

The Spider Calculus

Computing in Active Graphs

Benjamin C. Pierce
University of Pennsylvania
bcpierce@cis.upenn.edu

Alessandro Romanel
CoSBI and Università di Trento
romanel@cosbi.eu

Daniel Wagner
University of Pennsylvania
wagnerdm@seas.upenn.edu

March 25, 2010

Abstract

We explore a new class of process calculi, collectively called *spider calculi*, in which processes inhabit the nodes of a directed graph, evolving and communicating by local structural mutations. We study a variety of spider calculi, analyze their expressive power, and identify a *kernel spider calculus* that is both minimal and expressive. In particular, processes in the kernel calculus can construct arbitrary finite graphs, encode common data structures, and implement the communication primitives of the π -calculus. Finally, we show how an even simpler variant of the kernel calculus, in which the universe is an *undirected* graph, can encode the directed variant.

1 Introduction

The execution environment for modern software systems is fundamentally graph-structured. On both large and small scales (from routers and fiber to processors, memories, and buses), software components inhabit different physical or logical locations, and information must cross links for them to cooperate. Ordinarily, this graph structure is hidden behind simpler abstractions, such as point-to-point internet communication or the shared memory abstraction of a multiprocessor. But there are some situations where we want to deal with it explicitly.

For example, the implementations of the abstraction layers themselves must deal directly with the underlying graph structure. This includes many internet protocols, such as those for routing, broadcast, name resolution, and so on. There are also applications that work directly with the graph structure to increase efficiency. Content distribution networks such as Akamai [21] and Coral [11] fall in this category, as do numerous distributed graph algorithms, for example algorithms for answering reachability queries on streaming graphs [24] and approximating shortest paths with incomplete knowledge of the graph [15].

In all these applications, the graph structure is relatively fixed. In other cases, the graph may change as the program evolves—for example, in routing algorithms for ad hoc mobile networks. Indeed, there are cases where the program itself may alter the topology of the graph in which it is working. Peer-to-peer networks do just this, establishing logical graph structures for organizing communication over the location-transparent abstraction of internet routing. Also, some well-known parallel algorithms operate on virtual graphs: Delaunay mesh triangulation modifies graphs in which nodes represent physical locations [18], and n -body simulations often use graphs in which each node represents a volume of the space being simulated [2].

In the theoretical literature, there has been a corresponding interest in foundational models that explicitly embody notions of locality and connectivity. Pi-calculus variants such as Nomadic Pict [25] and the

Distributed π -calculus [14] model mobile computation in the internet, allowing direct communication only between processes that have migrated to the same location. Cardelli and Gordon’s Ambient Calculus [7], the biologically-inspired Brane Calculus [4], and their many variants [3, 5, 6, 19] refine the full point-to-point connectivity of the Distributed π -calculus by structuring locations into a tree; process movement is restricted to paths in the tree, and processes can alter the tree structure in the immediate neighborhood of their current location. Cardelli and Gardner’s 3π [8] model location within a coordinate system, so that movement is associated with a physical change of coordinates. Connectivity in 3π is decided in part at runtime; processes may refuse to receive a message based on the relative location of the sender.

We study here a new class of systems, dubbed *Spider Calculi*, that generalize the ideas of the Ambient Calculus to arbitrary graphs. Our goal in this paper is to take the first steps into this new design space: to explore the expressiveness of various primitives and to experiment with basic programming idioms. One outcome of this investigation is that we identify a particular calculus, the *kernel spider calculus* (or sometimes just “the spider calculus”) that strikes an attractive balance between expressiveness and simplicity.

2 Overview

To limit the design space to a manageable size, we start by adopting two fundamental constraints. First, we consider computing in edge-labeled directed graphs—that is, the “universe” for a computation (called the *web*, naturally) is a graph with labeled edges and anonymous nodes. (Other choices, such as labeling the nodes, may be reasonable, but we do not study them here.) We call the edges *links* to emphasize the labeling. We place no restriction on links or labels: there can be self-links, multiple links between any particular pair of nodes, or multiple links with the same label. Second, we prohibit any action at a distance. Each computational process (“spider”) is associated with a particular node in the graph, and only the links incident to that node are visible to the spider. To observe or modify links elsewhere in the graph, the spider must first travel there.

To describe computations in graphs, we need three things. First, we need a notation for describing the graphs themselves. Second, we need a notation for local computations at the nodes, including data structures, conditionals, loops, communication, synchronization, and so forth. And third, we need ways for processes to navigate and observe the graph. In the interest of parsimony, we have tried to combine these three as much as possible. In particular, there is no need for “local data”; as in the λ -calculus and π -calculus, which encode local data in function structure and process structure, respectively, all the data and control structures we need can be encoded in the web structure.

One consequence of this choice is that there are really two distinct kinds of links: links in the “real graph” and links that are created and used for some spider’s local computation. To avoid interference between the two kinds, and also to avoid interference between the local computations of distinct spiders, it is useful to make one last assumption: that link names are *scoped*—that there is a way to generate fresh link names that are only known to the spider that generates them—and that a spider can neither observe nor affect links whose names it does not know. Formally, we follow the π -calculus and its relatives by introducing the *restriction* operator ν for this purpose.

For the spiders themselves, we again follow the lead of the π -calculus. Spiders are written using a few generic combinators—an inert “null process,” parallel composition, and replication—plus a small set of primitive *actions*. The actions express atomic steps of the computation that modify and navigate the graph. Our main focus in this paper is on exploring the space of possible actions, striving for a compact, expressive set. To assess compactness, Section 7 proves a number of *independence* results, showing how some sets of actions can encode others. To assess expressiveness, we use two benchmarks: building arbitrarily shaped finite graphs and emulating the π -calculus. Our experience experimenting with spider programming suggests that any calculus that can do these two things is expressive enough to capture a broad range of computations in and on graphs.

Finally, after experimenting with versions of the calculus based on both directed and undirected graphs, we have chosen to present the calculus in terms of directed graphs. Directed links—that is, links accessible from only one “side”—seem to make programming slightly easier, though we discuss in Section 8 the possibility

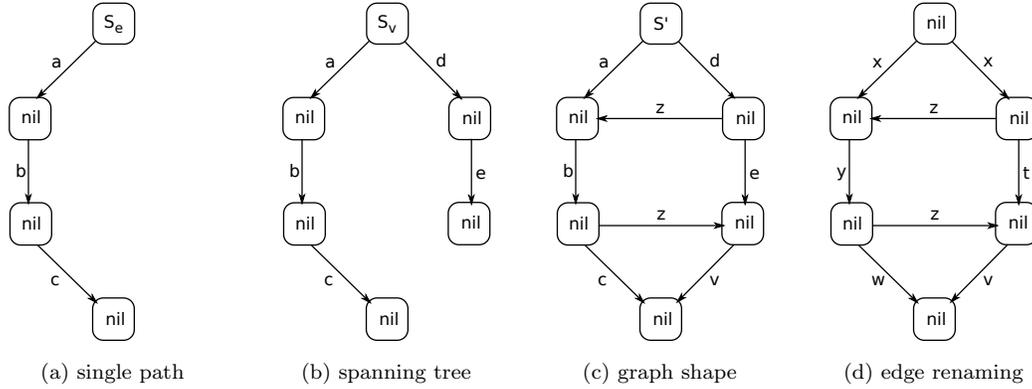


Figure 1: Building a finite graph

that the undirected version of the calculus is more fundamental.

3 Example

Before coming to formalities, let's look at a nontrivial program in the spider calculus, as a concrete demonstration of the generic features of the language and to give a flavor of the kind of primitives we are considering.

Any “graph computing calculus” worth its salt should certainly be expressive enough to construct finite graphs of any shape. For instance, consider the graph in Figure 1d. We'd like to write a spider program that builds this graph, beginning with a single node containing a single spider. Figures 1a, 1b, and 1c show intermediate webs; these will act as subgoals.

The first task is to build a new node with edges to it. We introduce three primitives for this purpose: **create**, **copy**, and **reverse**. The behavior of the **create** action is sketched in Figure 2a, which represents one rule in the reduction semantics (given formally in the next section). The small dashed lines represent any other links incident to the node; these are not affected by the **create** operation. The figure leaves implicit the fact that there may be other spiders running at this node, which are also unaffected. This is an instance of a general design principle: spiders interact *only* by observing each others' effects on the graph. Also, the continuation T of **create** remains at the same node rather than moving to the new node; this is a free choice in the definition of the primitive.

The **create** operation builds a single directed link from the creating node to the new one. A little later, we will also need another edge pointing back from the new node. The **copy** and **reverse** actions, sketched in Figures 2b and 2c, make this possible. The **copy** primitive simply creates an additional link with the same source and target as another link, while **reverse** swaps the source and target of an existing link. The three can be combined—

$$S = \text{create } x. \text{ copy } x \text{ as } x. \text{ reverse } x. S'$$

to create a new node with forward and backward links named x to it. (The **reverse** primitive chooses nondeterministically between the two available x links, but since they are identical, the outcome is the same.) This is a common enough task that we will use **createboth** $x. S'$ as shorthand for the above spider S .

We can now use our ability to create a single new node to create a path with unique names for each edge on the path. (These names are temporary; the last step will be replacing them with the actual labels that we want on these edges. Using distinct names during construction avoids ambiguity in cases where one node in the final graph has two links with the same name, like the topmost node in Figure 1d.) After the execution of a single **createboth** x operation, we have a new node accessible by a link named x , but no spider at the new node. To put one there, we use the **go** action, depicted in Figure 2d. Together, the **createboth** and **go** actions are enough to construct any finite path; the spider S_b shown in Figure 3 demonstrates how to build

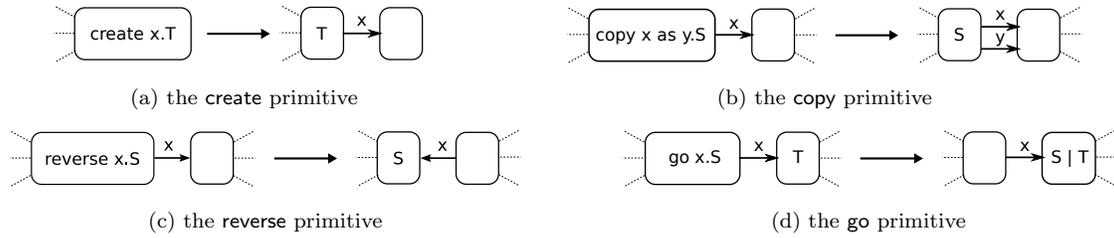


Figure 2: Primitives used in subgoal 1a

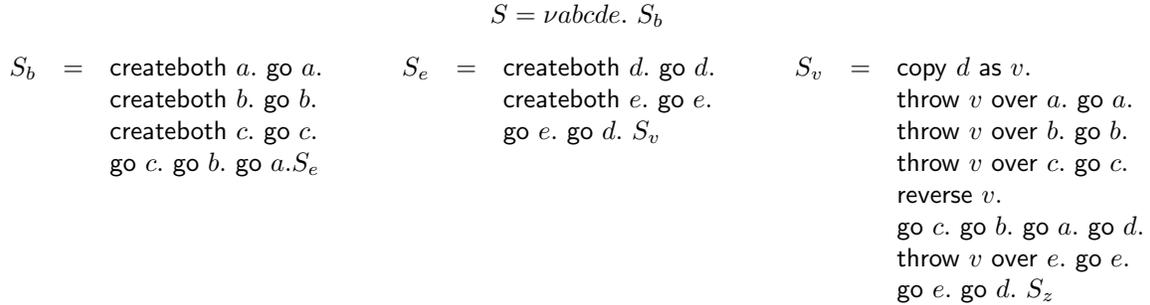


Figure 3: Parts of the spider that builds the graph in Figure 1d.

the path in Figure 1a. (The redundant round-trip over c clarifies the pattern for building arbitrary paths.) The spider S_e in Figure 3 finishes building the spanning tree by adding another path, then continues as S_v .

The next task is to flesh out the spanning tree. We can add a new edge between existing nodes with `copy x as y` , then relocate its endpoints using the `throw` action depicted in Figure 4. (As we will see later, `throw` has other important uses.) The spider S_v in Figure 3 demonstrates this process for building the v edge of our final graph, creating it initially at the top of the graph by copying a and then throwing each end down the spanning tree to its final destination. The spider S_z (not shown) is similar, putting the two z edges into place, then continuing as S_r .

We now have a graph with the desired shape with edges in all the right places, but with the wrong names. To rectify this, we introduce the `rename` primitive, depicted in Figure 5. The spider S_r shown in Figure 6 demonstrates the use of this primitive. This leaves us a graph with the right shape, but with some extra links named a , b , c , d , and e . One way to tidy these up would use the `delete` primitive¹ pictured in Figure 7. Alternatively, we can leave them, but use the restriction operator ν to ensure that they cannot affect the behavior of other spiders. This gives us the final spider S shown in Figure 3.

¹See Section 7 for a discussion of `delete`.

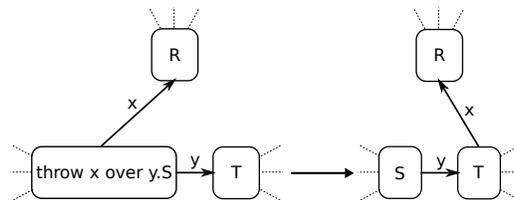


Figure 4: The `throw` primitive used for constructing cycles

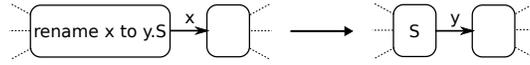


Figure 5: The rename primitive

```

Sr = go a. go b.
      rename c to w. go b.
      rename b to y. go a.
      rename a to x. go d.
      rename e to t. go d.
      rename d to x. nil
  
```

Figure 6: Renaming edges to match the desired graph.

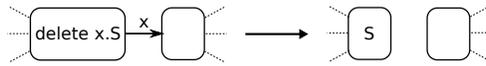


Figure 7: The delete primitive, which can be used for synchronization

$$S = S_{abc} \mid S_{de} \mid S_{watchdog}$$

$S_{abc} =$ createboth <i>a</i> . go <i>a</i> . createboth <i>b</i> . go <i>b</i> . createboth <i>c</i> . go <i>c</i> . go <i>c</i> . go <i>b</i> . go <i>a</i> . create <i>done</i> . nil	$S_{de} =$ createboth <i>d</i> . go <i>d</i> . createboth <i>e</i> . go <i>e</i> . go <i>e</i> . go <i>d</i> . create <i>done</i> . nil	$S_{watchdog} =$ delete <i>done</i> . delete <i>done</i> . S_v
--	--	--

Figure 8: Another way to write S

$$\begin{aligned}
V, W, Z &::= \text{Nil} \mid \nu x. W \mid V \mid W \mid [S]^i \mid i \xrightarrow{x} j \\
R, S, T &::= \text{nil} \mid \nu x. S \mid S \mid T \mid !S \mid M. S \\
M &::= \text{create } x \mid \text{go } x \mid \text{copy } x \text{ as } y \mid \\
&\quad \text{rename } x \text{ to } y \mid \text{throw } x \text{ over } y \mid \\
&\quad \text{reverse } x
\end{aligned}$$

Figure 9: Syntax of the Spider Calculus

$$\begin{aligned}
\text{fn}(\text{Nil}) &= \emptyset & \text{fn}([S]^i) &= \text{fn}(S) \cup \{i\} \\
\text{fn}(\nu x. W) &= \text{fn}(W) \setminus \{x\} & \text{fn}(W \mid V) &= \text{fn}(W) \cup \text{fn}(V) \\
\text{fn}(M.S) &= \text{fn}(M) \cup \text{fn}(S) & \text{fn}(i \xrightarrow{x} j) &= \{x, i, j\} \\
\text{fn}(\text{nil}) &= \emptyset & \text{fn}(!S) &= \text{fn}(S) \\
\text{fn}(\nu x. P) &= \text{fn}(P) \setminus \{x\} & \text{fn}(S \mid T) &= \text{fn}(S) \cup \text{fn}(T) \\
\text{fn}(\text{rename } x \text{ to } y. S) &= \{x, y\} \cup \text{fn}(S) & \text{fn}(\text{create } x. S) &= \{x\} \cup \text{fn}(S) \\
\text{fn}(\text{copy } x \text{ as } y. S) &= \{x, y\} \cup \text{fn}(S) & \text{fn}(\text{go } x. S) &= \{x\} \cup \text{fn}(S) \\
\text{fn}(\text{throw } x \text{ over } y. S) &= \{x, y\} \cup \text{fn}(S) & \text{fn}(\text{reverse } x. S) &= \{x\} \cup \text{fn}(S)
\end{aligned}$$

Figure 10: Definition of free names for webs and spiders

The approach outlined so far can be improved by parallelizing. For example, to parallelize the spanning tree creation, we can build the two branches in parallel rather than sequentially. The spiders S_{abc} and S_{de} shown in Figure 8 each build one branch of the tree. These two spiders can be run in parallel using the parallel composition operator $S_{abc} \mid S_{de}$. However, since the next phase of the construction relies on the existence of the complete spanning tree, it waits for S_{abc} and S_{de} to signal their completion by creating a link at the root node with an agreed-upon name (*done*). Transforming the program from sequential to parallel in this way is a global process. Nevertheless, this does not compromise our design goal of locality, because it is a program-level transformation, not a graph-level transformation. To observe when the completion links appear, the spider at the root node uses the **delete** action, which waits until a link with a particular name exists, then removes it from the web. The parallel version of S is shown in Figure 8.

4 Formal Definition

In the syntax of the spider calculus (Figure 9), we use V, W, Z to range over webs and R, S, T to range over spiders. Both grammars include syntactic forms for restriction and for binary and nullary parallel composition; when these forms appear at the top level of a spider, the structural congruence “promotes” them to the analogous forms at the level of the web. The graph structure of the web is represented as a parallel composition of nodes and edges, with node identity determined by name. For example, we regard a web with one node labeled x containing a spider $S \mid T$ as structurally equivalent to a web with two nodes labeled x , one containing S and one containing T . (This may appear to be a significant difference from Ambients, where a solution with a single node is *not* equivalent to a solution with two nodes of the same name, but the appearance is deceptive. Ambient node names correspond to *edge* names in the spider calculus.) The form $[S]^i$ denotes a spider S living at node i , and $i \xrightarrow{x} j$ denotes an edge from i to j labeled x .

We use the variable M to range over primitive actions and use lower-case letters for variable names. As usual, the spider $M. S$ blocks until M can be executed and then continues as S . The replication of spider S is written $!S$. (There is no replication form at the level of webs and no structural congruence rule for

$$\begin{array}{l}
V =_{\alpha} W \Rightarrow V \equiv W \quad \alpha\text{-conversion} \\
\\
V \mid \text{Nil} \equiv V \quad (\text{par-nil}) \\
V \mid (W \mid X) \equiv (V \mid W) \mid X \quad (\text{par-assoc}) \\
V \mid W \equiv W \mid V \quad (\text{par-com}) \\
\\
\nu x. \nu y. W \equiv \nu y. \nu x. W \quad (\text{res-res}) \\
V \mid \nu x. W \equiv \nu x. (V \mid W) \quad (\text{res-par}) \\
\text{if } x \notin \text{fn}(V) \\
\\
[\text{nil}]^i \equiv \text{Nil} \quad (\text{trans-nil}) \\
[S \mid T]^i \equiv [S]^i \mid [T]^i \quad (\text{trans-par}) \\
[\nu x. S]^i \equiv \nu x. [S]^i \quad (\text{trans-res}) \\
\text{if } x \neq i
\end{array}$$

Figure 11: Structural congruence laws

$$\begin{array}{l}
\frac{W \longrightarrow W'}{W \mid V \longrightarrow W' \mid V} \quad (\text{red-par}) \\
\\
\frac{W \longrightarrow W'}{\nu x. W \longrightarrow \nu x. W'} \quad (\text{red-res}) \\
\\
\frac{W \equiv W' \longrightarrow V' \equiv V}{W \longrightarrow V} \quad (\text{red-struct}) \\
\\
[!S]^i \longrightarrow [S]^i \mid [!S]^i \quad (\text{red-repl}) \\
\\
[\text{throw } x \text{ over } y. S]^i \mid i \xrightarrow{x} j \mid i \xrightarrow{y} k \longrightarrow [S]^i \mid k \xrightarrow{x} j \mid i \xrightarrow{y} k \quad (\text{red-dthrow}) \\
\\
[\text{create } x. S]^i \longrightarrow [S]^i \mid \nu j. i \xrightarrow{x} j \quad (\text{red-dcreate}) \\
\text{where } j \notin \{i, x\} \\
\\
[\text{rename } x \text{ to } y. S]^i \mid i \xrightarrow{x} j \longrightarrow [S]^i \mid i \xrightarrow{y} j \quad (\text{red-drename}) \\
\\
[\text{copy } x \text{ as } y. S]^i \mid i \xrightarrow{x} j \longrightarrow [S]^i \mid i \xrightarrow{x} j \mid i \xrightarrow{y} j \quad (\text{red-dcopy}) \\
\\
[\text{go } x. S]^i \mid i \xrightarrow{x} j \longrightarrow [S]^j \mid i \xrightarrow{x} j \quad (\text{red-dgo})
\end{array}$$

Figure 12: Operational semantics

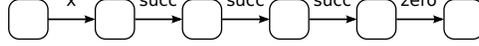


Figure 13: A number stored at location x in the structural encoding

replication; instead, replication is implemented using a reduction rule.) We abbreviate $M.\text{nil}$ as M and $\nu x_1. \dots \nu x_n. W$ as $\nu x_1, \dots, x_n. W$ or $\nu \tilde{x}. W$.

The structural congruence and operational semantics are mostly standard. Figure 10 gives the definition of free names (ν is the only name binder). Figure 11 defines the structural congruence. Name restrictions via ν may be floated in and out of parallel compositions, node boundaries, and other restrictions provided this does not orphan any names or cause any clashes. Figure 12 gives the operational semantics, which just formalizes the informal diagrams we have already seen.²

5 Programming

5.1 Data Structures

There is no provision in the core calculus for data like numbers, strings, lists, and so on. In this subsection, we will give three examples showing how to encode a natural number; each of these examples generalizes naturally to more complicated data types.

Passive encoding The first encoding uses the graph structure of a web in a straightforward way. In this encoding, a number is a location with either a link named *succ* pointing to another number or a link named *zero*. Figure 13 shows the value 3 encoded in a web. Using numbers in this encoding is straightforward: you choose one of two spiders based on the existence of one of the above links. For example, suppose we have a number stored at the bidirectional link x . (By bidirectional, we mean that there are two copies of the link pointing in opposite directions.) Then the following snippet will compute the predecessor of x and store it in y :

`go x. (copy succ as y. throw y over x | rename zero to zero. go x. copy x as y)`

The first two actions in the parallel composition depend on *succ* and *zero* existing, respectively; if the location really is a number, this means that only one of the two continuations will fire.

Active encoding In the second encoding, a number is represented as a spider that repeatedly answers requests for its value. There is an globally agreed-upon link name *val*; a number will wait for a link named *val* to appear, then go to that link, create either a *succ* or a *zero* link, and continue as its predecessor (if it has one). Here is the 0 spider:

$$S_0 = !\nu t. \text{rename } val \text{ to } t. \text{go } t. \text{create } zero$$

Assuming S_n is the spider representing the number n , here is the spider representing S_{n+1} :

$$S_{n+1} = !\nu t. \text{rename } val \text{ to } t. \text{go } t. \text{create } succ. S_n$$

Consuming a number in this representation is similar in spirit to the previous representation. For example, assuming there is a bidirectional link x pointing to a location that contains (only) a number, here is a spider

²One notable difference from pi-calculus (and variants of ambient calculus with communication primitives) is that the operational semantics doesn't involve substitution of one name for another. This makes it a little surprising that there is a direct translation of the pi-calculus into the spider calculus. The intuition for why this works is that while we don't have substitution, we do have aliasing, in the form of two links (with different names) that point to the same node.

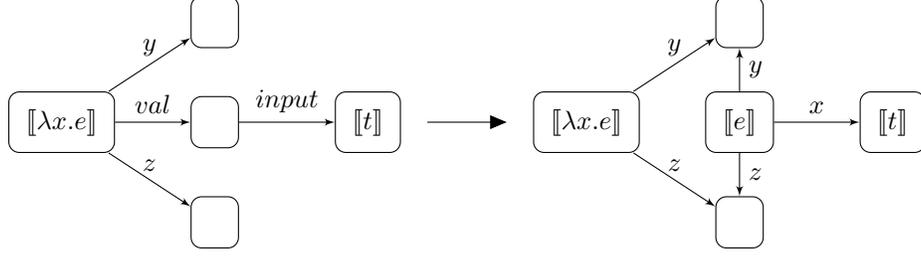


Figure 14: Emulating beta-reduction in the Spider Calculus, assuming $\text{fn}(e) = \{x, y, z\}$

$$\begin{array}{ll}
 \llbracket \lambda x. e \rrbracket & = \text{!} \nu t. \text{ rename } val \text{ to } t. \\
 & \quad \text{closure } \text{fn}(e) \setminus \{x\} \text{ at } t. \\
 & \quad \nu x. \text{ rename } input \text{ to } x. \llbracket e \rrbracket \\
 \llbracket x \rrbracket & = \text{!throw } val \text{ over } x \\
 \llbracket e_1 e_2 \rrbracket & = \nu t_1, t_2. \text{ createboth } t_1. \text{ create } t_2. \\
 & \quad (\text{ closure } \text{fn}(e_1) \text{ at } t_1. \llbracket e_1 \rrbracket \\
 & \quad | \text{ closure } \text{fn}(e_2) \text{ at } t_2. \llbracket e_2 \rrbracket \\
 & \quad | \text{ go } t_1. \text{ rename } t_1 \text{ to } val \\
 & \quad | \text{ copy } t_2 \text{ as } input)
 \end{array}$$

Figure 15: Encoding the λ -calculus

that creates a link y pointing to a location containing that number's predecessor:

$$\begin{array}{l}
 \text{go } x. \nu t. \text{ createboth } t. \text{ copy } t \text{ as } val. \text{ go } t. \\
 (\text{ rename } succ \text{ to } succ. \text{ go } t. \text{ rename } t \text{ to } y. \text{ throw } y \text{ over } x \\
 | \text{ rename } zero \text{ to } zero. \text{ go } t. \text{ go } x. \text{ copy } x \text{ as } y)
 \end{array}$$

Again, one of the two processes in the parallel composition will be stuck forever.

Church encoding The final encoding we describe encodes the λ -calculus into webs, then uses the standard Church-encoding of numbers. In encoded λ -calculus webs, there are two kinds of locations, namely function locations and argument locations. A function location has a spider waiting for a link named val that points to an argument location; an argument location has a link named $input$ pointing to a function location. Function application involves transforming an argument location into a function location by consuming the $input$ link, then awaiting val links. Each variable name in the λ -calculus corresponds to a link in the spider calculus that points to the location holding the value associated with that name. The translation of a variable simply forwards any val requests along the link corresponding to that variable's name, that is, to the location associated with that variable.

We model closures as nodes with a link for each bound name. We often have to copy and transmit an entire closure, so we introduce an abbreviation; if $S = \{x_1, \dots, x_n\}$ is a set of names, then $\text{closure } S \text{ at } t. T$ will stand for

$$\text{copy } x_1 \text{ as } x_1. \dots \text{ copy } x_n \text{ as } x_n. \text{ throw } x_1 \text{ over } t. \dots \text{ throw } x_n \text{ over } t. \text{ go } t. T.$$

Figure 14 sketches the evolution of a beta reduction, and Figure 15 shows the complete encoding of the λ -calculus.

5.2 Encoding the π -calculus

In the overview, we chose two benchmarks for the expressiveness of the calculus: building finite graphs and emulating the π -calculus. Section 3 tackled the former; this section tackles the latter. As a reminder, the syntax, structural congruence rules, and semantics of the standard (synchronous, choice-free) π -calculus are given in Figure 16.

$P, Q, R ::= \bar{x}y.P \mid x(y).P \mid (\nu x)P \mid !P \mid P \mid Q$	
$P \mid Q \equiv Q \mid P$	p-par-comm
$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$	p-par-assoc
$P \mid \text{nil} \equiv P$	p-sc-par-nil
$(\nu x)(P \mid Q) \equiv P \mid (\nu x)Q$ if $x \notin \text{fn}(P)$	p-res-par
$(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$	p-res-res
$\bar{x}y \mid x(z).P \longrightarrow P\{y/z\}$	p-red-comm
$P \longrightarrow P' \Longrightarrow (x)P \longrightarrow (x)P'$	p-red-res
$P \longrightarrow P' \Longrightarrow P \mid Q \longrightarrow P' \mid Q$	p-red-par
$P \mid Q \longrightarrow P' \mid Q' \Longrightarrow !P \mid Q \longrightarrow !P' \mid Q'$	p-red-rep
$P \equiv P' \longrightarrow Q' \equiv Q \Longrightarrow P \longrightarrow Q$	p-red-struct

Figure 16: Semantics of the π -calculus.

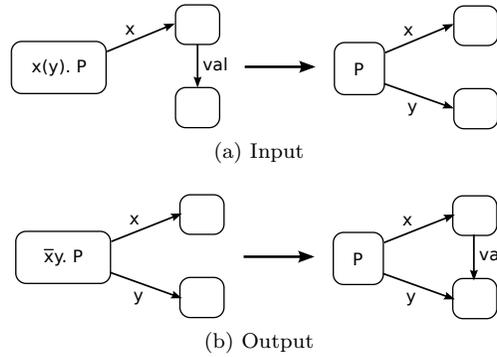


Figure 17: The execution of encoded π -calculus primitives

The idea is simple: all of the translations of π processes run in a single location (which we call *root*), and a π channel is represented by a location with links named *val* pointing to the messages waiting to be sent. Using a channel in the π -calculus requires knowing a name for it; using a channel in the Spider Calculus requires having an edge from *root* to the location for that channel. Figures 17a and 17b sketch the high-level behavior of encoded input and output processes. The full definition of the encoding is given in Figure 18.

5.3 Encoding Recursion Using Replication

There is a well-known conversion [23] in the π -calculus from recursion into replication. In broad strokes, a recursion creates a new channel and a process that repeatedly reads from that channel; when the process is ready to recurse, it sends a message on that channel, allowing an additional copy of the replicated read to make progress. A straightforward analogue in the Spider Calculus doesn't quite work, because the *go* primitive may take a spider to a different location than the replication it is trying to signal. The following example suggests a modified translation that deals with this problem.

Suppose we are given a web whose graph shape is a binary tree: each node has two edges named *left* and *right*. Additionally, there is a spider in this web that will be traveling through the tree, leaving messages about which direction it went at each node via links named *goleft* and *goright*. Our task is to write a spider which will follow these clues. If we had a μ operator in the Spider Calculus, the following spider would do

$$\begin{aligned}
[[P]_S^\pi &= [[S]^\pi \mid [[P]^\pi]^{root} \\
[[\{x_1, \dots, x_k\}]^\pi &= (root \xrightarrow{x_1} j_1 \mid \dots \mid root \xrightarrow{x_k} j_k) \\
[[\text{nil}]^\pi &= \text{nil} \\
[[(\nu x)P]^\pi &= \nu x. (\text{create } x \mid [[P]^\pi) \\
[[!P]^\pi &= ![P]^\pi \\
[[P \mid Q]^\pi &= [[P]^\pi \mid [[Q]^\pi \\
[[\bar{x}y]^\pi &= \nu z, z'. S_1 \\
&\quad \text{where} \\
&\quad S_1 = \text{copy } x \text{ as } z. S_2 \\
&\quad S_2 = \text{copy } y \text{ as } z'. S_3 \\
&\quad S_3 = \text{throw } z' \text{ over } z. S_4 \\
&\quad S_4 = \text{go } z. S_5 \\
&\quad S_5 = \text{rename } z' \text{ to } val \\
[[x(y).P]^\pi &= \nu z, z', y. T_1 \\
&\quad \text{where} \\
&\quad T_1 = \text{copy } x \text{ as } z. T_2 \\
&\quad T_2 = \text{copy } z \text{ as } z'. T_3 \\
&\quad T_3 = \text{reverse } z'. T_4 \\
&\quad T_4 = \text{go } z. T_5 \\
&\quad T_5 = \text{rename } val \text{ to } y. T_6 \\
&\quad T_6 = \text{throw } y \text{ over } z'. T_7 \\
&\quad T_7 = \text{go } z'. [[P]^\pi \\
&\quad \text{provided } z, z' \notin \text{fn}(P)
\end{aligned}$$

Figure 18: Encoding the π -calculus in directed webs

nicely:

$$\mu S. (\text{delete } goleft. \text{go left. } S) \mid (\text{delete } goright. \text{go right. } S)$$

We can achieve the same effect with replication by using a *home* location where the replication lives, plus one or more *current* locations where running incarnations of the replication are executing. We will connect these locations with private links named ℓ . This invariant will need to be preserved as the running spiders move around the web; we can do this by throwing (a copy of) the ℓ link before each `go` command. A reverse link from the home location to a current location can serve both as the signal for the replication to spin off a new copy and as the pointer to where it should begin executing. Thus, our path-following spider would look like this:

$$\begin{aligned}
&\nu \ell. \text{createboth } \ell. \text{go } \ell. \\
&!\nu t. \text{rename } \ell \text{ to } t. \text{go } t. \\
&(\text{delete } goleft. \text{throw } \ell \text{ over } left. \text{go left. copy } \ell \text{ as } \ell. \text{reverse } \ell) \mid \\
&(\text{delete } goright. \text{throw } \ell \text{ over } right. \text{go right. copy } \ell \text{ as } \ell. \text{reverse } \ell)
\end{aligned}$$

This idea can be generalized without too much trouble to arbitrary recursive spiders; details are omitted. (A little care is needed to ensure that the `throw` and `go` primitives choose the same edge.)

6 Reasoning About Spiders

We often want to compare the behavior of two spiders or webs, especially to claim that they behave the same way. To make these claims precise, we can define a standard form of *contextual equivalence* for the Spider Calculus. As usual, we begin with the concept of a *barb*, then define both strong and weak versions of barbed bisimulation, bisimilarity, and equivalence.

6.1 Contextual Equivalence

Since spiders can only interact with each other via modifications to the shape of the web, the interesting characteristics of a web are embodied in the evolution of its graph's shape. The barb $W \downarrow_i^x$ indicates the existence of a particular edge at a particular node: $W \downarrow_i^x$ means $W \equiv \nu \tilde{x}. (i \xrightarrow{x} j \mid V)$, where x and i are not in \tilde{x} (though j may be). We define a *strong barbed bisimulation* R to be a symmetric relation on webs such that, whenever $V R W$,

- if $V \downarrow_i^x$, then $W \downarrow_i^x$, and
- if $V \longrightarrow V'$, then $W \longrightarrow W'$ with $V' R W'$, for some W' .

The largest strong barbed bisimulation, denoted \sim , is called *strong barbed bisimilarity*.

To define strong barbed equivalence, we introduce *contexts*, which are webs with exactly one occurrence of Nil. We use C and D to range over contexts, and use the notation $C[W]$ to mean the web where the one occurrence of Nil in C is replaced by the web W . Strong barbed equivalence is denoted by \sim^c ; $V \sim^c W$ means $C[V] \sim C[W]$ for all contexts C .

The weak versions of these relations rely on the observation predicate $W \Downarrow_i^x$, which means there is a web W' with $W \longrightarrow^* W'$ and $W' \downarrow_i^x$. Then, a *weak barbed bisimulation* R is a symmetric relation on webs such that whenever $V R W$,

- if $V \Downarrow_i^x$, then $W \Downarrow_i^x$, and
- if $V \longrightarrow^* V'$, then $W \longrightarrow^* W'$ and $V' R W'$, for some W' .

Weak barbed bisimilarity is the largest weak barbed bisimulation, denoted \approx , and *weak barbed equivalences* like $V \approx^c W$ mean $C[V] \approx C[W]$ for all contexts C .³

When there is no ambiguity, we will abbreviate $[S]^i \approx^c [T]^i$ to $S \approx^c T$ (and similarly for the other relations defined in this section).

Lemma 6.1. *The following facts about bisimulations hold.*

1. *The identity relation $Id_{\mathcal{W}}$ is both a strong barbed bisimulation and a weak barbed bisimulation.*
2. *Each class of bisimulation is closed under composition, inverse, and union.*
3. *Strong barbed equivalence is a strong barbed bisimulation, and weak barbed equivalence is a weak barbed bisimulation.*
4. *Strong barbed bisimulations are weak barbed bisimulations, too.*
5. *Structural congruence \equiv is a strong barbed bisimulation.*

³Later, we will vary which primitive actions are available. One subtlety of the above definitions is that they implicitly depend on the available primitives (via the reduction relation); this means that any results about these relations will need to explicitly state which actions are available.

6.2 Correctness of the π Encoding

We would like to talk about the relationship between π processes and their encodings; however, the encodings are *longer* than the π processes in the sense that it takes many reduction steps in the spider calculus to emulate one reduction step of the π calculus. This means that to precisely characterize the relationship between the two calculi, we must allow a single π process to be related not only to its encoding, but also to reductions of translations where a particular π redex has been chosen for emulation, but has not yet executed all of its reduction steps. The $P \sim_{\pi} W$ relation, which can be found in the full version of the paper, formalizes this notion. The correctness of the encoding is stated in terms of this relation: any step in the π world can be mirrored in the spider world by related webs, and any step in the spider world can be mirrored in the π world by related processes.

7 The Design Space of Spider Primitives

There is a broad class of Spider Calculus variants that share the basic web and spider syntax and semantics, but choose different sets of primitive actions. The particular set we have presented so far is minimal (in the sense that they are mutually independent) and expressive (in the sense that many other useful actions can be encoded in them), but there are other reasonable choices with equivalent expressive power. To go beyond this degree of expressiveness, we can consider two primitives that strictly expand the power of the system: **merge** and **absent**. Adding **merge** doesn't greatly change the fundamental character of the system, but we only know of one example where it is needed: encoding the **open** primitive of ambients; **absent**, on the other hand, seems to be both more generally useful and a more radical extension.

As evidence of the independence of the primitives we have chosen, we provide two things for each: a schematic of a graph transformation that is possible using that primitive (and perhaps some others), and an invariant that is preserved by the other primitives but that is broken by the given transformation. The schematics are meant to be interpreted “up to garbage” – for example, ignoring chunks of the web with no free names. Figure 19 lists the transformations and invariants. For example, consider the **throw** action. Define a *self-link* to be a link that points from and to the same node. We observe that execution of each of the other actions (**create**, **copy**, **go**, **rename**, and **reverse**) preserves a critical property: if the web before the action's execution had no self-links, then the web after the action's execution also has no self-links. The **throw** primitive, however, can violate that property; the web

$$W = i \xrightarrow{x} j \mid i \xrightarrow{y} j \mid [\text{throw } x \text{ over } y]^i$$

has no self-links, but executing the **throw** at node i will create a self-link named x at j . Therefore, no sequence of other primitives can emulate a **throw**.

There are several additional primitives (that are not necessarily independent). The **build** and **self** primitives (shown in Figure 20) are sometimes convenient, but are readily emulated with the primitives that are already available. The **merge** primitive shown there makes it possible to “join” two nodes together; after the merge, the joined node contains all the spiders and edges from both nodes. The **merge** primitive is inspired by the **open** action of mobile ambients, and is needed to make a natural encoding of ambients in the Spider Calculus. However, most other uses of **merge** can be replaced with other primitives like **copy** and **go**. Formalizing **merge** requires some work—including the addition of a syntactic form for identifying two nodes—but does not cause any significant theoretical problems.

There is one final category of primitives to consider. We can think of **rename** as an action that tests whether an edge exists; for example **rename** x to x is an action that waits for a link named x to appear, then continues. Is there a way to wait until an edge *disappears*? In short, the answer is no, and there are concrete tasks that require such a test; for example, if we would like to do something when two nodes have different identities. Thus, we considered adding the **absent** primitive, which only executes when an edge with a given name does *not* exist. Unfortunately, spiders with such primitives violate a key monotonicity property, namely that if $W \longrightarrow W'$ then $V \mid W \longrightarrow V \mid W'$.

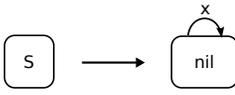
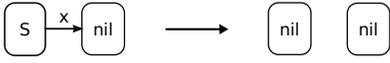
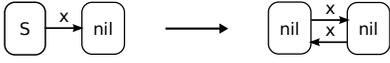
Primitive	Invariant	Graph transformation
create	the number of links	
copy	the number of nodes with two incoming links	
go	the number of non-nil spiders at j	
throw	the number of self links	
rename	the existence of visible links	
reverse	the number of cycles	

Figure 19: Reductions that support the independence of the primitives.

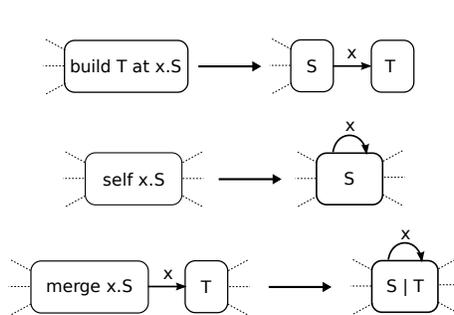


Figure 20: Pictorial representation of the semantics of some additional primitives

Purpose	Condition	Possible primitives	Additional power
node creation		create build	
mobility	with create	go	
cycle creation	with create	self copy	merge
mobility and cycle creation	with build, choose two	go copy	merge
topology changes		throw	
hiding		rename	
edge reversal		reverse	
absence detection			absent

Figure 21: A recipe for choosing directed primitives

Figure 21 summarizes our exploration of these primitives. To create a minimal system that is at least as expressive as the Spider Calculus, simply choose one primitive from each row in the table. Some primitives influence the choices of others; these conditions are shown in the second column. For example, a system that has `create` for node creation will need `go` and one of `self`, `copy`, or `merge` to achieve our benchmarks. A system with `build` for node creation can recover mobility and cycle creation from any two of `copy`, `go`, and `merge`. Figure 22 shows how to emulate any primitives you may leave out during this process. We conjecture that each of the encodings given preserve weak barbed congruence, i.e., if $\llbracket \cdot \rrbracket$ is the encoding for any one of the primitives (and expanded with syntactic congruence rules), then

Conjecture 7.1. $\llbracket W \rrbracket \approx^c W$

8 The Undirected Spider Calculus

Another natural variant of the Spider Calculus involves considering undirected webs, that is, webs in which links are visible to spiders on both ends of the link. Syntactically, we distinguish undirected links $i \overset{x}{\leftrightarrow} j$ from directed links $i \overset{x}{\rightarrow} j$, and introduce a structural congruence axiom $i \overset{x}{\leftrightarrow} j \equiv j \overset{x}{\leftrightarrow} i$ that allows undirected links to be “reversed.” We will also distinguish primitives that act on undirected links from primitives that act on directed links; Figure 23 gives the replacement for the syntactic category M of primitives, Figure 24 gives the definition of free names for them, and Figure 25 gives their operational semantics. A variant that includes both directed and undirected links could potentially also include some semantic rules allowing the mixing of edge types, such as the rules suggested in Figure 26.

The undirected Spider Calculus may actually be more *fundamental* than the directed system: it is possible to emulate a directed graph in an undirected graph by fixing a globally known name dir , then expanding directed edges $i \overset{x}{\rightarrow} j$ into a pair of undirected edges with a *waypoint* node $\nu k. i \overset{x}{\leftrightarrow} k \mid k \overset{dir}{\leftrightarrow} j$. Figure 27 shows this pictorially, using a dotted box to indicate the scope restriction of the waypoint node. Figure 28 gives an encoding of all the directed primitives; we conjecture that extending this to an encoding of directed webs gives a correct emulation of directed webs in undirected ones.

Theorem 8.1. *For any directed web W , if $W \longrightarrow W'$, then $\llbracket W \rrbracket \longrightarrow \llbracket W' \rrbracket$.*

Conjecture 8.1. *For any directed web W , if $\llbracket W \rrbracket \longrightarrow W'$, then there is a directed web W'' such that $W \longrightarrow W''$ and $W' \approx \llbracket W'' \rrbracket$.*

Finally, we have constructed a recipe (similar to the one given in Section 7) for choosing variants of the undirected Spider Calculus with different sets of primitives. Figure 29 summarizes our findings. Figure 30 gives the encodings.

<pre> [[create x. S]] = build nil at x. [[S]] </pre>	<pre> [[self x. S]] = νt. create t. merge t. [[S]] </pre>
<pre> [[self x. S]] = νt. build (copy t as t. throw t over t) at t. reverse t. rename t to x. [[S]] </pre>	<pre> [[self x. S]] = νt. create t. copy t as t. reverse t. go t. copy t as x. throw x over x. go t. [[S]] </pre>
<pre> [[copy x as y. S]] = νt, t'. rename x to t. self t'. throw t' over t. go t. self t. throw t over t'. go t'. rename t to x. rename t to y. [[S]] </pre>	<pre> [[build S at x. T]] = νt. create t. copy t as t. reverse t. go t. ([[S]] go t. rename t to x. [[T]]) </pre>
<pre> [[go x. S]] = νt, t'. build (merge t'. [[S]]) at t. copy x as t'. throw t' over t </pre>	

Figure 22: Writing directed primitives in terms of each other.

$$M ::= \text{ucreate } x \mid \text{ugo } x \mid \text{ucopy } x \text{ as } y \mid \text{urename } x \text{ to } y \mid \text{uthrow } x \text{ over } y$$

Figure 23: Undirected spider primitives

$$\begin{array}{ll}
\text{fn}(\text{urename } x \text{ to } y. S) = \{x, y\} \cup \text{fn}(S) & \text{fn}(\text{ucreate } x. S) = \{x\} \cup \text{fn}(S) \\
\text{fn}(\text{ucopy } x \text{ as } y. S) = \{x, y\} \cup \text{fn}(S) & \text{fn}(\text{ugo } x. S) = \{x\} \cup \text{fn}(S) \\
\text{fn}(\text{uthrow } x \text{ over } y. S) = \{x, y\} \cup \text{fn}(S) &
\end{array}$$

Figure 24: Free names for undirected spider primitives

$$[\text{uthrow } x \text{ over } y. S]^i \mid i \overset{x}{\leftrightarrow} j \mid i \overset{y}{\leftrightarrow} k \quad (\text{red-throw})$$

$$\longrightarrow [S]^i \mid k \overset{x}{\leftrightarrow} j \mid i \overset{y}{\leftrightarrow} k$$

$$[\text{ucreate } x. S]^i \longrightarrow [S]^i \mid \nu j. i \overset{x}{\leftrightarrow} j \quad (\text{red-create})$$

where $j \notin \{i, x\}$

$$[\text{urename } x \text{ to } y. S]^i \mid i \overset{x}{\leftrightarrow} j \longrightarrow [S]^i \mid i \overset{y}{\leftrightarrow} j \quad (\text{red-rename})$$

$$[\text{ucopy } x \text{ as } y. S]^i \mid i \overset{x}{\leftrightarrow} j \longrightarrow [S]^i \mid i \overset{x}{\leftrightarrow} j \mid i \overset{y}{\leftrightarrow} j \quad (\text{red-copy})$$

$$[\text{ugo } x. S]^i \mid i \overset{x}{\leftrightarrow} j \longrightarrow [S]^j \mid i \overset{x}{\leftrightarrow} j \quad (\text{red-go})$$

Figure 25: Semantics of the undirected primitives

$$[\text{throw } x \text{ over } y. S]^i \mid i \overset{x}{\leftrightarrow} j \mid i \overset{y}{\leftrightarrow} k \quad (\text{red-dthrow-hybrid})$$

$$\longrightarrow [S]^i \mid k \overset{x}{\leftrightarrow} j \mid i \overset{y}{\leftrightarrow} k$$

$$[\text{uthrow } x \text{ over } y. S]^i \mid i \overset{x}{\leftrightarrow} j \mid i \overset{y}{\leftrightarrow} k \quad (\text{red-uthrow-hybrid})$$

$$\longrightarrow [S]^i \mid k \overset{x}{\leftrightarrow} j \mid i \overset{y}{\leftrightarrow} k$$

Figure 26: Optional semantics for the hybrid system.

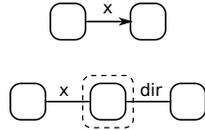


Figure 27: How to encode a directed edge into undirected webs.

$\llbracket \text{copy } x \text{ as } . S \rrbracket_d = \nu x', c, t. S_1$ $S_1 = \text{urename } x \text{ to } x'. S_2$ $S_2 = \text{ucrosslink } t \text{ on } x' \text{ dir. } S_3$ $S_3 = \text{ucreate } c. S_4$ $S_4 = \text{uthrow } t \text{ over } c. S_5$ $S_5 = \text{ugo } c. S_6$ $S_6 = \text{urename } t \text{ to } \text{dir. } S_7$ $S_7 = \text{ugo } c. S_8$ $S_8 = \text{urename } c \text{ to } x. S_9$ $S_9 = \text{urename } x' \text{ to } x. \llbracket S \rrbracket_d$	$\llbracket \text{reverse } x. S \rrbracket_d = \nu y. S_{10}$ $S_{10} = \text{urename } x \text{ to } y. S_{11}$ $S_{11} = \text{ugo } y. S_{12}$ $S_{12} = \text{urename } \text{dir} \text{ to } x. S_{13}$ $S_{13} = \text{ugo } y. S_{14}$ $S_{14} = \text{urename } y \text{ to } \text{dir. } \llbracket S \rrbracket_d$
$\llbracket \text{throw } x \text{ over } y. S \rrbracket_d = \nu x', y', t. S_{15}$ $S_{15} = \text{urnboth } x \ y \text{ to } x' \ y'. S_{16}$ $S_{16} = \text{ucrosslink } t \text{ on } y' \ \text{dir. } S_{17}$ $S_{17} = \text{uthrow } x' \ \text{over } t. S_{18}$ $S_{18} = \text{ugo } t. S_{19}$ $S_{19} = \text{urename } x' \ \text{to } x. S_{20}$ $S_{20} = \text{ugo } t. S_{21}$ $S_{21} = \text{urename } y' \ \text{to } y. \llbracket S \rrbracket_d$	$\llbracket \text{create } x. S \rrbracket_d = \nu z. S_{22}$ $S_{22} = \text{ucreate } z. S_{23}$ $S_{23} = \text{ugo } z. S_{24}$ $S_{24} = \text{ucreate } \text{dir. } S_{25}$ $S_{25} = \text{ugo } z. S_{26}$ $S_{26} = \text{urename } z \ \text{to } x. \llbracket S \rrbracket_d$
$\llbracket \text{rename } x \text{ to } y. S \rrbracket_d =$ $\text{urename } x \ \text{to } y. \llbracket S \rrbracket_d$	$\llbracket \text{go } x. S \rrbracket_d = \nu x', t. S_{27}$ $S_{27} = \text{urename } x \ \text{to } x'. S_{28}$ $S_{28} = \text{ucrosslink } t \ \text{on } x' \ \text{dir. } S_{29}$ $S_{29} = \text{urename } x' \ \text{to } x. S_{30}$ $S_{30} = \text{ugo } t. \llbracket S \rrbracket_d$
$\text{ucrosslink } x \ \text{on } a \ b. S = \nu y, z. S_{31}$ $S_{31} = \text{ucopy } a \ \text{as } y. S_{32}$ $S_{32} = \text{ugo } y. S_{33}$ $S_{33} = \text{ucopy } b \ \text{as } z. S_{34}$ $S_{34} = \text{uthrow } y \ \text{over } z. S_{35}$ $S_{35} = \text{ugo } z. \text{ugo } y. S_{36}$ $S_{36} = \text{urename } y \ \text{to } x. S$	$\text{urnboth } x \ y \ \text{to } x' \ y'. S = \nu \ell. S_{37}$ $S_{37} = \text{ucreate } \ell. !S_{38}$ $S_{38} = \text{udelete } \ell. S_{39}$ $S_{39} = \text{urename } x \ \text{to } x'. S_{40}$ $S_{40} = \text{utryrn } y \ \text{into } y' \ \text{do } S \ \text{orelse } S_{41}$ $S_{41} = \text{urename } x' \ \text{to } x. S_{42}$ $S_{42} = \text{urename } y \ \text{to } y. S_{43}$ $S_{43} = \text{ucreate } \ell$
$\text{utryrn } x \ \text{into } y \ \text{do } S \ \text{orelse } T = \nu \ell, n. S_{44}$ $S_{44} = S_{45} \mid S_{46} \mid S_{48}$ $S_{45} = \text{ucreate } \ell$ $S_{46} = \text{udelete } \ell. S_{47}$ $S_{47} = \text{ucreate } n. T$ $S_{48} = \text{urename } x \ \text{to } y. (S_{49} \mid S_{50})$ $S_{49} = \text{udelete } \ell. S$ $S_{50} = \text{udelete } n. S_{51}$ $S_{51} = \text{urename } y \ \text{to } x$	

Figure 28: Converting directed primitives to undirected ones.

Purpose	Condition	Possible primitives	Additional power
node creation		ucreate ubuild	
cycle creation		ucopy uself	umerge
mobility	with ubuild	ugo	umerge
mobility	with ucreate	ugo	
node identification	with ucreate		umerge
topology changes		uthrow	
hiding		urename	
absence detection			uabsent

Figure 29: A recipe for full-featured sets of undirected primitives

$\llbracket \text{ugo } x. S \rrbracket = (t, t')$ ubuild (umerge $t'. \llbracket S \rrbracket$) at t . ucopy x as t' . uthrow t' over t	$\llbracket \text{ubuild } S \text{ at } x. T \rrbracket = \nu t$. ucreate t . ugo t . $(\llbracket T \rrbracket \mid \text{ugo } t. \text{urename } t \text{ to } x. \llbracket S \rrbracket)$
$\llbracket \text{uself } x. S \rrbracket = \nu t$. ucreate t . umerge t . urename t to x . $\llbracket S \rrbracket$	$\llbracket \text{uself } x. S \rrbracket = (t, t')$ ucreate t . ucopy t as t' . ugo t' . uthrow t over t' . ugo t' . urename t to x . $\llbracket S \rrbracket$
$\llbracket \text{uself } x. S \rrbracket =$ ubuild (umerge $x. \llbracket S \rrbracket$) at x	$\llbracket \text{ucreate } x. S \rrbracket =$ ubuild nil at $x. \llbracket S \rrbracket$
$\llbracket \text{ucopy } x \text{ as } y. S \rrbracket = \nu t$. uself t . uthrow t over x . urename t to y . $\llbracket S \rrbracket$	

Figure 30: Writing undirected primitives in terms of each other

9 Related Work

Mobile ambients [5], a key inspiration for the Spider Calculus, structure processes into edge-labeled trees, much like the Spider Calculus’s edge-labeled graphs. Many Spider Calculus actions are direct analogues of mobile ambient actions; in particular, the `in x` and `out` capabilities of mobile ambients are analogous to the `go x` action of spiders, and the `open` capability is analogous to `merge`—and they cause similar sorts of headaches (the danger of code injection, etc.)! An interesting difference is that, whereas `open` is a fundamental operation of ambients, omitting `merge` from the Spider Calculus still leaves us with an expressive language.

Safe ambients [19] and boxed ambients [3] propose different restrictions that permit a more controlled style of programming than original ambients. Safe ambients introduce co-actions, and only allow actions to fire when they are properly paired with an identical co-action in the destination ambient. In essence, an action represents a request and a co-action represents permission. Boxed ambients replace `open` with special communication channels between a parent and its children. Processes stay within a single ambient, and only the communication topology changes. The Spider Calculus has no analog of safe ambients’ co-actions or of boxed ambients’ restricted mobility; indeed, the introduction of either feature seems at first glance to severely cripple the calculus. But it would be interesting to look for similar restrictions with better properties.

The brane calculus [4] shares many ideas with safe ambients, but it has two kinds of locations (the “membrane” and the “fluid”), which must alternate within the tree. This leads us to wonder whether it might be interesting to study a generalization of the Spider Calculus with processes at both nodes and links.

The distributed π -calculus [14] and Nomadic Pict [25] both have explicit, named locations. Processes can go to any location whose name they know, that is, the connection topology for any given process is the complete graph on the set of known location names. The formal presentation of the Spider Calculus also has explicit locations, but processes themselves cannot refer to the location names in any way; only the local connectivity—which may not be complete—is known. The distributed π -calculus’ notion of located channels is quite similar to the Spider Calculus’ notion of located links.

The 3π calculus [8] proposes an alternative form of location. Rather than keeping locations abstract, 3π uses a 3D coordinate system (actually, a full 3D affine transformation) as its set of locations. This provides great expressiveness for modeling physical systems.

The Chorus language [20] proposes another approach—intuitively rather similar to that of the Spider Calculus—to computing in graphs. Chorus models synchronization with neighborhoods: variables in any particular neighborhood are synchronous, and the basic operations merge and split neighborhoods. Synchronization in the Spider Calculus is implicit: certain edges may be interpreted as locks. Nevertheless, we believe that Chorus and the Spider Calculus may both prove to be good models for the same sorts of sparse parallel algorithms: physical simulations, computing spanning trees, n -body problems, social networking simulations, and sparse matrix computations.

Bigraphs [22] offer an abstract framework for a great variety of process calculi, including located ones. Bigraphs share some characteristics with spiders—in particular, nodes and named edges. However, the structure of nodes and edges differ significantly from webs: nodes are arranged in a tree structure, they have prescribed *arities* telling how many edges must be incident to them, and edges can connect arbitrarily many nodes. The result is that the nodes and edges of bigraphs are used for significantly different purposes than the nodes and edges of a web. Nevertheless, it seems likely that the Spider Calculus could be presented as an instance of the bigraph framework.

The Distributed Join Calculus [10] extends the join calculus (a pi-calculus variant) by adding locations and primitives for mobility. Although locations are arranged in a tree, processes may migrate directly to any location whose name they know. When a process migrates, it brings along any sublocations as well. In the Spider Calculus, by contrast, processes cannot refer directly to names of locations; only local migration along named edges is allowed. Also, the Spider Calculus separates process migration primitives from primitives that modify the topology.

Meld [1] is another language for distributed computing on networks with shifting connection topologies, but with some key differences. Like the Spider Calculus, there are nodes and links arranged in a graph structure, and computation occurs at the nodes, but in Meld, each node runs the same program. Meld’s roots are in logic programming; each node generates base facts (representing local connectivity or observations

of the node’s environment, for example) and inferred facts (which may be inferred from either facts local to the current node or facts from logically connected nodes). Nodes themselves do not influence the connection topology, and migration does not make sense, since all nodes are executing the same program.

Since the Spider Calculus is concerned with local transformations of graph structures, we should also mention *graph rewriting* (also called *graph transformations* or *graph grammars*) [9]. The root of its foundational theory lies in the sixties, as an extension of the theory of string transformations, and its applications range from layout algorithms to robotics, where new frameworks and tools (e.g., embedded graph grammars) have been introduced and used [16]. Connections between graph grammars and process calculi have been studied in terms of semantics and behavioural theory [12, 17, 13]. One interesting question in this connection is what forms of graph rewriting can be encoded in the Spider Calculus.

10 Future Work

We have presented only informal arguments for the correctness of our programs. In particular, the theory of contextual equivalence in the spider calculus is not yet sufficiently developed to prove the correctness of the pi-calculus encoding or the encoding of directed webs into undirected ones. More broadly, our examination of related work suggests several possible directions of interest: relating the Spider Calculus to graph rewriting or improving safety via typing, co-actions, or more restrictive primitives.

10.0.1 Acknowledgments

Many thanks to Davide Sangiorgi, Giorgio Ghelli, Andrew Gordon, David Walker, Michael Kearns, Martin Hofmann, and Arnaud Sahuguet for insightful discussions and pointers to proof techniques. Thanks also to the moca mailing list, who suggested many relevant papers.

References

- [1] M.P. Ashley-Rollman, P. Lee, S.C. Goldstein, P. Pillai, and J.D. Campbell. A Language for Large Ensembles of Independently Executing Nodes. In *Proceedings of the 25th International Conference on Logic Programming*, page 280. Springer, 2009.
- [2] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *nature*, 324:4, 1986.
- [3] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. *Lecture Notes in Computer Science*, pages 38–63, 2001.
- [4] L. Cardelli. Brane calculi. In *CMSB*, volume 4, pages 257–278. Springer, 2004.
- [5] L. Cardelli, G. Ghelli, and A.D. Gordon. Mobility types for mobile ambients. *Lecture Notes in Computer Science*, 1644:230–239, 1999.
- [6] L. Cardelli and A.D. Gordon. Types for mobile ambients. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 79–92. ACM New York, NY, USA, 1999.
- [7] L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.
- [8] Luca Cardelli and Philippa Gardner. Processes in Space. Technical Report 4, Imperial College London Department of Computing, 2009.
- [9] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of graph grammars and computing by graph transformation: vol. 3: concurrency, parallelism, and distribution*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

- [10] C. Fournet and G. Gonthier. The join calculus: a language for distributed mobile programming. *Lecture Notes in Computer Science*, 2395:268–332, 2002.
- [11] M.J. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with Coral.
- [12] F. Gadducci. Graph rewriting for the pi-calculus. *Mathematical Structures in Computer Science*, 17(3):407–437, 2007.
- [13] F. Gadducci and U. Montanari. A concurrent graph semantics for mobile ambients. *Electr. Notes Theor. Comput. Sci.*, 45, 2001.
- [14] M. Hennessy. *A distributed pi-calculus*. Cambridge Univ Pr, 2007.
- [15] M.R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997.
- [16] E. Klavins. Programmable Self Assembly. *IEEE Control Systems Magazine*, 27:43–56, 2007.
- [17] B. König. A graph rewriting semantics for the polyadic calculus. In *ICALP Satellite Workshops*, pages 451–458, 2000.
- [18] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L.P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, page 222. ACM, 2007.
- [19] F. Levi and D. Sangiorgi. Mobile safe ambients. *ACM Transactions on Programming Languages and Systems*, 25(1):1–69, 2003.
- [20] R. Lubliner, S. Chaudhuri, and P. Černý. Parallel programming with object assemblies. 2009.
- [21] B. Maggs. Global internet content delivery. In *Proc. 1st IEEE/ACM Int. Symposium on Cluster Computing and the Grid, CCGrid*, 2001.
- [22] R. Milner. *Space and Motion of Communicating Agents*. Cambridge Univ Press, 2009.
- [23] D. Sangiorgi and D. Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2003.
- [24] G. Unel, F. Fischer, and B. Bishop. Answering reachability queries on streaming graphs. *Stream Reasoning*, 2009.
- [25] PT Wojciechowski and P. Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, 2000.