Dynamic Slicing of Lazy Functional Programs Based on Redex Trails [*]

Claudio Ochoa
DIA, Technical University of Madrid
Campus de Montegancedo s/n, E-28660 Boadilla del Monte (Spain)
(`cochoa@fi.upm.es`*)*

Josep Silva
DSIC, Technical University of Valencia
Camino de Vera s/n, E-46022 Valencia (Spain)
(`jsilva@dsic.upv.es`*)*

Germán Vidal
DSIC, Technical University of Valencia
Camino de Vera s/n, E-46022 Valencia (Spain)
(`gvidal@dsic.upv.es`*)*

Abstract. Tracing computations is a widely used methodology for program debugging. Lazy languages, however, pose new demands on tracing techniques because following the *actual* trace of a computation is generally useless. Typically, tracers for lazy languages rely on the construction of a *redex trail*, a graph that stores the reductions performed in a computation. While tracing provides a significant help for locating bugs, the task still remains complex. A well-known debugging technique for imperative programs is based on *dynamic slicing*, a method for finding the program statements that influence the computation of a value for a specific program input.

In this work, we introduce a novel technique for dynamic slicing in first-order lazy functional languages. Rather than starting from scratch, our technique relies on (a slight extension of) redex trails. We provide a notion of dynamic slice and introduce a method to compute it from the redex trail of a computation. We also sketch the extension of our technique to deal with a functional *logic* language. A clear advantage of our proposal is that one can enhance existing tracers with slicing capabilities with a modest implementation effort, since the same data structure (the redex trail) can be used for both tracing and slicing.

Keywords: lazy functional programming, debugging, slicing, redex trails

1. Introduction

In lazy functional programming languages, following the *actual* trace of a computation is often useless due to the *on demand* style of evaluation. In the language Haskell (Peyton Jones, 2003), this drawback has been

[*] A preliminary version of this paper appeared in the Proceedings of PEPM'04 (Ochoa et al., 2004). This work has been partially supported by EU (FEDER) and Spanish MEC under grants TIN2004-00231, TIN2005-09207-C03-02, and *Acción Integrada* HA2006-0008.

overcome by introducing higher-level models that hide the details of lazy evaluation. This is the case of the following systems: Hood (Gill, 2000), Freja (Nilsson and Sparud, 1997), Buddha (Pope, 2006; Pope and Naish, 2003), and Hat (Sparud and Runciman, 1997; Wallace et al., 2001). Some of these approaches are based on the construction of a *redex trail* (Sparud and Runciman, 1997), a directed graph which records copies of all values and redexes (*red*ucible *ex*pressions) of a computation. Basically, redex trails allow us to present different—more intuitive—views of a given computation (e.g., by showing function calls with fully evaluated arguments). Wallace et al. (2001) introduced an extended trail, the *augmented redex trail* (ART), in order to cover all previous three approaches (i.e., those followed by Hood, Freja and Hat).

Braßel et al. (2004) introduce an instrumented version of an operational semantics for the kernel of a lazy functional *logic* language like Curry (Hanus, 2000) that returns not only the computed values and bindings, but also a trail of the computation. Similarly to the ART approach, this trail can also be used to perform tracing and algorithmic debugging, as well as to observe data structures through a computation. In contrast to previous approaches, the correctness of the computed trail is formally proved, which amounts to saying that it contains all (and only) the reductions performed in a computation.

While tracing-based debugging provides a powerful tool for inspecting erroneous computations, it should be clear that the task still remains complex. A well-known debugging technique for imperative programs is known as *slicing* (Weiser, 1984), a method for decomposing programs by analyzing their data and control flow. Intuitively speaking, a *program slice* consists of the statements which are (potentially) related to the values computed at some program point and/or variable, referred to as a *slicing criterion.* Program slices are usually computed from a *program dependence graph* (Ferrante et al., 1987; Kuck et al., 1981) that makes explicit both the data and control dependences for each operation in a program. Program dependences can be traversed backwards or forwards—from the slicing criterion—giving rise to so-called *backward* and *forward* slices, respectively. Additionally, slices can be *dynamic* or *static*, depending on whether a concrete program's input is provided or not. A detailed overview of program slicing has been presented by Tip (1995).

Essentially, a backward slice consists of the parts of the program that (potentially) affect the values computed from the slicing criterion. In contrast, a forward slice consists of the statements which are dependent on the slicing criterion, a statement being *dependent* on the slicing criterion if the values computed at that statement depend on the values computed at the slicing criterion or if the values computed at the slicing criterion determine whether the statement under consideration is

```
(1) read(n);           (1) read(n);           (1)
(2) i := 1;            (2) i := 1;            (2)
(3) sum := 0;          (3)                    (3) sum := 0;
(4) prod := 1;         (4) prod := 1;         (4)
(5) while i <= n do    (5) while i <= n do    (5)
(6)   sum := sum + i;  (6)                    (6)   sum := sum + i;
(7)   prod := prod * i; (7)   prod := prod * i; (7)
(8)   i := i + 1;      (8)   i := i + 1;      (8)
(9) write(sum);        (9)                    (9) write(sum);
(10)write(prod);       (10)write(prod);       (10)

        (a)                    (b)                    (c)
```

Figure 1. Forward and backward slicing —an example

executed (Tip, 1995). Consider, e.g., the example (Tip, 1995) depicted in Figure 1 (a) for computing the sum and the product of the sequence of numbers `1,2,...,n`. Figure 1 (b) shows a backward slice of the program w.r.t. the slicing criterion `(10,prod)` while Figure 1 (c) shows a forward slice w.r.t. the slicing criterion `(3,sum)`.

When debugging functional programs, we usually start from a particular computation that outputs an incorrect value. In this situation, dynamic slicing may help the programmer in finding the location of the bug by extracting a program slice which only contains the sentences whose execution influenced the incorrect value. Therefore, we are mainly interested in dynamic slicing (Korel and Laski, 1988), which only reflects the actual dependences of the erroneous execution, thus producing smaller—more precise—slices than static slicing (Tip, 1995).

In this work, we present the first dynamic backward slicing technique for a first-order lazy functional—or functional logic—language, i.e., the first-order component of languages like Haskell or Curry. Rather than starting from scratch, our technique relies on a slight extension of redex trails. To be precise, we extend (a simplified version of) the redex trail model of Braßel et al. (2004) in order to also store the *location*, in the program, of each reduced expression. A dynamic slice is then computed by (i) first gathering the set of nodes—in the redex trail—which are *reachable* from the slicing criterion and, then, (ii) by deleting (or hiding) those expressions of the original program whose locations do not appear in the set of collected nodes (see the next section for an informal overview of our debugging technique).

From our point of view, dynamic slicing may provide a complementary—rather than alternative—approach to tracing-based debugging. A clear advantage of our approach is that one can enhance existing tracers with slicing capabilities with a modest implementation effort, since the same data structure—the redex trail—is used for both tracing and slicing.

The main contributions of this work can be summarized as follows:

– We introduce a flexible notion of *slicing criterion* for a first-order lazy functional (logic) language. Previous definitions, e.g., (Biswas, 1997a; Reps and Turnidge, 1996), only considered a projection of the entire program as a slicing criterion.

– We extend (a simplified version of) the instrumented semantics of Braßel et al. (2004) in order to have an extended redex trail that also stores the positions, in the source program, of each reduced expression, which is essential for slicing.

– We provide a notion of dynamic slice and introduce a method to compute it from the extended redex trail of a computation.

– We show how tracing—based on redex trails—and slicing can be combined into a single framework. Moreover, the viability of our approach is supported by a prototype implementation of an integrated debugging tool.

This article is organized as follows. In the next section, we present an informal overview of our approach to combine tracing and slicing into a single framework. Section 3 presents the syntax of the considered language. Then, Section 4 formalizes an instrumented semantics that builds an extended redex trail of a computation which also includes the program positions of expressions. Section 5 defines the main concepts involved in dynamic slicing and introduces a method to compute dynamic slices from the extended redex trail. Section 6 presents some implementation issues of a prototypical implementation which integrates both tracing and slicing. Section 7 includes a comparison to related work and, finally, Section 8 concludes and points out several directions for further research. Proofs of technical results can be found in an appendix.

2. Combining Tracing and Slicing

In this section, we present an overview of a debugging tool for lazy functional languages that combines both tracing and dynamic slicing based on redex trails.

A *redex trail*—which was originally introduced by Sparud and Runciman (1997) in the context of lazy functional programs—is defined as a directed graph which records copies of all values and redexes (*red*ucible *ex*pressions) of a computation, with a backward link from

```
main = printMax (minmax [Z, S Z])

printMin t = case t of { Pair x y → printNat x }
printMax t = case t of { Pair x y → printNat y }

printNat n = case n of { Z → 0; (S m) → 1 + printNat m }

fst t = case t of { Pair x y → x }
snd t = case t of { Pair x y → y }

minmax xs = case xs of
            { y:ys → case ys of
                      { []   → Pair y y;
                        z:zs → let m = minmax (z:zs)
                               in Pair (min y (fst m))
                                       (max y (snd m)) } }
min x y = ite (leq x y) x y
max x y = ite (leq x y) y x

ite x y z = case x of { True → y; False → z }

leq x y = case x of
          { Z     → False;
           (S n) → case y of { Z → False; (S m) → leq n m } }
```

Figure 2. Example program `minmax`

each reduct (and its proper subexpressions) to the parent redex that created it. The ART (*Augmented Redex Trail*) model of the Haskell tracer Hat (Wallace et al., 2001) mainly extends redex trails by also including forward links from every redex to its reduct. Braßel et al. (2004) introduce a data structure which shares many features with the ART model but also includes a special treatment to cope with non-determinism (i.e., disjunctions and flexible case structures) so that it can be used for functional *logic* languages. Furthermore, in contrast to previous approaches, the redex trails of Braßel et al. (2004) are defined by instrumenting an operational semantics for the language (rather than by a program transformation) and their correctness is formally proved. Let us present this tracing technique by means of an example.

Example 1. Consider the program[1] shown in Figure 2—inspired by a similar example by Liu and Stoller (2003)—where data structures are built from

```
data Nat     = Z | S Nat
data Pairs   = Pair Nat Nat
data ListNat = Nil | Cons Nat ListNat
```

[1] Although we consider a first-order language in this work, here and in the following examples we omit some of the brackets and write function applications as in Haskell for the sake of readability.

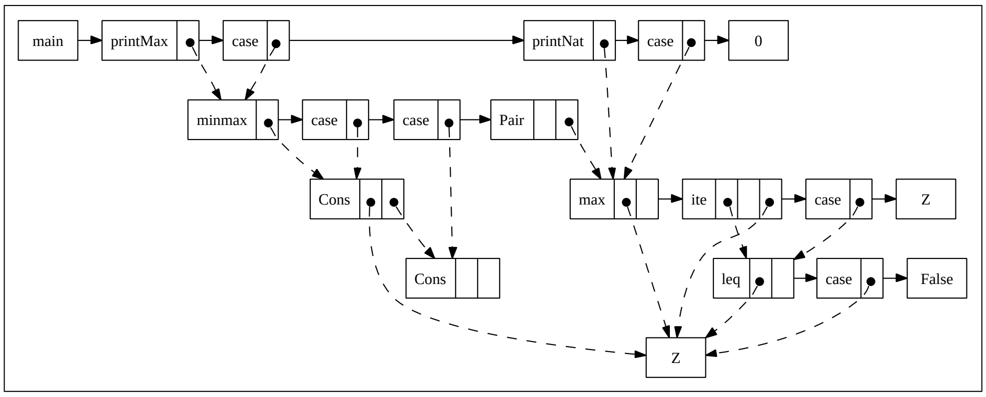


Figure 3. Redex trail for Example 1

As it is common practice in functional languages, we use "`[]`" and "`:`" as a shorthand for `Nil` and `Cons`.

From now on, we assume that computations always start from the distinguished function `main` which has no arguments. The execution of the program in Figure 2 should compute the maximum of the list `[Z, S Z]`, i.e., `S Z`, and thus return `1`; however, it returns `0`.

In order to trace the execution of this program, the tracing tool of Braßel et al. (2004) builds the redex trail shown in Figure 3. In this redex trail, we can distinguish two kinds of arrows:[2]

Successor arrows: There is a successor arrow, denoted by a solid arrow, from each redex to its reduct (e.g., from the node labeled with `main` to the node labeled with `printMax` in Figure 3).

Argument arrows: Arguments of both function and constructor calls are denoted by a pointer to the corresponding expression, which is denoted by a dashed arrow. When the evaluation of an argument is not required in a computation, we have a null pointer for that argument (e.g., the first argument of the node labeled with `Pair` in Figure 3).

From the computed trail, the user can inspect the trace of a computation. In our example, the tracing tool initially shows the following top-level computation for `main`:

```
0 = main
0 = printMax (Pair _ Z)
```

[2] The redex trails of Braßel et al. (2004) also contain a third kind of arrows—the *parent* arrows—that we ignore in this work because they are not necessary for computing program slices.

```
0 = printNat Z
0 = 0
```

where "_" denotes an argument whose evaluation is not needed (i.e., a null pointer in the redex trail). Each row shows a pair "*val* = *exp*" where *exp* is an expression and *val* is its (possibly partial) value. Note that this top-level computation corresponds to the six topmost nodes in Figure 3, where the two nodes labeled with a `case` structure are ignored to simplify the presentation. Note also that *function arguments appear as much evaluated as needed in the complete computation* in order to ease the understanding of the trace.

From the trace above, the user can easily see that the argument of `printMax` is incorrect because the second argument of constructor `Pair`, i.e., the maximum of the input list, should be `(S Z)` rather than `Z`. If the user selects the argument of `printMax`, the expression `(Pair _ Z)`, the following subtrace is shown:

```
0        = printMax (Pair _ Z)
Pair _ Z = minmax   (Z : _ : _)
Pair _ Z = Pair     _           Z
```

Here, the user can conclude that the *evaluation* of `minmax` (rather than its *definition*!) contains a bug because it returns `Z` (the second element of `Pair`) as the maximum of a list headed by `Z`, ignoring the remaining elements of the list (which were not evaluated in the computation).

Now, the user can either try to figure out the location of the bug by computing further subtraces (which ones?) or by isolating the code fragment which is responsible of the wrong result by *dynamic slicing* and, then, inspecting the (hopefully smaller) slice.

We propose the second approach and will show that incorporating dynamic slicing into existing redex-trail based tracers is simple and powerful. Essentially, our approach allows the user to easily obtain the program slice associated with any entry *val* = *exp* in the trace of a computation. Optionally, the user may provide a *pattern* π that indicates which part of *val* is of interest and which part is not. Then, a dynamic slice including all program constructs that are (potentially) needed to compute the relevant part of *val*—according to π—from *exp* is shown. Therefore, in our context, the slicing criterion is given by a tuple of the form $\langle exp, \pi \rangle$.

We consider that patterns for slicing are built from data constructor symbols, the special symbol $\bot$ (which denotes a subexpression of the value whose computation is not relevant), and $\top$ (a subexpression which is relevant). Consider, e.g., the slicing criterion

$$\langle \texttt{minmax}\ (\texttt{Z} : \_ : \_),\ \texttt{Pair}\ \bot\ \top \rangle$$

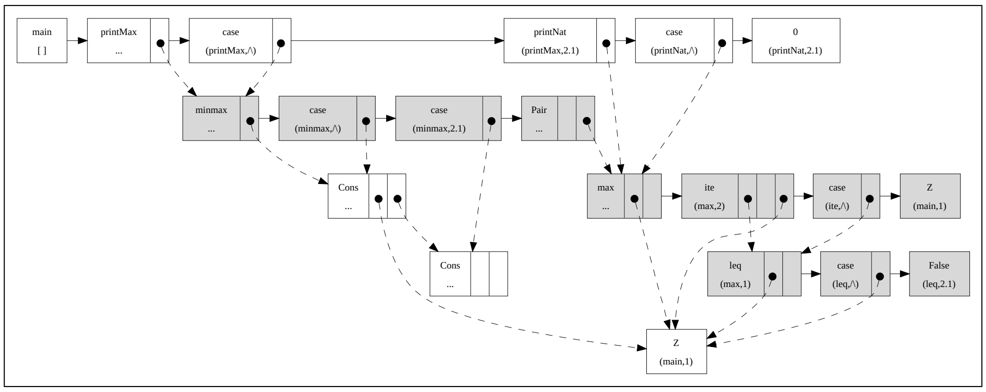


Figure 4. Extended redex trail for Example 1

associated with the erroneous entry of the previous trace for `minmax`. Intuitively, it determines the extraction of a slice with the expressions that are involved in the computation of the second argument of `Pair` starting from the call to `minmax`. We note, however, that the expressions involved in the evaluation of the *arguments* of the call to `minmax` will not be part of the slice. This is reasonable since the user identified a bug in the fact that `minmax (Z : _ : _)` evaluated to a wrong result, assuming that the arguments of `minmax` were already evaluated.

Similarly to Reps and Turnidge (1996), we consider the computation of *backward* slices where the information given by the slicing pattern π should be propagated backwards through the computation to determine the program constructs that are necessary to compute the relevant part of *exp* (according to π).

In order to compute the associated program slice, we extend the redex trail model in order to also store the position, in the source program, of every expression in the redex trail. Program positions are denoted by a pair $(\mathit{fun}, \mathit{pos})$ where *fun* is a function symbol and *pos* is a sequence of natural numbers that precisely identify a subexpression in the definition of *fun* (see Section 4.3 for a detailed definition).

For the example above, the extended redex trail including *some* of the program positions—more details on this example can be found in Section 5—is shown in Figure 4. The leftmost shadowed node stores the expression associated with the slicing criterion, while the remaining shadowed nodes store the expressions which are reachable from the slicing criterion (according to the slicing pattern).

The computed slice is thus the set of program positions in the shadowed nodes of the extended redex trail. The corresponding program slice is shown in Figure 5, where expressions that do not belong to the slice appear in gray. By inspecting the slice, we can check that `minmax` only calls to function `max` which, in turn, calls to functions `ite`, a

```
main = printMax (minmax [Z, S Z])

printMin t = case t of { Pair x y → printNat x }
printMax t = case t of { Pair x y → printNat y }

printNat n = case n of { Z → 0; (S m) → 1 + printNat m }

fst t = case t of { Pair x y → x }
snd t = case t of { Pair x y → y }

minmax xs = case xs of
            { y:ys → case ys of
                      { []    → Pair y y;
                        z:zs → let m = minmax (z:zs)
                                in Pair (min y (fst m))
                                        (max y (snd m)) } }
min x y = ite (leq x y) x y
max x y = ite (leq x y) y x

ite x y z = case x of { True → y; False → z }

leq x y = case x of
          { Z    → False;
           (S n) → case y of { Z → False; (S m) → leq n m } }
```

Figure 5. Slice of program `minmax`

standard conditional, and `leq` (for “less than or equal to”). However, the only case which is not deleted from the definition of `leq` can be read as “`Z` is *not* less than or equal to any natural number”, which is clearly wrong.

3. The Language

In this work, we consider *flat* programs (Hanus and Prehofer, 1999), a convenient standard representation for first-order functional (logic) programs which makes explicit the pattern matching strategy by the use of case expressions. It constitutes the kernel of modern declarative multi-paradigm languages like Curry (Hanus, 1997; Hanus, 2000) and Toy (López-Fraguas and Sánchez-Hernández, 1999). In addition, we assume in the following that flat programs are *normalized*, i.e., *let* constructs are used to ensure that the arguments of functions and constructors are always variables (not necessarily pairwise different). This is essential to express *sharing* without the use of complex graph structures. A simple normalization algorithm can be found in (Albert et al., 2005). Basically, this algorithm introduces one new let construct for each non-variable argument of a function or constructor call, e.g., $f(e)$ is transformed into “*let* $x = e$ *in* $f(x)$.”

P	$::=$	$D_1 \ldots D_m$		*Domains*
D	$::=$	$f(x_1, \ldots, x_n) = e$		
$Exp \ni e$	$::=$	x	(variable)	Variables:
	$\mid$	$c(x_1, \ldots, x_n)$	(constructor call)	$\mathcal{X} = \{x, y, z, \ldots\}$
	$\mid$	$f(x_1, \ldots, x_n)$	(function call)	
	$\mid$	$let\ x = e_1\ in\ e_2$	(let binding)	Constructors:
	$\mid$	$e_1\ or\ e_2$	(disjunction)	$\mathcal{C} = \{a, b, c, \ldots\}$
	$\mid$	$case\ x\ of\ \{\overline{p_n \to e_n}\}$	(rigid case)	
	$\mid$	$fcase\ x\ of\ \{\overline{p_n \to e_n}\}$	(flexible case)	Defined functions:
p	$::=$	$c(x_1, \ldots, x_n)$	(flat pattern)	$\mathcal{F} = \{f, g, h, \ldots\}$

Figure 6. Syntax for normalized flat programs

The syntax for normalized flat programs is shown in Figure 6, where $\overline{o_n}$ denotes the *sequence of objects* $o_1, \ldots, o_n$. A program P consists of a sequence of function definitions D such that the left-hand side has pairwise different variable arguments. The right-hand side is an expression e composed of variables, data constructors, function calls, let bindings where the local variable x is only visible in e_1 and e_2, disjunctions (e.g., to represent set-valued functions), and case expressions. In general, a case expression has the following form:[3]

$$(f)case\ x\ of\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots; c_k(\overline{x_{n_k}}) \to e_k\}$$

where x is a variable, $c_1, \ldots, c_k$ are different constructors of the type of x, and $e_1, \ldots, e_k$ are expressions. The *pattern variables* $\overline{x_{n_i}}$ are locally introduced and bind the variables of e_i. The difference between *case* and *fcase* only shows up when the argument x evaluates (at run time) to a free variable: *case* suspends[4] whereas *fcase* non-deterministically binds this variable to the pattern in a branch of the case expression.

Laziness (or neededness) of computations will show up in the description of the behavior of function calls and case expressions. In a function call, parameters are not evaluated but directly passed to the body of the function. In a case expression, the outermost symbol of the case argument is required. Therefore, the case argument should be evaluated to *head normal form*, i.e., a variable or an expression with a constructor at the outermost position. Consequently, the associated operational semantics describes the evaluation of expressions only to head normal form. This is not a restriction because the evaluation to normal form can be reduced to head normal form computations (Hanus and Prehofer, 1999).

[3] We write $(f)case$ for either *fcase* or *case*.

[4] In our context, a suspended computation represents a failure. In general, if one also considers a concurrent conjunction operator, then a suspended computation may resume when a variable is bound in a different thread; see, e.g., (Hanus, 2000).

Extra variables are those variables in a rule which do not occur in the left-hand side. Such extra variables are intended to be instantiated by flexible case expressions. In the following, we assume that all extra variables x are explicitly introduced in flat programs by a direct circular let binding of the form "$let\ x = x\ in\ e$". In this work, we call such variables which are bound to themselves *logical variables.*

4. Building the Extended Trail

In this section, we extend the instrumented semantics of Braßel et al. (2004) so that the computed redex trail also stores *program positions*; this additional information will become useful to identify precisely the location in the source program of each expression in the trail.

In order to keep the paper self-contained and to ease the understanding, we introduce our instrumented semantics in a stepwise manner. Furthermore, we consider a pure (first-order) functional language (i.e., we consider neither disjunctions nor flexible case structures) in Sections 4.1, 4.2, and 4.3; the treatment of disjunctions and flexible cases is sketched in Section 4.4.

4.1. The Basic Operational Semantics

First, we present the basic (small-step) operational semantics of Albert et al. (2005). This semantics obeys the following naming conventions:

$$\Gamma \in Heap = \mathcal{X} \to Exp \qquad\qquad v \in Value ::= x \mid c(\overline{x_n})$$

Each configuration of the small-step semantics is a triple, $\langle \Gamma, e, S\rangle$, where Γ is a heap, e is an expression (often called the *control* of the small-step semantics), and S is a stack. A *heap* is a partial mapping from variables to expressions (the *empty heap* is denoted by `[]`). The value associated with variable x in heap Γ is denoted by $\Gamma[x]$. We use the notation $\Gamma[x \mapsto e]$ to extend the heap Γ with a mapping from variable x to expression e. In a heap Γ, a logical variable x is represented by a circular binding of the form $\Gamma[x] = x$. A *stack* (a list of variable names and case alternatives where the empty stack is denoted by `[]`) is used to represent the current context. A *value*—a head normal form—is a constructor-rooted term (i.e., a term rooted by a constructor symbol) or a logical variable (w.r.t. the associated heap).

The transition rules are shown in Figure 7. A variable is evaluated by means of the rules varcons and varexp, depending on whether the variable is bound in the heap to a constructor-rooted term or to an arbitrary expression. Once a value is eventually computed, the computation either terminates—when the stack is empty—or this value is used to update the current heap (rule val).

varcons

$\langle \Gamma[x \mapsto t], x, S\rangle \Longrightarrow \langle \Gamma[x \mapsto t], t, S\rangle$ where t is constructor-rooted

varexp

$\langle \Gamma[x \mapsto e], x, S\rangle \Longrightarrow \langle \Gamma[x \mapsto e], e, x : S\rangle$ where e is not constructor-rooted and $e \neq x$

val

$\langle \Gamma[x \mapsto e], v, x : S\rangle \Longrightarrow \langle \Gamma[x \mapsto v], v, S\rangle$ where v is a value

fun

$\langle \Gamma, f(\overline{x_n}), S\rangle \Longrightarrow \langle \Gamma, \rho(e), S\rangle$ where $f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$

let

$\langle \Gamma, \mathit{let}\ x = e_1\ \mathit{in}\ e_2, S\rangle \Longrightarrow \langle \Gamma[y \mapsto \rho(e_1)], \rho(e_2), S\rangle$ where $\rho = \{x \mapsto y\}$ and y is a fresh variable

case

$\langle \Gamma, \mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\}, S\rangle \Longrightarrow \langle \Gamma, x, \{\overline{p_k \to e_k}\} : S\rangle$

select

$\langle \Gamma, c(\overline{y_n}), \{\overline{p_k \to e_k}\} : S\rangle \Longrightarrow \langle \Gamma, \rho(e_i), S\rangle$ where $i \in \{1, \ldots k\}$, $p_i = c(\overline{x_n})$, and $\rho = \{\overline{x_n \mapsto y_n}\}$

Figure 7. Basic small-step operational semantics

Rule fun performs a simple function unfolding; here, we assume that the considered program P is a global parameter of the calculus.

In rule let, we rename the local variable with a fresh name in order to avoid variable name clashes.

The evaluation of case expressions proceeds in two stages. First, rule case starts the evaluation of the case argument and pushes the alternatives $\{\overline{p_k \to e_k}\}$ on top of the stack. If a constructor-rooted term is computed, then rule select is used to select the appropriate branch and continue with the evaluation of this branch.

Example 2. Consider the function `leq` of Example 1 (where the bug is now corrected), together with the initial call `(leq Z (S Z))`. The corresponding normalized flat program is the following:

```
main = let x3 = Z in
       let x1 = Z in
       let x2 = S x3 in leq x1 x2

leq x y = case x of
          { Z     → True;
           (S n) → case y of { Z     → False;
                               (S m) → leq n m } }
```

The complete computation with the rules of Figure 7 is shown in Figure 8. For clarity, each computation step is labeled with the applied rule. Therefore, `main` computes the value `True`.

$$
\begin{array}{llllr}
\langle [\,], \texttt{main}, [\,]\rangle & \Longrightarrow_{\mathsf{fun}} & \langle [\,], & \texttt{let x3 = Z in ...}, & [\,]\rangle \\
 & \Longrightarrow_{\mathsf{let}} & \langle [\mathsf{x3} \mapsto \mathsf{Z}], & \texttt{let x1 = Z in ...}, & [\,]\rangle \\
 & \Longrightarrow_{\mathsf{let}} & \langle [\mathsf{x1} \mapsto \mathsf{Z}, \ldots], & \texttt{let x2 = S x3 in ...}, & [\,]\rangle \\
 & \Longrightarrow_{\mathsf{let}} & \langle [\mathsf{x2} \mapsto \mathsf{S\ x3}, \ldots], & \texttt{leq x1 x2}, & [\,]\rangle \\
 & \Longrightarrow_{\mathsf{fun}} & \langle [\ldots], & \texttt{case x1 of ...}, & [\,]\rangle \\
 & \Longrightarrow_{\mathsf{case}} & \langle [\mathsf{x1} \mapsto \mathsf{Z}, \ldots], & \texttt{x1}, & [\{\ldots\}]\rangle \\
 & \Longrightarrow_{\mathsf{varcons}} & \langle [\mathsf{x1} \mapsto \mathsf{Z}, \ldots], & \texttt{Z}, & [\{\ldots\}]\rangle \\
 & \Longrightarrow_{\mathsf{select}} & \langle [\ldots], & \texttt{True}, & [\,]\rangle
\end{array}
$$

Figure 8. Derivation of Example 2

4.2. A Semantics for Tracing

Now, we present the instrumented semantics of Braßel et al. (2004) to construct a redex trail of the computation. However, in contrast to (Braßel et al., 2004), we do not consider the "parent" relation because it is not useful for computing program slices. Basically, the trail is a directed graph with nodes identified by references[5] that are labeled with expressions. In the construction of this trail, the following conventions are adopted:

- $r \mapsto e$ denotes that the node with reference r is labeled with expression e. Successor arrows are denoted by $r \underset{q}{\mapsto}$ which means that node q is the successor of node r. The notation $r \underset{q}{\mapsto} e$ is used to denote both $r \mapsto e$ and $r \underset{q}{\mapsto}$.
- Argument arrows are denoted by $x \rightsquigarrow r$ which means that variable x points to node r. This relation suffices to store function arguments in our context because only variable arguments are allowed in normalized programs. These arrows are also called *variable pointers.*

Configurations in the instrumented semantics are tuples of the form $\langle \Gamma, e, S, G, r\rangle$; here, r denotes the *current* reference—where e will be eventually stored—and G is the trail built so far. Analogously to the heap, $G[r \mapsto e]$ is used to extend the graph G with a new node r labeled with the expression e.

The rules of the instrumented semantics are shown in Figure 9. Both rules varcons and varexp add a new variable pointer for x to the current graph if it does not yet contain such a pointer. This is used to account for the fact that the value of x is *needed* in the computation. For this

[5] The domain for references is not fixed. One can use, e.g., natural numbers but more complex domains are also possible.

varcons

$$\begin{array}{ll} \langle \Gamma[x \mapsto t], x, S, G, r\rangle & \\ \Longrightarrow \langle \Gamma[x \mapsto t], t, S, G \bowtie (x \rightsquigarrow r), r\rangle & \text{where } t \text{ is constructor-rooted} \end{array}$$

varexp

$$\begin{array}{ll} \langle \Gamma[x \mapsto e], x, S, G, r\rangle & \text{where } e \text{ is not constructor-} \\ \Longrightarrow \langle \Gamma[x \mapsto e], e, x : S, G \bowtie (x \rightsquigarrow r), r\rangle & \text{rooted and } e \neq x \end{array}$$

val

$$\langle \Gamma[x \mapsto e], v, x : S, G, r\rangle \Longrightarrow \langle \Gamma[x \mapsto v], v, S, G, r\rangle \qquad \text{where } v \text{ is a value}$$

fun

$$\begin{array}{ll} \langle \Gamma, f(\overline{x_n}), S, G, r\rangle & \text{where } f(\overline{y_n}) = e \in P,\ \rho = \{\overline{y_n \mapsto x_n}\} \\ \Longrightarrow \langle \Gamma, \rho(e), S, G[r \underset{q}{\mapsto} f(\overline{x_n})], q\rangle & \text{and } q \text{ is a fresh reference} \end{array}$$

let

$$\begin{array}{ll} \langle \Gamma, let\ x = e_1\ in\ e_2, S, G, r\rangle & \text{where } \rho = \{x \mapsto y\}, \\ \Longrightarrow \langle \Gamma[y \mapsto \rho(e_1)], \rho(e_2), S, G[r \underset{q}{\mapsto} \rho(let\ x = e_1\ in\ e_2)], q\rangle & y \text{ is fresh variable,} \\ & \text{and } q \text{ is a fresh ref.} \end{array}$$

case

$$\begin{array}{ll} \langle \Gamma, case\ x\ of\ \{\overline{p_k \to e_k}\}, S, G, r\rangle & \text{where } q \text{ is a} \\ \Longrightarrow \langle \Gamma, x, (\{\overline{p_k \to e_k}\}, r) : S, G[r \mapsto case\ x\ of\ \{\overline{p_k \to e_k}\}], q\rangle & \text{fresh reference} \end{array}$$

select

$$\begin{array}{ll} \langle \Gamma, c(\overline{y_n}), (\{\overline{p_k \to e_k}\}, r') : S, G, r\rangle & \text{where } i \in \{1, \ldots k\},\ p_i = c(\overline{x_n}), \\ \Longrightarrow \langle \Gamma, \rho(e_i), S, G[r \mapsto c(\overline{y_n}), r' \underset{q}{\mapsto}], q\rangle & \rho = \{\overline{x_n \mapsto y_n}\}, \text{ and } q \text{ is a fresh ref.} \end{array}$$

Figure 9. Instrumented semantics for tracing computations

purpose, the auxiliary function "$\bowtie$" is introduced:

$$G \bowtie (x \rightsquigarrow r) = \begin{cases} G[x \rightsquigarrow r] & \text{if there is no } r' \text{ such that } (x \rightsquigarrow r') \in G \\ G & \text{otherwise} \end{cases}$$

Observe that this function is necessary to take care of variable sharing: if the value of a variable was already demanded in the computation, no new variable pointer should be added to the graph.

In the rule val, the current graph is not modified because computed values are only stored when they are *used* (e.g., in rule select).

In the rules fun and let, a new node r, labeled with the current expression in the control, is added to the graph. The successor is set to q, a fresh reference, which is now the current reference of the derived configuration.

In the rule case we now push on the stack, not only the case alternatives, but also the current reference. This reference will become useful later if the rule select is eventually applied, in order to set the right successor for the case expression (which is not yet known when rule case is applied). Note that no successor for node r is set in rule select because *values* are fully evaluated and, thus, have no successor.

When tracing computations, however, we are also interested in *failing* derivations. For this purpose, rule select-f of Figure 10 is added to the calculus. This rule applies when there is no matching branch in a case expression. Furthermore, since expressions are added to the select-f

$$\begin{array}{ll} \langle \Gamma, c(\overline{y_n}), (\{\overline{p_k \to e_k}\}, r') : S, G, r\rangle & \text{where } \not\exists i \in \{1, \ldots k\} \\ \Longrightarrow \langle \Gamma, \mathit{Fail}, S, G[r \mapsto c(\overline{y_n}), r' \underset{q}{\mapsto}], q\rangle & \text{such that } p_i = c(\overline{x_n}) \end{array}$$

success

$$\langle \Gamma, c(\overline{x_n}), \texttt{[]}, G, r\rangle \Longrightarrow \langle \Gamma, \Diamond, \texttt{[]}, G[r \mapsto c(\overline{x_n})], \Box\rangle$$

Figure 10. Tracing semantics: failing and success rules

$$\begin{array}{llll} \multicolumn{4}{l}{\langle [\,], \texttt{main}, [\,], \texttt{G}_\emptyset, 0\rangle} \\ \Longrightarrow_{\mathsf{fun}} & \langle [\,], & \texttt{let x3 = Z in ...}, & [\,], \texttt{G}_0[0 \underset{1}{\mapsto} \texttt{main}], \quad 1\rangle \\ \Longrightarrow_{\mathsf{let}} & \langle [\texttt{x3} \mapsto \texttt{Z}], & \texttt{let x1 = Z in ...}, & [\,], \texttt{G}_1[1 \underset{2}{\mapsto} \texttt{let x3 = Z in ...}], \quad 2\rangle \\ \Longrightarrow_{\mathsf{let}} & \langle [\texttt{x1} \mapsto \texttt{Z}, \ldots], & \texttt{let x2 = S x3 in ...}, & [\,], \texttt{G}_2[2 \underset{3}{\mapsto} \texttt{let x1 = Z in ...}], \quad 3\rangle \\ \Longrightarrow_{\mathsf{let}} & \langle [\texttt{x2} \mapsto \texttt{S x3}, \ldots], & \texttt{leq x1 x2}, & [\,], \texttt{G}_3[3 \underset{4}{\mapsto} \texttt{let x2 = S x3 in leq x1 x2}], \quad 4\rangle \\ \Longrightarrow_{\mathsf{fun}} & \langle [\ldots], & \texttt{case x1 of ...}, & [\,], \texttt{G}_4[4 \underset{5}{\mapsto} \texttt{leq x1 x2}], \quad 5\rangle \\ \Longrightarrow_{\mathsf{case}} & \langle [\texttt{x1} \mapsto \texttt{Z}, \ldots], & \texttt{x1}, & [(\{\ldots\}, 5)], \texttt{G}_5[5 \mapsto \texttt{case x1 of \{...\}}], \quad 6\rangle \\ \Longrightarrow_{\mathsf{varcons}} & \langle [\texttt{x1} \mapsto \texttt{Z}, \ldots], & \texttt{Z}, & [(\{\ldots\}, 5)], \texttt{G}_6[\texttt{x1} \rightsquigarrow 6], \quad 6\rangle \\ \Longrightarrow_{\mathsf{select}} & \langle [\ldots], & \texttt{True}, & [\,], \texttt{G}_7[6 \mapsto \texttt{Z}, 5 \underset{7}{\mapsto}], \quad 7\rangle \\ \Longrightarrow_{\mathsf{success}} & \langle [\ldots], & \Diamond, & [\,], \texttt{G}_8[7 \mapsto \texttt{True}], \quad \Box\rangle \end{array}$$

Figure 11. Derivation of Example 3

graph *a posteriori*, it does not suffice to reach a "final" configuration (i.e., a configuration with a value in the control and an empty stack). An additional rule success (shown in Figure 10) is needed to store the result of the computation. In the derived configuration, we put the special symbol "$\Diamond$" to denote that no further steps are possible.

In the instrumented semantics, a computation starts with a configuration of the form $\langle \texttt{[]}, \texttt{main}, \texttt{[]}, \texttt{G}_\emptyset, r\rangle$, where $\texttt{G}_\emptyset$ denotes an empty graph and r an initial reference.

Example 3. Consider again the program of Example 2. By using the instrumented semantics of Figures 9 and 10, the derivation shown in Figure 11 is computed (here, natural numbers are used as references, where 0 is the initial reference). For clarity, we denote the graph of the i-th configuration of the computation by $\texttt{G}_\texttt{i}$ (thus we update $\texttt{G}_\texttt{i}$ in the $i+1$-th configuration).

4.3. Including Program Positions

In this section we extend the previous semantics so that the computed redex trail also includes *program positions*, which are used to uniquely determine the location—in the source program—of each stored expression.

Definition 1. (position, program position) A *position* is denoted by a sequence of natural numbers, where Λ is the empty sequence (i.e., the root position). They are used to address subexpressions of an expression viewed as a tree, as follows:[6]

$$
\begin{array}{ll}
e|_\Lambda = e & \text{for all expression } e\\
c(\overline{e_n})|_{i.w} = e_i|_w & \text{if } i \in \{1,\ldots,n\}\\
f(\overline{e_n})|_{i.w} = e_i|_w & \text{if } i \in \{1,\ldots,n\}\\
\mathit{let}\ x = e\ \mathit{in}\ e'|_{1.w} = e|_w & \\
\mathit{let}\ x = e\ \mathit{in}\ e'|_{2.w} = e'|_w & \\
\mathit{case}\ e\ \mathit{of}\ \{\overline{p_n \to e_n}\}|_{1.w} = e|_w & \\
\mathit{case}\ e\ \mathit{of}\ \{\overline{p_n \to e_n}\}|_{2.i.w} = e_i|_w & \text{if } i \in \{1,\ldots,n\}
\end{array}
$$

A *program position* is a pair (g, w) that addresses the expression $e|_w$ in the right-hand side of the definition, $g(\overline{x_n}) = e$, of function g. Given a program P, $\mathcal{P}os(P)$ denotes the set of all program positions in P.

Note that variables in a let construct or patterns in a case expression are not addressed by program positions because they will not be considered in the slicing process. While other approaches to dynamic slicing, e.g., (Biswas, 1997b), consider some form of *explicit* labeling for each program expression, our program positions can be seen as a form of *implicit* labeling.

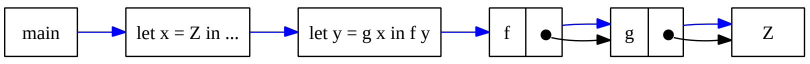
In order to add this additional information to the computed graph, two extensions are necessary. The first extension is straightforward: we label each mapping $x \mapsto e$ in the heap with the program position of expression e. Therefore, mappings in the heap have now the form $x \mapsto_{(g,w)} e$.

The second extension regards the addition of program positions to the nodes of the graph. In principle, we could extend the redex trail model so that $r \mapsto_{(g,w)}$ denotes that the expression which labels node r is located at program position (g, w). Unfortunately, this simple solution is not appropriate because *variables* are not stored in redex trails, but we still need to collect their program positions.

Example 4. Consider the following simple program:

```
main  = let x = Z in
        let y = g x in f y
f x = x
g x = x
```

By using the instrumented semantics of Figures 9 and 10, the derivation shown in Figure 12 is computed. The redex trail can be graphically depicted as follows:



[6] We consider arbitrary (not necessarily normalized) expressions for generality.

$$
\begin{array}{llll}
\multicolumn{4}{l}{\langle [\,], \mathtt{main}, [\,], \mathtt{G}_\emptyset, \mathtt{0}\rangle} \\
\Longrightarrow_{\mathsf{fun}} & \langle [\,], & \mathtt{let\ x\ =\ Z\ in\ ...}, & [\,], \mathtt{G_0[0} \underset{1}{\mapsto} \mathtt{main]}, \quad 1\rangle \\
\Longrightarrow_{\mathsf{let}} & \langle [\mathtt{x} \mapsto \mathtt{Z}], & \mathtt{let\ y\ =\ g\ x\ in\ f\ y}, & [\,], \mathtt{G_1[1} \underset{2}{\mapsto} \mathtt{let\ x\ =\ Z\ in\ ...]}, \quad 2\rangle \\
\Longrightarrow_{\mathsf{let}} & \langle [\mathtt{y} \mapsto \mathtt{g\ x},\ \mathtt{x} \mapsto \mathtt{Z}], & \mathtt{f\ y}, & [\,], \mathtt{G_2[2} \underset{3}{\mapsto} \mathtt{let\ y\ =\ g\ x\ in\ f\ y]}, \quad 3\rangle \\
\Longrightarrow_{\mathsf{fun}} & \langle [\mathtt{y} \mapsto \mathtt{g\ x},\ \mathtt{x} \mapsto \mathtt{Z}], & \mathtt{y}, & [\,], \mathtt{G_3[3} \underset{4}{\mapsto} \mathtt{f\ y]}, \quad 4\rangle \\
\Longrightarrow_{\mathsf{varexp}} & \langle [\mathtt{y} \mapsto \mathtt{g\ x},\ \mathtt{x} \mapsto \mathtt{Z}], & \mathtt{g\ x}, & [\mathtt{y}], \mathtt{G_4[y} \rightsquigarrow \mathtt{4]}, \quad 4\rangle \\
\Longrightarrow_{\mathsf{fun}} & \langle [\mathtt{y} \mapsto \mathtt{g\ x},\ \mathtt{x} \mapsto \mathtt{Z}], & \mathtt{x}, & [\mathtt{y}], \mathtt{G_5[4} \underset{5}{\mapsto} \mathtt{g\ x]}, \quad 5\rangle \\
\Longrightarrow_{\mathsf{varcons}} & \langle [\mathtt{y} \mapsto \mathtt{g\ x},\ \mathtt{x} \mapsto \mathtt{Z}], & \mathtt{Z}, & [\mathtt{y}], \mathtt{G_6[x} \rightsquigarrow \mathtt{5]}, \quad 5\rangle \\
\Longrightarrow_{\mathsf{val}} & \langle [\mathtt{y} \mapsto \mathtt{Z},\ \mathtt{x} \mapsto \mathtt{Z}], & \mathtt{Z}, & [\,], \mathtt{G_7(\equiv G_8)}, \quad 5\rangle \\
\Longrightarrow_{\mathsf{success}} & \langle [\mathtt{y} \mapsto \mathtt{Z},\ \mathtt{x} \mapsto \mathtt{Z}], & \Diamond, & [\,], \mathtt{G_8[5} \mapsto \mathtt{Z]}, \quad \Box\rangle
\end{array}
$$

Figure 12. Derivation of Example 4

Observe that the values of variables have not been stored in the graph. Therefore, the associated program slice containing the expressions in the redex trail would have the following form:

```
main  = let x = Z in
        let y = g x in f y
f x = x
g x = x
```

where functions `f` and `g` appear in gray because no expression in their right-hand sides belong to the slice. This is clearly wrong because the right-hand sides of both functions, `f` and `g`, have been evaluated.

In order to overcome this problem, we denote the mapping from nodes to program positions by $r \mapsto_{\mathcal{P}}$, where $\mathcal{P}$ is a *list* of program positions. Now, the meaning of $r \mapsto_{\mathcal{P}}$ is the following:

- the program position of the expression labeling node r is the first element of list $\mathcal{P}$ and
- if the tail of $\mathcal{P}$ is not empty, then it contains the program positions of the (chain of) variables whose evaluation was required in order to reach the expression labeling r.

Therefore, in the following, a configuration of the instrumented semantics is formally defined as follows:

Definition 2. (configuration) A configuration of the semantics is a tuple $\langle \Gamma, e, S, G, r, \mathcal{P}\rangle$ where $\Gamma \in$ *Heap* is the current heap, $e \in$ *Exp* is the expression to be evaluated (the control), S is the stack, G is a directed graph (the trail built so far), r is the current reference, and $\mathcal{P}$ is the list of program positions associated with e.

The new instrumented semantics, including the rules for failure and success, is shown in Figure 13.[7]

[7] As in Figure 9, we assume that q is always a fresh reference.

varcons

$\langle \Gamma[x \mapsto_{(g,w)} t], x, S, G, r, \mathcal{P}\rangle$
$\Longrightarrow \langle \Gamma[x \mapsto_{(g,w)} t], t, S, G \bowtie (x \rightsquigarrow r), r, (g,w) : \mathcal{P}\rangle$ where t is constructor-rooted

varexp

$\langle \Gamma[x \mapsto_{(g,w)} e], x, S, G, r, \mathcal{P}\rangle$
$\Longrightarrow \langle \Gamma[x \mapsto_{(g,w)} e], e, x : S, G \bowtie (x \rightsquigarrow r), r, (g,w) : \mathcal{P}\rangle$ where e is not constructor-rooted and $e \neq x$

val

$\langle \Gamma[x \mapsto_{(g',w')} e], v, x : S, G, r, (g,w) : \mathcal{P}\rangle$
$\Longrightarrow \langle \Gamma[x \mapsto_{(g,w)} v], v, S, G, r, (g,w) : \mathcal{P}\rangle$ where v is a value

fun

$\langle \Gamma, f(\overline{x_n}), S, G, r, \mathcal{P}\rangle$
$\Longrightarrow \langle \Gamma, \rho(e), S, G[r \underset{q}{\mapsto}_{\mathcal{P}} f(\overline{x_n})], q, [(f,\Lambda)]\rangle$ where $f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$

let

$\langle \Gamma, let\ x = e_1\ in\ e_2, S, G, r, \mathcal{P}\rangle$
$\Longrightarrow \langle \Gamma[y \mapsto_{(g,w.1)} \rho(e_1)], \rho(e_2), S, G[r \underset{q}{\mapsto}_{\mathcal{P}} \rho(let\ x = e_1\ in\ e_2)], q, [(g, w.2)]\rangle$
where $\rho = \{x \mapsto y\}$, y is a fresh variable, and $\mathcal{P} = (g,w) : \mathcal{P}'$ for some $\mathcal{P}'$

case

$\langle \Gamma, case\ x\ of\ \{\overline{p_k \rightarrow e_k}\}, S, G, r, \mathcal{P}\rangle$
$\Longrightarrow \langle \Gamma, x, (\{\overline{p_k \rightarrow e_k}\}, r) : S, G[r \mapsto_{\mathcal{P}} case\ x\ of\ \{\overline{p_k \rightarrow e_k}\}], q, [(g, w.1)]\rangle$
where $\mathcal{P} = (g,w) : \mathcal{P}'$ for some $\mathcal{P}'$

select

$\langle \Gamma, c(\overline{y_n}), (\{\overline{p_k \rightarrow e_k}\}, r') : S, G, r, \mathcal{P}\rangle$
$\Longrightarrow \langle \Gamma, \rho(e_i), S, G[r \mapsto_{\mathcal{P}} c(\overline{y_n}),$
$r' \underset{q}{\mapsto}_{(g,w):\mathcal{P}'}], q, [(g, w.2.i)]\rangle$ where $i \in \{1, \ldots k\}$, $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, and $(r' \mapsto_{(g,w):\mathcal{P}'}) \in G$ for some $\mathcal{P}'$

select-f

$\langle \Gamma, c(\overline{y_n}), (\{\overline{p_k \rightarrow e_k}\}, r') : S, G, r, \mathcal{P}\rangle$
$\Longrightarrow \langle \Gamma, Fail, S, G[r \mapsto_{\mathcal{P}} c(\overline{y_n}), r' \underset{q}{\mapsto}_{\mathcal{P}'}], q, [\,]\rangle$ where $\nexists i \in \{1, \ldots k\}$ such that $p_i = c(\overline{x_n})$, with $(r' \mapsto_{\mathcal{P}'}) \in G$

success

$\langle \Gamma, c(\overline{x_n}), [\,], G, r, \mathcal{P}\rangle \Longrightarrow \langle \Gamma, \Diamond, [\,], G[r \mapsto_{\mathcal{P}} c(\overline{x_n})], \Box, [\,]\rangle$

Figure 13. Instrumented semantics for tracing and slicing

In the first two rules, varcons and varexp, the evaluation of a variable x is demanded. Thus, the program position (g, w) of the expression to which x is bound in the heap is added to the current list of program positions $\mathcal{P}$. Here, $\mathcal{P}$ denotes a (possibly empty) list of program positions for the *chain* of variables that finishes in the current variable x (note that, in rule varexp, e may be a variable).

In rule val, the first element in the current list of program positions (i.e., the program position of the expression in the control) is used to label the binding that updates the current heap.

In rules fun, let and case, the current list of program positions is used to label the expression in the control which is stored in the graph. Note that, in rule fun, the list of program positions in the derived configuration is reset to $[(f, \Lambda)]$ because the expression in the control is the complete right-hand side of function f. Note also that, in rule let, the new binding $y \mapsto_{(g,w.1)} \rho(e_1)$, which is added to the heap, is labeled with the program position of expression e_1. This information may be necessary in rules varcons and varexp if the value of y is demanded.

In rule select, observe that the reference r'—which is stored in the stack by rule case—is used to determine the initial list of program positions in the derived configuration. Finally, in rules select-f and success an empty list of program positions is set because the control of the derived configuration contains either the special (constructor) symbol *Fail* or $\diamondsuit$, none of which occur in the program.

In order to perform computations with the instrumented semantics, we construct an *initial configuration* and apply the rules of Figure 13 until a *final configuration* is reached:

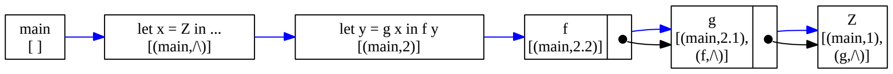
Definition 3. (initial configuration) An initial configuration has the form $\langle \texttt{[]}, \texttt{main}, \texttt{[]}, \texttt{G}_\emptyset, r, [\,]\rangle$, where $\texttt{G}_\emptyset$ denotes an empty graph, r a reference, and $[\,]$ an empty list of program positions (since there is no call to `main` from the right-hand side of any function definition).

Definition 4. (final configuration) A final configuration has the form $\langle \Gamma, \diamondsuit, \texttt{[]}, G, \Box, []\rangle$, where Γ is a heap containing the computed bindings and G is the extended redex trail of the computation.

We denote by $\Longrightarrow^*$ the reflexive and transitive closure of $\Longrightarrow$. A derivation $C \Longrightarrow^* C'$ is *complete* if C is an initial configuration and C' is a final configuration.

Clearly, the instrumented semantics is a conservative extension of the original small-step semantics of Figure 7, since the extensions impose no restriction on the application of the standard part of the semantics (the first three components of a configuration).

Example 5. Consider again the program of Example 4. By using the instrumented semantics of Figure 13, the derivation shown in Figure 14 is now computed. The redex trail can be graphically depicted as follows:



Therefore, in contrast to the situation in Example 4, the associated slice is now the complete program.

The correctness of the redex trail has been proved by Braßel et al. (2004). Now, we state the correctness of the computed program positions (the proof can be found in the appendix). Intuitively speaking, we

$\langle [\,], \mathtt{main}, [\,], \mathtt{G}_\emptyset, \mathtt{0}, [\,] \rangle$						
$\Longrightarrow_{\mathsf{fun}}$	$\langle [\,],$	$\mathtt{let\ x = Z\ in\ \ldots}$	$[\,],$	$\mathtt{G}_0[0 \underset{1}{\mapsto}_{[\,]} \mathtt{main}],$	$1,$	$[(\mathtt{main}, \Lambda)]\rangle$
$\Longrightarrow_{\mathsf{let}}$	$\langle [\mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}],$	$\mathtt{let\ y = g\ x\ in\ f\ y}$	$[\,],$	$\mathtt{G}_1[1 \underset{2}{\mapsto}_{[(\mathtt{main},\Lambda)]} \mathtt{let\ x = Z\ in} \ldots],$	$2,$	$[(\mathtt{main}, 2)]\rangle$
$\Longrightarrow_{\mathsf{let}}$	$\langle [\mathtt{y} \mapsto_{(\mathtt{main},2.1)} \mathtt{g\ x}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}],$	$\mathtt{f\ y},$	$[\,],$	$\mathtt{G}_2[2 \underset{3}{\mapsto}_{[(\mathtt{main},2)]} \mathtt{let\ y = g\ x\ in} \ldots],$	$3,$	$[(\mathtt{main}, 2.2)]\rangle$
$\Longrightarrow_{\mathsf{fun}}$	$\langle [\mathtt{y} \mapsto_{(\mathtt{main},2.1)} \mathtt{g\ x}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}],$	$\mathtt{y},$	$[\,],$	$\mathtt{G}_3[3 \underset{4}{\mapsto}_{[(\mathtt{main},2.2)]} \mathtt{f\ y}],$	$4,$	$[(\mathtt{f}, \Lambda)]\rangle$
$\Longrightarrow_{\mathsf{varexp}}$	$\langle [\mathtt{y} \mapsto_{(\mathtt{main},2.1)} \mathtt{g\ x}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}],$	$\mathtt{g\ x},$	$[\mathtt{y}],$	$\mathtt{G}_4[\mathtt{y} \rightsquigarrow 4],$	$4,$	$[(\mathtt{main}, 2.1), (\mathtt{f}, \Lambda)]\rangle$
$\Longrightarrow_{\mathsf{fun}}$	$\langle [\mathtt{y} \mapsto_{(\mathtt{main},2.1)} \mathtt{g\ x}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}],$	$\mathtt{x},$	$[\mathtt{y}],$	$\mathtt{G}_5[4 \underset{5}{\mapsto}_{[(\mathtt{main},2.1),(\mathtt{f},\Lambda)]} \mathtt{g\ x}],$	$5,$	$[(\mathtt{g}, \Lambda)]\rangle$
$\Longrightarrow_{\mathsf{varcons}}$	$\langle [\mathtt{y} \mapsto_{(\mathtt{main},2.1)} \mathtt{g\ x}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}],$	$\mathtt{Z},$	$[\mathtt{y}],$	$\mathtt{G}_6[\mathtt{x} \rightsquigarrow 5],$	$5,$	$[(\mathtt{main}, 1), (\mathtt{g}, \Lambda)]\rangle$
$\Longrightarrow_{\mathsf{val}}$	$\langle [\mathtt{y} \mapsto_{(\mathtt{main},1)} \mathtt{Z}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}],$	$\mathtt{Z},$	$[\,],$	$\mathtt{G}_7(\equiv \mathtt{G}_8),$	$5,$	$[(\mathtt{main}, 1), (\mathtt{g}, \Lambda)]\rangle$
$\Longrightarrow_{\mathsf{success}}$	$\langle [\mathtt{y} \mapsto_{(\mathtt{main},1)} \mathtt{Z}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}],$	$\Diamond,$	$[\,],$	$\mathtt{G}_8[5 \mapsto_{[(\mathtt{main},1),(\mathtt{g},\Lambda)]} \mathtt{Z}],$	$\Box,$	$[\,]\rangle$

Figure 14. Derivation of Example 5

prove that the first program position in the list of program positions attached to every configuration correctly addresses the location of the expression in the control of this configuration.

In what follows, given a configuration of the form $C = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle$, we let $ctrl(C) = e$ and $pos(C) = \mathcal{P}$.

Theorem 1. Let P be a program and $(C \Longrightarrow^* C')$ be a complete derivation in P, where G' is the graph in the final configuration C'. If $(r \mapsto_{\mathcal{P}} e) \in G'$, then either $\mathcal{P} = [\,]$ (with $e = \mathtt{main}$) or $\mathcal{P} = [(g_k, w_k), \ldots, (g_1, w_1)]$, $k \geqslant 1$, and the following conditions hold:

1. there exists a subderivation $C_1 \Longrightarrow \ldots \Longrightarrow C_j$ of $C \Longrightarrow^* C'$, $j \geqslant k$, such that $pos(C_j) = [(g_k, w_k), \ldots, (g_1, w_1)]$, and

2. for all $i = 1, \ldots, j$, there exists a rule $g'_i(\overline{x_{n_i}}) = e_i \in P$ such that $e_i|_{w'_i} = ctrl(C_i)$, and

   – $g'_i = g_i$ and $w'_i = w_i$ (for $i = 1, \ldots, k$)

   – $g'_i = g_k$ and $w'_i = w_k$ (for $i = k+1, \ldots, j$)

   with $e_j|_{w_j} = e$.

4.4. Logical Features

In this section, we sketch the extension of the instrumented semantics to deal with the logical features of the language: disjunctions and flexible case structures. The new rules are shown in Figure 15.[8]

Rule or is used to non-deterministically choose one of the disjuncts. It is self-explanatory and proceeds analogously to rules fun, let or case.

Besides the original rule for case expressions (rule case), we add a new rule fcase to deal with flexible case expressions. In this case, a symbol f is stored in the stack preceding the case alternatives to indicate that the case expression is flexible. Note also that, to be precise, this implies that rule select in Figure 13 should also be slightly changed in order to accept a pair $(f\{\overline{p_k \to e_k}\}, r')$ on top of the stack.

In contrast to the rule select, the rule guess is used when we have a *logical variable* in the control and the alternatives of a flexible case expression on the stack. Then, this rule non-deterministically chooses one alternative and continues with the evaluation of this branch. Since the names of logical variables are not relevant, the new node r is labeled with a special symbol $LogVar$. The successor of this node is then set to q which is labeled with the selected binding for the logical variable. The list of program positions for node q is empty because patterns—and their local variables—introduced by instantiation do not have an associated program position. Finally, observe that we use $(\_,\_)$ to denote a *null* program position in the heap.

Rule guess-f applies when we have a logical variable in the control and (the alternatives of) a *rigid* case expression on top of the stack, i.e., when the computation *suspends*. Analogously to rule select-f, an empty list of program positions is set in the derived configuration.

Finally, rule success-x is the counterpart of rule success when the computed value is a logical variable.

The extension of Theorem 1 to cope with the new rules does not involve any additional complexity (basically, Lemma 1 in the appendix should be extended in order to also consider these new rules).

5. Computing the Slice

In this section, we formalize the concept of dynamic slice and introduce a method to compute it from the extended redex trail.

As in the previous section, we consider a pure (first-order) functional language without disjunctions or flexible case structures; the treatment of these language features is sketched in Section 5.2.

We define a slice as a set of program positions:

[8] We assume that q and s are always fresh references.

or

$$\langle \Gamma, e_1 \; or \; e_2, S, G, r, \mathcal{P}\rangle \Longrightarrow \langle \Gamma, e_i, S, G[r \underset{q}{\mapsto}_{\mathcal{P}} e_1 \; or \; e_2], q, [(g, w.i)]\rangle$$

where $i \in \{1,2\}$ and $\mathcal{P} = (g,w) : \mathcal{P}'$ for some $\mathcal{P}'$

fcase

$$\langle \Gamma, \mathit{fcase} \; x \; \mathit{of} \; \{\overline{p_k \to e_k}\}, S, G, r, \mathcal{P}\rangle$$
$$\Longrightarrow \langle \Gamma, x, (f\{\overline{p_k \to e_k}\}, r) : S, G[r \mapsto_{\mathcal{P}} \mathit{fcase} \; x \; \mathit{of} \; \{\overline{p_k \to e_k}\}], q, [(g, w.1)]\rangle$$

where $\mathcal{P} = (g,w) : \mathcal{P}'$ for some $\mathcal{P}'$

guess

$$\langle \Gamma[y \mapsto_{(h,w')} y], y, (f\{\overline{p_k \to e_k}\}, r') : S, G, r, \mathcal{P}\rangle$$
$$\Longrightarrow \langle \Gamma[y \mapsto_{(\_,\_)} \rho(p_i), \overline{y_n \mapsto_{(\_,\_)} y_n}], \rho(e_i), S,$$
$$G[r \underset{q}{\mapsto}_{\mathcal{P}} \mathit{LogVar}, q \mapsto_{[\,]} \rho(p_i), y \rightsquigarrow r, r' \underset{s}{\mapsto}_{(g,w):\mathcal{P}'}], s, [(g, w.2.i)]\rangle$$

where $i \in \{1, \ldots k\}$, $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, $(r' \mapsto_{(g,w):\mathcal{P}'}) \in G$ for some $\mathcal{P}'$, and $\overline{y_n}$ are fresh variables

guess-f

$$\langle \Gamma[y \mapsto_{(g,w)} y], y, (\{\overline{p_k \to e_k}\}, r') : S, G, r, \mathcal{P}\rangle$$
$$\Longrightarrow \langle \Gamma[y \mapsto_{(g,w)} y], \mathit{Fail}, S, G[r \mapsto_{\mathcal{P}} \mathit{LogVar}, y \rightsquigarrow r, r' \underset{s}{\mapsto}_{[\,]}], s, [\,]\rangle$$

where $(r' \mapsto) \in G$

success-x

$$\langle \Gamma[x \mapsto_{(g,w)} x], x, \texttt{[]}, G, r, \mathcal{P}\rangle$$
$$\Longrightarrow \langle \Gamma[x \mapsto_{(g,w)} x], \Diamond, \texttt{[]}, G[r \overset{p}{\mapsto}_{\mathcal{P}} \mathit{LogVar}, x \rightsquigarrow r], \Box, [\,]\rangle$$

Figure 15. Instrumented semantics for tracing and slicing: logical features

Definition 5. (slice) Let P be a program. A slice for P is any set $\mathcal{W}$ of program positions with $\mathcal{W} \subseteq \mathcal{P}os(P)$.

Observe that we are not interested in producing *executable* slices but in computing the program positions of the expressions that influence the slicing criterion. This is enough for our purposes, i.e., for showing the original program with some distinguished expressions.

In the literature of dynamic slicing for imperative programs, the slicing criterion usually identifies a concrete point of the execution history, e.g., $\langle \mathtt{n} = 2, 8^1, \mathtt{x}\rangle$, where $\mathtt{n} = 2$ is the input for the program, 8^1 denotes the *first* occurrence of statement 8 in the execution history, and x is the variable we are interested in. In principle, it is not difficult to extend this notion of slicing criterion to a *strict* functional language. For instance, a slicing criterion could be given by a pair $\langle f(\overline{v_n}), \pi\rangle$, where $f(\overline{v_n})$ is a function call that occurs during the execution of the program—whose arguments are fully evaluated because we consider a strict language—and π is a restriction on the output of $f(\overline{v_n})$.

Unfortunately, extending this notion to a *lazy* functional language is not trivial. For instance, identifying a function call $f(\overline{e_n})$ in the actual execution trace is impractical because $\overline{e_n}$ are usually rather complex expressions. Luckily, if a tracing tool like the one presented in Section 2 is available—i.e., a tracing tool that shows function calls with arguments as much evaluated as needed in the full computation—then we can still consider a slicing criterion of the form $\langle f(\overline{v_n}), \pi\rangle$. In this case, $f(\overline{v_n})$

should be understood as "a function call $f(\overline{e_n})$ whose arguments $\overline{e_n}$ are evaluated to $\overline{v_n}$ in the full computation".

The connection with the tracing tool (as described in Section 2) should be clear: given a (sub)trace of the form

$$\begin{array}{lcl} val_1 & = & f_1(v_1^1, \ldots, v_m^1) \\ val_2 & = & f_2(v_1^2, \ldots, v_m^2) \\ \ldots & & \\ val_k & = & f_k(v_1^k, \ldots, v_m^k) \end{array}$$

a slicing criterion has the form $\langle f_i(v_1^i, \ldots, v_m^i), \pi \rangle$, where π denotes a restriction on val_i. Therefore, the user can easily produce a valid slicing criterion from the trace of a computation by only giving π (if any, since a default value $\top$ can also be used). Traces usually contain a special symbol, "_", to denote that some subexpression is missing because its evaluation was not required in the computation. Consequently, we will also accept expressions containing this special symbol (see below).

Definition 6. (slicing criterion) Let P be a program and $(C_0 \Longrightarrow^* C_m)$ a complete derivation.[9] A slicing criterion for P w.r.t. $(C_0 \Longrightarrow^* C_m)$ is a tuple $\langle f(\overline{pv_n}), \pi \rangle$ such that $f \in \mathcal{F}$ is a defined function symbol (arity $n \geqslant 0$), $pv_1, \ldots, pv_n \in PValue$ are partial values, and $\pi \in Pat$ is a slicing pattern. The domain $PValue$ obeys the following syntax:

$$pv \in PValue \ ::= \ \_ \mid x \mid c(\overline{pv_k})$$

where $c \in \mathcal{C}$ is a constructor symbol (arity $k \geqslant 0$) and "_" is a special symbol to denote any non-evaluated expression.

The domain Pat of slicing patterns is defined as follows:

$$\pi \in Pat \ ::= \ \bot \mid \top \mid hnf \mid c(\overline{\pi_k})$$

where $\bot$ denotes that the associated evaluation is not relevant, $\top$ denotes that the full evaluation (i.e., to *normal form*) is relevant, *hnf* denotes that only the evaluation to head normal form is relevant, and $c(\overline{\pi_k})$ that the evaluation to a head normal form $c(\overline{x_k})$, $c \in \mathcal{C}$, is relevant (if any) and that the evaluation of every argument x_i is relevant if it is relevant according to π_i.[10]

Slicing patterns are particularly useful when we are only interested in *part* of the result of a function call. For instance, if we know that a

[9] Here, and in the following, we use the notation $C_0 \Longrightarrow^* C_m$ as a shorthand to denote a derivation of m steps of the form $C_0 \Longrightarrow C_1 \Longrightarrow \ldots \Longrightarrow C_m$.

[10] Our patterns for slicing can be seen as an extension of the *liveness patterns* used to perform dead code elimination by Liu and Stoller (2003).

function call should return a pair, and we are only interested in the first component of this pair, we could use the slicing pattern $(\top, \bot)$; if a function returns a list, and we are only interested in the first two elements of this list, we could use the slicing pattern $\top : (\top : \bot)$. Usually, we are only interested in the computation of a value and, thus, the slicing pattern *hnf* can be used. Of course, if we are interested in the full evaluation of an expression (i.e., as much as needed in the considered computation), we can use the slicing pattern $\top$.

In order to compute a slice, we first need to identify the configuration associated with a slicing criterion (scc) because only the program positions in the next configurations are potential candidates for the slice:

$$C_1 \Longrightarrow C_2 \Longrightarrow \ldots \Longrightarrow \overbrace{\underbrace{C_i}_{\text{scc}} \Longrightarrow C_{i+1} \ldots \Longrightarrow C_n}^{\text{candidates for the slice}}$$

For this purpose, we need some auxiliary definitions. In what follows, given a configuration of the form $C = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle$, we let $\mathit{heap}(C) = \Gamma$, $\mathit{stack}(C) = S$, and $\mathit{graph}(C) = G$.

First, we introduce the notion of *lookup* configuration (Braßel et al., 2004), i.e., a configuration that demands the evaluation of a given variable for the first time in a computation.

Definition 7. (lookup configuration) Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m \geqslant 0$, be a derivation. A configuration C_i, $0 \leqslant i \leqslant m$, is a lookup configuration of $\mathcal{D}$ iff $\mathit{ctrl}(C_i) = x$ and there exists no configuration C_j, $j < i$, with $\mathit{ctrl}(C_j) = x$.

When a lookup configuration $\langle \Gamma, x, S, G, r, \mathcal{P} \rangle$ appears in a computation, a variable pointer $x \rightsquigarrow r$ is added to the graph to point out that the value of x is *demanded* in the computation:

Definition 8. (demanded variable) Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m \geqslant 0$, be a derivation. We say that a variable x is demanded in $\mathcal{D}$ iff there exists a lookup configuration C_i, $0 \leqslant i \leqslant m$, such that $\mathit{ctrl}(C_i) = x$.

The following function is useful to extract the partial value denoted by an expression. For this purpose, outer constructors are left untouched, non-demanded variables and operation-rooted terms are replaced by the special symbol "_" (i.e., "not a value"), and demanded variables are dereferenced.

Definition 9. Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a complete derivation. Given an expression e, we denote the partial value of e in $\mathcal{D}$ by

$val_{\mathcal{D}}(e)$, where function val is defined as follows:

$$val_{\mathcal{D}}(e) = \begin{cases} \mathsf{val}^{\mathsf{v}}_{\mathcal{D}}(x) & \text{if } e = x \text{ and } x \text{ is demanded in } \mathcal{D} \\ c(\overline{val_{\mathcal{D}}(x_n)}) & \text{if } e = c(\overline{x_n}) \\ \_ & \text{otherwise} \end{cases}$$

The auxiliary function $\mathsf{val}^{\mathsf{v}}_{\mathcal{D}}$ is used to dereference a variable w.r.t. (the heap computed in) a derivation:

$$\mathsf{val}^{\mathsf{v}}_{\mathcal{D}}(x) \ = \ \begin{cases} \mathsf{val}^{\mathsf{v}}_{\mathcal{D}}(y) & \text{if } \Gamma[x] = y \text{ and } x \neq y \\ c(\overline{val_{\mathcal{D}}(x_n)}) & \text{if } \Gamma[x] = c(\overline{x_n}) \\ \_ & \text{otherwise} \end{cases}$$

where $\Gamma = heap(C_m)$ is the heap in the last configuration of $\mathcal{D}$.

Let us illustrate this function with some examples. Consider three derivations, $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$, in which the computed heaps in the last configurations are as follows:[11]

$$\begin{array}{lcllll} \Gamma_1 & = & [\ \mathtt{x} \mapsto \mathtt{y}, & \mathtt{y} \mapsto \mathtt{(S\ n)}, & \mathtt{n} \mapsto \mathtt{Z} &] \\ \Gamma_2 & = & [\ \mathtt{x} \mapsto \mathtt{y}, & \mathtt{y} \mapsto \mathtt{(S\ z)}, & \mathtt{z} \mapsto \mathtt{Z}, & \mathtt{n} \mapsto \mathtt{Z}\ ] \\ \Gamma_3 & = & [\ \mathtt{m} \mapsto \mathtt{(f\ Z)}, & \mathtt{y} \mapsto \mathtt{(S\ n)}, & \mathtt{n} \mapsto \mathtt{Z} &] \end{array}$$

Here $\mathtt{Z}, \mathtt{S} \in \mathcal{C}$ are constructors, $\mathtt{f} \in \mathcal{F}$ is a defined function symbol, and $\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{n}, \mathtt{m} \in \mathcal{X}$ are variables. Assuming that variables $\mathtt{x}, \mathtt{y}, \mathtt{z}$ are demanded in the considered computations and $\mathtt{m}, \mathtt{n}$ are not, we have:

$$\begin{array}{lcl} val_{\mathcal{D}_1}(\mathtt{C\ x\ (f\ y)}) & = & \mathtt{C\ (S\ \_)\ \_} \\ val_{\mathcal{D}_2}(\mathtt{C\ x\ n}) & = & \mathtt{C\ (S\ Z)\ \_} \\ val_{\mathcal{D}_3}(\mathtt{C\ m\ y}) & = & \mathtt{C\ \_\ \ \ \ (S\ \_)} \end{array}$$

We can now formally identify the configuration associated with a slicing criterion.

Definition 10. (SC-configuration) Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a complete derivation. Given a slicing criterion $\langle f(\overline{pv_n}), \pi\rangle$, the associated SC-configuration C_i, $0 \leqslant i \leqslant m$, must fulfill the following conditions:[12]

– $ctrl(C_i) = f(\overline{x_n})$ and

– $pv_i = val_{\mathcal{D}}(x_i)$ for all $i = 1, \ldots, n$.

[11] Here we ignore the program positions that label the bindings in the heap since they are not relevant for computing the partial value.

[12] If there are several configurations that fulfill these conditions (e.g., if the same function is called several times in the computation with the same arguments), we choose the first of such configurations.

Intuitively speaking, the SC-configuration associated with a slicing criterion $\langle f(\overline{pv_n}), \pi \rangle$ is a configuration, C_i, whose control has the form $f(\overline{x_n})$ and the variables $\overline{x_n}$ are eventually bound to the partial values $\overline{pv_n}$ in the considered derivation.

Clearly, the notion of SC-configuration is only partially defined since it may happen that the conditions of the above definition do not hold (e.g., if the slicing criterion refers to a function that has not been evaluated in the considered computation). For simplicity, in the following we assume that there is always an SC-configuration associated to any slicing criterion.

For instance, given the program and derivation of Example 5, the SC-configuration associated with the slicing criterion $\langle \mathtt{f\ Z}, \top \rangle$ is the configuration

$$\begin{array}{l} \langle \ [\mathtt{y} \mapsto_{(\mathtt{main},2.1)} \mathtt{g\ x}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}], \quad \mathtt{f\ y}, \ [\,], \\ \quad \mathtt{G}[2 \underset{3}{\mapsto}_{[(\mathtt{main},2)]} \mathtt{let\ y = g\ x\ in} \ldots], \ 3, \quad [(\mathtt{main}, 2.2)] \ \rangle \end{array}$$

since its control is $(\mathtt{f\ y})$ and the variable $\mathtt{y}$ is bound to $\mathtt{Z}$ in the heap $[\mathtt{y} \mapsto_{(\mathtt{main},1)} \mathtt{Z}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}]$ of the final configuration of the derivation.

We also need the notion of *complete subderivation*:

Definition 11. (complete subderivation) Let P be a program and $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$ a complete derivation in P. Let C_i, $0 \geqslant i \geqslant m$, be a configuration. The subderivation $C_i \Longrightarrow^* C_j$, $i \geqslant j \geqslant m$, is the complete subderivation for C_i in $\mathcal{D}$ iff the following conditions hold:

1. $\mathit{ctrl}(C_j) \in \mathit{Value}$,
2. $\mathit{stack}(C_i) = \mathit{stack}(C_j)$, and
3. $\nexists k.\ i < k < j$, $\mathit{stack}(C_i) = \mathit{stack}(C_k)$ and $\mathit{ctrl}(C_k) \in \mathit{Value}$.

Consider, for instance, the derivation of Example 5 which is shown in Figure 16. Here, the complete subderivation for the configuration

$$\langle [\mathtt{y} \mapsto_{(\mathtt{main},2.1)} \mathtt{g\ x}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}], \mathtt{g\ x}, [\mathtt{y}], \mathtt{G}[\mathtt{y} \rightsquigarrow 4], 4, [(\mathtt{main}, 2.1), (\mathtt{f}, \Lambda)] \rangle$$

is shown in a box (where graphs and program positions are ignored for conciseness).

Now, we can already introduce our notion of dynamic slice (in the following, the auxiliary function *hd* returns the first element of a list):

Definition 12. (dynamic slice, ds) Let P be a program and let $\mathcal{D} = (C_0 \Longrightarrow^* C_n)$ be a complete derivation in P. Let $\langle f(\overline{pv_l}), \pi \rangle$ be a slicing criterion and C_i its associated SC-configuration. The dynamic slice for P w.r.t. the slicing criterion $\langle f(\overline{pv_l}), \pi \rangle$ is given by

$$\{\mathit{hd}(\mathit{pos}(C_i))\} \cup \mathit{ds}([C_{i+1}, \ldots, C_j], \pi, \{\,\}, \mathcal{D})$$

$$
\begin{array}{llllll}
\multicolumn{6}{l}{\langle [\,], \mathtt{main}, [\,], \mathtt{G}_\emptyset, 0, [\,] \rangle} \\
\Longrightarrow_{\mathsf{fun}} & \langle [\,], & \mathtt{let}\ \mathtt{x} = \mathtt{Z}\ \mathtt{in}\ \ldots & [\,], & \mathtt{G}_1, & 1,\ \mathcal{P}_1 \rangle \\
\Longrightarrow_{\mathsf{let}} & \langle [\mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}], & \mathtt{let}\ \mathtt{y} = \mathtt{g}\ \mathtt{x}\ \mathtt{in}\ \mathtt{f}\ \mathtt{y} & [\,], & \mathtt{G}_2, & 2,\ \mathcal{P}_2 \rangle \\
\Longrightarrow_{\mathsf{let}} & \langle [\mathtt{y} \mapsto_{(\mathtt{main},2.1)} \mathtt{g}\ \mathtt{x}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}] & \mathtt{f}\ \mathtt{y}, & [\,], & \mathtt{G}_3, & 3,\ \mathcal{P}_3 \rangle \\
\Longrightarrow_{\mathsf{fun}} & \langle [\mathtt{y} \mapsto_{(\mathtt{main},2.1)} \mathtt{g}\ \mathtt{x}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}] & \mathtt{y}, & [\,], & \mathtt{G}_4, & 4,\ \mathcal{P}_4 \rangle \\
\hline
\Longrightarrow_{\mathsf{varexp}} & \langle [\mathtt{y} \mapsto_{(\mathtt{main},2.1)} \mathtt{g}\ \mathtt{x}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}] & \mathtt{g}\ \mathtt{x}, & [\mathtt{y}], & \mathtt{G}_5, & 4,\ \mathcal{P}_5 \rangle \\
\Longrightarrow_{\mathsf{fun}} & \langle [\mathtt{y} \mapsto_{(\mathtt{main},2.1)} \mathtt{g}\ \mathtt{x}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}] & \mathtt{x}, & [\mathtt{y}], & \mathtt{G}_6, & 5,\ \mathcal{P}_6 \rangle \\
\Longrightarrow_{\mathsf{varcons}} & \langle [\mathtt{y} \mapsto_{(\mathtt{main},2.1)} \mathtt{g}\ \mathtt{x}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}] & \mathtt{Z}, & [\mathtt{y}], & \mathtt{G}_7, & 5,\ \mathcal{P}_7 \rangle \\
\hline
\Longrightarrow_{\mathsf{val}} & \langle [\mathtt{y} \mapsto_{(\mathtt{main},1)} \mathtt{Z}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}] & \mathtt{Z}, & [\,], & \mathtt{G}_8, & 5,\ \mathcal{P}_8 \rangle \\
\Longrightarrow_{\mathsf{success}} & \langle [\mathtt{y} \mapsto_{(\mathtt{main},1)} \mathtt{Z}, \mathtt{x} \mapsto_{(\mathtt{main},1)} \mathtt{Z}] & \Diamond, & [\,], & \mathtt{G}_9, & \Box,\ \mathcal{P}_9 \rangle
\end{array}
$$

Figure 16. A complete subderivation

$$
\mathit{ds}([C_j], \pi, V, \mathcal{D}) \;=\; \mathit{pos}(C_j) \cup \mathit{ds\_aux}(C_j, \pi, V, \mathcal{D})
$$

$$
\mathit{ds}([C_i, \ldots, C_j], \pi, V, \mathcal{D}) \;=\; \qquad [i < j]
$$

$$
\begin{cases}
\mathit{pos}(C_i) \;\cup\; \mathit{ds}([C_{i+1}, \ldots, C_j], \pi, V', \mathcal{D}) & \text{if } \mathit{ctrl}(C_i) = (\mathit{let}\ x = e\ \mathit{in}\ e') \\
 & \text{where } V' = V \cup \{x\} \\
\mathit{pos}(C_i) \;\cup\; \mathit{ds}([C_{i+1}, \ldots, C_j], \pi, V, \mathcal{D}) & \text{if } \mathit{ctrl}(C_i) \text{ is a function call} \\
\mathit{pos}(C_i) \;\cup\; \mathit{ds}([C_{k+1}, \ldots, C_j], \pi, V, \mathcal{D}) & \text{if } \mathit{ctrl}(C_i) = (\mathit{case}\ x\ \mathit{of}\ \ldots) \text{ and } x \notin V \\
 & \text{where } C_{i+1} \Longrightarrow^* C_k \text{ is the complete} \\
 & \text{subderivation for } C_{i+1} \text{ in } \mathcal{D} \\
\mathit{pos}(C_i) \;\cup\; \mathit{ds}([C_{i+1}, \ldots, C_k], \mathit{hnf}, V, \mathcal{D}) & \text{if } \mathit{ctrl}(C_i) = \mathit{case}\ x\ \mathit{of}\ \ldots \text{ and } x \in V \\
\qquad\quad \cup\; \mathit{ds}([C_{k+1}, \ldots, C_j], \pi, V, \mathcal{D}) & \text{where } C_{i+1} \Longrightarrow^* C_k \text{ is the complete} \\
 & \text{subderivation for } C_{i+1} \text{ in } \mathcal{D} \\
\mathit{ds}([C_{i+1}, \ldots, C_j], \pi, V, \mathcal{D}) & \text{otherwise}
\end{cases}
$$

$$
\mathit{ds\_aux}(C_j, \pi, V, \mathcal{D}) = \bigcup_{(x,\pi') \in \mathit{ctrl}(C_j) \sqcap_V \pi} \mathit{ds}([C_x, \ldots, C_{x'}], \pi', V, \mathcal{D})
$$

where $C_j \Longrightarrow^* C_x$ in $\mathcal{D}$, $\mathit{ctrl}(C_x) = x$, C_x is a lookup configuration, and $C_x \Longrightarrow^* C_{x'}$ is the complete subderivation for C_x in $\mathcal{D}$

$$
v \sqcap_V \pi \;=\; \begin{cases}
\{(x_i, \pi_i) \mid x_i \in V,\ \pi_i \neq \bot, \text{ and } i \in \{1, \ldots, m\}\} & \text{if } v = c(\overline{x_m}) \text{ and } \pi = c(\overline{\pi_m}) \\
\{(x_i, \top) \mid x_i \in V \text{ and } i \in \{1, \ldots, m\}\} & \text{if } e = c(\overline{x_m}) \text{ and } \pi = \top \\
\{\,\} & \text{otherwise}
\end{cases}
$$

Figure 17. Extracting a dynamic slice from a derivation (function *ds*)

where $C_i \Longrightarrow^* C_j$ is the complete subderivation for C_i in $\mathcal{D}$ and the definition of the auxiliary function *ds* is shown in Figure 17.

We note that function *ds* is not applied to the SC-configuration C_i since it may store not only the program position of $f(\overline{x_n})$ but also the positions of some variables whose evaluation introduced this function call (see the discussion in Section 4.3). Therefore, in order to avoid gathering these undesired positions, we only collect the position in the head of the list of positions in C_i and then apply function *ds* to the subderivation starting in the next configuration.

```
main = let x = pair in f x

f x = case x of
      { C w1 w2 → case w1 of { Z → case w2 of { Z → Z } } }

pair = let z = zero in g z

g z = let y = one in case z of
                     { Z → case y of
                            { S w → case w of
                                     { Z → let v1 = Z in
                                            let v2 = Z in C v1 v2 } } }
zero = Z
one  = let v = Z in S v
```

Figure 18. Program of Example 6

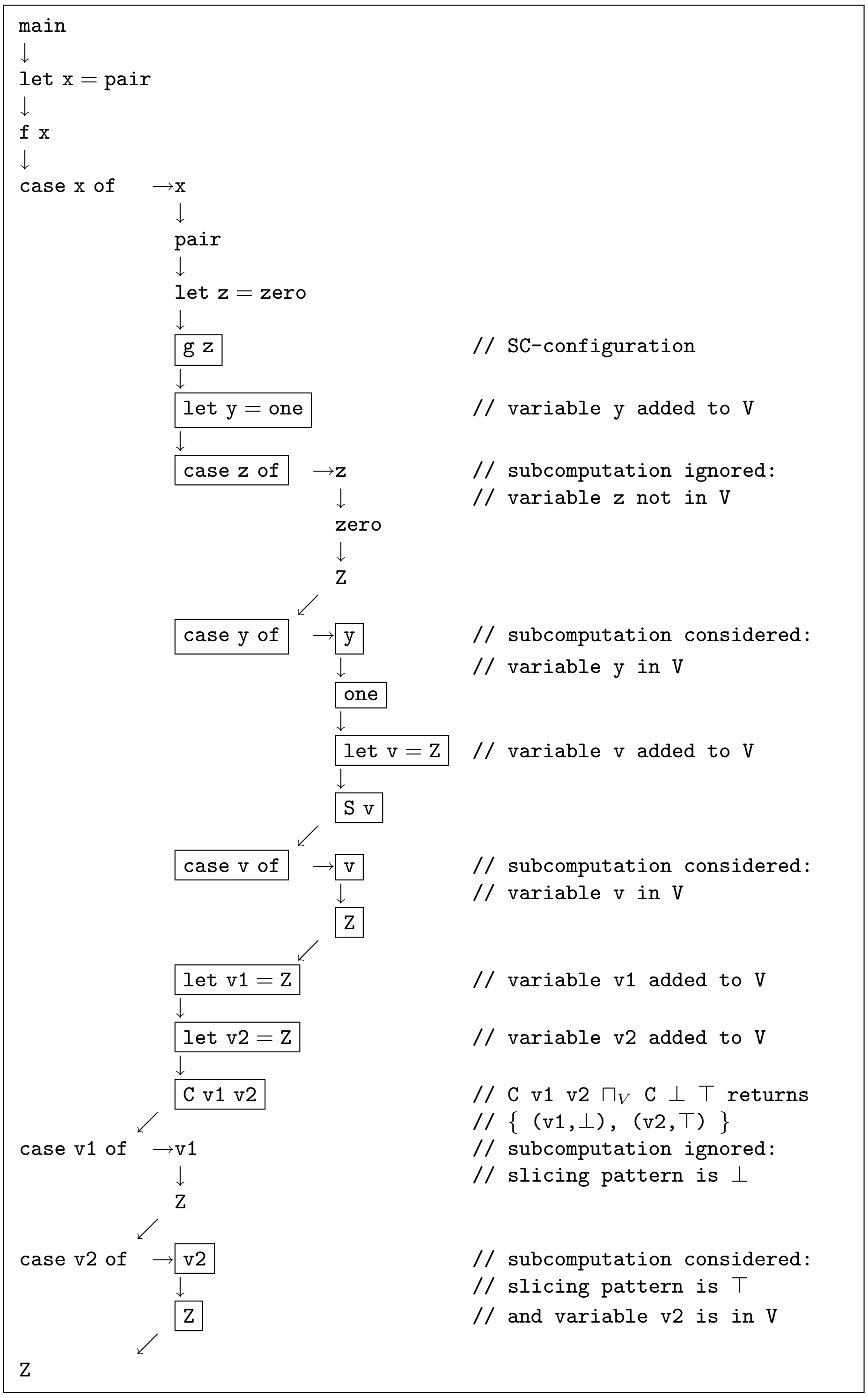
Note also that the third parameter of function *ds* is used to store the set of variables whose evaluation is relevant for the slice. Essentially, we consider that every variable introduced in a let expression *before* the SC-configuration is not relevant, according to the assumption that the evaluation of the arguments in the call of the slicing criterion should not be taken into account. This set, initially empty, is then updated whenever a let expression is found.

Let us now informally explain how a dynamic slice is computed through a simple example.

Example 6. Consider the program shown in Figure 18. Given the slicing criterion $\langle$`g Z`, `C` $\perp \top\rangle$, the computation described in Figure 19 illustrates the way in which function *ds* proceeds. For simplicity, we only show the control of each configuration. Here, the computation for function `main` is shown in the leftmost column; then, each time a case expression demands the evaluation of its argument, the computation moves one level to the right (and, analogously, when the demanded value is computed, it moves back one level to the left). In this computation, the expressions of those configurations whose program positions are collected by *ds* are shown in a box.

5.1. Using the Redex Trail

According to Definition 12, one could develop a slicing tool by implementing a meta-interpreter for the instrumented operational semantics (Figure 13) and, then, extracting the program positions from these configurations using function *ds* (Figure 17). Our aim in this work, however, is to compute the slice from the extended redex trail, *since it is already available in existing tracing tools.*



```
main
↓
let x = pair
↓
f x
↓
case x of   →x
             ↓
             pair
             ↓
             let z = zero
             ↓
             g z                          // SC-configuration
             ↓
             let y = one                  // variable y added to V
             ↓
             case z of   →z               // subcomputation ignored:
                          ↓               // variable z not in V
                          zero
                          ↓
                          Z
                        ↙
             case y of   →y               // subcomputation considered:
                          ↓               // variable y in V
                          one
                          ↓
                          let v = Z       // variable v added to V
                          ↓
                          S v
                        ↙
             case v of   →v               // subcomputation considered:
                          ↓               // variable v in V
                          Z
                        ↙
             let v1 = Z                   // variable v1 added to V
             ↓
             let v2 = Z                   // variable v2 added to V
             ↓
             C v1 v2                      // C v1 v2 ⊓V C ⊥ ⊤ returns
           ↙                              // { (v1,⊥), (v2,⊤) }
case v1 of   →v1                          // subcomputation ignored:
              ↓                           // slicing pattern is ⊥
              Z
           ↙
case v2 of   →v2                          // subcomputation considered:
              ↓                           // slicing pattern is ⊤
              Z                           // and variable v2 is in V
           ↙
Z
```

Figure 19. Computation of a dynamic slice (Example 6)

From the extended redex trail, the computation of a dynamic slice proceeds basically as follows: first, the node associated with the slicing criterion is identified (the *SC-node*, the counterpart of the SC-configuration); then, the set of program positions of the nodes which are *reachable* from the SC-node (taking into account the slicing pattern) are collected.

In order to identify the SC-node, we first need to define the counterpart of function $val_{\mathcal{D}}$:

Definition 13. Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a complete derivation and $G = graph(C_m)$. The partial value associated with variable x in G is given by $val_G(x)$, where function val_G is defined as follows:

$$val_G(x) = \begin{cases} \mathsf{val}^{\mathsf{v}}_{\mathsf{G}}(q) & \text{if } (x \rightsquigarrow r) \in G \text{ and } (r \underset{q}{\mapsto}) \in G \\ c(\overline{val_G(x_n)}) & \text{if } (x \rightsquigarrow r) \in G \text{ and } (r \mapsto c(\overline{x_n})) \in G \\ \_ & \text{otherwise} \end{cases}$$

The auxiliary function $\mathsf{val}^{\mathsf{v}}_{\mathsf{G}}$ is used to follow the successor chain in a graph:

$$\mathsf{val}^{\mathsf{v}}_{\mathsf{G}}(r) = \begin{cases} \mathsf{val}^{\mathsf{v}}_{\mathsf{G}}(q) & \text{if } (r \underset{q}{\mapsto}) \in G \\ c(\overline{val_G(x_n)}) & \text{if } (r \mapsto c(\overline{x_n})) \in G \\ \_ & \text{otherwise} \end{cases}$$

Observe the similarities between functions $val_{\mathcal{D}}$ and val_G. The main difference is that, in order to get the final value of a variable, $val_{\mathcal{D}}$ looks at the bindings in the heap of the final configuration of a computation, while val_G follows a path in the graph of the final configuration of this computation.

Now, the node associated with a slicing criterion in the extended redex trail can be determined as follows:

Definition 14. (SC-node) Let P be a program and $(C_0 \Longrightarrow^* C_m)$ a complete derivation, where $G = graph(C_m)$. Given a slicing criterion $\langle f(\overline{pv_n}), \pi \rangle$, the associated SC-node must fulfill the conditions:[13]

[13] As in Definition 10, if there are several nodes that fulfill these conditions, we choose the first one following a traversal of the graph that is defined as follows:

$$traversal(G, r) = \begin{cases} r; traversal(G, q'); traversal(G, q) & \text{if } (r \underset{q}{\mapsto} case\ x\ of\ \ldots) \in G \\ & \text{and } (x \rightsquigarrow q') \in G \\ r; traversal(G, q) & \text{otherwise, with } (r \underset{q}{\mapsto}) \in G \end{cases}$$

i.e., whenever we find a case expression, we first follow the evaluation of the case argument and then the selected branch. The initial call would be $traversal(G, r)$ where G is the computed graph and r is the reference of the initial configuration.

$$
\begin{array}{l}
DS(G,r,\pi,V) \;= \\
\left\{
\begin{array}{ll}
\mathcal{P} \,\cup\, DS\_AUX(G,c(\overline{x_n}),r,\pi,V) & \text{if } (r \mapsto_{\mathcal{P}} c(\overline{x_n})) \in G \\[1ex]
\mathcal{P} \,\cup\, DS(G,q,\pi,V') & \text{if } (r \underset{q}{\mapsto_{\mathcal{P}}} \mathit{let}\ x = e\ \mathit{in}\ e') \in G, \text{ where } V' = V \cup \{x\} \\[1ex]
\mathcal{P} \,\cup\, DS(G,q,\pi,V) & \text{if } (r \underset{q}{\mapsto_{\mathcal{P}}} \mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\}) \in G \text{ and } x \notin V \\[1ex]
\mathcal{P} \,\cup\, DS(G,q',\mathit{hnf},V) & \text{if } (r \underset{q}{\mapsto_{\mathcal{P}}} \mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\}) \in G \text{ and } x \in V, \\
\quad\;\cup\, DS(G,q,\pi,V) & \text{where } (x \leadsto q') \in G \\[1ex]
\mathcal{P} \,\cup\, DS(G,q,\pi,V) & \text{otherwise, where } (r \underset{q}{\mapsto_{\mathcal{P}}}) \in G
\end{array}
\right. \\
\hline
DS\_AUX(G,c(\overline{x_n}),r,\pi,V) = \bigcup_{(x,\pi') \in c(\overline{x_m}) \sqcap_V \pi} DS(G,r_x,\pi',V) \;\; \text{where } (x \leadsto r_x) \in G
\end{array}
$$

Figure 20. Extracting a dynamic slice from a redex trail (function *DS*)

– $(r \mapsto f(\overline{x_n})) \in G$ and

– $pv_i = val_G(x_i)$ for all $i = 1, \ldots, n$.

Analogously to the notion of SC-configuration, the SC-node is only partially defined since it may happen that the conditions of the above definition do not hold. For simplicity, in the following we assume that there is always an SC-node associated to any slicing criterion.

We now introduce a function to compute a dynamic slice from the program positions of the nodes that are reachable from the SC-node (according to the slicing pattern).

Definition 15. (function DS) Let G be an extended redex trail. Let $\langle f(\overline{pv_n}), \pi\rangle$ be a slicing criterion and r the reference of its associated SC-node. A slice of $\langle f(\overline{pv_n}), \pi\rangle$ in G is given by $\{hd(\mathcal{P})\} \cup DS(G, q, \pi, \{\,\})$, where $(r \underset{q}{\mapsto_{\mathcal{P}}}) \in G$ and function DS is defined in Figure 20.

The parallelisms between functions *ds* and *DS* should be clear. While traversing the complete subcomputation for the slicing criterion, both functions

– consider the subcomputations for those variables introduced in a let expression *after* the slicing criterion and

– ignore the subcomputations for those variables introduced in a let expression *before* the slicing criterion.

Once the value of the slicing criterion is reached, both functions use the slicing pattern to determine which further subcomputations should be considered.

The following theorem shows that slices computed by function *DS* are indeed dynamic slices according to Definition 12 (the proof can be found in the appendix):

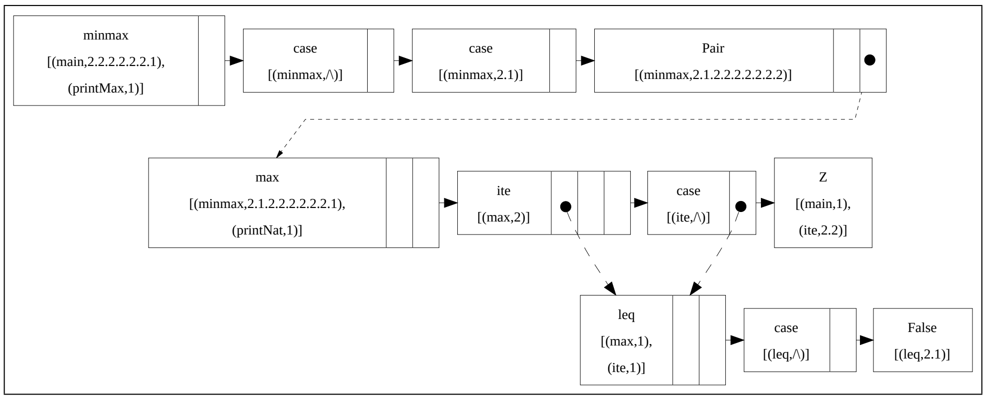


Figure 21. Reachable nodes w.r.t. ⟨`minmax` (`Z` : _ : _), `Pair` _ `Z`, `1`, `Pair` $\perp$ $\top$⟩

Theorem 2. (completeness and minimality) Let P be a program and $(C_0 \Longrightarrow^* C_m)$, $m > 0$, a complete derivation, with $G = \mathit{graph}(C_m)$. Let $\langle f(\overline{pv_n}), \pi\rangle$ be a slicing criterion and $\mathcal{W}$ be the dynamic slice computed according to Definition 12. Then, we have $DS(G, r, \pi, \{\,\}) = \mathcal{W}$, where r is the SC-node associated with the slicing criterion.

Example 7. Consider the (normalized version) of the program in Example 1 and the slicing criterion ⟨`minmax (Z : _ : _)`, `Pair` $\perp$ $\top$⟩. As we have already seen in Section 2, the extended redex trail of Figure 4 is computed. Figure 21 shows the SC-node together with the nodes which are reachable from the SC-node according to the slicing pattern (`Pair` $\perp$ $\top$), including the associated lists of program positions.[14] The computed slice is the one already shown in Figure 5.

5.2. Logical Features

In this section, we sketch the extension of the previous developments in this section in order to cope with the logical features of the considered functional logic language.

The first notion that should be extended is that of slicing criterion. Definition 6 is not appropriate in a lazy functional *logic* language because of non-determinism, since the same function call may be reduced to different values within the same computation. Consider, e.g., the following function definition:

```
coin = Z or (S Z)
```

[14] Note that the program positions correspond to the normalized version of the program shown in Figure 2, i.e., also including let expressions to have only variable arguments.

Here, `coin` can be non-deterministically reduced to `Z` and `(S Z)`. In order to identify a concrete occurrence of the given call, we add another element to the slicing criterion: the computed value.[15] Therefore, a slicing criterion is defined in the functional logic setting as a tuple $\langle f(\overline{v_n}), v, \pi\rangle$, where $f(\overline{v_n})$ is a function call whose arguments appear as much evaluated as needed in the complete computation, v is the computed value, and π is a pattern that determines the interesting part of the computed value, e.g., $\langle \mathtt{coin}, \mathtt{Z}, \top\rangle$.

Regarding the notions of SC-configuration and SC-node, only functions $val_{\mathcal{D}}$ and val_G should be slightly extended in order to also consider a logical variable as a value. For this purpose, auxiliary function $\mathsf{val}^{\mathsf{v}}_{\mathcal{D}}$ is defined as follows:

$$\mathsf{val}^{\mathsf{v}}_{\mathcal{D}}(x) \;=\; \begin{cases} \mathsf{val}^{\mathsf{v}}_{\mathcal{D}}(y) & \text{if } \Gamma[x] = y \text{ and } x \neq y \\ c(\overline{val_{\mathcal{D}}(x_n)}) & \text{if } \Gamma[x] = c(\overline{x_n}) \\ \mathit{LogVar} & \text{if } \Gamma[x] = x \\ \_ & \text{otherwise} \end{cases}$$

Correspondingly, function $\mathsf{val}^{\mathsf{v}}_{\mathsf{G}}$ is defined as follows:

$$\mathsf{val}^{\mathsf{v}}_{\mathsf{G}}(r) \;=\; \begin{cases} \mathsf{val}^{\mathsf{v}}_{\mathsf{G}}(q) & \text{if } (r \underset{q}{\mapsto}) \in G \\ c(\overline{val_G(x_n)}) & \text{if } (r \mapsto c(\overline{x_n})) \in G \\ \mathit{LogVar} & \text{if } (r \mapsto \mathit{LogVar}) \in G \\ \_ & \text{otherwise} \end{cases}$$

Finally, functions ds and DS require no significant changes, i.e., the only difference is that the underlying calculus also includes the rules of Figure 15.

Analogously to the development in the previous section, the extension of our results and, particularly, of Theorem 2 to cope with the logical features of the language does not involve significant difficulties, because the results from (Braßel et al., 2004) required for proving this theorem (see the appendix) have been already proved for a language which includes these logical features.

6. Implementation Issues

In order to check the usefulness and practicality of the ideas presented in this work, we have developed a prototype implementation of the

[15] Observe that we can still have an expression like `coin` + `coin` that evaluates to the same value (`S Z`) in different ways. Here, we implicitly consider the first of such computations. This is somehow a limitation, but it is not realistic to expect that users are able to distinguish among different computations which share both the same initial function call and the same final value.

$(\vert e\vert)_\Lambda = \Lambda$	for all expression e
$(\vert c(\overline{e_n})\vert)_{i.w} = i.(\vert e_i\vert)_w$	if $i \in \{1,\ldots,n\}$
$(\vert f(\overline{e_n})\vert)_{i.w} = i.(\vert e_i\vert)_w$	if $i \in \{1,\ldots,n\}$
$(\vert let\ x = e\ in\ e'\vert)_{1.w} = p_x.(\vert e\vert)_w$	where p_x is the position of x in e'
$(\vert let\ x = e\ in\ e'\vert)_{2.w} = (\vert \sigma(e')\vert)_w$	where $\sigma = \{x \mapsto e\}$
$(\vert (f)case\ e\ of\ \{\overline{p_n \rightarrow e_n}\}\vert)_{1.w} = 1.(\vert e\vert)_w$	
$(\vert (f)case\ e\ of\ \{\overline{p_n \rightarrow e_n}\}\vert)_{2.i.w} = 2.i.(\vert e_i\vert)_w$	if $i \in \{1,\ldots,n\}$

Figure 22. Conversion of positions from normalized flat programs to flat programs

$[\vert e\vert]^n_\Lambda = (n,\Lambda)$	for all expression e
$[\vert c(\overline{e_k})\vert]^n_{i.w} = (m, i.v)$	if $i \in \{1,\ldots,k\}$ and $[\vert e_i\vert]^n_w = (m,v)$
$[\vert f(\overline{e_k})\vert]^n_{i.w} = (m, i.v)$	if $i \in \{1,\ldots,k\}$ and $[\vert e_i\vert]^n_w = (m,v)$
$[\vert (f)case\ e\ of\ \{\overline{p_k \rightarrow e_k}\}\vert]^n_{2.i.w} = (n+m, v)$	if $i \in \{1,\ldots,k\}$ and $[\vert e_i\vert]^l_w = (m,v)$, where $l = brs(\overline{p_{i-1} \rightarrow e_{i-1}})$

Figure 23. Conversion of positions from flat programs to source programs

debugging tool that has been sketched in Section 2. It combines both tracing and slicing by extending a previous tracer for Curry programs (Braßel et al., 2004). Three main extensions were necessary:

– First, we modified the instrumented interpreter of Braßel et al. (2004) that generates the redex trail in order to also include program positions. Program positions are added in such a way that they do not interfere with the tracing process.

– Then, we defined algorithms to convert program positions for normalized flat programs—the programs considered in the instrumented semantics—to source Curry programs. This is necessary to be able to show program slices at an adequate level of abstraction. The kernel of the implemented conversion algorithms is shown in Figures 22 and 23. For simplicity, here we consider that let expressions are not allowed in source Curry programs, i.e., they are only used to take care of sharing.

  Given an expression e in a normalized flat program and its position w, we have that $(|e|)_w$ returns the position of e in the corresponding flat (non-normalized) program. Basically, transforming a non-normalized flat program into a flat program consists in applying inlining to every let expression. Consequently, function $(|\ |)$ only changes those positions inside a let expression in order to reflect these inlining steps. For instance, the position 2 addresses the call `f x` in the right-hand side of the following normalized rule:

```
main = let x = Z in f x
```

The associated position in the non-normalized rule

```
main = f Z
```

is $(|\texttt{let x} = \texttt{Z in f x}|)_2 = (|\texttt{f Z}|)_\Lambda = \Lambda$.

Then, function $[|\ |]$ takes an expression e in a flat program and its position w, so that $[|e|]^1_w$ returns the position of e in the corresponding source program. Transforming a flat program into a source Curry program mainly consists in replacing case expressions by pattern matching in the left-hand sides of different rules. Therefore, function $[|\ |]$ only changes positions inside a case expression so that these prefixes which are used to identify a particular case branch are deleted and the rule number within the function definition is determined instead. The superscript in function $[|\ |]$ is used to calculate the rule number within a definition (so it is initialized to 1). The auxiliary function *brs* is used to count the number of case branches within a list of expressions. Here, we assume that source Curry programs have no case expressions and that case expressions have always a variable argument.[16]

For instance, consider the following flat rule:

```
g x = case x of {Z → Z; S y → f y}
```

The position 2.2 addresses the call to (`f y`) in the right-hand side of the second branch of the case expression. Given the associated source Curry rule:

```
g Z     = Z
g (S y) = f y
```

the position associated with (`f y`) is given by $[|\texttt{case x of } \{\texttt{Z} \rightarrow \texttt{Z};\ \texttt{S y} \rightarrow \texttt{f y}\}|]^1_{2.2} = (1+1, \Lambda) = (2, \Lambda)$ (i.e., the right-hand side of the second rule), since $[|\texttt{f y}|]^1_\Lambda = (1, \Lambda)$.

We note, however, that there exist some features of Curry that are not covered in our current prototype, such as higher-order functions and local definitions.

– Finally, we have implemented a viewer that takes a slicing criterion and an extended redex trail and shows the associated program slice. This viewer first determines the SC-node (according to Definition 14) and then implements function *DS* to collect the set of program positions in the slice. The program positions for the

[16] This constraint is reasonable because flat programs obtained by translating source Curry programs without case expressions always fulfill it.

considered normalized flat program are first converted to their associated positions in the source Curry program. Then, the source Curry program is shown to the user by highlighting the expressions whose position belong to the slice.

Let us note that the complexity of the slicing algorithm is not a main concern in this work. Our slicing technique is built on top of an existing tracer. Therefore, if an efficient implementation of the tracer is available, then slicing would also run efficiently because only a slight extension is necessary (adding program positions to redex trails). Only the viewer, which takes care of computing the SC-node and gathering the set of program positions which are reachable from this node, requires some additional cost. Clearly, gathering the reachable program positions from the redex trail can be done in linear time if the repeated inspection of the same variable is avoided, which can easily be achieved by means of some sort of memoization. To be more precise, we increase the run time of the tracer by a constant factor that depends on the size of the redex trail. This is not a problem in practice since traceable programs usually produce small to medium size trails (otherwise, tracers cannot deal with them).

7. Related Work

Despite the fact that the usefulness of slicing techniques has been known for a long time in the imperative paradigm—see a list of successful applications in (Tip, 1995)—there are very few approaches to program slicing in the context of declarative languages. In the following, we review the closest to our work. Reps and Turnidge (1996) presented a backward slicing algorithm for first-order functional programs. This work is mainly oriented towards performing specialization by computing slices for *part* of a function's output, which is specified by means of a projection. In contrast to our approach, they consider a *strict* language and *static* slicing. We note that static slicing is less useful for debugging since it is generally less precise than dynamic slicing. Moreover, only a *fixed* slicing criterion is considered: the output of the whole program. Biswas (1997a, 1997b) considered a higher-order strict functional language and developed a technique for dynamic backward slicing based on labeling program expressions and then constructing an execution trace. Again, only a fixed slicing criterion is considered (although the extension to more flexible slicing criteria is discussed).

Liu and Stoller (2003) defined an algorithm for static backward slicing in a first-order strict functional language, where their main interest is the removal of dead code. Although the underlying semantics is different (lazy vs. strict), our slicing patterns are inspired by their *liveness patterns*.

Hallgren (2003) presented (as a result of the Programatica project) the development of a slicing tool for Haskell. Although there is little information available, it seems to rely on the construction of a graph of functional dependences (i.e., only function names and their dependences are considered). Therefore, in principle, it should produce bigger—less precise—slices than our technique.

Rodrigues and Barbosa (2005) introduced a novel approach to slicing functional programs which is based on the use of projection (for backward slicing) and hiding (for forward slicing) functions as slicing criteria. When these functions are composed with the original program, they lead to the identification of the intended slice. A positive aspect of this approach is that no data structure storing program dependences is needed.

The closest approach to our work is the *forward* slicing technique for lazy functional logic programs introduced by Vidal (2003). In this approach, a slight extension of a partial evaluation procedure for functional logic programs is used to compute program slices. Since partial evaluation naturally supports partial data structures, the distinction between static and dynamic slicing is not necessary (the same algorithm can be applied in both cases). This generality is a clear advantage of (Vidal, 2003), but it often implies a loss of precision.

Our slicing technique is based on an extension of the redex trail introduced by Braßel et al. (2004). This work introduces an instrumented operational semantics for lazy functional logic programs that generates a redex trail of a computation. This redex trail shares many features with the ART model of Wallace et al. (2001) but there is an important difference: while the ART model of Wallace et al. (2001) is defined in terms of a program transformation, Braßel et al. (2004) present a direct, semantics-based definition of redex trails. This improves the understanding of the tracing process and allows one to prove properties like "every reduction occurring in a computation can be observed in the trail". Chitil (2001) presents a first approach to such a direct definition by introducing an augmented small-step operational semantics for a core of Haskell that generate ARTs. Recently, he has also proposed a source-based trace exploration (Chitil, 2005) for the higher-order lazy functional language Haskell which combines ideas from algorithmic debugging, traditional stepping debuggers, and dynamic slicing. Although this proposal shares many features with our approach, no correctness results are provided (since no semantics-based definition of redex trails is available).

This work improves and extends (Ochoa et al., 2004). In particular, we present a revised version of the instrumented semantics with *lists* of program positions, we give more details on the implementation of a slicing tool (e.g., we provide algorithms to convert program positions from normalized flat programs to source Curry programs), and we provide proofs of technical results.

8. Conclusions and Future Work

In this work, we have presented a new scheme to perform dynamic slicing in a lazy first-order functional language. For this purpose, we have introduced a non-trivial notion of slicing criterion which is particularly well suited for lazy evaluation. Then, we have formalized the notion of dynamic (backward) slice associated to a given computation and have provided a constructive method to obtain this slice from the redex trail associated to the considered computation. This allows one to easily combine tracing and slicing into a single framework that builds the redex trail of the computation as a common device for both tasks. Finally, we have also sketched the extension of our developments to the case of a functional *logic* language including disjunctions and case expressions with free variables.

As future work, we plan to investigate the definition of appropriate methods for the *static* slicing of functional programs. In this case, we think that our approach could still be used but a sort of *abstract* redex trail becomes necessary for guaranteeing its finiteness even when no input data are provided. Some preliminary results along this direction can be found in (Cheda et al., 2007).

Acknowledgements

We gratefully acknowledge the anonymous referees, also of PEPM 2004, whose detailed comments and suggestions greatly improved this paper. We also thank Julia Lawall for her careful revision of the paper.

References

Albert, E., M. Hanus, F. Huch, J. Olvier, and G. Vidal: 2005, ‘Operational Semantics for Declarative Multi-Paradigm Languages’. *Journal of Symbolic Computation* **40**(1), 795–829.

Biswas, S.: 1997a, ‘A Demand-Driven Set-Based Analysis’. In: *Proc. of the 24th ACM Symp. on Principles of Programming Languages (POPL’97)*. pp. 372–385, ACM Press.

Biswas, S.: 1997b, ‘Dynamic Slicing in Higher-Order Programming Languages’. Ph.D. thesis, Department of CIS, University of Pennsylvania.

Braßel, B., M. Hanus, F. Huch, and G. Vidal: 2004, 'A Semantics for Tracing Declarative Multi-Paradigm Programs'. In: *Proc. of the 6th Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'04)*. pp. 179–190, ACM Press.

Cheda, D., J. Silva, and G. Vidal: 2007, 'Static Slicing of Rewrite Systems'. In: *Proc. of the 15th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP 2006)*. Electronic Notes in Theoretical Computer Science 177:123-136.

Chitil, O.: 2001, 'A Semantics for Tracing'. In: *13th Int'l Workshop on Implementation of Functional Languages (IFL 2001)*. pp. 249–254, Ericsson Computer Science Laboratory.

Chitil, O.: 2005, 'Source-Based Trace Exploration'. In: *16th Int'l Workshop on Implementation of Functional Languages (IFL 2004)*. pp. 126–141, Springer LNCS 3474.

Ferrante, J., K. Ottenstein, and J. Warren: 1987, 'The Program Dependence Graph and Its Use in Optimization'. *ACM Transactions on Programming Languages and Systems* **9**(3), 319–349.

Gill, A.: 2000, 'Debugging Haskell by Observing Intermediate Data Structures'. *Electronic Notes in Theoretical Computer Science* **41**(1). Proc. of the 4th Haskell Workshop.

Hallgren, T.: 2003, 'Haskell Tools from the Programatica Project'. In: *Proc. of the ACM Workshop on Haskell (Haskell'03)*. pp. 103–106, ACM Press.

Hanus, M.: 1997, 'A Unified Computation Model for Functional and Logic Programming'. In: *Proc. of the 24th ACM Symp. on Principles of Programming Languages (POPL'97)*. pp. 80–93, ACM.

Hanus, M.: 2000, 'Curry: An Integrated Functional Logic Language'. Available at: `http://www.informatik.uni-kiel.de/~curry/`.

Hanus, M. and C. Prehofer: 1999, 'Higher-Order Narrowing with Definitional Trees'. *Journal of Functional Programming* **9**(1), 33–75.

Korel, B. and J. Laski: 1988, 'Dynamic Program Slicing'. *Information Processing Letters* **29**(3), 155–163.

Kuck, D., R. Kuhn, D. Padua, B. Leasure, and M. Wolfe: 1981, 'Dependence Graphs and Compiler Optimization'. In: *Proc. of the 8th Symp. on the Principles of Programming Languages (POPL'81), SIGPLAN Notices*. pp. 207–218.

Liu, Y. and S. Stoller: 2003, 'Eliminating Dead Code on Recursive Data'. *Science of Computer Programming* **47**, 221–242.

López-Fraguas, F. and J. Sánchez-Hernández: 1999, 'TOY: A Multiparadigm Declarative System'. In: *Proc. of the 10th Int'l Conf. on Rewriting Techniques and Applications (RTA'99)*. pp. 244–247, Springer LNCS 1631.

Nilsson, H. and J. Sparud: 1997, 'The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging'. *Automated Software Engineering* **4**(2), 121–150.

Ochoa, C., J. Silva, and G. Vidal: 2004, 'Dynamic Slicing Based on Redex Trails'. In: *Proc. of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM'04)*. pp. 123–134, ACM Press.

Peyton Jones, S. (ed.): 2003, *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

Pope, B.: 2006, 'A Declarative Debugger for Haskell'. Ph.D. thesis, The University of Melbourne, Australia.

Pope, B. and L. Naish: 2003, 'A Program Transformation for Debugging Haskell 98'. In: *Proc. of 26th Australasian Computer Science Conference (ACSC 2003)*, Vol. 16 of *Conferences in Research and Practice in Information Technology*. pp. 227–236, ACS.

Reps, T. and T. Turnidge: 1996, ‘Program Specialization via Program Slicing’. In: O. Danvy, R. Glück, and P. Thiemann (eds.): *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*. pp. 409–429, Springer LNCS 1110.

Rodrigues, N. and L. Barbosa: 2005, ‘Slicing Functional Programs by Calculation’. In: *Proc. of the Dagstuhl Seminar on Beyond Program Slicing*. Seminar n. 05451, Schloss Dagstuhl.

Sparud, J. and C. Runciman: 1997, ‘Tracing Lazy Functional Computations Using Redex Trails’. In: *Proc. of the 9th Int’l Symp. on Programming Languages, Implementations, Logics and Programs (PLILP’97)*. pp. 291–308, Springer LNCS 1292.

Tip, F.: 1995, ‘A Survey of Program Slicing Techniques’. *Journal of Programming Languages* **3**, 121–189.

Vidal, G.: 2003, ‘Forward Slicing of Multi-Paradigm Declarative Programs Based on Partial Evaluation’. In: *Logic-based Program Synthesis and Transformation (revised and selected papers from the 12th Int’l Workshop LOPSTR 2002)*. pp. 219–237, Springer LNCS 2664.

Wallace, M., O. Chitil, T. Brehm, and C. Runciman: 2001, ‘Multiple-View Tracing for Haskell: a New Hat’. In: *Proc. of the 2001 ACM SIGPLAN Haskell Workshop*. Universiteit Utrecht UU-CS-2001-23.

Weiser, M.: 1984, ‘Program Slicing’. *IEEE Transactions on Software Engineering* **10**(4), 352–357.

Appendix

A. Proofs of Technical results

A.1. Proof of Theorem 1

We begin with the proof of Theorem 1. First, we need the following auxiliary lemma that proves two useful invariants for the configurations in a derivation. In what follows, given a configuration of the form $C = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle$, we let $\mathit{heap}(C) = \Gamma$, $\mathit{ctrl}(C) = e$, $\mathit{stack}(C) = S$, $\mathit{graph}(C) = G$, $\mathit{ref}(C) = r$, and $\mathit{pos}(C) = \mathcal{P}$.

Lemma 1. Let P be a program, C_0 be an initial configuration, and $(C_0 \Longrightarrow^* C_m)$, $m \geqslant 0$, be a derivation (not necessarily complete) in P. For each configuration $C_i = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle$, $0 \leqslant i \leqslant m$, the following conditions hold:

($I1$) either $\mathcal{P} = []$ (and $e = \mathtt{main}$, $e = \Diamond$, or $e = \mathit{Fail}$) or $\mathcal{P} = (g, w) : \mathcal{P}'$ for some $\mathcal{P}'$ and there exists a rule $g(\overline{x_n}) = e_1 \in P$ such that $e_1|_w = e$ (up to variable renaming), and

($I2$) for all bindings $x \mapsto_{(g',w')} e'$ in the heap Γ, there exists a rule $g'(\overline{x_k}) = e_2$ in P such that $e_2|_{w'} = e'$ (up to variable renaming).

Proof. We prove the claim by induction on the length m of the considered computation.

Base case ($m = 0$) Then, the derivation contains only the initial configuration

$$\langle \mathtt{[]}, \mathtt{main}, \mathtt{[]}, \mathtt{G}_\emptyset, r, [\,] \rangle$$

and both invariants trivially hold.

Inductive case ($m > 0$) Let us consider an arbitrary derivation of the form

$$C_0 \Longrightarrow C_1 \Longrightarrow \cdots \Longrightarrow C_{m-1} \Longrightarrow C_m$$

By the inductive hypothesis, both (I1) and (I2) hold in C_{m-1}. Now, we prove that (I1) and (I2) also hold in C_m. If $\mathit{pos}(C_{m-1}) = []$, by definition of the instrumented semantics, C_{m-1} can only be an initial configuration (with $\mathit{ctrl}(C_{m-1}) = \mathtt{main}$) or a final one (with $\mathit{ctrl}(C_{m-1}) = \Diamond$ or $\mathit{ctrl}(C_{m-1}) = \mathit{Fail}$). Moreover, in our case, C_{m-1} can only be an initial configuration since we know that there exists a successor configuration C_m. Therefore, only rule fun can be applied to C_{m-1}. Here, invariant (I1) trivially holds in C_m since $\mathit{pos}(C_m) = [(\mathtt{main}, \Lambda)]$ clearly addresses the right-hand side of function `main`. Invariant (I2) also holds because the heap is not modified.

Otherwise, assume that $C_{m-1} = \langle \Gamma, e, S, G, r, (g,w) : \mathcal{P} \rangle$. We consider the following cases depending on the applied rule:

(varcons) Then $e = x$ is a variable. Consider that $\Gamma = \Gamma[x \mapsto_{(g',w')} t]$. It is immediate that invariant (I1) holds in the derived configuration $C_m = \langle \Gamma, t, S, G, r, (g',w') : (g,w) : \mathcal{P} \rangle$ because the invariant (I2) holds in C_{m-1}. Also, invariant (I2) trivially holds in C_m because the heap is not modified in the step.

(varexp) Then $e = x$ is a variable and the proof is perfectly analogous to the previous case.

(val) Then either e is a (logical) variable or a constructor-rooted term. In either case, invariant (I1) trivially holds in C_m because both the control and the program position are not changed. As for invariant (I2), since invariant (I1) holds in C_{m-1}, we know that program position (g,w) is a correct program position for v. Therefore, the update of the heap is correct and, since invariant (I2) already holds in C_{m-1}, it still holds in C_m.

(fun) Then $e = f(\overline{x_n})$ is a function call. Since the heap is not modified, invariant (I2) trivially holds in C_m. Moreover, since the expression in the control is the renamed (by ρ) right-hand side of function f, the program position (f, Λ) is obviously correct (up to the variable renaming ρ) and invariant (I1) follows.

(let) Then $e = (let\ x = e_1\ in\ e_2)$ is a let expression. Since invariant (I1) holds in C_{m-1}, by definition of position, $(g, w.2)$ correctly addresses the (renaming of) expression e_2 and, thus, invariant (I1) also holds in C_m. Similarly, $(g, w.1)$ correctly addresses the (renaming of) expression e_1 and, since this is the only modification on heap Γ and (I2) holds for C_{m-1}, (I2) also holds in C_m.

(case) Then $e = case\ x\ of\ \{\overline{p_k \to e_k}\}$. Since the heap is not modified, invariant (I2) trivially holds in C_m. Moreover, by definition of position, $(g, w.1)$ correctly addresses the case argument and, thus, invariant (I1) also holds in C_m.

(select) Then $e = c(\overline{x_n})$. Since the heap is not modified in C_m, invariant (I2) trivially holds. Moreover, it is easy to prove that the program position of the case expression that demanded the evaluation of $c(\overline{x_n})$ is (g', w'), which is stored in the graph. Therefore, by definition of position, $(g', w'.2.i)$ correctly addresses the expression $\rho(e_i)$ in the i-th branch of the case expression (up to the variable renaming ρ).

(select-f) Then $e = c(\overline{x_n})$. Since the heap is not modified in this step, invariant (I2) trivially holds in C_m. Moreover, since the list of program positions in C_m is $[\,]$, invariant (I1) also holds.

(success) Then $e = c(\overline{x_n})$. This case follows trivially because the heap is not modified and the list of program positions in C_m is $[\,]$.

According to the instrumented semantics of Figure 13, not all expressions in the control of a configuration are stored in the graph. The following definition, slightly extended from (Braßel et al., 2004) to include program positions (and ignore parent references), formalizes when a configuration contains an expression that will be stored in the graph (in the next step of the computation).

Definition 16. A configuration $\langle \Gamma, e, S, G, r, \mathcal{P}\rangle$ is *relevant* iff one of the following conditions hold:

1. e is a value and the stack S is not headed by a variable, or

2. e is a function call, a let expression, or a case expression.

The following lemma—a simplified version of the analogous lemma in (Braßel et al., 2004) where logical features are not considered—states that, for each relevant configuration in a derivation, the associated information is stored in the graph.

Lemma 2. Let $(C_0 \Longrightarrow^* C_n)$, $n > 0$, be a derivation. For each $i \in \{0, \ldots, n\}$, we have that $C_i = \langle \Gamma, e, S, G, r, \mathcal{P}\rangle$ is a relevant configuration iff $r \mapsto_{\mathcal{P}} e \in \mathit{graph}(C_{i+1})$ and $r \mapsto_{\mathcal{P}} e \notin G$.

Finally, we state and prove Theorem 1:

Theorem 1. Let P be a program and $(C \Longrightarrow^* C')$ be a complete derivation in P, where G' is the graph in the final configuration C'. If $(r \mapsto_{\mathcal{P}} e) \in G'$, then either $\mathcal{P} = [\,]$ (with $e =$ `main`) or $\mathcal{P} = [(g_k, w_k), \ldots, (g_1, w_1)]$, $k \geqslant 1$, and the following conditions hold:

1. there exists a subderivation $C_1 \Longrightarrow \ldots \Longrightarrow C_j$ of $C \Longrightarrow^* C'$, $j \geqslant k$, such that $\mathit{pos}(C_j) = [(g_k, w_k), \ldots, (g_1, w_1)]$, and

2. for all $i = 1, \ldots, j$, there exists a rule $g'_i(\overline{x_{n_i}}) = e_i \in P$ such that $e_i|_{w'_i} = \mathit{ctrl}(C_i)$, and

   – $g'_i = g_i$ and $w'_i = w_i$ (for $i = 1, \ldots, k$)

   – $g'_i = g_k$ and $w'_i = w_k$ (for $i = k+1, \ldots, j$)

   with $e_j|_{w_j} = e$.

Proof. If $\mathcal{P} = [\,]$ then, by Lemma 2, there must be some relevant (and non final) configuration C_i with $\mathit{ctrl}(C_i) = e$ and $\mathit{pos}(C_i) = [\,]$. By Lemma 1, since $\mathit{pos}(C_i) = [\,]$, we have that $e =$ `main`, $e = \Diamond$, or $e = \mathit{Fail}$. Finally, since C_i is not a final configuration, then $e =$ `main`.

Now, consider $\mathcal{P} = [(g_k, w_k), \ldots, (g_1, k_1)]$, $k \geqslant 1$. Since we have $(r \mapsto_{\mathcal{P}} e) \in G'$, by Lemma 2, there exists a relevant configuration $C_j = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle$. If $\mathcal{P} = [(g_k, w_k)]$ contains only one program position, the proof follows straightforwardly by Lemma 1 (with $k = 1, j = 1$). Otherwise, observe that only rules varcons, varexp and val can be applied to a configuration so that either the list of program positions is not modified or new program positions are added to it. Therefore, there must be a subderivation of the form $C_1 \Longrightarrow \ldots \Longrightarrow C_j$ of $C \Longrightarrow^* C'$ with $j = k$ (if rule val is never applied) or $j \geqslant k$ (otherwise) such that $pos(C_j) = [(g_k, w_k), \ldots, (g_1, k_1)]$, $k > 1$. Observe that the applications of rule val (if any) can only happen at the end of the subderivation, since once a value is set in the control, rules varcons and varexp are no longer applicable. By Lemma 1, we have that there exists a rule $g_i(\overline{x_{n_i}}) = e_i \in P$ such that $e_i|_{w_i} = ctrl(C_i)$, for all $i = 1, \ldots, k$. Finally, since only rule val is applied to $C_{k+1}, \ldots, C_j$, we have $e_k|_{w_k} = ctrl(C_i)$, for all $i = k+1, \ldots, j$, with $e_j|_{w_j} = e$, and the claim follows.

A.2. Proof of Theorem 2

First, we need some results from (Braßel et al., 2004) that we slightly modify to include program positions and ignore parent arrows: the notion of *successor* derivation and Propositions 1 and 2, which correspond, respectively, to Propositions 5.5 and 5.8 in (Braßel et al., 2004). First, we introduce the notion of successor derivation.

Definition 17. (successor derivation) Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a derivation. Let $C_i = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle$, $0 \leq i < m$, be a relevant configuration such that e is a function call, a let expression, or a case expression. Then, $C_i \Longrightarrow^* C_j$, $i < j \leq m$, is a successor subderivation of $\mathcal{D}$ iff (i) C_j is relevant, (ii) $stack(C_i) = stack(C_j)$, and (iii) there is no relevant configuration C_k, $i < k < j$, such that $stack(C_i) = stack(C_k)$.

Intuitively, a successor derivation represents a single reduction step including the associated subcomputations (if any). The following result states that, for each successor subderivation in a computation, the instrumented semantics adds a corresponding successor arrow to the graph, and vice versa.

Proposition 1. Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a derivation. For all i, j $(0 \leq i < j < m)$, there exists a successor subderivation

$$C_i = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle \Longrightarrow^* \langle \Gamma', e', S', G', r', \mathcal{P}' \rangle = C_j$$

in $\mathcal{D}$ iff $(r \underset{r'}{\mapsto}_{\mathcal{P}} e) \in graph(C_{j+1})$ and $(r' \underset{q}{\mapsto}_{\mathcal{P}'} e') \in graph(C_{j+1})$.

As mentioned before, when a lookup configuration $\langle\Gamma, x, S, G, r, p, \mathcal{P}\rangle$ appears in a computation, a variable pointer $x \rightsquigarrow r$ should be added to the graph. Moreover, reference r stores the *dereferenced value* of heap variable x. This is expressed by means of function Γ^* which is defined as follows:

$$\Gamma^*(x) = \begin{cases} \Gamma^*(y) & \text{if } \Gamma[x] = y \text{ and } x \neq y \\ \Gamma[x] & \text{otherwise} \end{cases}$$

The next proposition formally states that, for each variable whose value is *demanded* in a computation, we have an associated variable pointer in the graph (it is a slight extension of a similar result in (Braßel et al., 2004) in order to consider program positions).

Proposition 2. Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a derivation. For all i $(0 < i < m)$, there exists a lookup configuration $C_i = \langle\Gamma, x, S, G, r, \mathcal{P}\rangle$ in $\mathcal{D}$ iff $(x \rightsquigarrow r) \in \mathit{graph}(C_{j+1})$ and $r \mapsto_{\mathcal{P}'} e' \in \mathit{graph}(C_{j+1})$, where $C_j = \langle\Gamma', e', S', G', r, \mathcal{P}'\rangle$, $i \leq j < m$, is a relevant configuration and there is no relevant configuration C_k with $i < k < j$.

Observe that all the configurations in the subderivation $C_i \Longrightarrow^* C_{j-1}$ have a variable in the control. Therefore, only rules varcons and var-exp can be applied, which implies that their program positions are accumulated to the list of program positions in C_j.

In order to prove the completeness of the computed slice, we first need to prove that, given a slicing criterion, the associated SC-node correctly points to the node that stores the control of the associated SC-configuration. For this purpose, we need some preliminary results.

The next lemma shows that reductions in a derivation can also be obtained from the extended redex trail by following the successor relation.

Lemma 3. Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a complete derivation and $C_i = \langle\Gamma, e, S, G, r, \mathcal{P}\rangle$, $0 < i < m$, be a relevant configuration. There exists a complete subderivation $C_i \Longrightarrow^* C_j$ for C_i in $\mathcal{D}$ iff there exists a successor chain $(r \underset{q_1}{\mapsto}_{\mathcal{P}} e), (q_1 \underset{q_2}{\mapsto}_{\mathcal{P}_1} e_1), \ldots, (q_n \mapsto_{\mathcal{P}_n} e_n) \in \mathit{graph}(C_m)$ with $e_n = \mathit{ctrl}(C_j)$.

Proof. Since C_i is a relevant configuration, we have that $C_i \Longrightarrow^* C_j$ can be decomposed into a sequence of successor subderivations

$$\begin{array}{rcl} C_i = \langle\Gamma, e, S, G, r, \mathcal{P}\rangle & \Longrightarrow^* & \langle\Gamma_1, e_1, S_1, G_1, q_1, \mathcal{P}_1\rangle \\ & \Longrightarrow^* & \langle\Gamma_2, e_2, S_2, G_2, q_2, \mathcal{P}_2\rangle \\ & \ldots & \\ & \Longrightarrow^* & \langle\Gamma_n, e_n, S_n, G_n, q_n, \mathcal{P}_n\rangle \end{array}$$

Therefore, by Proposition 1, the claim follows.

The next lemma shows that, for each variable x demanded in a derivation $\mathcal{D}$, we have $val_{\mathcal{D}}(x) = val_G(x)$.

Lemma 4. Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a complete derivation and $C_i = \langle \Gamma, x, S, G, r, \mathcal{P} \rangle$, $0 < i < m$, be a lookup configuration. Then, $val_{\mathcal{D}}(x) = val_G(x)$.

Proof. We prove this claim by structural induction on the computed partial value pv.

(Base case) Then, pv is a constructor constant c. By definition of function val, we have: $val_{\mathcal{D}}(x) = \mathsf{val}^{\mathsf{v}}_{\mathcal{D}}(x) = \mathsf{val}^{\mathsf{v}}_{\mathcal{D}}(y_1) = \ldots = \mathsf{val}^{\mathsf{v}}_{\mathcal{D}}(y_n) = c$. Hence, we have $\Gamma^*(x) = c$. Moreover, by Proposition 2, we have that $(x \rightsquigarrow r) \in G$ and $(r \mapsto \Gamma^*(x)) \in G$, where $G = graph(C_m)$. Thus, $val_{\mathcal{D}}(x) = val_G(x) = c$.

(Inductive case) We now consider that the partial value pv is a constructor call $c(\overline{x_n})$. Then, $\Gamma[x] = c(\overline{x_n})$ and $val_{\mathcal{D}}(x) = \mathsf{val}^{\mathsf{v}}_{\mathcal{D}}(x) = c(\overline{val_{\mathcal{D}}(x_n)})$. By Proposition 2, we have that $(x \rightsquigarrow r) \in G$ and $(r \mapsto \Gamma^*(x)) \in G$, where $G = graph(C_m)$. Therefore, the claim follows by the inductive hypothesis since $val_{\mathcal{D}}(x_i) = val_G(x_i)$ for all $i = 1, \ldots, n$.

We now establish the equivalence between the notions of SC-node and SC-configuration for a given slicing criterion.

Theorem 3. Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$ be a complete derivation. Given a slicing criterion $\langle f(\overline{pv_n}), \pi \rangle$, the associated SC-configuration is $C_i = \langle \Gamma, f(\overline{x_n}), S, G, r, \mathcal{P} \rangle$ iff the associated SC-node is $r = ref(C_i)$.

Proof. Both the definition of SC-configuration and SC-node require two conditions. We consider each of them separately:

(1) Trivially, $C_i = \langle \Gamma, f(\overline{x_n}), S, G, r, \mathcal{P} \rangle$ is a relevant configuration. Then, by Lemma 2, we have that $G_{i+1} = G[r \mapsto_{\mathcal{P}} f(\overline{x_n})]$ and, thus, $(r \mapsto_{\mathcal{P}} f(\overline{x_n})) \in graph(C_m)$.

(2) This case follows by Lemma 4, since it implies that $val_{\mathcal{D}}(x_i) = val_G(x_i)$, with $(x_i \rightsquigarrow r_i) \in graph(C_m)$, for all $i = 1, \ldots, n$.

Finally, we state and prove Theorem 2. In the following, we say that a call, $ds([C_i, \ldots, C_j], \pi, V, \mathcal{D})$, is *relevant* if the configuration C_i is relevant.

Theorem 2. (completeness and minimality) Let P be a program and $(C_0 \Longrightarrow^* C_m)$, $m > 0$, a complete derivation, with $G = graph(C_m)$. Let

$\langle f(\overline{pv_n}), \pi\rangle$ be a slicing criterion and $\mathcal{W}$ be the dynamic slice computed according to Definition 12. Then, we have $DS(G, r, \pi, \{\,\}) = \mathcal{W}$, where r is the SC-node associated with the slicing criterion.

Proof. (Completeness) We prove a more general claim: we prove that for any, possibly incomplete, computation for ds, the same positions are also gathered by some, possibly incomplete, computation for DS.

Formally, let us denote a (possibly incomplete) evaluation of the dynamic slice using function ds as follows:

$$\mathcal{P}_0 \cup \overline{ds}_0 = \mathcal{P}_1 \cup \overline{ds}_1 = \ldots = \mathcal{P}_z \cup \overline{ds}_z$$

where

- each new equality is obtained from the previous one by unfolding a single call to ds one or more times until it reduces to a (possibly empty) union of relevant calls to ds,
- $\mathcal{P}_k$ is the already computed set of positions, and
- $\overline{ds}_k$ is the (possibly empty) union of relevant calls that have not been reduced yet

for all $k = 0, \ldots, z$.

Now, we prove that, for any $z \geq 0$, we have a corresponding (possibly incomplete) sequence of reductions of calls to DS:

$$\mathcal{P}'_0 \cup \overline{DS}_0 = \mathcal{P}'_1 \cup \overline{DS}_1 = \ldots = \mathcal{P}'_z \cup \overline{DS}_z$$

such that $\mathcal{P}'_z \supseteq \mathcal{P}_z$ and, for every call $ds(C'_z : CS_z, \pi_z, V_z, \mathcal{D})$ in $\overline{ds}_z$ (if any), we have that there exists a call $DS(G, r_z, \pi'_z, V'_z)$ in $\overline{DS}_z$ with $r_z = ref(C'_z)$, $(r_z \mapsto_{\mathcal{P}^z} e_z) \in G$, $\mathcal{P}^z = pos(C'_z)$, $e_z = ctrl(C'_z)$, $\pi'_z = \pi_z$, and $V'_z = V_z$.

We prove the claim by induction on the number z of relevant calls to ds reduced in the considered evaluation.

(Base case $z = 0$) Then, we have $\mathcal{P}_0 = \{hd(pos(C_i))\}$ and

$$\overline{ds}_0 = ds([C_{i+1}, \ldots, C_j], \pi, \{\,\}, \mathcal{D})$$

where C_i is the SC-configuration associated with the slicing criterion $\langle f(\overline{pv_n}), \pi\rangle$ and $C_i \Longrightarrow^* C_j$ is the complete subderivation for C_i in the considered computation. For simplicity, we assume that C_{i+1} is relevant (otherwise, we would proceed as in the inductive case, see below). By Theorem 3, the SC-node associated to $\langle f(\overline{pv_n}), \pi\rangle$ is $ref(C_i)$. Moreover, since C_i is relevant, by Lemma 2, we have $(r \underset{q}{\mapsto}_{\mathcal{P}} ctrl(C_i)) \in G$, with

$r = \mathit{ref}(C_i)$, $q = \mathit{ref}(C_{i+1})$, and $\mathcal{P} = \mathit{pos}(C_i)$. Hence, we construct the following initial expression for DS:

$$\overbrace{\{\mathit{hd}(\mathit{pos}(C_i))\}}^{\mathcal{P}'_0} \cup \overbrace{DS(G, \mathit{ref}(C_{i+1}), \pi, \{\ \})}^{\overline{DS}_0}$$

and the claim follows.

(Inductive case $z > 0$) Let us consider a possibly incomplete evaluation in which $z-1$ relevant calls to ds have been reduced. Then, by the inductive hypothesis, we have the following (possibly incomplete) sequence of reductions of calls to DS:

$$\mathcal{P}'_0 \cup \overline{DS}_0 = \mathcal{P}'_1 \cup \overline{DS}_1 = \ldots = \mathcal{P}'_{z-1} \cup \overline{DS}_{z-1}$$

such that $\mathcal{P}'_{z-1} \supseteq \mathcal{P}_{z-1}$ and, for every call

$$ds(C'_{z-1} : CS_{z-1}, \pi_{z-1}, V_{z-1}, \mathcal{D})$$

in $\overline{ds}_{z-1}$ (if any), there exists a call

$$DS(G, r_{z-1}, \pi'_{z-1}, V'_{z-1})$$

in $\overline{DS}_{z-1}$ such that $r_{z-1} = \mathit{ref}(C'_{z-1})$, $(r_{z-1} \mapsto_{\mathcal{P}^{z-1}} e_{z-1}) \in G$, $\mathcal{P}^{z-1} = \mathit{pos}(C'_{z-1})$, $e_{z-1} = \mathit{ctrl}(C'_{z-1})$, $\pi'_{z-1} = \pi_{z-1}$, and $V'_{z-1} = V_{z-1}$.

If there is no such call in $\overline{ds}_{z-1}$, the proof is done. Otherwise, let us consider a call of the form $ds(C'_{z-1} : CS_{z-1}, \pi_{z-1}, V_{z-1}, \mathcal{D})$. We denote by $\overline{ds}'_{z-1}$ the union of calls to ds in $\overline{ds}_{z-1}$ minus the considered one.

According to the definition of ds, we distinguish two cases, depending on whether CS_{z-1} is empty or not. Let us consider first that CS_{z-1} is not empty. Then, we consider the following cases:

– $\mathit{ctrl}(C'_{z-1}) = (\mathit{let}\ x = e\ \mathit{in}\ e')$. Then, according to the definition of ds, this call reduces to

$$\mathit{pos}(C'_{z-1}) \cup ds(CS_{z-1}, \pi_{z-1}, V_z, \mathcal{D})$$

where $V_z = V_{z-1} \cup \{x\}$.

Assume $CS_{z-1} = C^1_{z-1} : \ldots : C^p_{z-1} : CS_z$, where C^p_{z-1}, $p \geq 1$, is the first relevant configuration in CS_{z-1} (observe that, by construction, there must be at least one relevant configuration). Since $C^1_{z-1}, \ldots, C^{p-1}_{z-1}$ are not relevant configurations, by definition of ds, we have that

$$ds(CS_{z-1}, \pi_{z-1}, V_z, \mathcal{D}) = ds(C^p_{z-1} : CS_z, \pi_{z-1}, V_z, \mathcal{D})$$

by repeatedly applying the last case of ds. This is a consequence of the fact that, if a configuration is not relevant, its control can only contain a variable or a constructor call. Therefore, the next expression of the sequence is

$$\overbrace{\mathcal{P}_{z-1} \cup pos(C'_{z-1})}^{\mathcal{P}_z} \cup \overbrace{ds(C^p_{z-1} : CS_z, \pi_{z-1}, V_z, \mathcal{D}) \cup \overline{ds}'_{z-1}}^{\overline{ds}_z}$$

Now, we consider the call $DS(G, r_{z-1}, \pi'_{z-1}, V'_{z-1})$ in $\overline{DS}_{z-1}$ $r_{z-1} = ref(C'_{z-1})$, $(r_{z-1} \mapsto_{\mathcal{P}^{z-1}} e_{z-1}) \in G$, $\mathcal{P}^{z-1} = pos(C'_{z-1})$, $e_{z-1} = ctrl(C'_{z-1})$, $\pi'_{z-1} = \pi_{z-1}$, and $V'_{z-1} = V_{z-1}$. We denote by $\overline{DS}'_{z-1}$ the union of calls to DS in $\overline{DS}_{z-1}$ minus the above one.

First, since $C'_{z-1} \Longrightarrow^* C^p_{z-1}$ is a successor subderivation, by Proposition 1, we have

$$(r_a \underset{r_b}{\mapsto}_{\mathcal{P}_a} ctrl(C'_{z-1})) \in G \quad \text{and} \quad (r_b \underset{r'}{\mapsto}_{\mathcal{P}_b} ctrl(C^p_{z-1})) \in G$$

with

$$r_a = ref(C'_{z-1}),\ \mathcal{P}_a = pos(C'_{z-1}),\ r_b = ref(C^p_{z-1}),\ \mathcal{P}_b = pos(C^p_{z-1})$$

and r' a fresh reference. Hence, according to the definition of function DS, we have that

$$DS(G, r_{z-1}, \pi'_{z-1}, V'_{z-1}) = pos(C'_{z-1}) \cup DS(G, ref(C^p_{z-1}), \pi_{z-1}, V_z)$$

Therefore, the next expression of the sequence is now

$$\overbrace{\mathcal{P}'_{z-1} \cup pos(C'_{z-1})}^{\mathcal{P}'_z} \cup \overbrace{DS(G, ref(C^p_{z-1}), \pi_{z-1}, V_z) \cup \overline{DS}'_{z-1}}^{\overline{DS}_z}$$

and the claim follows since $\mathcal{P}'_{z-1} \supseteq \mathcal{P}_{z-1}$.

– $ctrl(C'_{z-1})$ is a function call. This case is perfectly analogous to the previous one with the only difference that $V_z = V_{z-1}$.

– $ctrl(C'_{z-1}) = case\ x\ of \ldots$ and $x \notin V_{z-1}$. Then, assume

$$CS_{z-1} = C^1_{z-1} : \ldots : C^p_{z-1} : C'_z : CS_z$$

where $C^1_{z-1} \Longrightarrow^* C^p_{z-1}$, $p > 1$, is the complete subderivation for C^1_{z-1} in the considered computation (i.e., the subderivation in which the variable x is evaluated) and that C'_z is relevant (otherwise, we would proceed as in the first case). By definition of ds, we have

$$\begin{array}{rl} & ds(C'_{z-1} : CS_{z-1}, \pi_{z-1}, V_{z-1}, \mathcal{D}) \\ = & \mathcal{P}os(C'_{z-1}) \cup ds(C'_z : CS_z, \pi_{z-1}, V_{z-1}, \mathcal{D}) \end{array}$$

Therefore, the next expression of the sequence is

$$\overbrace{\mathcal{P}_{z-1} \cup pos(C'_{z-1})}^{\mathcal{P}_z} \cup \overbrace{ds(C'_z : CS_z, \pi_{z-1}, V_{z-1}, \mathcal{D}) \cup \overline{ds}'_{z-1}}^{\overline{ds}_z}$$

Now, we consider the call $DS(G, r_{z-1}, \pi'_{z-1}, V'_{z-1})$ in $\overline{DS}_{z-1}$ such that $r_{z-1} = ref(C'_{z-1})$, $(r_{z-1} \mapsto_{\mathcal{P}^{z-1}} e_{z-1}) \in G$, $\mathcal{P}^{z-1} = pos(C'_{z-1})$, $e_{z-1} = ctrl(C'_{z-1})$, $\pi'_{z-1} = \pi_{z-1}$, and $V'_{z-1} = V_{z-1}$. We denote by $\overline{DS}'_{z-1}$ the union of calls to DS in $\overline{DS}_{z-1}$ minus the above one.

First, since $C'_{z-1} \Longrightarrow^* C'_z$ is a successor subderivation, by Proposition 1, we have

$$(r_a \underset{r_b}{\mapsto}_{\mathcal{P}_a} ctrl(C'_{z-1})) \in G \quad \text{and} \quad (r_b \underset{r'}{\mapsto}_{\mathcal{P}_b} ctrl(C'_z)) \in G$$

with

$$r_a = ref(C'_{z-1}),\ \mathcal{P}_a = pos(C'_{z-1}),\ r_b = ref(C'_z),\ \mathcal{P}_b = pos(C'_z)$$

and r' a fresh reference. Hence, according to the definition of function DS, we have that

$$DS(G, r_{z-1}, \pi'_{z-1}, V'_{z-1}) = pos(C'_{z-1}) \cup DS(G, ref(C'_z), \pi_{z-1}, V_{z-1})$$

Therefore, the next expression of the sequence is now

$$\overbrace{\mathcal{P}'_{z-1} \cup pos(C'_{z-1})}^{\mathcal{P}'_z} \cup \overbrace{DS(G, ref(C'_z), \pi_{z-1}, V_{z-1}) \cup \overline{DS}'_{z-1}}^{\overline{DS}_z}$$

and the claim follows since $\mathcal{P}'_{z-1} \supseteq \mathcal{P}_{z-1}$.

– $ctrl(C'_{z-1}) = case\ x\ of \ldots$ and $x \in V_{z-1}$. Assume now that we have $CS_{z-1} = C^1_{z-1} : \ldots : C^p_{z-1} : C'_z : CS_z$ where $C^1_{z-1} \Longrightarrow^* C^p_{z-1}$, $p > 1$, is the complete subderivation for C^1_{z-1} in the considered computation (i.e., the subderivation in which the variable x is evaluated) and that C'_z is relevant (otherwise, we would proceed as in the first case). By definition of ds, we have

$$\begin{array}{l} ds(C'_{z-1} : CS_{z-1}, \pi_{z-1}, V_{z-1}, \mathcal{D}) \\ \qquad = \mathcal{P}os(C'_{z-1}) \cup ds([C^1_{z-1}, \ldots, C^p_{z-1}], hnf, V_{z-1}, \mathcal{D}) \\ \qquad\qquad \cup\ ds(C'_z : CS_z, \pi_{z-1}, V_{z-1}, \mathcal{D}) \end{array}$$

Moreover, for simplicity, we assume that, in the sequence of configurations $C^1_{z-1}, \ldots, C^p_{z-1}$, the configuration C^2_{z-1} is the first relevant configuration. Hence, we have

$$\begin{array}{l} ds([C^1_{z-1}, \ldots, C^p_{z-1}], hnf, V_{z-1}, \mathcal{D}) \\ \qquad = ds([C^2_{z-1}, \ldots, C^p_{z-1}], hnf, V_{z-1}, \mathcal{D}) \end{array}$$

Therefore, the next expression of the sequence is

$$\begin{array}{l}\overbrace{\mathcal{P}_{z-1} \cup pos(C'_{z-1})}^{\mathcal{P}_z}\\ \qquad\qquad\qquad \cup\ ds([C^2_{z-1}, \ldots, C^p_{z-1}], hnf, V_{z-1}, \mathcal{D})\\ \qquad\qquad\qquad \underbrace{\cup\ ds(C'_z : CS_z, \pi_{z-1}, V_{z-1}, \mathcal{D}) \cup \overline{ds}'_{z-1}}_{\overline{ds}_z}\end{array}$$

Now, we consider the call $DS(G, r_{z-1}, \pi'_{z-1}, V'_{z-1})$ in $\overline{DS}_{z-1}$ such that $r_{z-1} = ref(C'_{z-1})$, $(r_{z-1} \mapsto_{\mathcal{P}^{z-1}} e_{z-1}) \in G$, $\mathcal{P}^{z-1} = pos(C'_{z-1})$, $e_{z-1} = ctrl(C'_{z-1})$, $\pi'_{z-1} = \pi_{z-1}$, and $V'_{z-1} = V_{z-1}$. We denote by $\overline{DS}'_{z-1}$ the union of calls to DS in $\overline{DS}_{z-1}$ minus the above one.

On the one hand, since $C'_{z-1} \Longrightarrow^* C'_z$ is a successor subderivation, by Proposition 1, we have

$$(r_a \underset{r_b}{\mapsto}_{\mathcal{P}_a} ctrl(C'_{z-1})) \in G \quad \text{and} \quad (r_b \underset{r'}{\mapsto}_{\mathcal{P}_b} ctrl(C'_z)) \in G$$

with

$$r_a = ref(C'_{z-1}),\ \mathcal{P}_a = pos(C'_{z-1}),\ r_b = ref(C'_z),\ \mathcal{P}_b = pos(C'_z)$$

and r' a fresh reference. On the other hand, since $ctrl(C^1_{z-1}) = x$, we have two possibilities:

- If C^1_{z-1} is a lookup configuration (i.e., this is the first time that the value of x is demanded in the computation), since C^2_{z-1} is relevant, by Proposition 2, there exists

  $$(x \rightsquigarrow r_c) \in G \quad \text{and} \quad (r_c \underset{r''}{\mapsto}_{\mathcal{P}_c} ctrl(C^2_{z-1})) \in G$$

  with

  $$r_c = ref(C^1_{z-1}) = ref(C^2_{z-1}),\ \mathcal{P}_c = pos(C^2_{z-1}),\ r'' \text{ fresh}$$

  Hence, by definition of function DS, we have

  $$\begin{array}{rcl} DS(G, r_{z-1}, \pi'_{z-1}, V'_{z-1}) &=& pos(C'_{z-1})\\ && \cup\ DS(G, ref(C^2_{z-1}), hnf, V_{z-1})\\ && \cup\ DS(G, ref(C'_z), \pi_{z-1}, V_{z-1})\end{array}$$

  Therefore, the next expression of the sequence is now:

  $$\begin{array}{l}\overbrace{\mathcal{P}'_{z-1} \cup pos(C'_{z-1})}^{\mathcal{P}'_z}\\ \qquad\qquad\qquad \cup\ DS(G, ref(C^2_{z-1}), hnf, V_{z-1})\\ \qquad\qquad\qquad \underbrace{\cup\ DS(G, ref(C'_z), \pi_{z-1}, V_{z-1}) \cup \overline{DS}'_{z-1}}_{\overline{DS}_z}\end{array}$$

  and the claim follows since $\mathcal{P}'_{z-1} \supseteq \mathcal{P}_{z-1}$.

- Consider now that C^1_{z-1} is not a lookup configuration, i.e., the value of x has already been demanded in the computation. Hence, the heap contains a binding that maps x to a value. Therefore, in this case, we have that $p = 2$, $ctrl(C^2_{z-1})$ is the (already computed) value of x, and

$$\overbrace{\mathcal{P}_{z-1} \cup pos(C'_{z-1})}^{\mathcal{P}_z} \\ \cup\ ds([C^2_{z-1}], hnf, V_{z-1}, \mathcal{D}) \\ \underbrace{\cup\ ds(C'_z : CS_z, \pi_{z-1}, V_{z-1}, \mathcal{D}) \cup \overline{ds}'_{z-1}}_{\overline{ds}_z}$$

is the next expression of the sequence. In the graph, however, we have

$$(x \rightsquigarrow r_d) \in G$$

such that r_d is not labeled with $ctrl(C^2_{z-1})$ but points to a path in the graph that ends in a reference labeled with $ctrl(C^2_{z-1})$. This is due to the fact that all occurrences of x in the graph are shared (see rules varcons and varexp in Fig. 13) and, thus, x points to the full path representing the original evaluation of x (associated to the first time it was demanded). Hence, there must be a successor chain

$$(r_d \underset{q^1_d}{\mapsto}_{\mathcal{P}^1_d} e^1_d), (q^1_d \underset{q^2_d}{\mapsto}_{\mathcal{P}^2_d} e^2_d), \ldots, (q^{n-1}_d \mapsto_{\mathcal{P}^n_d} e^n_d) \in G$$

such that $ctrl(C^2_{z-1}) = e^n_d$ and $pos(C^2_{z-1}) = \mathcal{P}^n_d$ (i.e., the computed value of x must be the same every time its value is demanded). Therefore, in this case, we construct the following expression:

$$\overbrace{\mathcal{P}'_{z-1} \cup pos(C'_{z-1}) \cup \mathcal{P}^1_d \cup \ldots \cup \mathcal{P}^{n-1}_d}^{\mathcal{P}'_z} \\ \cup\ DS(G, q^{n-1}_d, hnf, V_{z-1}) \\ \underbrace{\cup\ DS(G, ref(C'_z), \pi_{z-1}, V_{z-1}) \cup \overline{DS}'_{z-1}}_{\overline{DS}_z}$$

and the claim follows since $\mathcal{P}'_{z-1} \supseteq \mathcal{P}_{z-1}$, $(q^{n-1}_d \mapsto_{\mathcal{P}^n_d} e^n_d) \in G$, $\mathcal{P}^n_d = pos(C^2_{z-1})$, and $e^n_d = ctrl(C^2_{z-1})$.

– $ctrl(C'_{z-1})$ is a variable or a constructor call. This case is not possible since, then, C'_{z-1} would not be relevant.

Let us now consider that CS_{z-1} is an empty list. In this case, we have:

$$\begin{array}{l} ds([C'_{z-1}], \pi_{z-1}, V_{z-1}, \mathcal{D}) \\ \qquad = pos(C'_{z-1}) \cup ds\_aux(C'_{z-1}, \pi_{z-1}, V_{z-1}, \mathcal{D}) \end{array}$$

By definition, this expression is reduced as follows:

$$\begin{array}{l} ds([C'_{z-1}], \pi_{z-1}, V_{z-1}, \mathcal{D}) \\ \qquad = pos(C'_{z-1}) \cup ds(\overline{C}^1_x : CS^1_x, \pi^1_x, V_{z-1}, \mathcal{D}) \\ \qquad \qquad \qquad \quad \cup \ldots \\ \qquad \qquad \qquad \quad \cup\ ds(\overline{C}^v_x : CS^v_x, \pi^v_x, V_{z-1}, \mathcal{D}) \end{array}$$

so that C^i_x is the first (from C'_{z-1}) lookup configuration that fulfills the following conditions: its control stores a variable x^i in V_{z-1} and there exists a pair (x^i, π^i_x) in $ctrl(C'_{z-1}) \sqcap \pi_{z-1}$ for $i = 1, \ldots, v$. Here, we consider that $\overline{C}^i_x$ denotes the first relevant configuration from C^i_x and $\overline{C}^i_x : CS^v_x$ contains the list of configurations in the complete subderivation for $\overline{C}^i_x$ in $\mathcal{D}$, for all $i = 1, \ldots, v$.

By the inductive hypothesis, there exists a call to function DS of the form $DS(G, r_{z-1}, \pi'_{z-1}, V'_{z-1})$ such that $r_{z-1} = ref(C'_{z-1})$, $(r_{z-1} \mapsto_{\mathcal{P}^{z-1}} e_{z-1}) \in G$, $\mathcal{P}^{z-1} = pos(C'_{z-1})$, $e_{z-1} = ctrl(C'_{z-1})$, $\pi'_{z-1} = \pi_{z-1}$, and $V'_{z-1} = V_{z-1}$. By definition of function DS, we have:

$$\begin{array}{rcl} DS(G, r_{z-1}, \pi'_{z-1}, V'_{z-1}) & = & pos(C'_{z-1}) \\ & & \cup\ DS\_AUX(G, ctrl(C'_{z-1}), r_{z-1}, \pi_{z-1}, V_{z-1}) \end{array}$$

Now, by Proposition 2, we have

$$\begin{array}{rcl} DS\_AUX(G, ctrl(C'_{z-1}), r_{z-1}, \pi_{z-1}, V_{z-1}) & \subseteq & DS(G, ref(\overline{C}^1_x), \pi^1_x, V_{z-1}) \\ & & \cup \ldots \\ & & \cup\ DS(G, ref(\overline{C}^v_x), \pi^v_x, V_{z-1}) \end{array}$$

and the claim follows. Observe that DS_AUX may in principle compute additional positions since function ds_aux only considers *lookup* configurations and this restriction has no counterpart in function DS_AUX.

(Minimality) Analogously to the previous case, we prove a more general claim. First, we say that a call, $DS(G, r, \pi, V)$, is *collecting* if it is the first call to DS with reference r in the considered computation.[17] Let us consider a (possibly incomplete) computation for DS as follows:

$$\mathcal{P}'_0 \cup \overline{DS}_0 = \mathcal{P}'_1 \cup \overline{DS}_1 = \ldots = \mathcal{P}'_z \cup \overline{DS}_z$$

where

[17] The notion of *collecting* call in the evaluation of DS is essentially the counterpart of the notion of *relevant* call in the evaluation of ds.

- each new equality is obtained from the previous one by reducing a single call to DS one or more times until it reduces to a (possibly empty) union of collecting calls,
- $\mathcal{P}'_k$ is the already computed set of positions, and
- $\overline{DS}_k$ is the (possibly empty) union of collecting calls that have not been evaluated yet

for all $k = 0, \ldots, z$.

Now, we prove that, for any $z \geq 0$, there is a (possibly incomplete) sequence of calls to ds:

$$\mathcal{P}_0 \cup \overline{ds}_0 = \mathcal{P}_1 \cup \overline{ds}_1 = \ldots = \mathcal{P}_z \cup \overline{ds}_z$$

such that $\mathcal{P}_z \supseteq \mathcal{P}'_z$ and, for every call $DS(G, r_z, \pi'_z, V'_z)$ in $\overline{DS}_z$ with $(r_z \mapsto_{\mathcal{P}^z} e_z) \in G$, there exists a call $ds([C'_z], \pi_z, V_z, \mathcal{D})$ in $\overline{ds}_z$ with $ref(C'_z) = r_z$, $pos(C'_z) = \mathcal{P}^z$, $ctrl(C'_z) = e_z$, $\pi_z = \pi'_z$, and $V_z = V'_z$.

The proof is similar to the proof of completeness since the needed results (Propositions 1 and 2, Lemma 2, and Theorem 3) hold in both directions. Actually, the positions gathered by DS and ds are always the same except in one case: when considering that the reference r_{z-1} is labeled with a case expression $case\ x\ of \ldots$ and $x \in V_{z-1}$. In the following, we only show the proof of this case.

In this case, we consider a collecting call $DS(G, r_{z-1}, \pi'_{z-1}, V'_{z-1})$ with $(r_{z-1} \underset{r_z}{\mapsto}_{\mathcal{P}^{z-1}} case\ x\ of \ldots) \in G$ and $x \in V'_{z-1}$. We denote by $\overline{DS}'_{z-1}$ the union of calls to DS in $\overline{DS}_{z-1}$ minus the above one.

By the inductive hypothesis, we have that $\mathcal{P}_{z-1} \supseteq \mathcal{P}'_{z-1}$ and there exists a call $ds(C'_{z-1} : CS_{z-1}, \pi_{z-1}, C_{z-1}, \mathcal{D})$ in $\overline{ds}_{z-1}$ with $ref(C'_{z-1}) = r_{z-1}$, $ctrl(C'_{z-1}) = case\ x\ of \ldots$, $pos(C'_{z-1}) = \mathcal{P}^{z-1}$, $\pi_{z-1} = \pi'_{z-1}$, and $V_{z-1} = V'_{z-1}$. By definition of function DS, we have

$$\begin{array}{rl} DS(G, r_{z-1}, \pi'_{z-1}, V'_{z-1}) = \mathcal{P}^{z-1} & \cup\ DS(G, r_x, hnf, V'_{z-1}) \\ & \cup\ DS(G, r_z, \pi'_{z-1}, V'_{z-1}) \end{array}$$

where $(x \rightsquigarrow r_x) \in G$. Therefore, the next expression in the reduction sequence is

$$\overbrace{\mathcal{P}'_{z-1} \cup \mathcal{P}^{z-1}}^{\mathcal{P}'_z} \cup \overbrace{DS(G, r_x, hnf, V'_{z-1}) \cup DS(G, r_z, \pi'_{z-1}, V'_{z-1}) \cup \overline{DS}'_{z-1}}^{\overline{DS}_z}$$

Since r_z is the successor of r_{z-1}, we have by Proposition 1 that there exists a successor derivation of the form $C'_{z-1} \Longrightarrow C_x \Longrightarrow^* C'_x \Longrightarrow C'_z$ in $\mathcal{D}$ where $C_x \Longrightarrow^* C'_x$ is the complete subderivation for C_x.

Let $(r_x \mapsto_{\mathcal{P}_x} e_x) \in G$. By Proposition 2, there exists a lookup configuration C; assume that C' is the first relevant configuration from

C, where $ctrl(C') = e_x$, and $pos(C') = \mathcal{P}_x$. Now, we distinguish two cases:

- Assume first that $C'_{z-1} \Longrightarrow C$ (i.e., the lookup configuration C is C_x). In this case, by definition of function ds, we construct the expression

  $$\begin{array}{ll} \overbrace{\mathcal{P}_{z-1} \cup pos(C'_{z-1})}^{\mathcal{P}_z} & \cup\ ds([C', \ldots, C'_x], hnf, V_{z-1}, \mathcal{D}) \\ & \underbrace{\cup\ ds(C'_z : CS_z, \pi_{z-1}, V_{z-1}, \mathcal{D}) \cup \overline{ds}'_{z-1}}_{\overline{ds}_z} \end{array}$$

  and the claim follows by the induction hypothesis analogously to the corresponding case in the proof of completeness.

- Assume now that the next configuration, C_x, is not a lookup configuration in $\mathcal{D}$. In this case, we have $C_x \Longrightarrow C'_x$, C'_x stores in the control the value of x, and the call $DS(G, r_x, hnf, V'_{z-1})$ is not a collecting call. Hence, there must be a successor chain in the graph:

  $$(r_x \underset{q_x^1}{\mapsto}_{\mathcal{P}_x^1} e_x^1), (q_x^1 \underset{q_x^2}{\mapsto}_{\mathcal{P}_x^2} e_x^2), \ldots, (q_x^{n-1} \mapsto_{\mathcal{P}_x^n} e_x^n) \in G$$

  such that $ctrl(C'_x) = e_x^n$ and $pos(C'_x) = \mathcal{P}_x^n$ (i.e., the computed value of x is the same in both cases). Then, we have

  $$\begin{array}{rcl} DS(G, r_x, hnf, V'_{z-1}) & = & \mathcal{P}_x^1 \cup DS(G, q_x^1, hnf, V'_{z-1}) \\ & = & \ldots \\ & = & \mathcal{P}_x^1 \cup \ldots \cup \mathcal{P}_x^n \end{array}$$

  such that $\mathcal{P}_x^1 \cup \ldots \cup \mathcal{P}_x^{n-1} \subseteq \mathcal{P}_{z-1}$ because the associated calls are not collecting. Therefore, by definition of function ds, we construct the expression

  $$\begin{array}{l} \overbrace{\mathcal{P}_{z-1} \cup pos(C'_{z-1}) \cup \mathcal{P}_x^1 \cup \ldots \cup \mathcal{P}_x^{n-1}}^{\mathcal{P}_z} \\ \qquad\qquad \cup\ ds([C'_x], hnf, V_{z-1}, \mathcal{D}) \\ \qquad\qquad\quad \underbrace{\cup\ ds(C'_z : CS_z, \pi_{z-1}, V_{z-1}, \mathcal{D}) \cup \overline{ds}'_{z-1}}_{\overline{ds}_z} \end{array}$$

  and the claim follows by the inductive hypothesis.