□, □)`, which would force enough extra computation to, say, compute `Cons(□, Nil)`. The demand pattern \Box was a “question”; in this case `Nil` was the answer. This is not so different from simply running a lazy computation to obtain a value with embedded thunks, and then forcing those thunks afterwards by running them individually as top-level lazy computations. But what we can do in addition is re-run that program lazily from

scratch using $\text{Cons}(\square, \text{Nil})$ as a demand pattern, since we now know it is capable of computing at least that much output. For example, one could save the state of a GUI when an application was shut down. The saved state could then be used as a demand pattern to force the computation to the state it was in when it was shut down, allowing the interaction between the user and the application to be resumed where it left off.

A particularly useful application would be for a provenance or debugging system based on traces where traces are discarded when not being used and re-computed when needed, as per Nilsson’s piecemeal tracing, discussed in §7.2.1 above. Rather than re-running the entire program just to re-compute some small part of the trace, it can be re-computed using the demand pattern that was active when it was discarded. A third application would be for lazy pattern-matching: a disjunctive pattern such as $\text{Nil} \mid \text{Cons}(\square, \text{Nil} \mid \text{Cons}(\square, \square))$ represents a partitioning of the list data type corresponding to three exhaustive and non-overlapping constructor patterns in a case expression. This demand pattern expresses precisely the extent to which the scrutinee must be evaluated in order to select the appropriate branch.

7.2.6 Primitive operations

In Chapter 5, §5.2.1, we noted that our treatment of primitive operations is rather unrealistic, because we require the interpretation $\hat{\oplus}$ of every operation to be strict. To permit primitives with more fine-grained input-output dependencies, we can weaken the requirement to insist only that $\hat{\oplus}$ preserve meets, like evaluation under the hole-propagation semantics. In other words the operation must satisfy

$$(c_1 \hat{\oplus} c_2) \sqcap (c_1' \hat{\oplus} c_2') = (c_1 \sqcap c_1') \hat{\oplus} (c_2 \sqcap c_2')$$

where c is either a primitive constant or \square . This permits primitives like \times to be non-strict, so that for example $\mathbf{0} \times \square = \mathbf{0}$. But a binary primitive must be strict in at least *one* of its arguments. If $\mathbf{0} \times \square = \mathbf{0}$ and $\square \times \mathbf{0} = \mathbf{0}$ but $\square \times \square = \square$, then \times will not preserve meets. In particular, this rules out Plotkin’s *parallel* or operator [Plo77]. Berry’s notion of *stable function* [Ber79] is related, although we did not explore this connection in the thesis.

For slicing, every primitive operation must, for any pair of inputs c_1 and c_2 , provide a function $\hat{\oplus}_{c_1, c_2}^{-1}$ which is the lower adjoint of $\hat{\oplus}$ when restricted to prefixes of c_1 and c_2 . Such adjoints exist because of the requirement that each $\hat{\oplus}$ be meet-preserving. In practice a definition must be provided by the implementation; it must be monotonic, and related to $\hat{\oplus}$ in the following way. Suppose $\hat{\oplus}_{c_1, c_2}^{-1}(c) = (c_1', c_2')$. Then

1. $c_1' \hat{\oplus} c_2' \sqsupseteq c$
2. if $c_1'' \hat{\oplus} c_2'' \sqsupseteq c$, then $(c_1', c_2') \sqsubseteq (c_1'', c_2'')$.

Then the hole-propagation rules for primitive operations need to be replaced by a single rule which delegates to $\hat{\oplus}_{c_1, c_2}^{-1}$, using the values cached on the primitive operation trace as the values of c_1 and c_2 .

What the specification above does is specify abstractly the properties an operation must satisfy in order for it to be plugged into our system without compromising the minimality or correctness of slicing for computations it participates in. Therefore, generalised to arbitrary values rather than just primitive constants, this approach can also model *external* operations or components for which traces are not available; see §7.2.1 above.

Bibliography

- [AACP12] Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. A core calculus for provenance. In *Proceedings of the First Conference on Principles of Security and Trust (POST)*, pages 410–429. Springer, 2012.
- [ABD07] Umut A. Acar, Matthias Blume, and Jacob Donham. A consistent semantics of self-adjusting computation. In *Programming Languages and Systems, Proceedings of the 16th European Symposium on Programming (ESOP’07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 458–474, Berlin / Heidelberg, 2007. Springer.
- [ABLW⁺10] Umut A. Acar, Guy Blelloch, Ruy Ley-Wild, Kanat Tangwongsan, and Duru Turkoglu. Traceable data types for self-adjusting computation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’10*, pages 483–496, New York, NY, USA, 2010. ACM.
- [Aca05] Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computing Science, Carnegie Mellon University, 2005.
- [BAD⁺01] 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.
- [BCV08] Peter Buneman, James Cheney, and Stijn Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Transactions on Database Systems*, 33(4):28, November 2008.
- [Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Company, Inc., Boston, MA, USA, 2002.
- [Ber79] Gérard Berry. *Modèles complètement adéquats et stables des lambda-calculs typés*. Thèse de doctorat d’état, Université Paris VII, 1979.
- [BF05] Rajendra Bose and James Frew. Lineage retrieval for scientific data processing: a survey. *ACM Computing Surveys*, 37(1):1–28, 2005.
- [BHHV04] B. Brassel, M. Hanus, F. Huch, and G. Vidal. A semantics for tracing declarative multi-paradigm programs. In *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP ’04*, pages 179–190, New York, NY, USA, 2004. ACM.
- [Bis97] Sandip K. Biswas. *Dynamic Slicing in Higher-Order Programming Languages*. PhD thesis, University of Pennsylvania, 1997.
- [BJ97] Simon P. Booth and Simon B. Jones. Walk backwards to happiness - debugging by time travel. In *Automated and Algorithmic Debugging*, pages 171–183, 1997.
- [BKT01] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *Proceedings of the 8th International Conference on Database Theory, ICDT ’01*, pages 316–330, London, UK, 2001. Springer-Verlag.
- [Bor79] Alan Hamilton Borning. *ThingLab – A Constraint-Oriented Simulation Laboratory*. Palo Alto Research Center XEROX, Palo Alto, California, July 1979.
- [BS95] Karen L. Bernstein and Eugene W. Stark. Formally defining debuggers: A comparison of three approaches. In Mireille

- Ducassé, editor, *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, pages 261–275. IRISA-CNRS, 1995.
- [CB97] Chris Clack and Lee Braine. Object-oriented functional spreadsheets. In *Proceedings of the 10th Glasgow Workshop on Functional Programming (GlaFP '97)*, 1997.
- [CCBF11] Stephen Chang, John Clements, Eli Barzilay, and Matthias Felleisen. Stepping lazy programs (pre-print). *Computing Research Repository (CoRR)*, abs/1108.4706, 2011.
- [CDHA11] Yan Chen, Joshua Dunfield, Matthew A. Hammer, and Umut A. Acar. Implicit self-adjusting computation for purely functional programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP '11*, pages 129–141, New York, NY, USA, 2011. ACM.
- [CFF01] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *Proceedings of the 10th European Symposium on Programming Languages and Systems, ESOP '01*, pages 320–334, London, UK, 2001. Springer-Verlag.
- [Che11] James Cheney. A formal framework for provenance security. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium, CSF '11*, pages 281–293, Washington, DC, USA, 2011. IEEE Computer Society.
- [Chi01] Olaf Chitil. A semantics for tracing. In Thomas Arts and Markus Mohnen, editors, *Draft Proceedings of the 13th International Workshop on Implementation of Functional Languages, IFL 2001*, pages 249–254, Sweden, September 2001. Ericsson Computer Science Laboratory.
- [Chi05] Olaf Chitil. Source-based trace exploration. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004*, LNCS 3474, pages 126–141. Springer, March 2005.
- [CJPR08] Andrew Cirillo, Radha Jagadeesan, Corin Pitcher, and James Riely. TAPIDO: trust and authorization via provenance and integrity in distributed objects. In *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems, ESOP'08/ETAPS'08*, pages 208–223, Berlin, Heidelberg, 2008. Springer-Verlag.
- [CK06] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European Conference on Programming Languages and Systems, ESOP'06*, pages 294–308, Berlin, Heidelberg, 2006. Springer-Verlag.
- [CL07] Olaf Chitil and Yong Luo. Structure and properties of traces for functional programs. *Electronic Notes in Theoretical Computer Science*, 176(1):39–63, 2007.
- [CMRW03] Nick Collins, Alex McLean, Julian Rohrerhuber, and Adrian Ward. Live coding in laptop performance. *Organised Sound*, 8(3):321–330, December 2003.
- [CRC⁺03] Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, and Malcolm Wallace. Testing and tracing lazy functional programs using Quickcheck and Hat. In *4th Summer School in Advanced Functional Programming*, number 2638 in LNCS, pages 59–99. Springer, 2003.
- [CS09] Diego Cheda and Josep Silva. State of the practice in algorithmic debugging. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 246:55–70, August 2009.
- [DF08] Susan B. Davidson and Juliana Freire. Provenance and scientific workflows: challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08*, pages 1345–1350, New York, NY, USA, 2008. ACM.
- [DGHL⁺80] Veronique Donzeau-Gouge, Gerard Huet, Bernard Lang, Gilles Kahn, et al. Programming environments based on structured editors: the MENTOR experience. Technical Report 26, INRIA, 1980.
- [dHRvE95] 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.

- [DIL04] Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. In *SODA '04: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 281–290, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.
- [Dmi01] M. Dmitriev. Towards flexible and safe technology for runtime evolution of Java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference*, 2001.
- [dS91] Fabio Q.B. da Silva. *Correctness Proofs of Compilers and Debuggers: an Approach Based on Structured Operational Semantics*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1991.
- [DSST86] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 109–121, New York, NY, USA, 1986. ACM Press.
- [Edw04] Jonathan Edwards. Example centric programming. *SIGPLAN Notices*, 39(12):84–91, 2004.
- [Edw05] 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.
- [Edw07] Jonathan Edwards. No ifs, ands, or buts: uncovering the simplicity of conditionals. In *OOPSLA '07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 639–658, New York, NY, USA, 2007. ACM.
- [EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP '97: Proceedings of the Second ACM SIGPLAN International Conference on Functional programming*, pages 263–273, New York, NY, USA, 1997. ACM.
- [Ell07] Conal M. Elliott. Tangible functional programming. *SIGPLAN Notices*, 42(9):59–70, 2007.
- [FAS94] Peter Fritzon, Mikhail Auguston, and Nahid Shahmehri. Using assertions in declarative and operational models for automated debugging. *Journal of Systems and Software*, 25(3):223 – 239, 1994.
- [FB09] Matthew Flatt and Eli Barzilay. Keyword and optional arguments in PLT Scheme. In *Proceedings of the Tenth Workshop on Scheme and Functional Programming*, 2009.
- [FGT08] J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated XML: queries and provenance. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '08, pages 271–280, New York, NY, USA, 2008. ACM.
- [Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
- [FT98] John Field and Frank Tip. Dynamic dependence in term rewriting systems and its application to program slicing. *Information and Software Technology*, 40(11–12):609–636, November/December 1998.
- [Geo00] Carlisle E. George. EROSI – visualising recursion and discovering new errors. In *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '00, pages 305–309, New York, NY, USA, 2000. ACM.
- [GKT07] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, pages 31–40, New York, NY, USA, 2007. ACM.
- [Gra94] Paul Graham. *On Lisp*. Prentice-Hall, 1994.
- [GS10] Lars Grammel and Margaret-Anne Storey. A survey of mashup development environments. In Mark Chignell, James Cordy, Joanna Ng, and Yelena Yesha, editors, *The Smart Internet*, pages 137–151. Springer-Verlag, Berlin, Heidelberg, 2010.
- [Hal84] Daniel Conrad Halbert. *Programming by example*. PhD thesis, University of California, Berkeley, 1984.

- [Han02] Keith Hanna. Interactive visual functional programming. In *ICFP '02: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 145–156, New York, NY, USA, 2002. ACM Press.
- [Han03] Christopher Michael Hancock. *Real-time programming and the big ideas of computational literacy*. PhD thesis, Massachusetts Institute of Technology, 2003.
- [HKS⁺07] Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. A formal model of dataflow repositories. In *Proceedings of the 4th International Conference on Data Integration in the Life Sciences, DILS'07*, pages 105–121, Berlin, Heidelberg, 2007. Springer-Verlag.
- [HN05] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1049–1096, November 2005.
- [HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, December 1983.
- [HW85] Peter Henderson and Mark Weiser. Continuous execution: the VisiProg environment. In *Proceedings of the 8th International Conference on Software Engineering, ICSE '85*, pages 68–74, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.
- [JCC⁺11] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2011.
- [Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Series in Computer Science. Prentice-Hall International, 1987. ISBN 0-13-453325-9.
- [Jon03] Simon L. Peyton Jones. Haskell 98: Introduction. *Journal of Functional Programming*, 13(1):0–6, 2003.
- [KAM05] Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. Design requirements for more flexible structured editors from a study of programmers' text editing. In *Extended Abstracts on Human Factors in Computing Systems, CHI EA '05*, pages 1557–1560, New York, NY, USA, 2005. ACM.
- [KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised⁵ report on the algorithmic language Scheme. *SIGPLAN Notices*, 33:26–76, 1998.
- [KDC05] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *ATEC '05: Proceedings of the Annual Conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2005. USENIX Association.
- [KHC91] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring semantics: a formal framework for specifying, implementing, and reasoning about execution monitors. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI '91*, pages 338–352, New York, NY, USA, 1991. ACM.
- [KW87] Raghu Karinthi and Mark Weiser. Incremental re-execution of programs. In *PLDI*, pages 38–44, 1987.
- [Lam07] Butler Lampson. Principles for computer system design. In *ACM Turing Award lectures*. ACM, New York, NY, USA, 2007.
- [Lew03] Bil Lewis. Debugging backwards in time. In Michiel Ronsse and Koen De Bosschere, editors, *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, September 2003.
- [LH80] Henry Lieberman and Carl Hewitt. A session with Tinker: interleaving program testing with program design. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming, LFP '80*, pages 90–99, New York, NY, USA, 1980. ACM.
- [LWAB12] Ruy Ley-Wild, UmutA. Acar, and Guy Blelloch. Non-monotonic self-adjusting computation. In Helmut Seidl, editor, *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 476–496. Springer, Berlin / Heidelberg, 2012.

- [LWFA08] Ruy Ley-Wild, Matthew Fluet, and Umut A. Acar. Compiling self-adjusting programs with continuations. *SIGPLAN Notices*, 43(9):321–334, 2008.
- [McC02] James McCartney. Rethinking the computer music language: SuperCollider. *Computer Music Journal*, 26(4):61–68, December 2002.
- [Nau10] David R. Naugler. Concurrent programming in Erlang: pre-conference workshop. *Journal of Computing Sciences in Colleges*, 25(5):6–7, May 2010.
- [NF92] Henrik Nilsson and Peter Fritzon. Algorithmic debugging for lazy functional languages. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, PLILP ’92, pages 385–399, London, UK, 1992. Springer-Verlag.
- [NF94] Henrik Nilsson and Peter Fritzon. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(03):337–369, 1994.
- [NHFP08] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’08, pages 37–49, New York, NY, USA, 2008. ACM.
- [Nil99] Henrik Nilsson. Tracing piece by piece: affordable debugging for lazy functional languages. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 36–47, Paris, France, September 1999. ACM Press.
- [Noo11] Harold Noonan. Identity. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter edition, 2011.
- [NS96] Henrik Nilsson and Jan Sparud. The evaluation dependence tree: an execution record for lazy functional debugging. Research Report LiTH-IDA-R-96-23, Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden, August 1996.
- [OED09] *Oxford English Dictionary*. Oxford University Press, Oxford, UK, third edition edition, 2009.
- [Oka99] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1999.
- [OSV08] Claudio Ochoa, Josep Silva, and Germán Vidal. Dynamic slicing of lazy functional programs based on redex trails. *Higher Order Symbolic Computation*, 21(1-2):147–192, 2008.
- [PACL12] Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. Functional programs that explain their work. In *ICFP ’12: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, 2012.
- [Pei98] Charles Sanders Peirce. Harvard lectures on pragmatism. In The Peirce Edition Project, editor, *The Essential Peirce*, volume 2. Indiana University Press, Bloomington, IN, USA, 1998.
- [Per82] Alan J. Perlis. Epigrams on programming. *SIGPLAN Notices*, 17(9):7–13, 1982.
- [Per86] Luís Pereira. Rational debugging in logic programming. In Ehud Shapiro, editor, *Third International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 203–210. Springer, Berlin / Heidelberg, 1986.
- [Per04] Roly Perera. Refactoring: to the Rubicon... and beyond! In *OOPSLA ’04: Companion to the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 2–3, New York, NY, USA, 2004. ACM Press.
- [Per08] 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.
- [Per10] Roly Perera. First-order interactive programming. In M. Carro and R. Peña, editors, *PADL 2010: Proceedings of the 12th International Symposium on the Practical Aspects of Declarative Languages*, volume 5937 of *Lecture Notes in Computer Science*, pages 186–200, Heidelberg, 2010. Springer.

- [PF93] Ken Perlin and David Fox. Pad: an alternative approach to the computer interface. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 57–64, New York, NY, USA, 1993. ACM.
- [Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.
- [PT11] Guillaume Pothier and Éric Tanter. Summarized trace indexing and querying for scalable back-in-time debugging. In *Proceedings of the 25th European Conference on Object-Oriented Programming*, ECOOP'11, pages 558–582, Berlin, Heidelberg, 2011. Springer-Verlag.
- [Rei07] Steven P. Reiss. Visual representations of executing programs. *Journal of Visual Languages and Computing*, 18(2):126–148, April 2007.
- [Rep93] Alexander Repenning. *Agentsheets: a tool for building domain-oriented dynamic, visual environments*. PhD thesis, University of Colorado at Boulder, Boulder, CO, USA, 1993.
- [RRB99] James Reichwein, Gregg Rothermel, and Margaret Burnett. Slicing spreadsheets: an integrated methodology for spreadsheet testing and debugging. In *Proceedings of the 2nd conference on Conference on Domain-Specific Languages - Volume 2*, DSL'99, page 3, Berkeley, CA, USA, 1999. USENIX Association.
- [SC06] Josep Silva and Olaf Chitil. Combining algorithmic debugging and program slicing. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pages 157–166, New York, NY, USA, 2006. ACM.
- [SCH08] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *IEEE Symposium on Security and Privacy*, pages 369–383, 2008.
- [Sep00] Marvin S. Seppanen. Modeling for application: developing industrial strength simulation models using Visual Basic for Applications (VBA). In *Proceedings of the 32nd Conference on Winter Simulation*, WSC '00, pages 77–82, San Diego, CA, USA, 2000. Society for Computer Simulation International.
- [SG10] Andrew Sorensen and Henry Gardner. Programming with time: cyber-physical programming with impromptu. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '10, pages 822–834, New York, NY, USA, 2010. ACM.
- [Sha83] Ehud Y. Shapiro. *Algorithmic program debugging*. ACM Distinguished Dissertations. MIT Press, Cambridge, MA, USA, 1983.
- [SHB⁺07] Gareth Stoye, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(4), August 2007.
- [Shn83] B. Shneiderman. Direct manipulation: A step beyond programming languages. *Computer*, 16(8):57–69, August 1983.
- [Smi87] Randall B. Smith. Experiences with the alternate reality kit: an example of the tension between literalism and magic. In *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface*, CHI '87, pages 61–67, New York, NY, USA, 1987. ACM.
- [SN95] Jan Sparud and Henrik Nilsson. The architecture of a debugger for lazy functional languages. In Mireille Ducassé, editor, *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG '95)*, Saint-Malo, France, May 1995.
- [SN10] Neil Sculthorpe and Henrik Nilsson. Keeping calm in the face of change. *Higher-Order and Symbolic Computation*, 23(2):227–271, June 2010.
- [SPG05] Yogesh Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005.
- [SR97] Jan Sparud and Colin Runciman. Tracing lazy functional computations using redex trails. In *PLILP '97: Proceedings of*

- the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 291–308, London, UK, 1997. Springer-Verlag.
- [SS02] Dana Shapira and James A. Storer. Edit distance with move operations. In *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, CPM '02, pages 85–98, London, UK, 2002. Springer-Verlag.
- [Ste77] Guy Lewis Steele, Jr. Debunking the “expensive procedure call” myth or, procedure call implementations considered harmful or, LAMBDA: The ultimate GOTO. In *Proceedings of the 1977 Annual Conference*, ACM '77, pages 153–162, New York, NY, USA, 1977. ACM.
- [Sut63] Ivan E. Sutherland. Sketchpad: a man-machine graphical communication system. In *Proceedings of the May 21-23, 1963, Spring Joint Computer Conference*, AFIPS '63 (Spring), pages 329–346, New York, NY, USA, 1963. ACM.
- [SXC⁺11] Robin Salkeld, Wenhao Xu, Brendan Cully, Geoffrey Lefebvre, Andrew Warfield, and Gregor Kiczales. Retroactive aspects: programming in the past. In *Proceedings of the Ninth International Workshop on Dynamic Analysis*, WODA '11, pages 29–34, New York, NY, USA, 2011. ACM.
- [SZ10] William N. Sumner and Xiangyu Zhang. Memory indexing: canonicalizing addresses across executions. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 217–226, New York, NY, USA, 2010. ACM.
- [TA95] Andrew Tolmach and Andrew W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(02):155–200, 1995.
- [Tan90] Steven L. Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages and Computing*, 1(2), 1990.
- [Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.
- [Vic11] Bret Victor. Up and down the ladder of abstraction. <http://worrydream.com/LadderOfAbstraction>, 2011.
- [Vic12a] Bret Victor. Inventing on principle. <http://worrydream.com/#!/InventingOnPrinciple>, 2012.
- [Vic12b] Bret Victor. Learnable programming. <http://worrydream.com/#!/LearnableProgramming>, 2012.
- [Wad71] Christopher Peter Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. D.Phil. thesis, Oxford University, 1971.
- [WC04] Ge Wang and Perry R. Cook. On-the-fly programming: Using code as an expressive musical instrument. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 138–143, 2004.
- [WCBR01] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for Haskell: a new Hat. In Ralf Hinze, editor, *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 151–170, Firenze, Italy, September 2001. Universiteit Utrecht UU-CS-2001-23.
- [Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
- [WF93] Rickard Westman and Peter Fritzson. Graphical user interfaces for algorithmic debugging. In Peter A. Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 273–286. Springer-Verlag, Berlin / Heidelberg, 1993.
- [WL90] Nicholas Wilde and Clayton Lewis. Spreadsheet-based interactive graphics: from prototype to tool. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Empowering People*, CHI '90, pages 153–160, New York, NY, USA, 1990. ACM.
- [Wol91] Stephen Wolfram. *Mathematica: a system for doing mathematics by computer (2nd ed.)*. Addison-Wesley Longman Publishing Company, Inc., Redwood City, CA, USA, 1991.
- [WT96] Lisa K. Wells and Jeffrey Travis. *LabVIEW for everyone: graphical programming made even easier*. Prentice-Hall, Inc.,

Upper Saddle River, NJ, USA, 1996.

- [XQZ⁺05] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30:1–36, March 2005.
- [XSZ08] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 238–248, New York, NY, USA, 2008. ACM.
- [Zel02] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 1–10, New York, NY, USA, 2002. ACM.

Appendices

A Additional Proofs

We write *IH* for “induction hypothesis”.

A.1 Proof of Theorem 3

Proof. Suppose $v, T \Downarrow^{-1} \rho, e$. We only show part (1); part (2) follows from part (1) by determinism (Lemma 5). So suppose $u \sqsubseteq v$. We proceed by induction on the derivation. In each case, if $u = \square$ the conclusion is immediate. We present only the variable and application cases. The others are similar, but simpler.

Case

$$\frac{}{v, x \Downarrow^{-1} \square_{\Gamma.x \mapsto v}, x} v \neq \square$$

We can immediately derive

$$\frac{}{u, x \Downarrow^{-1} \square_{\Gamma.x \mapsto u}, x} u \neq \square$$

noting that $\square_{\Gamma.x \mapsto u} \sqsubseteq \square_{\Gamma.x \mapsto v}$.

Case

$$\frac{v, T \Downarrow^{-1} \rho'[f \mapsto v_1][x \mapsto v_2], e' \quad v_2, T_2 \Downarrow^{-1} \rho_2', e_2' \quad v_1 \sqcup \langle \rho', \text{fun } f(x).e' \rangle, T_1 \Downarrow^{-1} \rho_1', e_1'}{v, T_1 T_2 \triangleright \langle \gamma, \text{fun } f(x).T \rangle \Downarrow^{-1} \rho_1' \sqcup \rho_2', e_1' e_2'} v \neq \square$$

We want to derive

$$\frac{u, T \Downarrow^{-1} \rho[f \mapsto u_1][x \mapsto u_2], e \quad u_2, T_2 \Downarrow^{-1} \rho_2, e_2 \quad u_1 \sqcup \langle \rho, \text{fun } f(x).e \rangle, T_1 \Downarrow^{-1} \rho_1, e_1}{u, T_1 T_2 \triangleright \langle \gamma, \text{fun } f(x).T \rangle \Downarrow^{-1} \rho_1 \sqcup \rho_2, e_1 e_2} u \neq \square$$

with $(\rho_1 \sqcup \rho_2, e_1 e_2) \sqsubseteq (\rho_1' \sqcup \rho_2', e_1' e_2')$.

First premise. Since $u \sqsubseteq v$, we have $(\rho[f \mapsto u_1][x \mapsto u_2], e) \sqsubseteq (\rho'[f \mapsto v_1][x \mapsto v_2], e')$ by the IH. Then $\rho \sqsubseteq \rho'$ and $u_1 \sqsubseteq v_1$ and $u_2 \sqsubseteq v_2$ by Equation 5.1, with $e \sqsubseteq e'$.

Second premise. Since then $u_2 \sqsubseteq v_2$, we have $(\rho_2, e_2) \sqsubseteq (\rho_2', e_2')$ again by the IH.

Third premise. Since $u_1 \sqsubseteq v_1$ and $\rho \sqsubseteq \rho'$ and $e \sqsubseteq e'$, and the join $v_1 \sqcup \langle \rho', \text{fun } f(x).e' \rangle$ exists, the join $u_1 \sqcup \langle \rho, \text{fun } f(x).e \rangle$ also exists and is smaller. We then have $(\rho_1, e_1) \sqsubseteq (\rho_1', e_1')$ by the IH.

RHS of conclusion. Since $\rho_1 \sqsubseteq \rho_1'$ and $\rho_2 \sqsubseteq \rho_2'$ and the join $\rho_1' \sqcup \rho_2'$ exists, the join $\rho_1 \sqcup \rho_2$ exists and is smaller than $\rho_1' \sqcup \rho_2'$. \square

A.2 Proof of Theorem 4

Proof. Suppose $\rho, e \Downarrow v, T$. If $v = \square$, then the conclusion is immediate. Otherwise we proceed by induction on the derivation, considering only the variable and application cases. The other cases are similar, but simpler.

Case

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

We can immediately derive

$$\frac{}{\rho(x), x \Downarrow^{-1} \square_{\Gamma.x \mapsto \rho(x)}, x} \rho(x) \neq \square$$

noting that $\square_{\Gamma.x \mapsto \rho(x)} \sqsubseteq \rho$.

Case

$$\frac{\rho, e_1 \Downarrow v_1, T_1 \quad \rho, e_2 \Downarrow v_2, T_2 \quad \rho'[f \mapsto v_1][x \mapsto v_2], e \Downarrow v, T}{\rho, e_1 e_2 \Downarrow v, T_1 T_2 \triangleright \langle \gamma, \mathbf{fun} f(x).T \rangle} v_1 = \langle \rho', \mathbf{fun} f(x).e \rangle$$

We want to derive

$$\frac{v, T \Downarrow^{-1} \rho''[f \mapsto u_1][x \mapsto u_2], e' \quad u_2, T_2 \Downarrow^{-1} \rho_2, e_2' \quad u_1 \sqcup \langle \rho'', \mathbf{fun} f(x).e' \rangle, T_1 \Downarrow^{-1} \rho_1, e_1'}{v, T_1 T_2 \triangleright \langle \gamma, \mathbf{fun} f(x).T \rangle \Downarrow^{-1} \rho_1 \sqcup \rho_2, e_1' e_2'} v \neq \square$$

with $(\rho_1 \sqcup \rho_2, e_1' e_2') \sqsubseteq (\rho, e_1 e_2)$.

First premise. By the IH, T explains v and $(\rho''[f \mapsto u_1][x \mapsto u_2], e') \sqsubseteq (\rho'[f \mapsto v_1][x \mapsto v_2], e)$. Note that then $\rho'' \sqsubseteq \rho'$ and $u_1 \sqsubseteq v_1$ and $u_2 \sqsubseteq v_2$ by Equation 5.1, and also $e' \sqsubseteq e$.

Second premise. By the IH, T_2 explains v_2 , and then:

$$\begin{aligned} & (\rho_2, e_2') \\ \stackrel{\text{def}}{=} & \text{tr-uneval}_{T_2}(u_2) \\ \sqsubseteq & \text{tr-uneval}_{T_2}(v_2) \quad (u_2 \sqsubseteq v_2) \\ \sqsubseteq & (\rho, e_2) \quad (\text{IH}) \end{aligned}$$

Third premise. By the IH, T_1 explains v_1 . Because $u_1 \sqsubseteq v_1$ and $\langle \rho'', \mathbf{fun} f(x).e' \rangle \sqsubseteq v_1$, the join $u_1' \stackrel{\text{def}}{=} u_1 \sqcup \langle \rho'', \mathbf{fun} f(x).e' \rangle$ exists and is smaller than v_1 . Then:

$$\begin{aligned} & (\rho_1, e_1') \\ \stackrel{\text{def}}{=} & \text{tr-uneval}_{T_1}(u_1') \\ \sqsubseteq & \text{tr-uneval}_{T_1}(v_1) \quad (u_1' \sqsubseteq v_1) \\ \sqsubseteq & (\rho, e_1) \quad (\text{IH}) \end{aligned}$$

RHS of conclusion. Since $\rho_1, \rho_2 \sqsubseteq \rho$ the join $\rho_1 \sqcup \rho_2$ exists. \square

A.3 Proof of Lemma 6

Proof. Suppose S explains v , and $u \sqsubseteq v$ and $T \sqsupseteq S$. Note that if $\Gamma \vdash S$ then $\Gamma \vdash T$. By Theorem 3, S explains u . We proceed by induction on the derivation of $u, S \Downarrow^{-1} \rho, e$. The proof is straightforward and similar to a proof of determinacy, and so we present only the hole, variable, function and application cases.

Case

$$\frac{}{\Box, S \Downarrow^{-1} \Box_\Gamma, \Box}$$

Since $\Gamma \vdash T$ we can immediately derive

$$\frac{}{\Box, T \Downarrow^{-1} \Box_\Gamma, \Box}$$

Case

$$\frac{}{u, x \Downarrow^{-1} \Box_{\Gamma, x \mapsto u}, x} u \neq \Box$$

Since $S = x = T$, the conclusion is immediate.

Case

$$\frac{}{\langle \rho, \text{fun } f(x).e \rangle, \text{fun } f(x).e' \Downarrow^{-1} \rho, \text{fun } f(x).e} e \sqsubseteq e'$$

Since T is then of the form $\text{fun } f(x).e''$ with $e' \sqsubseteq e''$, we can immediately derive

$$\frac{}{\langle \rho, \text{fun } f(x).e \rangle, \text{fun } f(x).e'' \Downarrow^{-1} \rho, \text{fun } f(x).e} e \sqsubseteq e''$$

Case

$$\frac{u, S \Downarrow^{-1} \rho[f \mapsto u_1][x \mapsto u_2], e \quad u_2, S_2 \Downarrow^{-1} \rho_2, e_2 \quad u_1 \sqcup \langle \rho, \text{fun } f(x).e \rangle, S_1 \Downarrow^{-1} \rho_1, e_1}{u, S_1 \ S_2 \triangleright \langle \gamma, \text{fun } f(x).S \rangle \Downarrow^{-1} \rho_1 \sqcup \rho_2, e_1 \ e_2} u \neq \Box$$

We want to derive

$$\frac{u, T \Downarrow^{-1} \rho'[f \mapsto u_1'][x \mapsto u_2'], e' \quad u_2', T_2 \Downarrow^{-1} \rho_2', e_2' \quad u_1' \sqcup \langle \rho', \text{fun } f(x).e' \rangle, T_1 \Downarrow^{-1} \rho_1', e_1'}{u, T_1 \ T_2 \triangleright \langle \gamma, \text{fun } f(x).T \rangle \Downarrow^{-1} \rho_1' \sqcup \rho_2', e_1' \ e_2'} u \neq \Box$$

with $(\rho_1 \sqcup \rho_2, e_1 \ e_2) = (\rho_1' \sqcup \rho_2', e_1' \ e_2')$. Since S explains u and $T \sqsupseteq S$, we have $(\rho'[f \mapsto u_1'][x \mapsto u_2'], e') = (\rho[f \mapsto u_1][x \mapsto u_2], e)$ by the IH. Note that $\rho' = \rho$ and $u_1' = u_1$ and $u_2' = u_2$ by Equation 5.1, and also $e' = e$. Then since S_2 explains $u_2 = u_2'$ and $T_2 \sqsubseteq S_2$, we have $(\rho_2', e_2') = (\rho_2, e_2)$ by the IH. Similarly, since S_1 explains $u_1 \sqcup \langle \rho, \text{fun } f(x).e \rangle = u_1' \sqcup \langle \rho', \text{fun } f(x).e' \rangle$, we also have $(\rho_1', e_1') = (\rho_1, e_1)$ by the IH. □

A.4 Proof of Theorem 5

Proof. Suppose $\rho, e \Downarrow v, T$ where $\Gamma \vdash T$, and $(u, S) \sqsubseteq (v, T)$ with $u, S \Downarrow^{-1} \rho', e'$. We want that $\rho', e' \Downarrow_{\text{ref}} u'$ with $u' \sqsupseteq u$. We proceed by induction on the derivation of $u, S \Downarrow^{-1} \rho', e'$ and inversion on the derivation of $\rho, e \Downarrow v, T$, presenting only the hole, variable, function and application cases. Recall Equation 5.1.

Case

$$\frac{}{\Box, S \Downarrow^{-1} \Box_{\Gamma}, \Box}$$

We can immediately derive $\Box_{\Gamma}, \Box \Downarrow_{\text{ref}} \Box$.

Case

$$\frac{}{u, x \Downarrow^{-1} \Box_{\Gamma.x \mapsto u}, x} u \neq \Box$$

We can immediately derive

$$\frac{}{\Box_{\Gamma.x \mapsto u}, x \Downarrow_{\text{ref}} (\Box_{\Gamma.x \mapsto u})(x)}$$

noting that $(\Box_{\Gamma.x \mapsto u})(x) = u$.

Case

$$\frac{}{\langle \rho, \text{fun } f(x).e \rangle, \text{fun } f(x).e' \Downarrow^{-1} \rho, \text{fun } f(x).e} e \sqsubseteq e'$$

We can immediately derive

$$\frac{}{\rho, \text{fun } f(x).e \Downarrow_{\text{ref}} \langle \rho, \text{fun } f(x).e \rangle}$$

Case

$$\frac{u, S \Downarrow^{-1} \rho''[f \mapsto u_1][x \mapsto u_2], e' \quad u_2, S_2 \Downarrow^{-1} \rho_2, e_2' \quad u_1 \sqcup \langle \rho'', \text{fun } f(x).e' \rangle, S_1 \Downarrow^{-1} \rho_1, e_1'}{u, S_1 S_2 \triangleright \langle \gamma, \text{fun } f(x).S \rangle \Downarrow^{-1} \rho_1 \sqcup \rho_2, e_1' e_2'} u \neq \Box$$

By inversion we have

$$\frac{\rho, e_1 \Downarrow v_1, T_1 \quad \rho, e_2 \Downarrow v_2, T_2 \quad \rho'[f \mapsto v_1][x \mapsto v_2], e \Downarrow v, T}{\rho, e_1 e_2 \Downarrow v, T_1 T_2 \triangleright \langle \gamma, \text{fun } f(x).T \rangle} v_1 = \langle \rho', \text{fun } f(x).e \rangle$$

First we show that each premise of the unevaluation derivation yields a prefix of the evaluand of the corresponding premise of the evaluation derivation. Since $(u, S) \sqsubseteq (v, T)$, we have $(\rho''[f \mapsto u_1][x \mapsto u_2], e') \sqsubseteq (\rho'[f \mapsto v_1][x \mapsto v_2], e)$ by Lemma 7. Then since $(u_2, S_2) \sqsubseteq (v_2, T_2)$, we have $(\rho_2, e_2') \sqsubseteq (\rho, e_2)$ by Lemma 7. And then since $(u_1 \sqcup \langle \rho'', \text{fun } f(x).e' \rangle, S_1) \sqsubseteq (v_1, T_1)$, we have $(\rho_1, e_1') \sqsubseteq (\rho, e_1)$ by Lemma 7.

We now want to derive

$$\frac{\rho_1 \sqcup \rho_2, e_1' \Downarrow_{\text{ref}} u_1' \quad \rho_1 \sqcup \rho_2, e_2' \Downarrow_{\text{ref}} u_2' \quad \rho'''[f \mapsto u_1'][x \mapsto u_2'], e'' \Downarrow_{\text{ref}} u'}{\rho_1 \sqcup \rho_2, e_1' e_2' \Downarrow_{\text{ref}} u'} u_1' = \langle \rho''', \text{fun } f(x).e'' \rangle$$

with $u' \sqsubseteq u$.

First premise. Noting that $\rho_1 \sqsubseteq \rho$ and $\rho_2 \sqsubseteq \rho$ and $e_1' \sqsubseteq e_1$, we have

$$\begin{aligned}
& u_1 \sqcup \langle \rho'', \text{fun } f(x).e' \rangle \\
& \sqsubseteq \text{eval}(\rho_1, e_1') \quad (\text{IH}) \\
& \sqsubseteq \text{eval}(\rho_1 \sqcup \rho_2, e_1') \stackrel{\text{def}}{=} u_1' \quad (\text{Theorem 2}) \\
& \stackrel{\text{def}}{=} \langle \rho''', \text{fun } f(x).e'' \rangle \\
& \sqsubseteq \text{eval}(\rho, e_1) \stackrel{\text{def}}{=} v_1 \quad (\text{Theorem 2}) \\
& \stackrel{\text{def}}{=} \langle \rho', \text{fun } f(x).e \rangle
\end{aligned}$$

Note that $u_1 \sqsubseteq u_1' \sqsubseteq v_1$ and $\rho'' \sqsubseteq \rho''' \sqsubseteq \rho'$ and $e' \sqsubseteq e'' \sqsubseteq e$.

Second premise. Similarly, noting that $e_2' \sqsubseteq e_2$ we have

$$\begin{aligned}
& u_2 \\
& \sqsubseteq \text{eval}(\rho_2, e_2') \quad (\text{IH}) \\
& \sqsubseteq \text{eval}(\rho_1 \sqcup \rho_2, e_2') \stackrel{\text{def}}{=} u_2' \quad (\text{Theorem 2}) \\
& \sqsubseteq \text{eval}(\rho, e_2) \stackrel{\text{def}}{=} v_2 \quad (\text{Theorem 2})
\end{aligned}$$

Third premise. Using the inequalities established for the first and second premises, we note that $\rho''[f \mapsto u_1][x \mapsto u_2] \sqsubseteq \rho'''[f \mapsto u_1'][x \mapsto u_2'] \sqsubseteq \rho'[f \mapsto v_1][x \mapsto v_2]$. Then we have:

$$\begin{aligned}
& u \\
& \sqsubseteq \text{eval}(\rho''[f \mapsto u_1][x \mapsto u_2], e') \quad (\text{IH}) \\
& \sqsubseteq \text{eval}(\rho'''[f \mapsto u_1'][x \mapsto u_2'], e'') \quad (\text{Theorem 2}) \\
& \stackrel{\text{def}}{=} u'
\end{aligned}$$

□

A.5 Proof of Theorem 8

Proof. Suppose $v, T \searrow \rho, S$. We want that there exist ρ', e such that $v, S \Downarrow^{-1} \rho', e$, and also that $S \sqsubseteq T$. We proceed by induction on the derivation.

Case

$$\frac{}{\Box, T \searrow \Box_\Gamma, \Box}$$

We can immediately derive

$$\frac{}{\Box, \Box \Downarrow^{-1} \Box_\Gamma, \Box}$$

Case

$$\frac{}{v, x \searrow \Box_\Gamma.x \mapsto v, x} v \neq \Box$$

Since $S = x = T$ we can immediately derive

$$\frac{}{v, x \Downarrow^{-1} \Box_\Gamma.x \mapsto v, x} v \neq \Box$$

The case for a primitive constant c is similar to the variable case.

Case

$$\frac{c_2, T_2 \searrow \rho_2, S_2 \quad c_1, T_1 \searrow \rho_1, S_1}{v, T_1 \oplus_{c_1, c_2} T_2 \searrow \rho_1 \sqcup \rho_2, S_1 \oplus_{c_1, c_2} S_2} v \neq \square$$

We want to derive:

$$\frac{c_2, S_2 \Downarrow^{-1} \rho_2', e_2 \quad c_1, S_1 \Downarrow^{-1} \rho_1', e_1}{v, S_1 \oplus_{c_1, c_2} S_2 \Downarrow^{-1} \rho_1' \sqcup \rho_2', e_1 \oplus e_2} v \neq \square$$

Both premises are immediate from the IH, with $S_1 \sqsubseteq T_1$ and $S_2 \sqsubseteq T_2$. The cases for pairs (T_1, T_2) , projections `fst` and `snd`, injections `inl` and `inr`, and `roll` and `unroll` are similar.

Case

$$\frac{}{\langle \rho, \text{fun } f(x).e \rangle, \text{fun } f(x).e' \searrow \square_\Gamma, \text{fun } f(x).e} e \sqsubseteq e'$$

We can immediately derive

$$\frac{}{\langle \rho, \text{fun } f(x).e \rangle, \text{fun } f(x).e \Downarrow^{-1} \rho, \text{fun } f(x).e} e \sqsubseteq e$$

Case

$$\frac{v, T \searrow \rho[f \mapsto v_1][x \mapsto v_2], S \quad v, T \Downarrow^{-1} _, e \quad v_2, T_2 \searrow \rho_2, S_2 \quad v_1 \sqcup \langle \rho, \text{fun } f(x).e \rangle, T_1 \searrow \rho_1, S_1}{v, T_1 T_2 \triangleright \langle \gamma, \text{fun } f(x).T \rangle \searrow \rho_1 \sqcup \rho_2, S_1 S_2 \triangleright \langle \gamma, \text{fun } f(x).S \rangle} v \neq \square$$

We want that $S_1 S_2 \triangleright \langle \gamma, \text{fun } f(x).S \rangle \sqsubseteq T_1 T_2 \triangleright \langle \gamma, \text{fun } f(x).T \rangle$, and also that we can derive:

$$\frac{v, S \Downarrow^{-1} \rho'[f \mapsto v_1'][x \mapsto v_2'], e' \quad v_2', S_2 \Downarrow^{-1} \rho_2', e_2 \quad v_1' \sqcup \langle \rho', \text{fun } f(x).e' \rangle, S_1 \Downarrow^{-1} \rho_1', e_1}{v, S_1 S_2 \triangleright \langle \gamma, \text{fun } f(x).S \rangle \Downarrow^{-1} \rho_1' \sqcup \rho_2', e_1 e_2} v \neq \square$$

First premise. By the IH, S explains v and $S \sqsubseteq T$. By Lemma 6, T also explains v , with $\text{tr-uneval}_T(v) = \text{tr-uneval}_S(v)$. Note that then $e = e'$. Moreover, $\rho'[f \mapsto v_1'][x \mapsto v_2'] = \rho[f \mapsto v_1][x \mapsto v_2]$ by Theorem 7 and the determinism of \searrow . Then $\rho' = \rho$ and $v_1' = v_1$ and $v_2' = v_2$ by Equation 5.1.

Second premise. S_2 explains $v_2 = v_2'$ with $S_2 \sqsubseteq T_2$ immediately by the IH.

Third premise. S_1 explains $v_1 \sqcup \langle \rho, \text{fun } f(x).e \rangle = v_1' \sqcup \langle \rho', \text{fun } f(x).e' \rangle$ immediately by the IH.

Case

$$\frac{v, T_1 \searrow \rho_1[x_1 \mapsto v_1], S_1 \quad \text{inl } v_1, T \searrow \rho, S}{v, \text{case } T \text{ of } \{\text{inl}(x_1).T_1; \text{inr}(x_2).e_2\} \searrow \quad \rho_1 \sqcup \rho, \text{case } S \text{ of } \{\text{inl}(x_1).S_1; \text{inr}(x_2).\square\}} v \neq \square$$

We want to show that $\text{case } S \text{ of } \{\text{inl}(x_1).T_1; \text{inr}(x_2).\square\} \sqsubseteq \text{case } T \text{ of } \{\text{inl}(x_1).T_1; \text{inr}(x_2).e_2\}$, and also that we can derive

$$\frac{v, S_1 \Downarrow^{-1} \rho_1'[x_1 \mapsto v_1'], e_1 \quad \text{inl } v_1', S \Downarrow^{-1} \rho', e}{v, \text{case } S \text{ of } \{\text{inl}(x_1).S_1; \text{inr}(x_2).\square\} \Downarrow^{-1} \quad \rho_1' \sqcup \rho', \text{case } e \text{ of } \{\text{inl}(x_1).e_1; \text{inr}(x_2).\square\}} v \neq \square$$

The case for the `case` form that takes the `inr` branch is similar.

First premise. By the IH, S_1 explains v and $S_1 \sqsubseteq T_1$. By Lemma 6, T_1 also explains v , with $\text{tr-uneval}_{T_1}(v) = \text{tr-uneval}_{S_1}(v)$. Then $\rho_1[x_1 \mapsto v_1] = \rho_1'[x_1 \mapsto v_1']$ by Theorem 7 and the determinism of \searrow , with $\rho_1 = \rho_1'$ and $v_1 = v_1'$ by Equation 5.1.

Second premise. S explains $v_1 = v_1'$ with $S \sqsubseteq T$ immediately by the IH. \square

A.6 Proof of Theorem 9

Proof. Suppose $v, T' \searrow \rho', S$, and any $T \sqsubseteq T'$ such that $v, T \Downarrow^{-1} \rho, e$. We proceed by induction on the derivation of $v, T \Downarrow^{-1} \rho, e$ and inversion on the derivation of $v, T' \searrow \rho', S$, using a stronger inductive hypothesis, namely that $(\rho', S) \sqsubseteq (\rho, T)$. Note that if $\Gamma \vdash T'$ then $\Gamma \vdash T$.

Case

$$\frac{}{\square, T \Downarrow^{-1} \square_\Gamma, \square} \quad \frac{}{\square, T' \searrow \square_\Gamma, \square}$$

The conclusion is immediate.

Case

$$\frac{}{v, x \Downarrow^{-1} \square_{\Gamma.x \mapsto v}, x} v \neq \square \quad \frac{}{v, x \searrow \square_{\Gamma.x \mapsto v}, x} v \neq \square$$

Again the conclusion is immediate.

Case

$$\frac{}{c, c \Downarrow^{-1} \square_\Gamma, c} \quad \frac{}{c, c \searrow \square_\Gamma, c}$$

Again the conclusion is immediate.

Case

$$\frac{c_2, T_2 \Downarrow^{-1} \rho_2, e_2 \quad c_1, T_1 \Downarrow^{-1} \rho_1, e_1}{v, T_1 \oplus_{c_1, c_2} T_2 \Downarrow^{-1} \rho_1 \sqcup \rho_2, e_1 \oplus e_2} v \neq \square \quad \frac{c_2, T_2' \searrow \rho_2', S_2 \quad c_1, T_1' \searrow \rho_1', S_1}{v, T_1' \oplus_{c_1, c_2} T_2' \searrow \rho_1' \sqcup \rho_2', S_1 \oplus_{c_1, c_2} S_2} v \neq \square$$

Since $T_1 \sqsubseteq T_1'$, we have $(\rho_1', S_1) \sqsubseteq (\rho_1, T_1)$ by the IH. Similarly, $T_2 \sqsubseteq T_2'$ and so $(\rho_2, S_2) \sqsubseteq (\rho_2, T_2)$ by the IH. Then $(\rho_1' \sqcup \rho_2', S_1 \oplus_{c_1, c_2} S_2) \sqsubseteq (\rho_1 \sqcup \rho_2, T_1 \oplus_{c_1, c_2} T_2)$. The cases for pairs (T_1, T_2) , projections `fst` and `snd`, injections `inl` and `inr`, and `roll` and `unroll` are similar.

Case

$$\frac{}{\langle \rho, \text{fun } f(x).e \rangle, \text{fun } f(x).e' \Downarrow^{-1} \rho, \text{fun } f(x).e} e \sqsubseteq e' \\ \frac{}{\langle \rho, \text{fun } f(x).e \rangle, \text{fun } f(x).e'' \searrow \square_\Gamma, \text{fun } f(x).\square} e \sqsubseteq e''$$

Then $\text{fun } f(x).e' \sqsubseteq \text{fun } f(x).e'$ immediate.

Case

$$\frac{v, T \Downarrow^{-1} \rho[f \mapsto v_1][x \mapsto v_2], e \quad v_2, T_2 \Downarrow^{-1} \rho_2, e_2 \quad v_1 \sqcup \langle \rho, \mathbf{fun} f(x).e \rangle, T_1 \Downarrow^{-1} \rho_1, e_1}{v, T_1 T_2 \triangleright \langle \gamma, \mathbf{fun} f(x).T \rangle \Downarrow^{-1} \rho_1 \sqcup \rho_2, e_1 e_2} v \neq \square$$

$$\frac{v, T' \searrow \rho'[f \mapsto u_1][x \mapsto u_2], S \quad v, T' \Downarrow^{-1} _, e' \quad u_2, T_2' \searrow \rho_2', S_2 \quad u_1 \sqcup \langle \rho', \mathbf{fun} f(x).e' \rangle, T_1' \searrow \rho_1', S_1}{v, T_1' T_2' \triangleright \langle \gamma, \mathbf{fun} f(x).T' \rangle \searrow \rho_1' \sqcup \rho_2', S_1 S_2 \triangleright \langle \gamma, \mathbf{fun} f(x).S \rangle} v \neq \square$$

Since $T \sqsubseteq T'$, we have $(\rho'[f \mapsto u_1][x \mapsto u_2], S) \sqsubseteq (\rho[f \mapsto v_1][x \mapsto v_2], T)$ by the IH. Note that $\rho' \sqsubseteq \rho$ and $u_1 \sqsubseteq v_1$ and $u_2 \sqsubseteq v_2$ by Equation 5.1, and also $S \sqsubseteq T$. Since $u_2 \sqsubseteq v_2$, we have $u_2, T_2 \Downarrow^{-1} \rho_2'', e_2'$ with $\rho_2'' \sqsubseteq \rho_2$ by Theorem 3. But since $T_2 \sqsubseteq T_2'$, we have $(\rho_2', S_2) \sqsubseteq (\rho_2'', T_2) \sqsubseteq (\rho_2, T_2)$ by the IH. Now note that since $T \sqsubseteq T'$, we have $e' = e$ by Lemma 6, and so $u_1' \stackrel{\text{def}}{=} u_1 \sqcup \langle \rho', \mathbf{fun} f(x).e' \rangle \sqsubseteq v_1 \sqcup \langle \rho, \mathbf{fun} f(x).e \rangle$. Then $u_1', T_1 \Downarrow^{-1} \rho_1'', e_1'$ with $\rho_1'' \sqsubseteq \rho_1$ by Theorem 3. And then since $T_1 \sqsubseteq T_1'$, we have $(\rho_1', S_1) \sqsubseteq (\rho_1'', T_1) \sqsubseteq (\rho_1, T)$ by the IH. Then $(\rho_1' \sqcup \rho_2', S_1 S_2 \triangleright \langle \gamma, \mathbf{fun} f(x).S \rangle) \sqsubseteq (\rho_1 \sqcup \rho_2, T_1 T_2 \triangleright \langle \gamma, \mathbf{fun} f(x).T \rangle)$.

Case

$$\frac{v, T_1 \Downarrow^{-1} \rho_1[x_1 \mapsto v_1], e_1 \quad \mathbf{inl} v_1, T \Downarrow^{-1} \rho, e}{v, \mathbf{case} T \text{ of } \{\mathbf{inl}(x_1).T_1; \mathbf{inr}(x_2).e_2\} \Downarrow^{-1} \quad \rho_1 \sqcup \rho, \mathbf{case} e \text{ of } \{\mathbf{inl}(x_1).e_1; \mathbf{inr}(x_2).\square\}} v \neq \square$$

$$\frac{v, T_1' \searrow \rho_1'[x_1 \mapsto u_1], S_1 \quad \mathbf{inl} u_1, T' \searrow \rho', S}{v, \mathbf{case} T' \text{ of } \{\mathbf{inl}(x_1).T_1'; \mathbf{inr}(x_2).e_2'\} \searrow \quad \rho_1' \sqcup \rho', \mathbf{case} S \text{ of } \{\mathbf{inl}(x_1).S_1; \mathbf{inr}(x_2).\square\}} v \neq \square$$

Since $T_1 \sqsubseteq T_1'$, we have $(\rho_1'[x_1 \mapsto u_1], S_1) \sqsubseteq (\rho_1[x_1 \mapsto v_1], T_1)$ by the IH. Then $\rho_1' \sqsubseteq \rho_1$ and $u_1 \sqsubseteq v_1$ by Equation 5.1. Since $u_1 \sqsubseteq v_1$, we have $\mathbf{inl} u_1, T \Downarrow^{-1} \rho'', e'$ with $\rho'' \sqsubseteq \rho$ by Theorem 3. But since $T \sqsubseteq T'$, we also have $(\rho', S) \sqsubseteq (\rho'', T) \sqsubseteq (\rho, T)$ by the IH. Then $(\rho_1' \sqcup \rho', \mathbf{case} S \text{ of } \{\mathbf{inl}(x_1).S_1; \mathbf{inr}(x_2).\square\}) \sqsubseteq (\rho_1 \sqcup \rho, \mathbf{case} T \text{ of } \{\mathbf{inl}(x_1).T_1; \mathbf{inr}(x_2).e_2\})$. The case for the \mathbf{case} form that takes the \mathbf{inr} branch is similar. \square

B Delta Visualisation in LambdaCalc

LambdaCalc is able to calculate the changes to a computation implied by a program edit. This closely relates our method to techniques for incremental computation such as self-adjusting computation (Related Work, §3.10). Both approaches support *differential execution*: execution that maps input changes to output changes. Combined with an efficient update algorithm, differential execution can be used to implement interactive programs that respond automatically to input changes, eliminating the imperative event-handling code normally used to implement this sort of reactive behaviour.

In this appendix, we explore differential execution in LambdaCalc, without considering efficient update. We report on our use of LambdaCalc to build part of LambdaCalc’s own UI. Using reflection, we used differential execution to implement a key part of the UI code which visualises the deltas that differential execution itself produces. We visualise the delta between two computations by treating the views themselves as LambdaCalc values that can be differentially computed. We show some examples of visualisation functions, and discuss some of the issues that arise when writing LambdaCalc code with differential execution in mind.

B.1 Visualising differences by differencing visualisations

LambdaCalc not only calculates deltas, but visualises them, inviting the question of how these deltas should be visualised. Our idea is to treat the visualisation of an interactive program as another interactive program: a meta-program which takes the state of an interactive program as input and produces a visualisation of that state as output. The visualisation program responds to changes in the computation being visualised – and potentially to changes in its own code – by producing changes to the visualisation. We reduce the problem of visualising differences to the problem of differencing visualisations. This simplifies the problem of visualising differences, but also allows us to explore LambdaCalc’s potential for differential computation.

A key component of this self-hosting approach is *reflection*, the ability to convert values back and forth between the meta-language, which in our current implementation is Haskell, and the object language, LambdaCalc. Terminology in the reflection literature varies, but one convention has that *reflection* coerces from object language to meta-language and *reification* coerces in the other direction. This should not be confused with the notion of reification used elsewhere in this thesis.

Figure B.1 illustrates our UI architecture schematically. We start with a pair of LambdaCalc programs, represented as Haskell values, whose execution delta we wish to visualise. We evaluate the two programs into their respective traces (1). The traces are then reified into a pair of LambdaCalc values (2). At the LambdaCalc level, the traces are visualised, producing a pair of views (3); these views are values of a data type representing text strings (which somewhat inaccurately we call “glyphs”), shaded backgrounds, borders and

other primitive graphical components, plus constructors for various kinds of visual composition. The views are then reflected back into Haskell (4). Finally, they are passed to a delta-rendering function for visualisation (5). Rather than producing a pair of renderings, the delta-rendering code produces a rendering of one of the views (the *current* view), augmented with information about how it differs from the other view. We render changes with the simple highlighting scheme introduced in Chapter 2, but a fancier implementation would report changes continuously using animation.

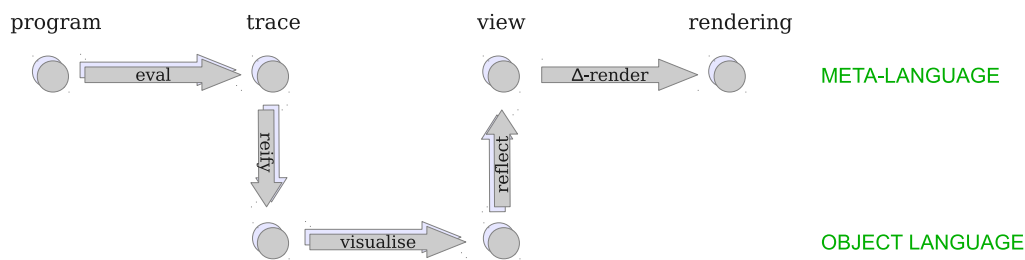


Figure B.1 Reflection-based UI architecture

By the delta-rendering stage the problem of visualising changes has been significantly simplified. There are no choices left to make about how to map trace changes to view changes, but only about how to present view changes. And all such changes are created equal: whether they arise as a consequence of changes to the thing being viewed, or as a result of changes to the visualisation code itself, changes to the view are rendered uniformly. The view becomes a medium that automatically advertises how its state is changing. This delta-rendering aspect of the UI cannot be implemented in LambdaCalc since it requires treating the views as term graphs rather than pure tree-structured values, whereas LambdaCalc values are pure. Currently we implement this as a layer of imperative Haskell that outputs to a GTK+ canvas.

The meaning of view changes is sometimes subtle and depends not only on what is being visualised but on specific details of how it was processed by the visualisation code. A novice programmer, or even an experienced programmer new to our system, would be unlikely to make immediate sense of every aspect of the UI's behaviour. But a hand-crafted change-visualisation scheme would likely suffer from these problems too, and be significantly more complex to boot. In the following section, we show some examples of visualisation functions and how the particular structure of the code can influence the differential behaviour of the UI. Coding decisions alter the circumstances under which view nodes are considered identical or distinct. This highlights a potential subtlety with differential computation in general, namely that one must often care about how code is written, rather than just how it behaves extensionally.

B.2 Coding for stability

Our visualisation code represents a different use case for interactive programming from those we have considered so far. Until now we have focused on allowing the programmer to see the impact of code changes on the intensional structure of a computation. For application development, we are more concerned with

extensional behaviour, i.e. how an interactive program maps input changes to output changes. The behaviour we see is a consequence both of the strategy we use for assigning indices to values described in Chapter 6 and of how we write the actual program. Studying this behaviour may help us design better differential execution schemes, or even different kinds of language better suited to differential execution. In this section we describe an idiom that has turned out to be essential for controlling the differential behaviour of the visualisation code, and illustrate with some examples from the code. The examples will also give the reader a feel for the complexity of the codebase, which is about 500 lines of LambdaCalc. Naturally, the figures are generated using the visualisation code itself.

For two LambdaCalc values to have the same identity in two different computations it is necessary, although not sufficient, that they be constructed by the same expression. Now suppose in our code we have a value which we want to construct with a particular constructor c , but for which where there are some conditional choices to be made to determine the arguments to be passed to c . Normally it would be common to duplicate the constructor code into each branch of the conditional logic. But the problem with this is that in different executions, different branches of the conditional may be taken and different expressions evaluated. So if we want it to be possible for this value to have the same identity in different executions, we have to swap things around so that there is a single constructor containing multiple copies of the conditional code.

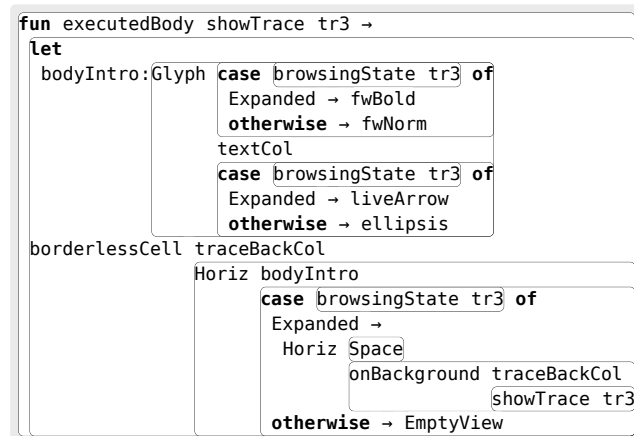


Figure B.2 Visualising executed function bodies

We see this pattern in the function `executedBody` shown in Figure B.2, which visualises the execution of a function body. We create a glyph, which we call `bodyIntro`, which will be in one of two states depending on whether the function body `tr3` is expanded. If the body is expanded, we want a control-flow arrow \rightarrow with bold font-weight; otherwise we want an ellipsis with normal font-weight. In both cases, we also want to pass `textCol` as the second component. We think of the three fields of the glyph as constituting its *state* because we would like the ellipsis to be able to *mutate into* the arrow, and vice-versa, without the identity of the glyph changing. That can only be achieved if there is a centralised point of construction for the glyph. But that means duplicating the conditional code that switches on the browsing state of `tr3`.

We see something similar in the `showValue` function in Figure B.3, which creates the view of a value v

which will appear in a value pane. Once again we centralise construction of a glyph, which we call `glyph`, so that in different executions it has the potential to be assigned the same identity. Here, this necessitates duplicating the case analysis of `v`. The idiom here is a little more problematic: we do not actually need the glyph in every branch of the conditional which follows the construction of `glyph`, but by centralising its construction, we end up having to construct it even for cases of `v` where we do not need it. This explains the otherwise clause that returns the dummy value "Unused".

When the conditional logic is too complicated to duplicate, an alternative is to construct an interim tuple in each branch of the conditional, holding the arguments for `c`. The tuple can be unpacked after the conditional and its components used to construct `c`. We see this in the `visualise` function given in Figure B.4, the top-level function responsible for visualising a computation in a cell. The goal here is to centralise the construction of a `RoundedCell` so that it will be uniquely associated with the computation `tr`, the argument to `visualise`. The cell has two components: a trace view `wt'` and an optional view pane `wv_opt` arranged in an "overlay", a kind of compact horizontal composition. To centralise the construction of the cell, we build an interim triple of the form `Pair(Pair(-, -), -)` containing the raw materials, store it in the variable `args_`, and then unpack the components of `args_` at the bottom of the function when we are ready to build the cell.

Unfortunately there is no generalisation of this centralised construction idiom that permits the constructor itself to be conditionally selected. This is because a given constructor expression must always mention a specific constructor. It is not therefore possible to write a `LambdaCalc` program where, for example, a value which was `Nil` in one execution becomes `Cons` in the next execution as the result of a different branch being taken. This feels like a limitation: the user can freely edit a `Nil` into a `Cons`, so it should equally be possible to "compute" such a change.

We end by saying something about accumulators, which can be problematic. Changing the identity of an argument to a function changes the identity of any values constructed in the body of the function. If one of these values is an accumulator, these changes of identity will cascade to all recursive invocations. However, this problem might not be as serious as it seems. Perhaps for historical reasons, functional programs tends to favour the use of lists when often more balanced data types would be more appropriate. Lists as normally defined have "right-bias": a right-fold of a binary operation over lists can be expressed directly as a list catamorphism, but a left-fold cannot, requiring an accumulator. In practice, we have found that we can commonly avoid accumulators by switching to a data type which is more redundant but also more symmetric. For example, vertical and horizontal view composition are monoids, for which the bracketing structure is irrelevant. For list-like structures, we can use a data type `RList` of *reversible lists* that combines both `Cons` cells and `Snoc` cells:

```
data RList a = Nil | Cons a (RList a) | Snoc (RList a) a
```

with similar properties.

In conclusion, with some care `LambdaCalc` can be used quite effectively for differential execution. The evidence is the delta visualisations we have been able to produce for this thesis. If programs are to exhibit useful differential behaviour, the programmer must adopt some idioms for constructing values, be conscious of the identity of values passed as argument to functions, and sometimes rethink the data types their functions operate on. Beyond these considerations, the programmer can for the most part write standard, purely

functional code. We could have made these general points more clearly, perhaps, with simpler examples that did not involve the meta-circular use of LambdaCalc to implement some of its own behaviour. On the other hand, this route has allowed us to demonstrate some of the potential for using interactive programming techniques to build non-trivial applications.

```

fun showValue showTrace v →
case v of
  NullValue → NullView
  otherwise →
    let
      showValue':showValue showTrace
    in
      RoundedCell
        case v of
          Constr(⟦_,_⟧) → Some borderCol
          otherwise → None
        Background valueBackCol
        let
          glyph:Glyph fwNorm
            textCol
              case v of
                ConstInt(n) → intToString n
                Constr(c,vs) → string c
                Op(op_) → string op_
                otherwise → "Unused."
          case v of
            ConstString(str) → showString str
            ConstInt(n) → glyph
            Constr(c,vs) →
              case isViewCtr c of
                True →
                  Horiz Separator borderCol
                    reflectWidth
                  Horiz Background reflectBackCol
                    reflect v
                  Separator borderCol
                    reflectWidth
                False →
                  Horiz glyph
                    case layoutVert c of
                      False →
                        argList showValue'
                          vs
                      True →
                        Horiz Space
                          constrArgs showValue'
                            vs
            Op(op_) → glyph
            Closure(f,x,e,rho) →
              showTrace TraceResult Unknown
                rho
                FunT f
                x
                e
                None
          otherwise →
            error v
            "[showValue] Impossible."

```

Figure B.3 Visualising values

```

fun visualise tr →
case trace tr of
NullTrace → NullView
otherwise →
let
wt_opt:case collapsed tr of
True → EmptyView
otherwise →
showTrace visualise
tr

let
traceOnly:Pair Pair None
wt_opt
EmptyView

let
args_:case trace tr of
OpT(_) → traceOnly
VarT(_) → traceOnly
otherwise →
case valueOpt tr of
None →
case trace tr of
ConstIntT(_) → traceOnly
ConstStringT(_) → traceOnly
otherwise →
Pair Pair Some borderCol
wt_opt
EmptyView
Some(v) →
let
const:isConstant tr
Pair Pair Some borderCol
case const of
True → EmptyView
False → wt_opt
case const of
True → EmptyView
otherwise →
LeftConnector borderCol
valueBackCol
RoundedCell Some borderCol
onBackground valueBackCol
showValue showTrace visualise
v

let
border:(fst fst args_)
let
wt':onBackground traceBackCol
snd fst args_
let
wv_opt:snd args_
RoundedCell border
Overlay wt'
wv_opt

```

Figure B.4 Visualising a computation in a cell