Chapter 8

EDGAR: Explicit Demand Graph Reduction

I have demonstrated how slivers and synchronized lazy aggregates can be realized using the concurrency, synchronization, and non-strictness features of OPERA. But since I have only described these features in an informal way, there are many potential ambiguities and unexplained subtleties concerning how they work. In particular, the details of how a rendezvous occurs at a synchron are far from clear.

In this chapter, I present a semantic model that precisely specifies the meaning of OPERA expressions. The model is based on *E*xplicit *D*emand *G*r*A*aph *R*eduction (EDGAR), a framework that I have developed for describing how computations unfold over time. Whereas many semantic frameworks focus on the *value* produced by a computation, EDGAR is designed to emphasize the *way* in which the value is computed. EDGAR represents a computation as a sequence of graphs, each one of which represents the control state and accessible data at a particular point in the computation. EDGAR shares many similarities with other models of computation (e.g., graph reduction, dataflow models, concurrency models, structured operational semantics). The key feature that distinguishes it from most other semantic models is its explicit representation of the flow of demand through a computation. This feature simplifies the explanation of OPERA's features for fine-grained operational control.

I present the semantics of OPERA in three parts. First, I explain the basics of the

EDGAR model. Then I focus on the EDGAR rules that implement OPERA's concurrency, synchronization, and non-strictness features. Finally, I describe how an OPERA expression can be evaluated by translating it into an initial EDGAR graph, turning the crank of the graph reduction process, and translating the final graph into an OPERA result. I conclude the chapter by relating EDGAR to other graph-based semantic frameworks.

8.1 The Basics of EDGAR

At the very highest level, EDGAR follows the basic recipe for an operational semantics framework [Plo81]. In an operational semantics, program execution states are represented by some sort of structured *configuration*, and there are *transition rules* that specify how the computation can step from one configuration to the next. Program execution is modelled by mapping the program to an *initial configuration* and iteratively applying transition rules until a suitable *final configuration* (one representing an answer) is obtained. The final configuration is then mapped to an appropriate answer domain.

8.1.1 Snapshots

In EDGAR, each configuration is called a *snapshot* because it represents a single moment in the dynamic evolution of a process. A snapshot is a graph consisting of interconnected program components. Figure 8.1 is a visual depiction of a sample snapshot that we will refer to in the ensuing discussion.

Each program component, called a *node*, can be viewed as a computational device that responds to a demand for a value by computing that value. Every node is has a *label* that indicates the behavior of the node. Each node posesses a set of labelled *input ports* that specify the arguments of the node and a set of labelled *output ports* that specify the results of the node. The number of input ports and output ports is dictated by the label of the node. Typically, a node has several input ports and one output port. Every node must have at least one port, but one of the input port set or the output port set may be empty.[1]

[1] In all of our examples, a node will have only 0 or 1 output ports, but in general it can have any number.

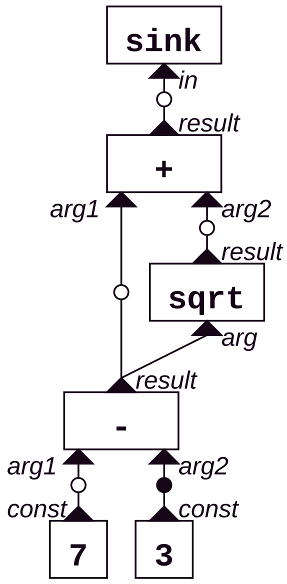


Figure 8.1: A simple snapshot.

In Figure 8.1, nodes are depicted as labelled rectangles, and ports are depicted as triangles attached to the nodes. An input port points into its node; an output port points away from its node. Ports are labelled with names that serve to distinguish them, as well as indicate their purpose.

Nodes are wired together much like the components of an electrical circuit. Each connection is indicated by a directed *wire* from an output port of the *source node* to an input port of the *target node*. Wires appear as lines in Figure 8.1. Technically, a wire connects a *source port* to a *target port*, but when there is no ambiguity it is convenient to refer to a wire as connecting two *nodes*. The set of wires entering the input ports of a node are its *input wires*, while those that leave its output ports are its *output wires*.

Intuitively, a wire is used for a two-step communication protocol between its source and target ports: the target port can request the value from its source port, and the source port can respond to the request by returning a value. Every wire has a *state* attribute that indicates the status of this communication protocol. There are three possible wire states:

1. *inactive*: No request has yet been made on the wire.

2. *requested*: A request has been made on the wire, but no value has been returned.

3. *returned*: A request has been made on the wire, and a value has been returned.

In Figure 8.1, an unannotated line is an inactive wire; a line with an empty circle is requested; and a line with a filled circle is returned. Wire states are the chief characteristic of EDGAR that distinguish it from traditional forms of graph reduction. In particular, the existence of a *requested* state is what makes EDGAR an "explicit demand" model.

The value returned by a returned wire is the source node of the wire. In EDGAR, only a subset of the nodes may appear at the source of a returned wire. These nodes are called *value nodes*. Examples of value nodes include constants, data structures, and procedures. The `+`, `-`, and `sqrt` nodes in Figure 8.1 examples of nodes that are not value nodes; they represent the applications of procedures but are not procedures themselves.

We shall see later that the state of a wire can change when a computation steps from one snapshot to another. As implied by the description of the wire states, the communication protocol on a wire over a sequence of snapshots is very limited:

- At most one request can ever be made on a wire.

- No value can be returned to a wire until a request has been made on that wire.

- At most one value can ever be returned to a wire.[2]

A wire, then, is a one-shot communication "fuse" that can be used for transmiting a single request and a single value before it is "used up". These restrictions on wires distinguish EDGAR from dataflow graph models, in which wires carry a stream of value tokens.

In a legal snapshot, every input port must be connected to exactly one wire, but an output port may be connected to any number of wires, including zero. Wire fan-out at an output port allows the result computed at one node to be shared as an input to other nodes. Wire sharing means that nodes can be arranged in directed acyclic graphs (DAGs) or even exhibit cycles (see Figure 7.4 on page 323 for an example of a cyclic graph).

[2]This is true only for the functional subset of EDGAR. In the presence of nodes that model side effects, the value represented by a wire may change over time. See [Tur94] for details.

The snapshot depicted in Figure 8.1 is an intermediate configuration in a simple numerical computation. The computation specifies the calculation of

$$(7-3)+\sqrt{(7-3)}$$

where the sharing of the `-` node indicates that the difference between 7 and 3 is to be calculated only once. The nodes labelled `7` and `3` are *numeric nodes* whose single output ports stand for the designated number. The nodes labelled `+`, `-`, and `sqrt` are *primop nodes*, whose output ports stand for the result of performing the specified operation on the argument values specified by the input wires.

The node labelled `sink` is a *sink node*, which serves as the primitive source of demand in a computation. Intuitively, a sink node "tugs on" (emits a request to) the wire connected to its single input port in order to initiate a computation. The computation initiated by a sink node successfully terminates when the wire into the sink node enters the returned state. Every computation must have at least one sink node, a distinguished node known as the *top-level sink node*. The top-level sink node loosely corresponds to the `eval` in a traditional Lisp `read-eval-print` loop.

It is worth emphasizing that a snapshot is just a particular moment in the evolution of dynamic process. No mention yet has been made to how a process steps from one snapshot to the next. Nevertheless, it is possible to deduce certain aspects of the history of a process from the wire states in a single snapshot. In Figure 8.1, for example, the wire annotations indicate that demand has propagated from the `sink` node through the `+` node to the `sqrt` node (where demand stops) and to the `-` node (through which demand has propagated to the `7` and `3` nodes). Furthermore, the `3` node has returned to the `-` node in response to the demand.

We shall soon see how the allowable transitions between snapshots are constrained in order to guarantee that a sensible history can be deduced from the wire states of any particular shapshot.

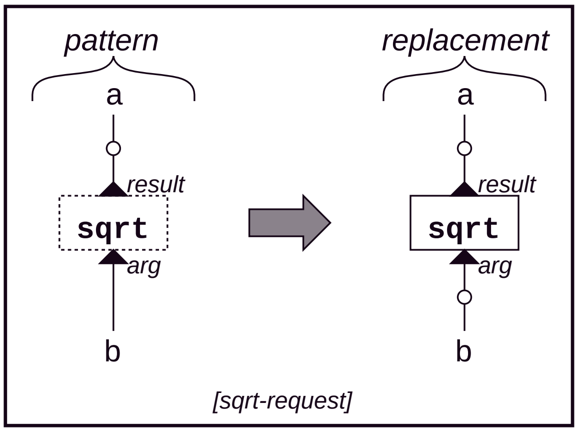


Figure 8.2: A rule that propagates demand through a `sqrt` node.

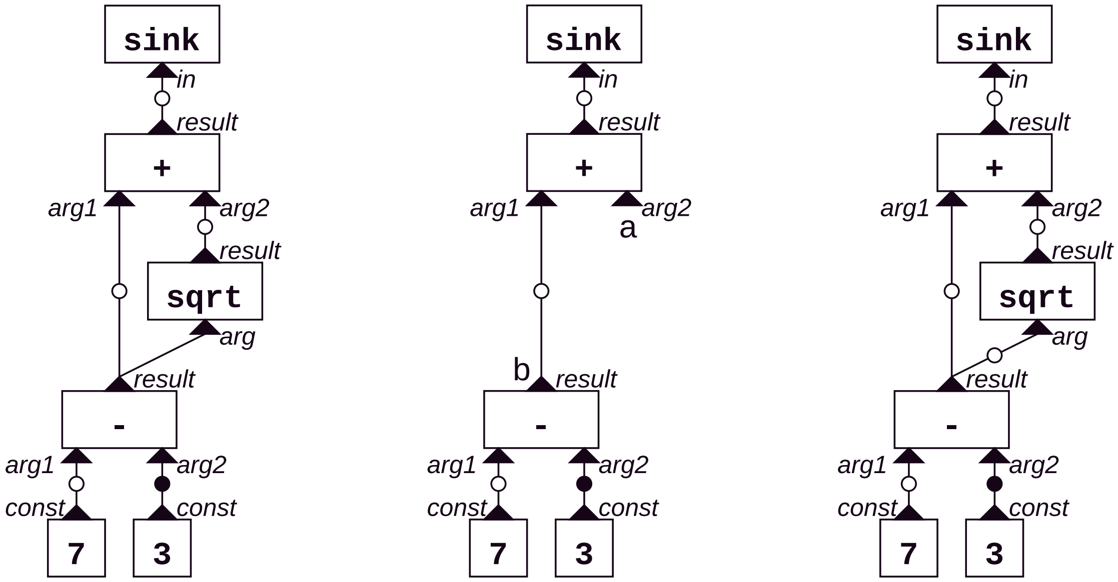
8.1.2 Rewrite Rules

The dynamic behavior of a computation is specified by a set of *rewrite rules* that transform one snapshot into another. It is the rewrite rules that dictate the dynamic behavior of nodes and the flow of demand and values through a sequence of snapshots.

A rewrite rule has two parts, a *pattern* and a *replacement*. The pattern is a partial graph structure in which wires may be attached to *pattern variables* instead of ports. A pattern is said to *match* a snapshot if it can be embedded in the snapshot. The replacement specifies how the snapshot structure matched by the pattern should be replaced in order to construct a new snapshot.

Figure 8.2 shows a simple rewrite rule for propagating demand through a `sqrt` node. The rule says that if a snapshot contains a `sqrt` node having at least one output wire in the requested state and its input wire in the inactive state, then the snapshot can be transformed into a new graph with the same structure as the original except that the input wire is in the requested state. The rule has been labelled with a name ([*sqrt-request*]) so that we can conveniently refer to it later.

A rewrite rule that matches a snapshot can be *applied* to the snapshot to yield a new snapshot. Removing the structure specified by a pattern from a snapshot that it matches leaves the *context* of the match. The new snapshot is constructed from the context by



(a) A snapshot matched by the [*sqrt-request*] rule.

(b) The context determined by the match.

(c) The result of the rule application.

Figure 8.3: Steps in an application of the [*sqrt-request*] rule.

filling the hole left by the deleted pattern with the structure specified by the replacement. The part of the initial snapshot that is not directly matched by the pattern is carried over unchanged into the final snapshot.

For example, the pattern of the [*sqrt-request*] rule matches the snapshot in Figure 8.3(**a**). The context determined by the match appears in Figure 8.3(**b**). Context ports that match a pattern variable in the pattern are labelled to indicate the match. The structure specified by the rule replacement is "glued" onto the context to form the snapshot that is the result of the rule application (Figure 8.3(**c**)). As shown in Figure 8.4, a rule application can be depicted by showing both the original snapshot and the resulting snapshot. (Henceforth, we will omit names on ports by assuming that input and output ports on a node appear in a canonical left-to-right order.)

Each rule pattern contains a distinguished node-matching element called its *locus*. In figure for the [*sqrt-request*] rule, the locus of the rule is the `sqrt` node, which is indicated by a dotted outline. The locus provides a convenient way to specify a particular node in a snapshot at which the match occurs. In a successful match, the node matched by the locus of a rule is said to be *enabled* by the rule. Whenever a snapshot contains a node enabled

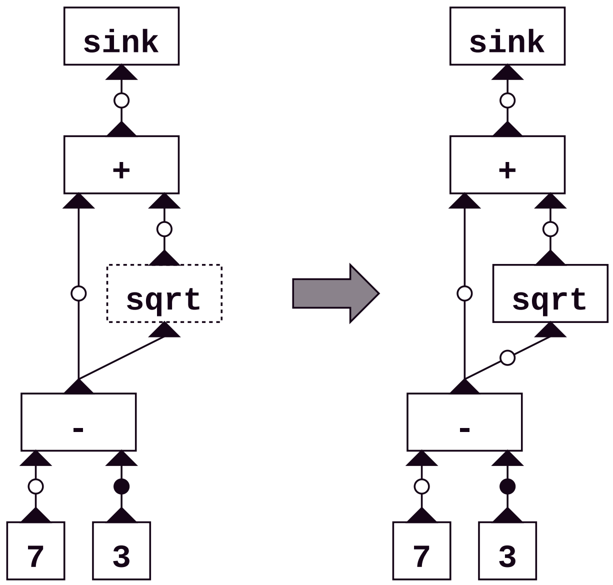


Figure 8.4: An application of the [*sqrt-demand*] rule. Port labels have been dropped here (and in following figures) because they are unambiguously determined by position.

by a rule, the rule can be applied at that node to yield a new snapshot. In Figure 8.4, the dotted outline of the `sqrt` node indicates that it is enabled by the [*sqrt-request*] rule.

Figure 8.5 shows all the EDGAR rules necessary for handling the kinds of nodes we have seen so far. The [*sink-request*] rule shows that a sink acts as a source of demand in the system. Although other nodes may propagate demand, sinks are EDGAR's only source of demand.

The [*constant-return*] rule is the only rule for handling constants (e.g., numbers, booleans, primitive procedures, etc.). Technically, [*constant-return*] is a *rule schema*[3] that stands for an infinite collection of rules; we obtain a particular rule by instantiating the metavariable C with any constant. The [*constant-return*] rule schema dictates that a constant responds to a request simply by returning. The filled-in circle on the output wire of a returned constant indicates that the wire has entered a "returned" state. Unlike certain representations of dataflow, the filled-in circle is just a state indicator and not a value-bearing token; the

[3]I do not emphasize the distinction between rules and rule schemas in the remainder of this discussion, and will sometimes loosely use "rule" where "rule schema" would be more appropriate.

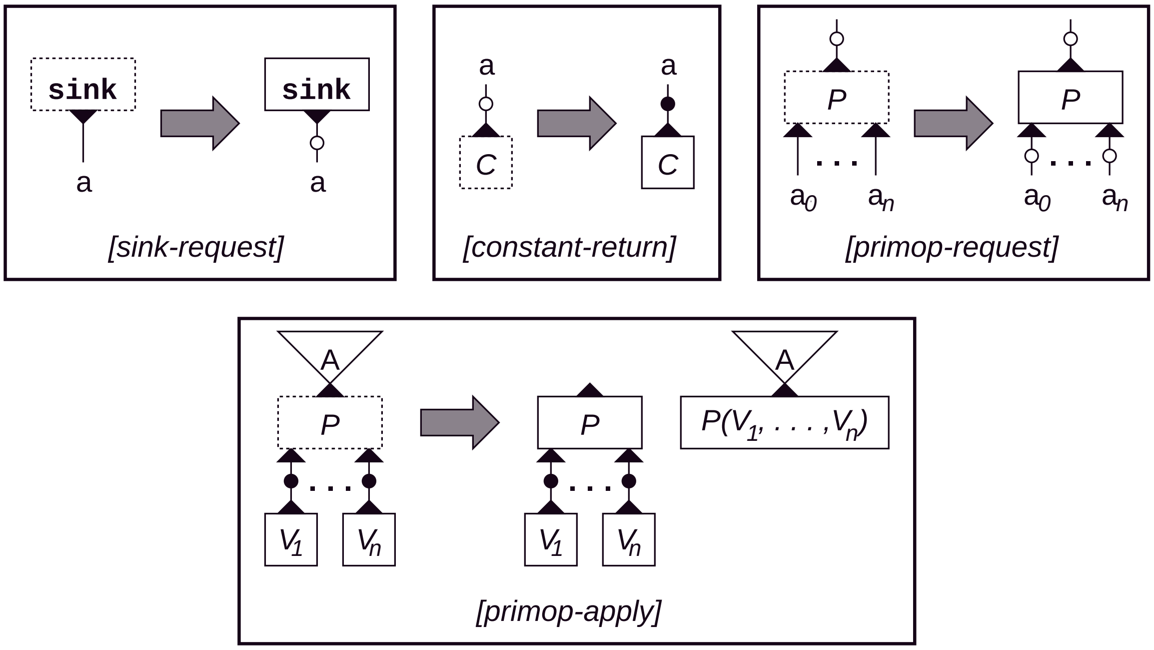


Figure 8.5:

returned value is simply the node at the source of a wire in the returned state.

There are two rules for handling primop nodes. The [*primop-request*] rule specifies that demand for the result of a n-argument primitive operation P propagates *in parallel* to all n arguments of the operation. This rule differs from the operand evalution semantics of most sequential languages, which typically require that the arguments are evaluated in a specified order, or at least some sequential order. The [*primop-request*] rule is one of the means by which concurrency is achieved in EDGAR.

The [*primop-apply*] rule specifies how the application of a primitive operator to values returns a result. An n-argument primop node P is only enabled by such a rule if each of its n input wires is in the returned state (i.e., a primitive application is strict). Each metavariable V_i can be instantiated with any value node that is an acceptable ith argument to P. The node labelled $P(V_1, \ldots, V_n)$ is the result of applying the primitive operator to the argument values. For example, applying a `+` primop node to a `1` node and `2` node would yield a `3` node.

The triangle labelled A at the output port of the primop node is the depiction of a

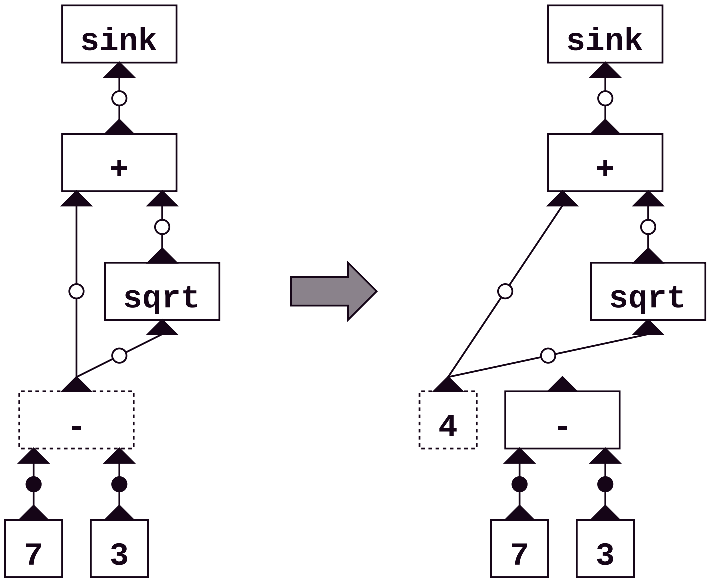


Figure 8.6: Application of a [*primop-apply*] rule for binary subtraction. Note that the `4` node is enabled in the resulting snapshot.

wire set pattern variable. Such a pattern variable matches the *entire* set of wires leaving a port, provided that the set is nonempty and contains at least one requested wire. The [*primop-apply*] rule says that all of the wires originally attached to the P node are "moved" to the output port of the node that is the result of the primitive application. Figure 8.6 shows a sample application of the rule obtained by instantiating the [*primop-apply*] rule with a binary subtraction operator.

8.1.3 Garbage Collection

As indicated by Figure 8.6, rule applications can sometimes result in nodes that are *inaccessible* from the top-level sink node. A node is inaccessible from a sink if there is no directed path of wires from the output port of the node to the input port of the root. In the binary subtraction example, the subgraph rooted at the `-` node is a disconnected component of the snapshot graph, all of whose nodes are inaccessible from the sink.

In order to be able to accurately model the space required by a computation, it is necessary to have some means for removing the inaccessible nodes from a snapshot. This process is called *garbage collection*. We will assume the existence of a garbage collection function, *gc*, that maps snapshots to snapshots by removing any nodes that are not accessible from the sink nodes of a snapshot.[4]

In the case of the [*primop-apply*] rule, the inaccessible nodes in the result are present because the rule specifies that the primop nodes and argument nodes should be copied from the initial snapshot to the final snapshot. Why not change the [*primop-apply*] rule avoid copying these nodes? The problem with this approach is that incorporating garbage collection into the rewrite rules leads to a number of complexities. For example, while it is always safe for the [*primop-apply*] rule to delete the primop node (since it is guaranteed to become inaccessible when all of its output wires are moved), the argument nodes cannot always be deleted since they may be accessible from some other part of the snapshot. Rather than trying to express the garbage collection conditions within the rewrite rules, it is simpler to handle garbage collection via a separate mechanism.

8.1.4 Transitions

Whenever a rule allows snapshot S to be rewritten into S', we say that there is a *transition* between S and $gc(S')$, written $S \Rightarrow gc(S')$. A transition combines a rule application and garbage collection into a single step of the computation. $\Rightarrow$ is a binary relation between snapshots; $\stackrel{*}{\Rightarrow}$ is the reflexive, transitive closure of this relation. When $S_1 \stackrel{*}{\Rightarrow} S_2$, we say that there is a *transition path* from S_1 to S_2.

In general, it may be possible to make several different transitions from a given snapshot. It is often the case that several nodes in a snapshot are enabled by one or more rules. For example, Figure 8.7 shows the two transitions that are possible from the snapshot in Figure 8.1.

Even though more than one rule application may be possible for a given snapshot, transitions are defined so that there is only one rule application per transition. This restriction corresponds to a computational model in which there is a single "processor" that can only apply one rule in each step of the computation. It is possible to imagine alternate approaches

[4]The sink nodes are so-called *roots* of the garbage collection. Eagon nodes are also garbage collection roots.

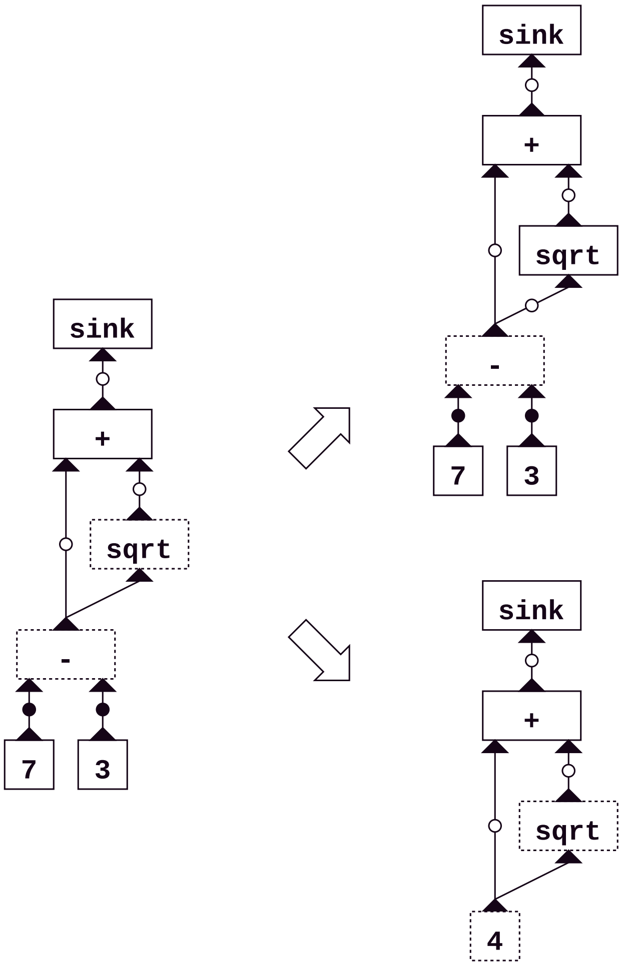


Figure 8.7: The two possible transitions for a snapshot in which two nodes are enabled.

in which more than one rule application would be allowed per transition. For example, a multi-processor model might allow several rule applications to occur in parallel within a single transition. However, allowing multiple applications introduces extra complexities (e.g. undesirable interactions between rules whose patterns might overlap when matching a graph). Since allowing only a single application per transition is sufficient for our purposes, we henceforth ignore the possibility of multiple rule applications per transition.

8.1.5 Computation

Here we formalize the notion of an EDGAR computation. First, a few definitions:

- An *initial snapshot* is a snapshot rooted at a sink node in which all wires are inactive.
- A *final snapshot* is a snapshot in which no rules are enabled.
- A *trace* is any sequence of snapshots such that there is a transition from each element of the sequence to the next.
- The *space* required by a snapshot is the sum of the number of nodes and the number of wires in the snapshot.

A *terminating computation* is a trace from an initial snapshot to a final snapshot. The *time* required by a terminating computation is the number of transitions made in the trace (i.e., one less than the length of the trace). The *space* required by a terminating computation is the maximum of the space of the snapshots in the computation.

For example, Figure 8.8 depicts a sample terminating computation as a "movie" of numbered snapshot "frames". The time of the sample computation is 13 transitions, and the space is 12 units (the first 6 frames require 6 nodes + 6 wires).

A *non-terminating computation* is an infinite trace beginning with an initial snapshot. The *time* required by a non-terminating computation is undefined. The *space* required by a non-terminating computation is the maximum of the space of the snapshots in the computation *if* this maximum is defined; otherwise the space is undefined.

A computation has an *outcome* that describes its fate. The outcome of a non-terminating computation is *bottom*. The outcome of a terminating computation is determined by its

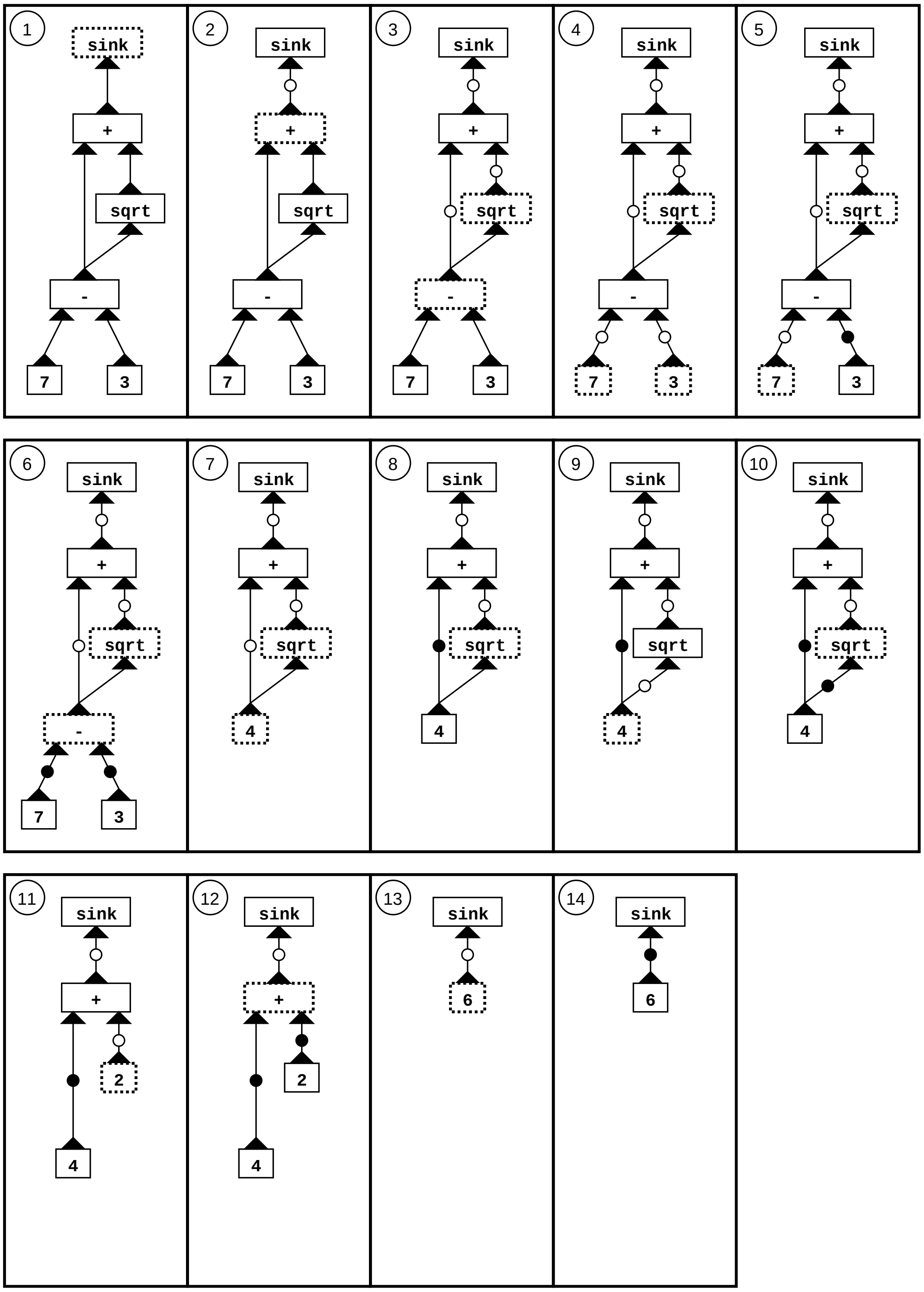


Figure 8.8: Depiction of a computation as a sequence of snapshots.

final snapshot. If the input wire to the root sink node is returned, then the outcome is a *result* whose value is the graph structure at the source of the wire. For example, the outcome of the computation depicted in Figure 8.8 is a result whose value is the constant node `6`. If the input wire to the root sink is not returned, then the outcome is *deadlock*. Deadlock indicates a computation that is "stuck" in a snapshot in which no rules are applicable. Figure 8.9 shows two final snapshots of a deadlocked computation. In **(a)**, the snapshot corresponds to a type error (`sqrt` cannot be applied to a boolean); in **(b)**, the snapshot is caught in an unresolvable dependency.

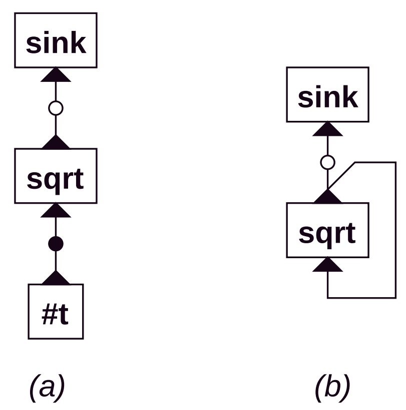
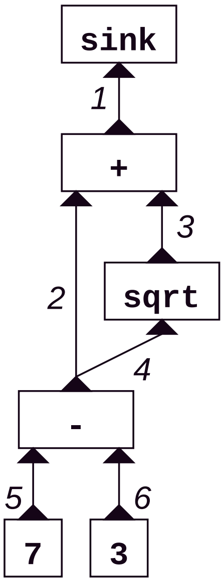


Figure 8.9: Two examples of deadlocked final snapshots.

8.1.6 Behavior

The *behavior* of an initial snapshot is the set of all computations that begin with that snapshot. A behavior often contains numerous computations due to the fact that several transitions may be possible from a given snapshot. The *time* and *space* required by of a behavior can be found by maximizing over the spaces and times required by the component computations (when these quantities are defined). The *outcomes* of a behavior is the set of outcomes for the component computations. Due to the fact that EDGAR supports side effects (data mutation and I/O), the nondeterminism of transitions can lead to multiple outcomes.

Consider the initial snapshot $S_{initial}$ used in Figure 8.8:



The wires in $S_{initial}$ have been labelled so that we can refer to them by number. The behavior of $S_{initial}$ is depicted in Figure 8.10 as a directed graph in which each vertex stands for a snapshot and each directed edge stands for a transition. Each vertex is a triple of numeric sets indicating the states of the wires labelled by the numbers. The triple is interpreted as

$$\langle inactive\ wires, requested\ wires, returned\ wires \rangle$$

A number can be a member of at most one of the three sets; if it is a member of none, then it is not present in the snapshot. For the simple case of $S_{initial}$, a triple uniquely identifies a snapshot in one of $S_{initial}$'s computations. Every path through the graph from $S_{initial}$ to S_{final} represents one of the computations in the behavior of $S_{initial}$. The shaded crosses are isomorphic subgraphs that only differ in whether or not wire 4 is inactive (left cross) or requested (right cross).

8.1.7 Global State

The above definition of behavior assumes that the set of computations emanating from an initial snapshot depends only on the snapshot. Indeed, in the absence of I/O operations or other operations on global state, each initial snapshot identifies a unique behavior. But

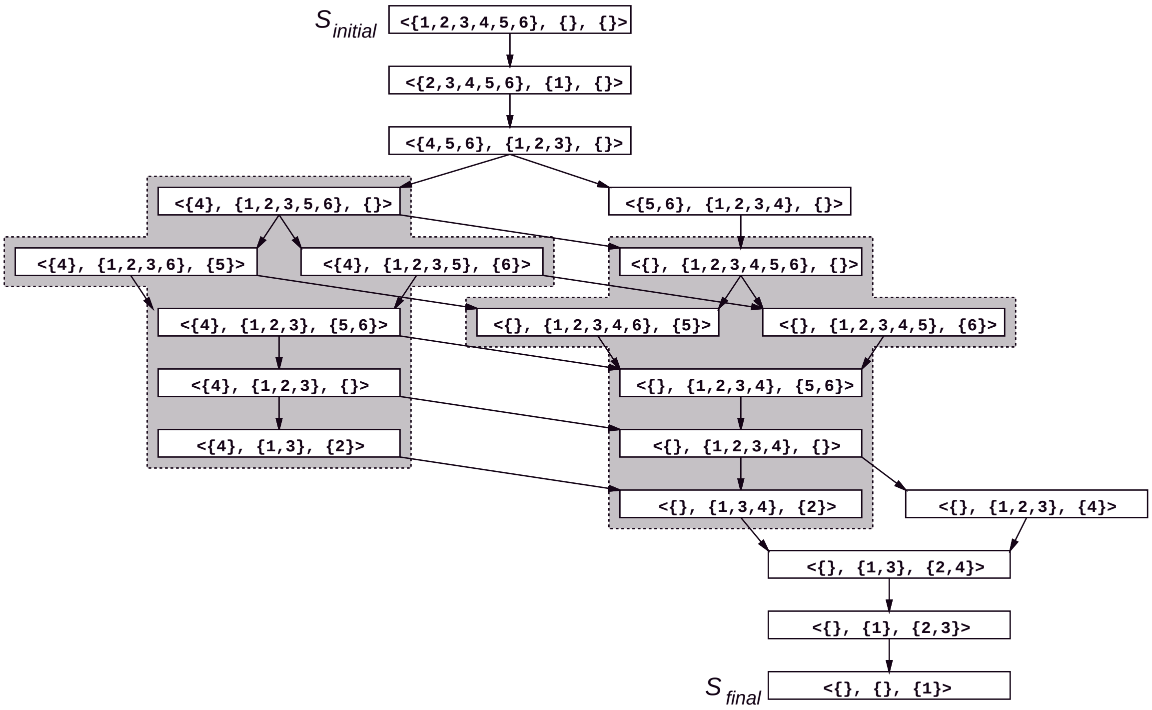


Figure 8.10: Depiction of the behavior of a simple initial snapshot.

in the presence of such operations, the behavior of an initial snapshot may depend on the global state, and individual computations may modify the global state.

There are many approaches to handling global state in the EDGAR framework. Here are some possibilities:

1. Assume that all global state is explicitly encoded in the initial snapshot and can be extracted from the final snapshot.

2. Change the definition of computation so that it is a triple of a trace, an initial state, and a final state such that the trace produces the final state from the initial one. Additionally extend the notion of outcome to include the final state. Behavior would still be defined as a set of computations.

3. Keep the definition of a computation the same, but change the definition of behavior to be a function from initial states to sets of computation/final-state pairs.

I will not pick a particular method, but will simply assume that global state is handled appropriately in situations where it is an issue.

8.2 The Details of EDGAR

Here we detail only those EDGAR rules that are essential for expressing the concurrency, synchronization, and non-strictness features of OPERA. The complete collection of EDGAR rules is described in [Tur94].

8.2.1 Procedures

The handling of procedures and procedure calls is shown in Figure 8.11. A `pcall` node is a general procedure application node. The [*pcall-request*] rule propagates demand for a `pcall` node to all of its input wires. Together, the [*primop-request*] rule (from the previous section) and the [*pcall-request*] rule implement a concurrent strategy for argument evaluation.

Nodes labelled `proc` represent procedures. Each `proc` node contains a *template*, a partial graph structure that specifies the body of the procedure. The template has an output port that represents the result of the body, and a number of input ports that represent the

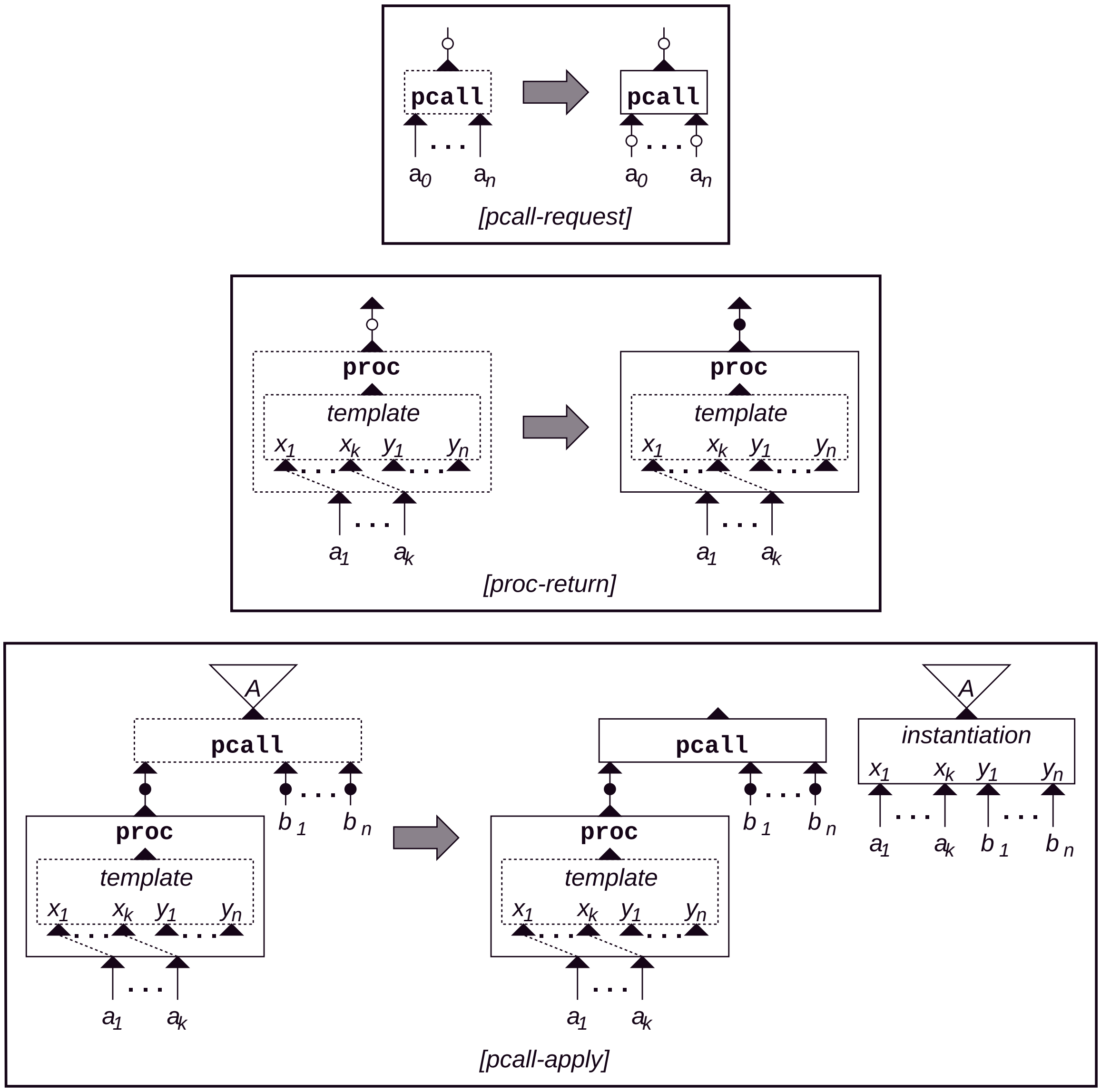


Figure 8.11: EDGAR rules for handling procedures.

values on which the body depends. The k input ports x_1, ..., x_k stand for the values of free variables used in the body, while the n input ports y_1, ..., y_n stand for the explicit arguments of the procedure. The `proc` node itself has k input ports that are attached to the values of the free variables (a_1, ..., a_k). These inputs correspond to the environment of a closure in the traditional environment model of procedures [ASS85]. The dotted lines connected the first k input ports on the template to the input ports of the `proc` node are intended to suggest that the template inputs will be connected to the `proc` inputs upon application of the procedure.

The [*proc-return*] rule simply says that a `proc` node is a value. The [*pcall-apply*] rule specifies the details of procedure application. The rule is not applicable until all the input wires are returned, which indicates that procedure calls are strict. The rule creates a copy of the template, called the *instantiation*, and connects wires to its ports as follows:

- The first k input wires of the instantiation are connected to the sources of the k input wires of the `proc` node. This supplies the values of the free variables as implicit arguments to the instantiation.
- The last n input wires of the instantiation are connected to the sources of the n operand wires of the `pcall` node. This supplies the values of the explicit arguments to the instantiation.
- All of the output wires of the `pcall` node are rerouted to the output port of the instantiation. This means that the value of the instantiation will be "returned" as the value of the `pcall` node.

8.2.2 Synchrons

I present two versions of the synchron rules. The simpler version, which ignores the complexities of the `precede!` operator, is shown in in Figure 8.12. The [*synchron-return*] rule treats `synchron` nodes as self-evaluating values. The [*simultaneous*] rule is a unification rule that ensures that all references to two unified synchrons point to the same object.

The [*rendezvous*] rule is the cornerstone of the OPERA semantics. In fact, the whole EDGAR system was design to express this rule in a reasonable way! A synchron output wire

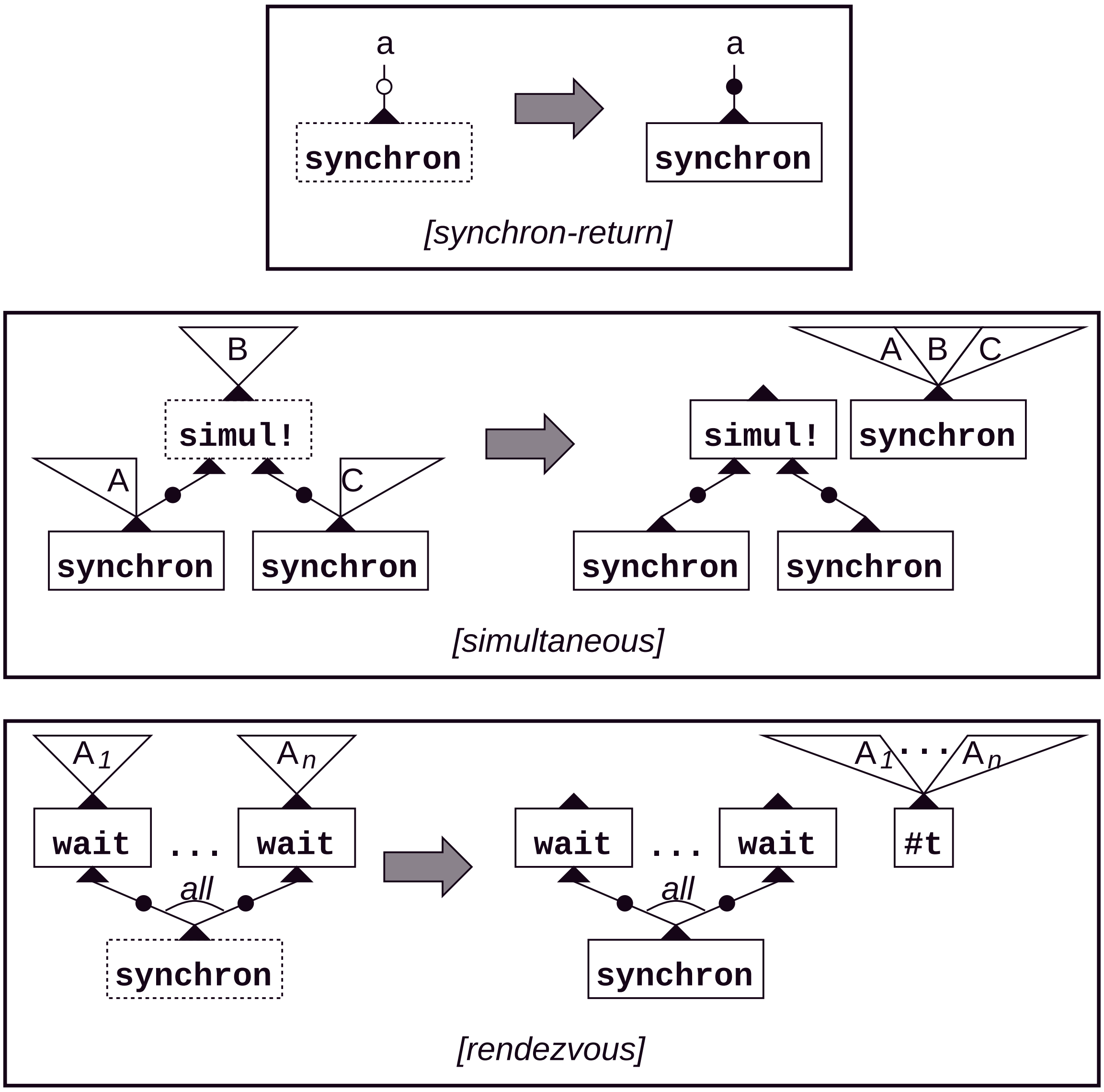


Figure 8.12: Simple versions of the EDGAR synchron rules.

that is in the returned state and attached to a demanded `wait` node is a *waiting pointer*. Any other output wire of a synchron is a *non-waiting pointer*. The rendezvous rule is only enabled when all of the output wires of a synchron are waiting pointers. The result of the rule is that a constant true node is returned to all output wires of all the wait nodes involved in the rendezvous. Note that the synchron is inaccessible after the [*rendezvous*] rule, and can be garbage collected. The rule essentially embodies a proof that the rendezvous is safe since every process that *could ever* wait on the synchron *is* waiting on the synchron.

The `precede!` operator complicates the handling of synchrons. The constraint that synchron A precedes synchron B will be represented by augmenting A with an input port that is connected to the output port of B via an inactive wire. Since the wire is a non-waiting pointer, B is blocked from rendezvousing as long at A is still accessible.

The modified rules necessary for handling synchrons in the presence of `precede!` are shown in Figure 8.13. Each synchron is assumed to have a set of input wires. The [*precede*] rule adds a new input wire to one synchron only if the other is not already a follower; the result returned by the `precede!` node is a constant true node. The [*simultaneous*] rule is modified to take the union of the followers of the two synchrons being combined. The [*rendezvous*] rule is trivially updated to indicate that nothing happens to the followers.

One last wrinkle concerning synchrons in the presence of `precede!` is that the *gc* garbage collection function must be modified to transform certain configurations of connected synchrons in order for OPERA to maintain the right storage behavior. The transformation is expressed as the [*compaction*] pseudo-rule in Figure 8.14. (Although expressed as a rewrite rule, the transformation is part of the *gc* garbage collection function, not the rewrite system). The pseudo-rule says that if a synchron with a single follower is only pointed at by other synchrons, it can safely be removed. This reduces the amount of space required by chains of unreferenced synchrons, which can occur in the *up* processing of filtered lists (see Section 5.5.4). Extending the rule to synchrons with more than one follower is not a good idea, because the transformation would actually *increase* the space by creating new wires.

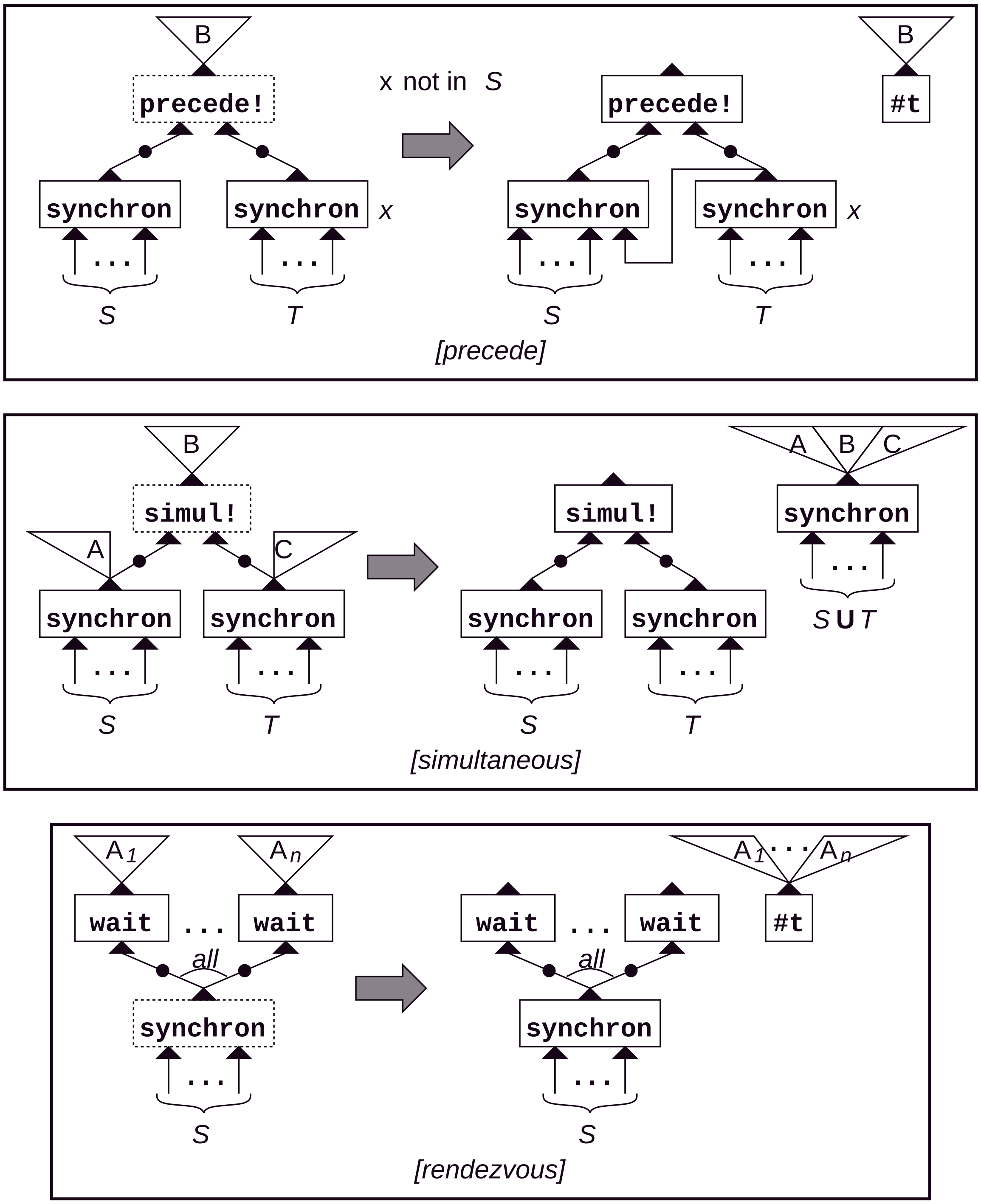


Figure 8.13: Extended versions of the EDGAR synchron rules.

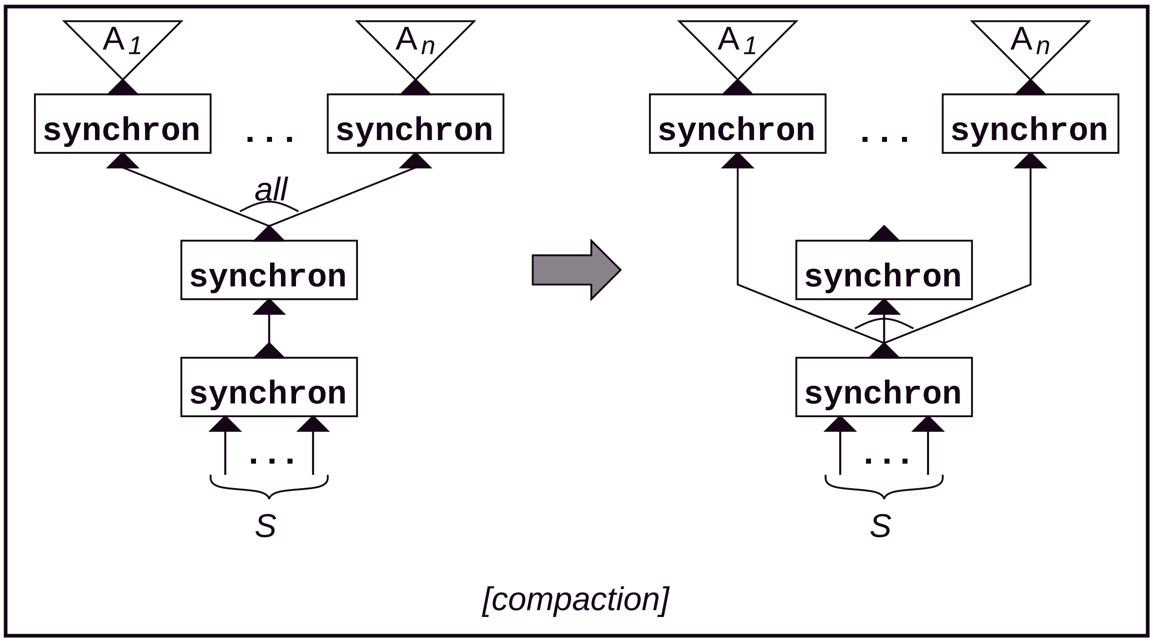


Figure 8.14: Pseudo-rule for synchron garbage collection that must be implemented by the *gc* function.

8.2.3 Excludons

In Opera, excludons and the `exclusive` construct are responsible for implementing mutual exclusion. The rules for manipulation excludons are shown in Figure 8.15. The [*excludon-return*] rule indicates that excludons are values. The side conditions on this rule implement the locking behavior of excludons; if an excludon has a returned output wire to one `exclusive`, it cannot return to another. The [*exclusive-request-lock*] rule requests an excludon, but this process will be blocked if the excludon has a returned wire to another `exclusive`. Once an `exclusive` has obtained a lock, it can evaluate the body ([*exclusive-with-lock*]). When the body returns ([*exclusive-release-lock*]), the lock is effectively released because garbage collection will remove the returned wire to the inaccessible `exclusive`.

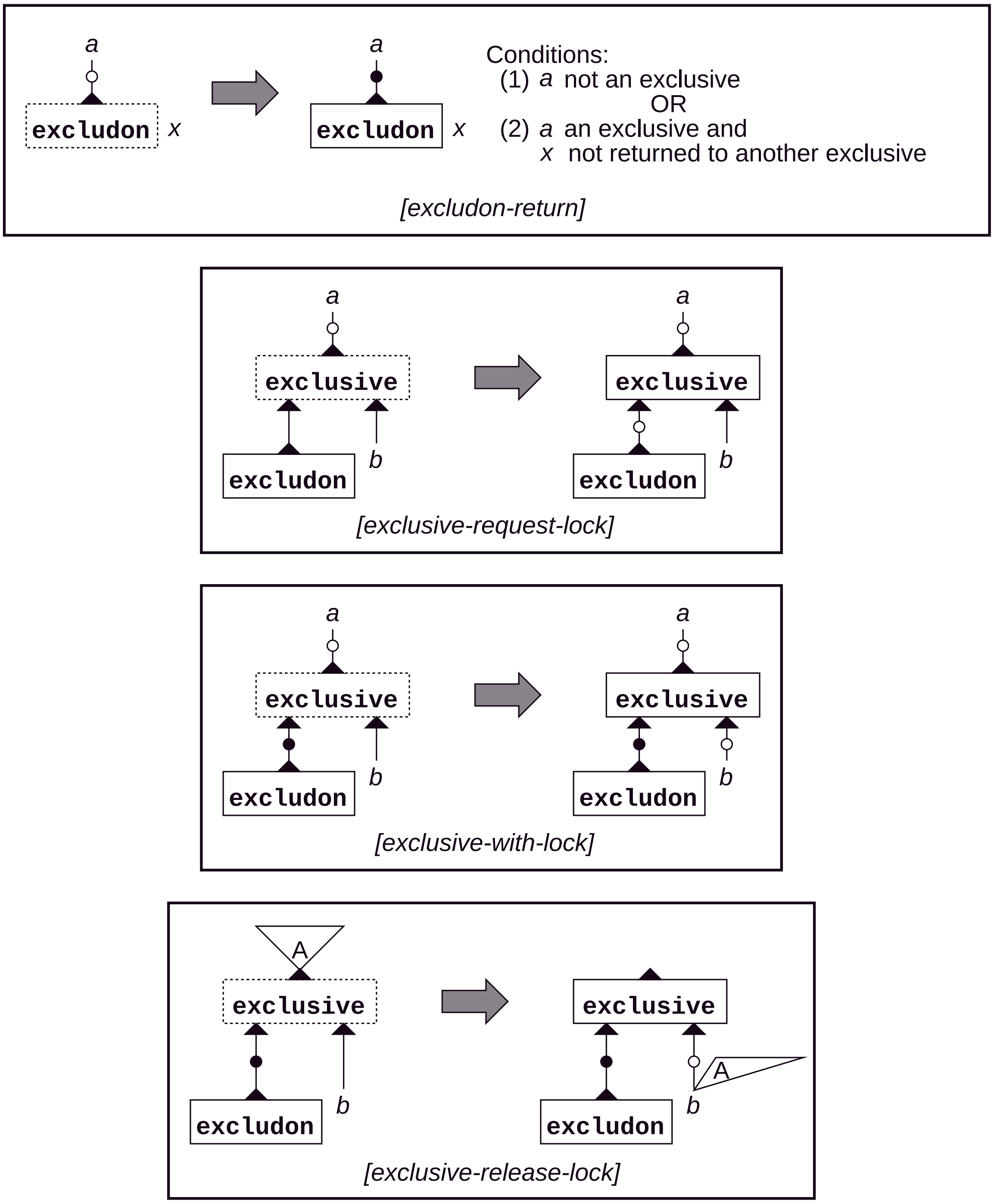


Figure 8.15: Rules for handling `excludons`.

8.2.4 Lazons and Eagons

OPERA's non-strictness is expressed in terms of lazons and eagons. The EDGAR rewrite rules for handling these to objects are shown in Figure 8.16. The rules indicate there is only a single difference between lazons and eagons: a lazon returns to a request without propagating the demand, while an eagon returns to a request *and* propagates demand. The dotted input wire to the `eagon` node in the [*eagon-touch*] rule indicates that the wire may be in any state.

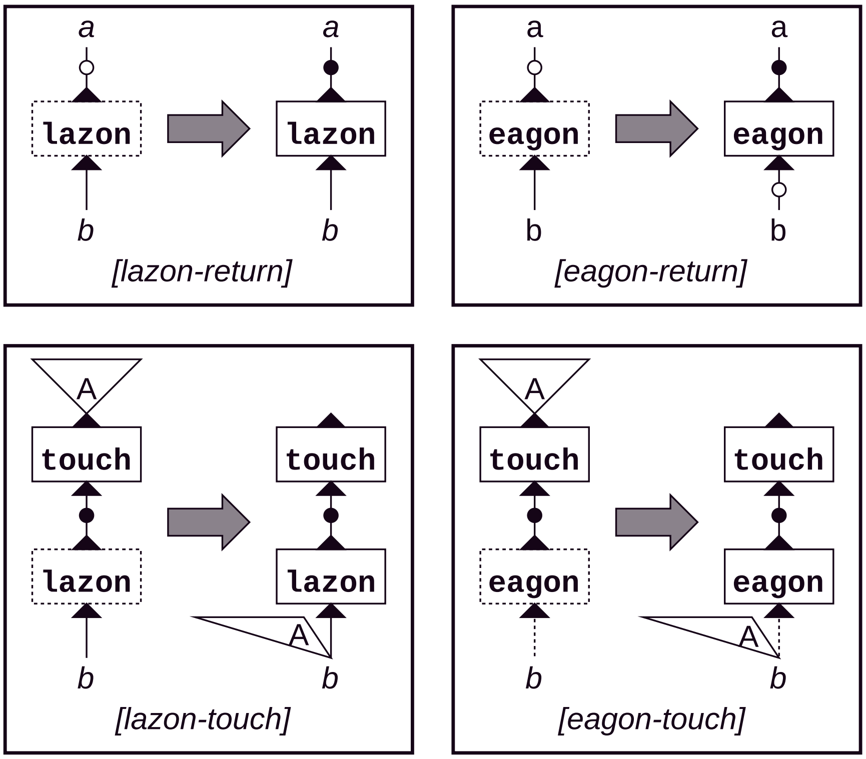


Figure 8.16: EDGAR rules for handling lazons and eagons.

8.3 Compiling OPERA into EDGAR

The semantics of OPERA can be formalized by describing how to translate an OPERA program into an EDGAR initial snapshot. The behavior of this snapshot is the meaning of the OPERA expression. The results of this snapshot are the possible values of the expression.

To simplify the presentation, we will introduce an intermediate language, OK, which is a slight variant of the OPERA kernel. We will then present the OPERA to EDGAR translation process in two stages: the translation from OPERA to OK, and the translation from OK to EDGAR.

8.3.1 OK

OK is a language that is a minor tweak of the OPERA kernel. The syntax of OK is summarized in Figure 8.17. (For comparison, see Figure 7.1 on page 296 for a summary of the OPERA kernel.) OK supports almost all the kernel syntax of OPERA. The differences are as follows:

- Unlike an OPERA program, an OK program includes no definitions. It consists of a single OK expression.
- Whereas OPERA's `quote` form can express arbitrary symbolic expressions as literals, OK's `quote` form can only express symbols.
- OK does not support the assignment form `(set!` I_{name} E_{val}`)`.
- OK includes the new form `(primop` O_{prim} E_{arg}`*)` to express the application of primitive operators. O_{prim} must be the name of one of OPERA's primitive procedures. `(primop + 1 2)` can be viewed as an "in-lined" version of `(+ 1 2)`.
- The forms `(seq1` E`*)` and `(seqn` E`*)`, which are syntactic sugar in OPERA, are considered to be kernel forms in OK.
- Except for the `seq1` and `seqn` forms, OK does not support any of the syntactic sugar of OPERA.

8.3.2 Translating OPERA to OK

The translation from OPERA to OK is performed by two functions:

1. $\mathcal{T}_{\text{prog}}$ maps an OPERA program to an OK program.

Kernel Grammar:

P	$\in$	Program	
E	$\in$	Expression	
I	$\in$	Identifier	
L	$\in$	Literal	
S	$\in$	Symbolic Expression	
O	$\in$	Operator Name	
P	::=	`(program` E_{body}`)`	[program]
E	::=	L	[literal expression]
	|	I	[variable reference]
	|	`(lambda (`I_{formal}`*)` E_{body}`)`	[abstraction]
	|	`(pcall` E_{proc} E_{arg}`*)`	[parallel application]
	|	`(pletrec ((`I_{name} E_{def}`)*)` E_{body}`)`	[parallel recursive bindings]
	|	`(if` E_{test} E_{then} E_{else}`)`	[conditional]
	|	`(quote` I`)`	[quoted expressions]
	|	`(lazon` E_{body}`)`	[suspension]
	|	`(eagon` E_{body}`)`	[future]
	|	`(exclusive` E_{excl} E_{body}`)`	[mutual exclusion]
	|	`(nex ((`I_{name} E_{def}`)*)` E_{body}`)`	[graphical bindings]
	|	`(nexrec ((`I_{name} E_{def}`)*)` E_{body}`)`	[graphical recursive bindings]
	|	`(primop` O_{prim} E_{arg}`*)`	[primitive application]
	|	`(seq1` E_{sequent}`*)`	[sequence/first]
	|	`(seqn` E_{sequent}`*)`	[sequence/last]
I	::=	*usual Scheme identifiers*	
L	::=	*usual Scheme literals*	

Figure 8.17: OK summary

2. $\mathcal{T}_{\text{exp}}$ maps an OPERA expression to an OK expression.

$\mathcal{T}_{\text{prog}}$ simply transforms an OPERA program into an OK program whose body is a `pletrec` expression:

$$\mathcal{T}_{\text{prog}}[\![\texttt{(program } E_{\text{body}} \texttt{ (define } I\ E\texttt{) ...)}]\!]$$
$$= \texttt{(program } \mathcal{T}_{\text{exp}}[\![\texttt{(pletrec ((}I_1\ E_1\texttt{) ...) } E_{\text{body}}\texttt{)}]\!]\texttt{)}$$

The $\mathcal{T}_{\text{exp}}$ function can be expressed as the composition of a number of source-to-source transformations I present each component transformation below in the order in which they are applied. I will treat each transformation as a separate phase, though in practice the phases can be interwoven into a transformation program that makes fewer passes over the OPERA expression (yet another potential application of slivers!):

Desugaring

The desugaring phase, $\mathcal{D}_{\text{exp}}$, transforms an OPERA expression into another OPERA expression by expanding all of the syntactic sugar forms. The syntactic sugar inherited from Scheme (except for `begin`) is expanded according to the rules given in [CR+91]. The OPERA-specific syntactic sugar is trivially expanded as follows:

$$\mathcal{D}_{\text{exp}}[\![\texttt{(begin } E \ldots\texttt{)}]\!] = \texttt{(seqn } \mathcal{D}_{\text{exp}}[\![E]\!] \ldots\texttt{)}$$

$$\mathcal{D}_{\text{exp}}[\![\texttt{(}E_{\text{proc}}\ E_{\text{arg}} \ldots\texttt{)}]\!] = \texttt{(pcall } \mathcal{D}_{\text{exp}}[\![E_{\text{proc}}]\!]\ \mathcal{D}_{\text{exp}}[\![E_{\text{arg}}]\!] \ldots\texttt{)}$$

$$\mathcal{D}_{\text{exp}}[\![\texttt{(let ((}I\ E\texttt{) ...) } E_{\text{body}}\texttt{)}]\!]$$
$$= \texttt{(pcall (lambda (}I \ldots\texttt{) } \mathcal{D}_{\text{exp}}[\![E_{\text{body}}]\!]\texttt{) } \mathcal{D}_{\text{exp}}[\![E]\!] \ldots\texttt{)}$$
; `plet` is handled similarly

$$\mathcal{D}_{\text{exp}}[\![\texttt{(letrec ((}I\ E\texttt{) ...) } E_{\text{body}}\texttt{)}]\!]$$
$$= \texttt{(pletrec ((}I\ \mathcal{D}_{\text{exp}}[\![E]\!]\texttt{) ...) } \mathcal{D}_{\text{exp}}[\![E_{\text{body}}]\!]\texttt{)}$$

$$\mathcal{D}_{\text{exp}}[\![\texttt{(seq } E \ldots\texttt{)}]\!] = \mathcal{D}_{\text{exp}}[\![\texttt{(seq1 \#t } E \ldots\texttt{)}]\!]$$

Though simple expansions exist for `seq1` and `seqn`, these forms are left unexpanded by the desugaring phase.

Quote removal

The quote removal phase, $\mathcal{Q}_{\text{exp}}$, transforms one OPERA expresion to another by lifting compound quoted expressions to top level and expanding them. A quoted expression is compound if the text of the quotation is a list or vector.

The lifting stage names all compound quoted expressions at top level and replaces their former positions with a variable reference. For example, lifting would transform:

```
(lambda () (list '(a b) 'd '(e (f g))))
```

into[5]

```
(let ((%1 '(a b))
      (%2 '(e (f g))))
  (lambda () (list %1 'd %2)))
```

The expansion stage, $\mathcal{X}_{\text{sexp}}$, uses the following transformations to expand quoted expressions:

$$\mathcal{X}_{\text{sexp}}[\![\texttt{'}L]\!] \;=\; L$$

$$\mathcal{X}_{\text{sexp}}[\![\texttt{'(}S\ \ldots\texttt{)}]\!] \;=\; \texttt{(list }\mathcal{X}_{\text{sexp}}[\![\texttt{'}S]\!]\ \ldots\texttt{)}$$

$$\mathcal{X}_{\text{sexp}}[\![\texttt{'(}S_{\text{car}}\ \texttt{.}\ S_{\text{cdr}}\texttt{)}]\!] \;=\; \texttt{(cons }\mathcal{X}_{\text{sexp}}[\![\texttt{'}S_{\text{car}}]\!]\ \mathcal{X}_{\text{sexp}}[\![\texttt{'}S_{\text{cdr}}]\!]\texttt{)}$$

$$\mathcal{X}_{\text{sexp}}[\![\texttt{'\#(}S_{\text{car}}\ \ldots\texttt{)}]\!] \;=\; \texttt{(vector }\mathcal{X}_{\text{sexp}}[\![\texttt{'}S_{\text{car}}]\!]\ \ldots\texttt{)}$$

$\mathcal{X}_{\text{sexp}}$ resembles the expander associated with Lisp's backquote notation, except that it works on quote notation instead. It leaves behind a tree of data constructors that will later create the structure denoted by the quoted expression. The leaves of the tree are literals and quoted symbols (quoted symbols are left untouched by $\mathcal{X}_{\text{sexp}}$).

Quote removal is necessary to accurately model Lisp's treatment of quoted expressions within EDGAR. In Lisp, the data structure of a quoted expression is created by the Lisp reader; every evaluation of the quoted expression returns this same data structure. In particular, the data structure can be mutated. Consider the following Scheme procedure:

[5] I use a `let` here to enhance readability, but since the $\mathcal{Q}_{\text{exp}}$ transformation is performed after the desugaring phase, the actual transformation needs to use the desugared form of `let`. In the interests of readability, I will similarly make use of other unavailable constructs in later examples; but in all cases, the transformation can be made precise without the offending constructs.

```
(let ((foo (lambda () '(1 2 3))))
  (begin (set-car! (foo) 17)
         (foo))))
```

The value of the above expression should be `(17 2 3)` because the `set-car!` mutates the value of the quoted list in the body of the `foo` procedure. Quote removal expresses this behavior by constructing the data structure for a quoted expression at top level (simulating the Lisp reader) and using naming to appropriately share the resulting value.

Assignment conversion

Since EDGAR does not support any kind of named entities, all names must be removed as part of the compilation process. Assignment conversion is one aspect of name removal. The assignment conversion phase, $\mathcal{A}_{\text{exp}}$, transforms OPERA expressions to OK expressions by removing all assignments — i.e., expressions of the form `(set!` I E`)`. Assignment conversion is described in [K+86]. The basic idea behind assignment conversion is to transform all assignments into mutation operations on cells (mutable one-slot data structures).

As an example, consider the OPERA expression:

```
(lambda (count)
  (lambda (inc)
    (seqn (set! count (+ count inc))
          count)))
```

The following OK expression results from applying the assignment conversion function, $\mathcal{A}_{\text{exp}}$, to the above OPERA expression:

```
(lambda (count)
  (let ((%count (cell count)))
    (lambda (inc)
      (seqn (cell-set! %count (+ (cell-ref %count) inc))
            (cell-ref %count)))))
```

The name `count` introduced by the `lambda` is shadowed by the `count` introduced by the `let`. The latter is bound to a cell containing the value of the former. The assignment to the former `count` is replaced by a `cell-set!` operation on the latter. Each variable reference to the former count is replaced by a `cell-ref` operation on the latter. Note that the `inc` variable, which is not assigned to, is left untouched.

In general, assignment conversion processes each name introduced by a `lambda` or `pletrec`. If the name is not assigned to in its lexical scope, it is left alone (like `inc`).

However, if the name is assigned in its lexical scope, then (1) a new variable is introduced and bound to a cell holding the value of the original; (2) every reference to the name in its lexical scope is replaced by an appropriate call to `cell-ref`; and (3) every assignment to the name in its lexical scope is replaced by an appropriate call to `cell-set!`.

Note that assignment conversion does *not* process names introduced by a `nex` or a `nexrec`. In fact, attempting to assign to a variable introduced by one of these constructs is an error.

Eta expansion

The eta expansion phase, $\mathcal{E}_{\mathrm{exp}}$, maps an OK expression to an OK expression by eta expanding [Bar84] every reference to a primitive procedure that does not occur in the operator position of a `pcall` application. For example:

$$\mathcal{E}_{\mathrm{exp}}[\![\texttt{(pcall f + (pcall + 1 2))}]\!]$$
$$= \texttt{(pcall f (lambda (a b) (pcall + a b)) (pcall + 1 2))}$$

Here, the first instance of `+` does not occur in the operator position of a pcall, so it is expanded in a `lambda` of two arguments (we assume that `+` takes exactly two arguments). Note that the expansion moves `+` into the operator position of a `pcall`. At the end of the eta expansion phase, every reference to a primitive procedure is in the operator position.

The name of a primitive procedure is only eta expanded if is not shadowed by a lexical variable. For instance, eta expansion acts as the identity on `(lambda (+) (f + 1))` because the name `+` introduced by the `lambda` is not a reference to the primitive procedure `+`.

In-lining of primitives

The in-lining phase, $\mathcal{I}_{\mathrm{exp}}$, maps OK expressions to OK expressions by converting every application of a primitive operator into a `primop` expression:

$$\mathcal{I}_{\mathrm{exp}}[\![\texttt{(pcall } O_{\mathrm{prim}}\ E_{\mathrm{arg}}\ \ldots\texttt{)}]\!] = (\mathrm{primop}\ O_{\mathrm{prim}}\ E_{\mathrm{arg}}\ \ldots)$$

This phase merely tags primitive operator applications as being special.

Touch introduction

The touch introduction phase, $\mathcal{H}_{\text{exp}}$, wraps an explicit `touch` around every expression that occurs in a touching context. The non-trivial clauses for the definition of $\mathcal{H}_{\text{exp}}$ are shown below:

$$\begin{aligned}
&\mathcal{H}_{\text{exp}}[\![\texttt{(if } E_{\text{test}}\ E_{\text{then}}\ E_{\text{else}}\texttt{)}]\!]\\
&\quad = \texttt{(if (touch } \mathcal{H}_{\text{exp}}[\![E_{\text{then}}]\!]\texttt{) } \mathcal{H}_{\text{exp}}[\![E_{\text{then}}]\!]\ \mathcal{H}_{\text{exp}}[\![E_{\text{else}}]\!]\texttt{)}\\
&\mathcal{H}_{\text{exp}}[\![\texttt{(pcall } E_{\text{rator}}\ E_{\text{rand}}\ \ldots\texttt{)}]\!]\\
&\quad = \texttt{(pcall (touch } \mathcal{H}_{\text{exp}}[\![E_{\text{rator}}]\!]\texttt{) } \mathcal{H}_{\text{exp}}[\![E_{\text{rand}}]\!]\ \ldots\texttt{)}\\
&\mathcal{H}_{\text{exp}}[\![\texttt{(exclusive } E_{\text{excl}}\ E_{\text{body}}\texttt{)}]\!]\\
&\quad = \texttt{(exclusive (touch } \mathcal{H}_{\text{exp}}[\![E_{\text{excl}}]\!]\texttt{) } \mathcal{H}_{\text{exp}}[\![E_{\text{body}}]\!]\texttt{)}\\
&\text{; This clause only applies if } O_{\text{prim}} \text{ is neither a constructor nor } \texttt{touch}.\\
&\mathcal{H}_{\text{exp}}[\![\texttt{(primop } O_{\text{prim}}\ E_{\text{arg}}\ \ldots\texttt{)}]\!]\\
&\quad = \texttt{(primop } O_{\text{prim}}\ \texttt{(touch } \mathcal{H}_{\text{exp}}[\![E_{\text{arg}}]\!]\texttt{) } \ldots\texttt{)}
\end{aligned}$$

The `primop` transformation is not applicable when O_{prim} is `touch` or when it is a constructor (i.e., `cons`, `vector`, and `cell`). The special case for constructors allows them to hold untouched lazons and eagons.

The straightforward touch introduction scheme sketched above can be optimized to avoid unnecessary touches. For example, it is never necessary to touch a literal, quoted expression, or `lambda`. In fact, it is only necessary to touch expressions that might evaluate to a lazon or eagon. Analysis techniques that conservatively approximate this property could reduce the number of `touch`es introduced.

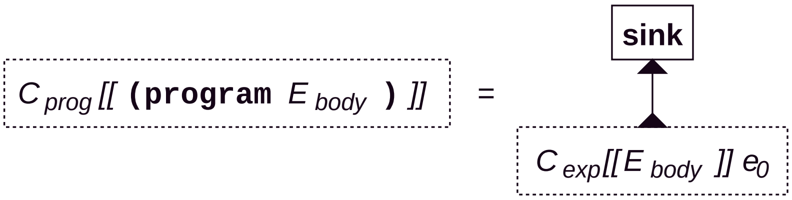
8.3.3 Translating OK to Edgar

Most of the messy details of compiling Opera to Edgar are handled by the Opera to OK translation described above. The second stage, translating OK to Edgar, is fairly straightforward. We will describe two compilation functions:

1. $\mathcal{C}_{\text{prog}}$ compiles an OK program to an Edgar initial graph.

2. $\mathcal{C}_{\text{exp}}$ compiles an OK expression to an Edgar graph.

$\mathcal{C}_{\text{prog}}$ compiles the OK program `(program` E_{body}`)` to an Edgar graph rooted at a `sink` node:



$\mathcal{C}_{\mathrm{exp}}$ takes two arguments, an OK expression and an environment, and returns a port of a EDGAR graph. The environment is a function mapping names to EDGAR ports. The empty environment, e_0, maps every name to a distinguished *unbound port* indicating that the name is not bound. The above compilation rule for $\mathcal{C}_{\mathrm{prog}}$ calls $\mathcal{C}_{\mathrm{exp}}$ on the program body and the empty environment.

Compiling simple expressions

$\mathcal{C}_{\mathrm{exp}}$ is a simple recursive descent compiler. Figure 8.18 shows sample clauses for its definition. A literal compiles to a constant node, a quoted symbol compiles to a symbol literal, and `lazon`, `if`, and `primop` expressions compile to `lazon`, `if`, and `primop` nodes. The clauses for `pcall`, `eagon`, `seq1`, `seqn`, and `exclusive` are similarly straightforward, and are not shown.

The only interesting clauses in the definition of $\mathcal{C}_{\mathrm{exp}}$ are for those expressions that manipulate the environment: I, `nex`, `nexrec`, `lambda`, and `pletrec`. When $\mathcal{C}_{\mathrm{exp}}$ reaches a variable reference I, it returns the port associated with I in the environment. If the name is associated with the unbound port, then the expression contains an unbound variable. (Note that such a name can't be a global reference to a primitive procedure since all these have been removed by the eta expansion and procedure in-lining phases of $\mathcal{T}_{\mathrm{exp}}$.)

Compiling `nex` and `nexrec`

The compilation of a `nex` expression extends the environment with the result of recursively compiling the binding expressions, and returns the result of compiling the body expression in the extended environment:

$$\mathcal{C}_{\mathrm{exp}}[\![\texttt{(nex ((}I_1\ \ E_1\texttt{)}\ \ldots\texttt{(}I_n\ \ E_n\texttt{))}\ \ E_{\mathrm{body}}\texttt{)}]\!]e = \mathcal{C}_{\mathrm{exp}}[\![E_{\mathrm{body}}]\!]e[\mathcal{C}_{\mathrm{exp}}[\![E_1]\!]e/I_1,\ldots,\mathcal{C}_{\mathrm{exp}}[\![E_n]\!]e/I_n]$$

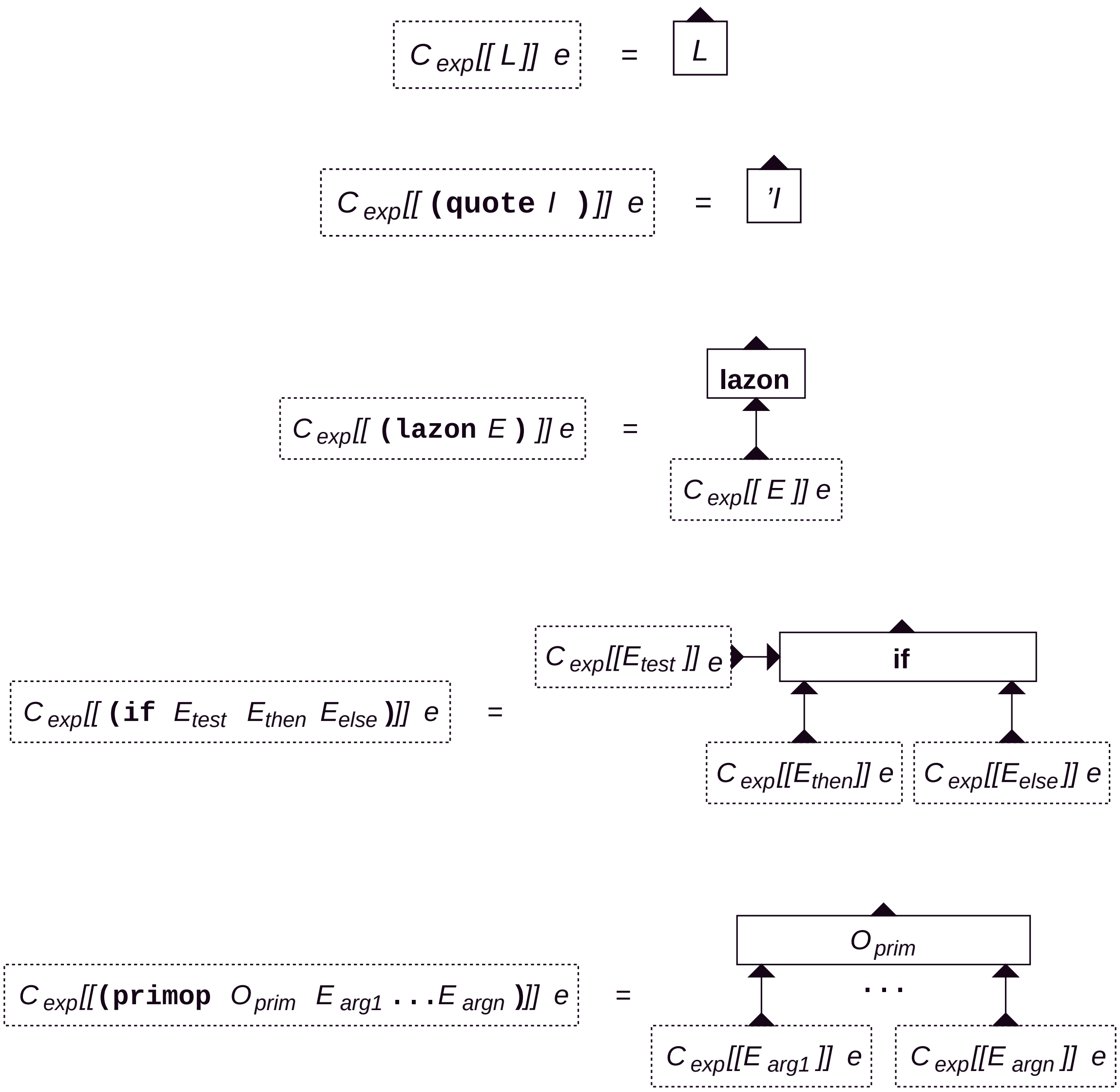


Figure 8.18: Sample clauses for the OK to EDGAR compiler.

(The notation $e[\ldots, v_i/I_\mathrm{i}, \ldots]$ stands for the result of extending environment e with the bindings between identifiers I_i and values v_i.) By allowing a subgraph to be named and referred to later by a variable, `nex` permits the creation of directed acyclic graphs.

`Nexrec` permits the full flexibility of graphs by allowing cyclic references through the environment:

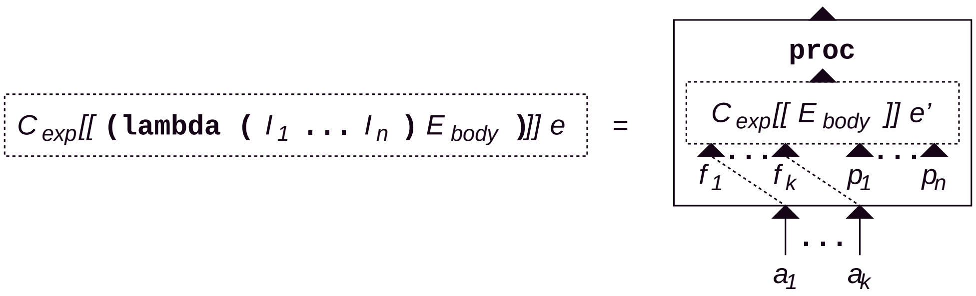
$$\mathcal{C}_{\mathrm{exp}}[\![\texttt{(nexrec ((}I_1\ \ E_1\texttt{)}\ \ldots\texttt{(}I_\mathrm{n}\ \ E_\mathrm{n}\texttt{))}\ \ E_{\mathrm{body}}\texttt{)}]\!]e \ = \mathcal{C}_{\mathrm{exp}}[\![E_{\mathrm{body}}]\!]e\,'$$
$$\text{where}\quad e\,' = e[\mathcal{C}_{\mathrm{exp}}[\![E_1]\!]e\,'/I_1, \ldots, \mathcal{C}_{\mathrm{exp}}[\![E_\mathrm{n}]\!]e\,'/I_\mathrm{n}]$$

The environment $e\,'$ is the fixed point obtained from extending e with bindings between the names I_i and the graphs resulting from the compilation of the value expressions E_i in $e\,'$.

Compiling `lambda`

The compilation of `lambda` expressions depends crucially on the notion of *free variables*. Suppose we view an OK expression as an abstract syntax tree whose leaves are literals, quoted symbols, and variable references, and whose internal nodes are the compound expressions types (e.g., `pcall`, `lambda`, `if`, etc.). Then an occurrence of a variable reference I within an OK expression is *free* if in the corresponding abstract syntax tree there is no I-binding `lambda`, `pletrec`, `nex`, or `nexrec` on the path from the reference to the root of the tree. In other words, I is free if it is not lexically bound by one of the binding constructs. The free variables of E is the set of the names of all the variable references that are free in E.

The compilation of a `lambda` expression in environment e yields a `proc` node whose template results from compiling the body of the `lambda` in an extended environment $e\,'$:



Suppose that the `lambda` expression being compiled contains k free variables, $J_1 \ldots J_k$. Then $e'$is defined as

$$e' = e[f_1/J_1, \ldots, f_k/J_k, p_1/I_1, \ldots, p_n/I_n]$$

where f_1, …, f_k, p_1, …, p_k are new ports. That is, the extended environment binds each free variable and parameter of the `lambda` expression to a new port. These new ports serve as placeholders for the graph structure that will eventually be the values of the variables.

The `proc` node itself has k input ports which are wired to the results of looking up the free variables J_1, …, J_n in the environment e. These inputs represent the values of the free variables used within the `lambda` expression. If the free variable was originally bound by a `lambda` or `pletrec`, the result of a lookup will be a placeholder port. However, if the variable was originally bound by a `nex` or `nexrec`, the result of the lookup may be the output port of a fragment of compiled graph structure.

Figure 8.19 shows the result of compiling the following `lambda` expression:

```
(lambda (a)
  (lambda (b)
    (pcall (lambda (c d)
             (* b
                (/ (- c d)
                   (+ c d))))
           (* a a)
           (* b b))))
```

For readability, the parameter ports on the procedure templates have been labelled with the corresponding parameter names. The three nested `proc` nodes in the figure correspond to the three nested `lambda`s of the expression. Note that the template of the innermost `proc` has only one free variable input port because the corresponding `lambda` expression has only one free variable (`b`).

Together, the `lambda` compilation strategy and the EDGAR rewrite rule for `pcall` describe the "plumbing" that allows argument values and the values of free variables to reach their destinations. The input ports to a template within a `proc` node stand for all the values that can potentially be used to compute the body of the template; these include the explicit parameters of the `lambda` as well as implicit parameters — i.e., free variables. A `proc` node holds free variable values as inputs so that they can be supplied as implicit argument values

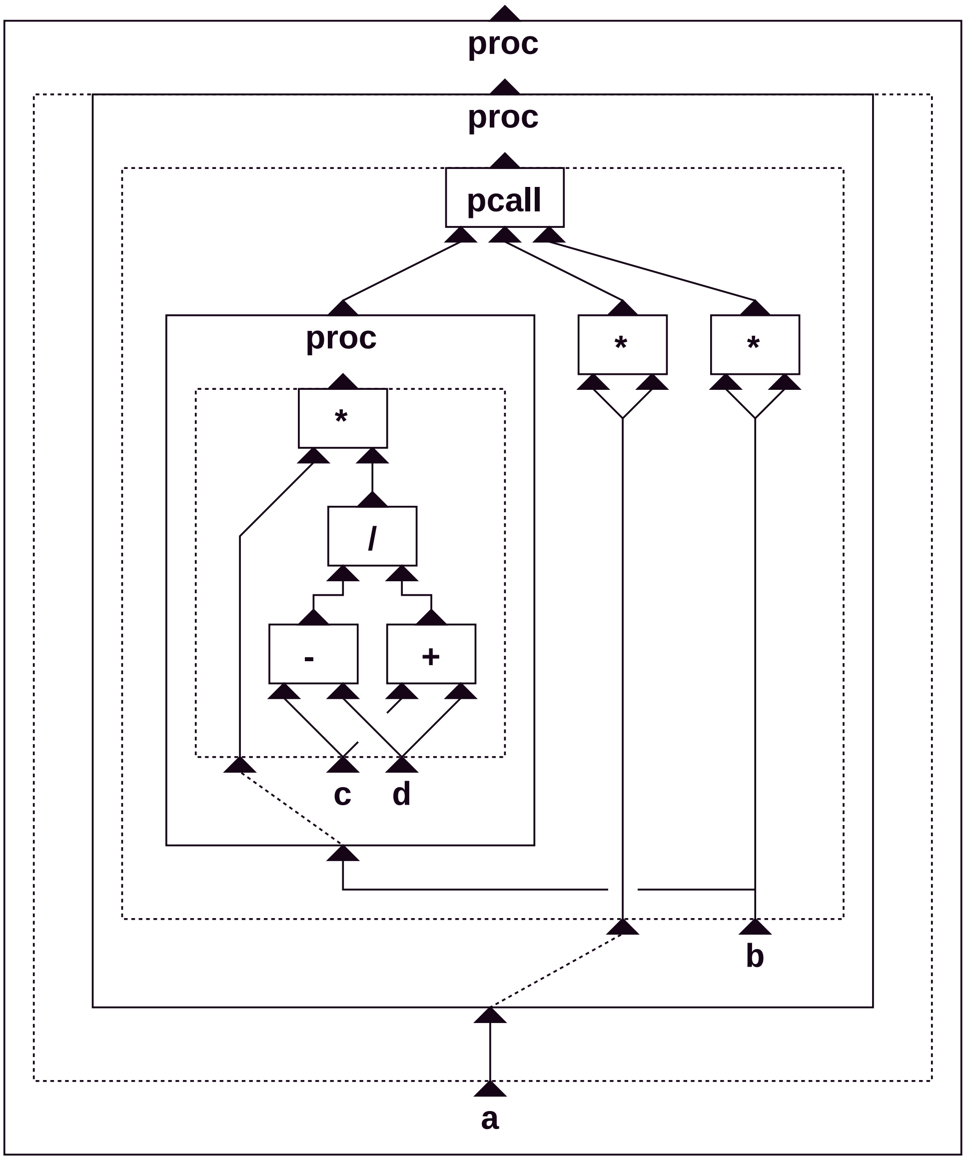


Figure 8.19: Result of compiling a `lambda` expression via $\mathcal{C}_{\mathrm{exp}}$.

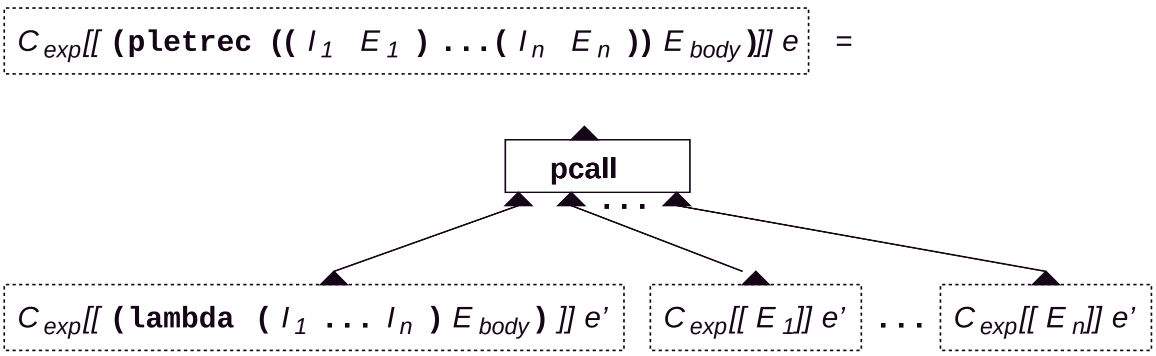
at the point of call. This strategy for managing free variables as implicit arguments is a graphical version of the *lambda lifting* technique for transforming nested procedures into top-level ones [Joh85].

An important detail of the `lambda` compilation process is that the implicit parameters to a procedure are *only* the free variables that appear textually inside of it. A `proc` node only holds onto values that are likely to be used in the computation of the template. In particular, a `proc` node does not hold onto the values of lexical variables that are not referenced by the template. This strategy has desirable consequences for garbage collection. Aggressively dropping references to the values of unused lexical variables means that those values can be garbage collected as soon as they are no longer needed.

In contrast, procedures in the traditional environment model [ASS85] hold onto the values of *all* lexically enclosing variables. As a result, the garbage collection of many provably useless values is delayed because they are spuriously held by a procedure. In programs that manipulate synchrons, the aggressive reference dropping inherent in `lambda` compilation strategy solves the riddle (raised in Section 7.1.3) of how deadlocks implied by the environment model are avoided in OPERA.

Compiling `pletrec`

The compilation of a `pletrec` expression involves the compilation of a `lambda` expression as a subcomponent:



This compilation rule is similar to the desugaring rule for `plet`,

$$\mathcal{D}_{\text{exp}}[\![\texttt{(plet ((}I\ E\texttt{) ...)}\ E_{\text{body}}\texttt{)}]\!] = \texttt{(pcall (lambda (}I\ \texttt{...)}\ \mathcal{D}_{\text{exp}}[\![E_{\text{body}}]\!]\texttt{)}\ \mathcal{D}_{\text{exp}}[\![E]\!]\ \texttt{...)},$$

except that the compilation rule uses an extended environment e' for compiling the subexpressions. The extended environment is the solution of the same fixed point equation used in the compilation of `nexrec`:

$$e' = e[\mathcal{C}_{\text{exp}}[\![E_1]\!]e'/I_1, \ldots, \mathcal{C}_{\text{exp}}[\![E_n]\!]e'/I_n]$$

8.3.4 Notes

We conclude the discussion of OPERA to EDGAR compilation with a few notes:

- *No names:* It is worth emphasizing in the context of OPERA to EDGAR compilation that EDGAR has no names. In fact, one of the most interesting aspects of the compilation process is how it translates all names into graphical connections.

- *Meaning of* OPERA *programs:* OPERA programs are compiled to EDGAR via the function $\mathcal{C}_{\text{prog}} \circ \mathcal{T}_{\text{prog}}$ that composes the two translation stages described above. It is easy to show that the result of this compilation process is an EDGAR initial graph. By the definition of $\mathcal{C}_{\text{prog}}$, it is rooted at a sink; and all of the clauses of $\mathcal{C}_{\text{exp}}$ only create graph structure in which all wires are inactive. We can then define the meaning of an OPERA program as the behavior of the initial graph into which it compiles. The possible values of the OPERA expression can be determined from the outcomes of the behaviors (see Section 8.1.6). In the case where an outcome is an EDGAR value node, it is easy to translate the graph rooted at that node into a legal OPERA value.

  This notion of meaning ignores the fact that the program might read data from or write data to some global entity (such as a file system). In general, the behavior of an OPERA program might depend on global state, and we would be interested in tracking any changes the program makes to the global state. We will assume that global state is handled in one of the ways suggested in Section 8.1.7, and that the notion of program meaning is changed to take into account the initial and final state.

- *Meaning of* OPERA *expressions:* All of the translation machinery described above assumed that we started with an OPERA *program*, not an OPERA *expression*. Yet, in practice, we might be concerned with the meaning of OPERA expressions evaluated by an interactive interpreter (such an interpreter is described in Section 9.1.2). This introduces several complexities. First, it would be necessary to accurately model state maintained by the interpreter (e.g., a global environment that keeps track of bindings introduced by top-level `define`s). Second, it would be necessary to extend the notion of the behavior of an EDGAR graph to express the changes made to the interpreter state by evaluating the graph. This is another example of the global state issue (see Section 8.1.7).

8.4 Alternatives and Extensions

The above discussions of EDGAR and the compilation of OPERA to EDGAR were structured to simplify the presentation. Here we briefly touch upon some extra issues that were ignored above.

- *Primop rules:* There is no real need for EDGAR to represent primitive applications with a special class of nodes. Instead of representing an addition by a special + *application* node, it would be more elegant to handle this situation using the general `pcall` with a special `+` *value* node as its operator. This would reduce the number of rewrite rules and obviate the need for the eta expansion and primitive in-lining phases of the OPERA to EDGAR compiler.

  Why, then, have primitive application nodes? Because without them, the rewrite rules can become more complex. Any rule involving a primitive application (e.g., rules involving `wait`, `touch`, `car`, etc.) would include extra `pcall` nodes. This would make it harder for both people and programs to reason about the rules.

- *Value returns:* The demand driven nature of EDGAR is intended to model the evaluation process of a Scheme interpreter. That is, demand arriving at a node is supposed to correspond to calling `eval` on the expression represented by that node. But the EDGAR rules introduced above fail to accurately model the return of values. For example, in the evaluation of the graph corresponding to `(+ 1 2)`, the newly created result node (`3`) must be demanded before its value is returned. But this implies that the Scheme interpreter calls `eval` on the result of a primitive application, which is definitely not the case.

This behavior can be fixed by dictating that demanded wires connected to the newly created node must immediately enter the returned state. But such a change complicates the primitive application rule. And it also raises new questions. If `(+ 1 2)` is demanded by *two* wires, should the newly created `3` immediately return to both? (I.e., in a concurrent Scheme evaluation model, would it make sense for a value to be returned to two threads at the same time?)

- *Trivial transitions:* In a typical trace, an large percentage of the transitions are trivial ones in which a value node is requested or is returned. These trivial transitions encumber step-by-step reasoning about EDGAR executions, and are a serious annoyance in the program animator based on the EDGAR model (Section 9.1.4).

  It is possible to remove trivial transitions by changing the way that values are handled. An alternative to having value nodes to respond to individual demands is to require that all output wires of every value node are always in the returned state. This alternate strategy is consistent with the interpretation that a returned wire indicates that the node at the source of the wire is a value node. It also corresponds to the common practice in operational semantics of classifying a subset of rewritable expressions as values [Mey, GJ90]. An additional benefit is that it greatly simplifies the handling of `quote`, since a compound quoted expression can be treated as a single value created before the program is executed.[6]

  Unfortunately, the alternate strategy for handling values complicates the EDGAR rewrite rules. At the high level, it invalidates the constraint that a wire cannot enter a returned state until it is first in the demanded state. It also complicates particular rules by increasing the number of cases that must be considered. For instance, it would be necessary to modify the [*pcall-request*] rule to indicate that demand is only propagated to unreturned input wires. New notations could simplify this change;

[6] However, even with the alternate value strategy, it would still be necessary to perform the lifting aspect of the quote removal transformation to get the appropriate sharing of quoted values.

we could introduce a wire pattern that matched only inactive or returned wires, and introduce a meta-rule that demanding a returned wire has no effect. However, such notations would complicate the already detailed rules of EDGAR.[7]

For simplicity, I stuck with the no-return-until-requested model in my presentation of the EDGAR rules above. However, my implementation actually uses the every-value-is-returned strategy.

8.5 Related Work

The development of EDGAR was heavily influenced by previous work in operational semantics and graph reduction. Graph reduction has long been used as an implementation technique for functional programming languages. Turner [Tur79] developed a means for compiling functional programs into graphs of three combinators (S, K, and I) and executing the resulting graphs by graph rewriting. Hughes [Hug82] introduced a method, super-combinators, for compiling a function into a specialized graph rewriting rule. Peyton Jones [Pey87] summarizes numerous issues relevant to graph reduction as an implementation technique.

While EDGAR is clearly a graph reduction framework, it differs in many respects from the work done in the functional programming community. First is a difference in emphasis: EDGAR is primarily a semantic framework, not an implementation technique. The EDGAR rules have been implemented as an interpreter, but only as a proof-of-concept; we make no claim that the EDGAR style of graph reduction is a good implementation technique. In this sense, EDGAR is closer to work on graph rewriting semantics [B+87, AA93].

Because implementation efficiency is not a concern, some of the concepts in EDGAR are cleaner than in other graph reduction frameworks. For example, the ability for EDGAR rules to "move" wires obviates the need for messy special mechanisms like overwriting application nodes and inserting indirection nodes (see [Pey87]). The fact that EDGAR deals with side effects, concurrency, and synchronization clearly separates it from much of the

[7] The added complexity may be due to the fact that using return tokens to represent values may just be a bad idea. The traditional approach of defining a class of value graphs might simplify the presentation of the alternate strategy. This possibility needs to be explored.

other work in the field. Although parallel graph reduction techniques exist [Pey87, Sme93], the focus is again on efficiency, not on semantics. Until relatively recently, side effects have been anathema to the functional programming community, so they rarely appear in graph reduction work (but see [AA93]).

An notable exception to the avoidance of side effects is Bawden's work on connection graphs (CGs) [Baw86, Baw92]. Bawden introduced (CGs) to explore issues of state and linearity in programming languages. Like EDGAR, CGs are a framework in which graph rewrite rules define the semantics of interconnected nodes with labelled ports. Because the systems have different goals, it is not surprising that they differ in many details: CG wires are undirected while EDGAR wires are directed; CGports must be connected to exactly one wire, while EDGAR output ports may be connected to zero or multiple wires; CG rewrite rule patterns may only have two nodes connected by a single wire, while EDGAR rewrite rule patterns may involve any connected set of nodes; CG uses node states to simulate evaluation, while EDGAR uses tokens on wires. Nevertheless, CGs exerted a tremendous influence on the development of OPERA. The notion that procedures hold onto the values of their free variables, the non-deterministic handling of side effects, and many aspects of the OPERA to Scheme compiler were inspired by Bawden's Scheme to CG compiler.

EDGAR's explicit representation of demand was inspired by the demand tokens used in the operational model for Gelernter and Jaganathan's Ideal Software Machine (ISM) [GJ90]. ISM is a model that ascribes sequential and parallel characteristics to dimensions of both space and time. The concurrency allowed by ISM (parallelism in the time dimension) is formally expressed in a semantic framework that combines aspects of Petri nets [Pet77] and graph rewriting. Like Petri nets, the ISM semantics uses tokens to represent concurrency and synchronization; in the case of ISM, the tokens are an explicit representation of demand. As in graph rewriting, the ISM semantics allows graph rewrites to occur where patterns are matched; the patterns usually involve tokens.

Not only was the basic ISM model adopted in EDGAR, but many of EDGAR's rewrite rules (such as those handling procedure calls, conditionals, and sequential evaluation) are almost identical to ones in the ISM semantics (modulo different notation). There are several major differences, however. Foremost is EDGAR's support of synchrons as a new synchronization technology; in fact, the *raison d'etre* of the EDGAR model is to explain synchrons. Another major difference is naming. We find that the ISM model handles naming in a rather ad hoc fashion; by eliminating names altogether, the EDGAR model emphasizes that naming is fundamentally a topological concern (i.e., names express connections). As far as we can determine, the ISM semantics for procedure call is not properly tail recursive; this is an EDGAR feature that is crucial for controlling space behavior. Finally, it appears that ISM's representation of closure environments holds onto values longer than necessary. To be fair, ISM was designed to be a general model for comparing different languages. We view EDGAR as an improved version of ISM targeted at faithfully modelling a wide range of operational issues.

A handful of other systems employ explicit representation of demand. Arvind and Pingali describe a mechanism for simulating demand-driven evaluation in a data flow model; they use data tokens to represent demand [PA85, PA86]. Ashcroft describes a system that combines demand flow (via entities called *questons*) with data flow (via entities called *datons*) [Ash86].

The visual notations used to represent EDGAR graphs were influenced by the notations used in the data flow community (e.g., [Den75, DK82]). We emphasize, however, that the execution models for EDGAR and data flow are fundamentally different. In data flow models, tokens representing values flow through mostly static networks; the same wire can be used transmit many different values. In EDGAR's dynamically-changing networks, a wire may only transmit a single value. Additionally, the demand-driven nature of EDGAR conrasts with the data-driven emphasis of data flow.

The visual notations of EDGAR bear some resemblance to plan calculus representations from the Programmer's Apprentice project [Ric81]. And, in fact, EDGAR was designed to support many of the same kinds of program decompositions considered in that project. EDGAR has the advantage that it is executable, but as a relatively low-level operational model, it does not straightforwardly support the high-level plans at the heart of the Programmer's Apprentice.

It is possible to encode EDGAR's graphical rewrite rules in a more traditional textual fashion. The substitution model described by Meyer [Mey], which was influenced by ideas in

EDGAR, is a step in this direction. Unlike most other text-based operational models, Meyer's model captures the sharing of values in data structures. However, from the viewpoint of pedagogy, we strongly believe that the visual notations provided by EDGAR are vastly superior to any textual mechanism for indicating sharing.